275.2

Foundations - Computers, Technology, & Security Book 2

SANS | GIAC CERTIFICATIONS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

SANS Foundations - Book 2

Table of Contents

Programming 1

Contents

This module is a basic introduction to programming in Python, a high-level language. We will first cover what a computer program is and how it works, before delving into writing basic programs. The content we will cover includes:

- What is a computer program?
- What different kinds of programming languages are there?
- What is Python?
- Running Python programs
- Printing text to the screen
- Variables
- Manipulating string variables
- Manipulating number variables
- How to convert types of variables to other types
- Lists, and how to modify them
- Dictionaries, and how to modify them
- Code comments

What is a Computer Program?

A computer program is a series of instructions to the processor of a computer. When you run a computer program, the instructions contained in the program are loaded into memory and then read by the processor. The processor performs each instruction in turn while the program runs.

Think of a computer program a bit like a shopping list. When you write the shopping list, it's a list of things you need to buy at the supermarket. When you are in the supermarket, you go through the list item by item, picking up each item in turn and placing it in your shopping basket.

Programming Languages

Computer programs aren't written in plain English like a shopping list would be. They are written in a programming language. Computers cannot understand English, but they also cannot understand programming languages. Programming languages exist for the benefit of humans, making it easier for us to write computer programs. Before a processor can understand the code that you've written, it first must be converted to a format that the processor understands. If you recall, the processor only understands binary, meaning all code that is 'native' to the processor is pure binary.

Compiled vs Interpreted

Depending on the programming language used, the process of converting that code into something the processor can understand is either 'compiling' or 'interpreting' that code. The difference being, when a programming language is compiled, it is converted into machine code and then saved that way as an executable file. When a programming language is interpreted, the code is converted and executed at the time the user runs the program, and the converted instructions are not saved. Programs written in interpreted languages are often just text files with the code in them. To execute a computer program written in an interpreted language, you have to feed the text file into the programming language's interpreter.

Computer programs that have been compiled are faster to execute because the conversion process has already happened. The downside is that you have to compile a version of the program for every different processor architecture. So if you wanted to run a program compiled for an intel x86_64 processor on an intel x86 processor, it would fail. Computer programs that are interpreted aren't limited by the processor architecture, but on the other hand, they are slower to execute because every time you run the program, the conversion process has to take place.

Programming Languages

Programming languages are often separated into the categories of high-level and low-level languages. A low-level programming language is closer to how the processor thinks, meaning low-level languages are quite difficult to learn. A high-level programming language, on the other hand, is just the opposite. It's a language where the code you write is more abstract from the machine code that it produces. By using high-level languages, much of the complexity of programming is removed, with the hard parts being done automatically by the compiler or interpreter.

So why would we use low-level programming languages when we have high-level ones? High-level programming languages remove a lot of the complexity, but that complexity still exists, it's just hidden from the programmer. That means there is a cost in efficiency. A programmer using a low-level language has more control over how the processor does things, which a programmer using a high-level language could never achieve.

That isn't to say there is no place for high-level programming languages either; it's just that each type of language has its place and its uses. If you want to write a program quickly and have it work and you don't care about the performance, use a high-level language. If you want to write a program that needs to maximise performance (such as an operating system, like Windows or Linux), then use a low-level language.

In this course, we are going to focus on two programming languages. We are going to look at Python and C.

Python

Python is a high-level programming language. While Python can be compiled, it is usually used as an interpreted language. It is considered very easy to use and very powerful as a programming language. It's very popular in the cyber security community because of its ease of use, and many security tools are written in this language.

C

C isn't strictly a low-level programming language, but it's close. Many people consider C to be the mother of all programming languages because a lot of other languages (particularly high-level languages) are written in C. For example, the Python interpreter is written in C. C is an important language to understand, even if you aren't very good at writing in it, because it allows us to explore aspects of programming that high-level languages hide from us, such as memory management.

Git & Version Control Systems

We can't really talk about programming without first briefiy touching on version control systems. Version control systems are powerful tools that enable you to track changes to programming projects. Using version control will enable you to look back at your code at any point in time in the past, collaborate with other developers and merge the work of many into a working product. The most prominent version control system at the moment is Git, and there are many services out there that can host Git projects. One of the most prominent is GitHub.

The Repository

A repository (or repo) is the place where a project lives. You can 'clone' a repository to your computer - taking a copy of it at that point in time. After you make changes, you can then add your changes to a 'commit' and give the commit a descriptive name. Once you have a commit, you can 'push' your changes up to the remote repository.

Commits

A commit is an incremental change to the repository. Every commit has a unique identifier and you can browse each commit to see what was changed and even revert the repository to how it was at the time of a particular commit.

Origin & Master

In Git terminology, the origin is the location of the remote repository and the master is the name of the default branch in the git repository. When you are ready to push a commit, you must tell git which location and which branch you wish to push your changes to. You do not necessarily have to push to master, you can use any other branch name.

Branches

Branches in Git allow you to work on specific features independently, without touching the code in master. Then, at a future date, you could merge the branch back into master when you are ready. A typical workfiow may be:

- The master branch is the production state of your application.
- The dev branch is the testing state of your application, which would be what you test on.
- Feature branches, one for each new feature in development.

Feature branches might then be merged into the dev branch for testing, and then the dev branch may then be merged into master once it is ready for production.

Of course, you are not obliged to use this workflow, this is just one of many examples of what is possible.

Merging & Pull Requests

Once you have branches, you would usually want to merge them at some point to consolidate all the new code. Typically, this would be done through a pull request (pr). A pull request will allow you to review all the code that would be changed in the merge and either approve or reject it. Typically, development teams would have rules in place surrounding pull requests. One example of a rule might be that two developers must review the request before it can be merged.

Pull requests on public repositories allow strangers to collaborate with you on your project. Anyone could contribute to your project, and then create a pull request, leaving it up to you to decide if you want to incorporate their code or not. This is the foundation of the open source movement.

Git Hands On

In this lab exercise you are going to be working with a project called `test-project` on a git server running inside your demo environment. This is very similar to using GitHub or a similar service. You will step through some of the major functionality developers use day to day.

Note this lab is tough, there are lots of commands you can mis-type and it is worth doing a few times until you can achieve the following without having to follow the instructions:

- Clone a repository
- Modify and push changes to master
- Create a development branch or two
- Modify and push changes to those branches
- Merge a branch in to master
- Tidy up a no longer needed branch
- Revert commits and push, preserving history
- Reset local changes and use git clean

Treat this as a challenge and try making some of our own files, folders and branches. If you get stuck do not be afraid to use the git man page, or search on the web!

Running Python Programs

We'll provide you with an online editor for running Python programs throughout this course, but it's also important to know how you would run these programs on your own. First, you'll need the Python interpreter, which you can find here: https://www.python.org/. Alternatively, Linux users can install Python using their package manager, and a version of Python comes with Mac OS X by default.

You'll notice that there are two versions of Python: Python 2 and Python 3. Which one you use is very important, because there are minor differences, which could cause problems. We'll be teaching you in Python 3, so make sure you get the correct version of the interpreter. Python 3 is newer than Python 2, though there is still significant fragmentation with some projects that have not been updated to Python 3.

Once you have the interpreter installed, you can open up your command line and run a Python script using:

```
$ python script.py
```

Where script.py is the filename of the Python program you want to run. Notice the file extension for Python scripts is '.py'.

Note: It can be useful to first determine which version of Python is installed on your system as the default using:

```
$ python --version
```

No matter which version is the default, there can sometimes be both Python 2 and Python 3 available on a system. You could find out by typing either:

```
$ python2 --version
```

or:

```
$ python3 --version
```

Printing in Python

Let's get to writing our first Python program. On most programming courses, the very first program you write tends to just output 'Hello, World!' to the screen. Let's not break with tradition.

Remember, computer programming is all about accuracy. One missing semi-colon, or using a lower-case letter where a capital letter is expected, will cause the entire program to fail. Pay attention to detail!

Variables

What are Variables?

That last section wasn't so interesting. That's because you can't really do anything cool in any programming language without using variables. A variable is a way of storing data in programs.

Think of a variable like a box with a label on it. Let's say your box has a label that says "Box A". Then you put a whole bunch of socks in the box. If someone asks you for some socks, you can just say "Oh, you'll want to look in Box A". But if, one day, you decide to remove all the socks and store some shoes in that box instead, you would still refer to that box as "Box A". The contents may have changed, but the label or the way of referring to the box remains the same.

Assigning values to a variable

In most programming languages, to assign a value to a variable, we use the equals (=) operator. Python follows this standard, so in Python, assigning a message to our variable looks like this:

```
user_text = "James was here."
print(user_text)
```

At any time later in my program, I can change the value of the `user_text` variable again like so:

```
user_text = "Emily was here."
print(user_text)

user_text = "Have you seen James lately?"
print(user_text)
```

The above code will create the below output.

```
Emily was here.
Have you seen James lately?
```

Naming Variables

You can name a variable almost anything you want, but there are a few guidelines you should follow to prevent errors, and to make your program easier for humans to read and understand. After all, programming languages are for people, not for computers, so we should always be thinking about the people trying to understand our code when we create variables.

- Variable names can contain only letters, numbers, and underscores.
- You can start a variable name with a letter or an underscore, but not with a number. So `variable_1` and `_variable1` are both ok, but`1_variable` won't work.
- Variable names are case sensitive:`James` and `james` are two different variables to a computer.
- You shouldn't use Python's built-in keywords or functions as variable names, such as `print` or`break`.
- Keep your variable names short but descriptive. A variable called `this_is_a_really_long_and_pointlessly_verbose_variable_name` is probably a bit long in most contexts, but likewise a variable calleda doesn't tell us much about what the variable's purpose is. Something short but also descriptive like `username` is much better.

Variable Types

Every variable has its own "type" that decides what kind of data it can store. Some of the most common data types are:

- **string:** A series of characters (basically, text). Note that these are always surrounded by quotes. If it isn't in quotes, it isn't a string. (In Python, it doesn't matter if you use single or double quotes, but in some programming languages it matters! You'll see more of this later when we talk about C.)
- **integer:** A whole number (could be positive or negative), e.g.: -2, -1, 0, 1, 2, 3...
- **float:** A number with a decimal point, eg: 3.14, 56.99998
- **boolean:** A True or False value. In some programming languages, booleans can be represented as 1 (True) and 0 (False).

Python is a dynamically typed programming language. This means you don't have to specify what type a variable is when you create it: it will shift its type depending on the context of the program. Other programming languages, like C, are statically typed. Statically typed languages require you to tell the program what type of data you want to store in the variable when you create it.

Here it an example of storing an integer variable in Python:

```
solved_cases = 42
```

In Python, if you're not sure what type your variable is at any given moment in your program, you can always check using the `type()` method.

```
variable_1 = '42'
variable_2 = 42
variable_3 = 4.2
variable_4 = True

print(type(variable_1))
print(type(variable_2))
print(type(variable_3))
print(type(variable_4))
```

The above should output:

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
```

Notice the capital "T" in "True" for the boolean we have assigned as `variable_4`? If you don't use an uppercase "T", the program will error because in Python boolean values are case sensitive. You must use "True" or "False", not "true" or "false".

Variables in Python

You should know quite a bit about variables from the last chapter. Remember, a string is surrounded by double quotes or single quotes (in Python anyway, in some languages it matters when you use a single or double quote). An integer, fioat or boolean doesn't need quotes. In Python, a boolean value must be written with a capital in the first letter, i.e. True or False.

Manipulating Strings

Strings are quite simple, but you can do a lot with this type of data. A string is a series of characters, such as a word or a sentence.

- `this` is a string.
- `So is this` is also a string.
- `<p>Text inside a paragraph tag</p>` is also a string if we want it to be.
- `lu3hd6h3309&%34-0+` is - you guessed it - a string.

You get the idea.

Strings, quotes, and escaping

In Python, anything inside quotes (" ") or (' ') is a string. The ability to use either a single or double quote gives us some fiexibility in Python, for instance:

```
sentence_1 = "Bob and Bill met for coffee at Starbucks."
sentence_2 = "James rushed by and accidentally knocked Bill's coffee on the floor."
sentence_3 = '"Oops, let me buy you a new coffee!" said James.'
```

In `sentence_2` we needed to use a single quote as an apostrophe inside the string, so we wrapped the string in double quotes. In `sentence_3` however, we needed to use the double quotes in the string, so we wrapped the string in single quotes.

But what would we do if we need to use *both* single and double quotes in our string?

```
sentence_4 = '"I spilled Bill's coffee," said James.'
print(sentence_4)
```

If we run the above, we'll get an error:

```
$ python program.py
  File "program.py", line 1
    sentence_4 = '"I spilled Bill's coffee," said James.'
                          ^
SyntaxError: invalid syntax
```

The string stored in sentence_4 uses both double quotes and single quotes as part of the string. In situations like this, we need to escape the quote within the string that's used as an apostrophe, otherwise the program will think our string is finished after the "I" in

"Bill's", and that causes an error because there are more characters the computer doesn't understand after what it thinks is the closing (').

```
sentence_4 = '"I spilled Bill\'s coffee," said James.'
```

We escape a character by using the backward slash (\). The backslash tells our program that the character immediately following it should be interpreted as part of the string rather than a signifier that the string has ended.

But what if we need to escape a backslash?

```
sentence_5 = "The backslash character '\' escapes things."
```

If we try printing out the above, it will print The backslash character '' escapes things. which isn't quite what we meant. In this case, we need another backslash.

```
sentence_5 = "The backslash character '\\' escapes things."
```

Printing the above will now do as we want: `The backslash character '\' escapes things.`

Be careful when using backslashes, as it's easy to cause an error if you don't pay attention to when your backslashes need to be escaped:

```
print("This can happens if you use a \")
```

The above gives you the below error.

```
$ python program.py
  File "program.py", line 1
    print("This can happens if you use a \")
                                          ^
SyntaxError: EOL while scanning string literal
```

Combining strings

Python lets us combine multiple strings together into one. In programming, combining strings is usually called "concatenating" strings.

```
first_name = 'Ada'
last_name = 'Lovelace'
full_name = first_name + ' ' + last_name

print(full_name)
```

The above example will output `Ada Lovelace` by concatenating 3 strings together: "Ada", the string held in the `first_name` variable; "Lovelace", the string held in the `last_name` variable; and a space, which we insert manually between these two strings.

This can be very helpful for us when programming as we can use variables as placeholders for data when the exact data is unknown, for example if we're asking for user input.

```
user_name = "Sarah"

print("Hello " + user_name + "!")
```

The above will output `Hello Sarah!`

Adding and stripping whitespace

A "whitespace" refers to a type of character that doesn't print anything out, but adds some sort of spacing formatting to the output. The spacebar, tab key, and return key on your computer all create a different kind of "whitespace".

We can add whitespace to our programs to format our output. Let's output our list of well-known agents so that each agent appears on a different line using the `\n` character combination to tell the program where we want our line breaks.

```
user_list = "Bill\nBob\nSarah\nMike"

print(user_list)
```

The above will print:

```
Bill
Bob
Sarah
Mike
```

We can add tabs to our output using the `\t` character combination, and we can even combine the new line and tabs together to create more complex layouts using

whitespace.

```
user_list = "User List:\n\tBill\n\tBob\n\tSarah\n\tMike"

print(user_list)
```

The above code will output as:

```
User List:
	Bill
  Bob
	Sarah
  Mike
```

On Windows systems, you'll need a combination of character sequences to create a new line, `\r\n`, while on Linux and Mac `\n` is enough on its own. This is why if you open a text file that someone originally wrote on a Mac on a Windows machine, the formatting will be messed up. (Thanks Bill!)

Sometimes we want to remove whitespace rather than add it in. For example, if we are copying an email address jane@email.com from our contact book and grab an extra space so the string is 'jane@email.com ', the program will see that email with a space tacked onto the end as a different string than just the email address without the space. So if we tried to compare 'jane@email.com' and 'jane@email.com ' in a program, they wouldn't match.

Python has built-in tools for helping us strip out these whitespaces when we don't want to consider them. We can strip whitespace off the left side of a string using `lstrip()` and off the right side of a string using `rstrip()`. And if we want to strip both sides at the same time, we can use `strip()`.

```
email = ' jane@email.com '

print("'" + email.lstrip() + "'\n'" + email.rstrip() + "'\n'" + email.strip() + "'")
```

The above example prints out the email with our different stripping methods, with the addition of some concatenated quotes and newline characters so we can better see what whitespace characters are being removed and which are being left.

The above will output like this:

```
'jane@email.com '
' jane@email.com'
'jane@email.com'
```

String Manipulation Practice

Being able to manipulate strings is a key skill, which we will take advantage of when we get to more cyber security focused examples. Imagine a password guessing tool where we have a dictionary of words and want to try variations of them together -- our string manipulation skills would enable us to build this!

Manipulating Numbers

When programming, numbers are very useful to us. We use them to perform mathematical comparisons of data, track our score in games, count things, and so on. Python treats numbers in a few different ways depending on how we use them.

Integers

Integers are whole numbers, and don't have a decimal in them. They can be positive or negative. Just like on a calculator, you can add (+), subtract (-), multiply (*) and divide (/) integers in Python.

```
print(4 + 2)
print(4 - 2)
print(4 * 2)
print(4 / 2)
```

For each `print()` function above, the program will output the result of the mathematical operation we put in. This will output:

```
6
2
8
2
```

That's pretty straightforward.

We can also use exponential operations, like 10 to the power of 4:

```
print(10 ** 4)
```

This will give us:

```
10000
```

We can also get the modulus, which allows us to return the remainder left over after we divide.

```
print(10 % 3)
```

This will divide 10 by 3, then print out the remainder leftover from that arithmetic.

```
1
```

Python allows us to specify the order we'd like to do our calculation in as well, using parenthesis to prioritise what operations we want done first.

```
print(8 + 4 * 10)
print((8 + 4) * 10)
```

The above will output:

```
48
120
```

Dividing Integers

Now let's take a closer look at dividing integers, because Python 2 does something a little bit unintuitive in some situations. Let's look at another division example. Remember this is python2 syntax! If you want to run it in the code editor and terminal you will need to run it as follows:

`python2 output.py`

```
print 5/2
```

If we run this, we get:

```
2
```

But that's not quite what we expect! There's a remainder missing there, since 5 doesn't divide neatly by 2. If we were doing this same operation on a calculator, we'd expect to get "2.5". In Python 2, when we divide integers it always gives us an integer back and truncates (removes) everything after the decimal.

If we want to get a decimal number back from our operation, we'll need to make sure we give it at least one decimal value to start.

```
print 5.0 / 2
print 5 / 2.0
print 5.0 / 2.0
```

All three of the above will print the same thing `2.5`.

Python 3 is comparatively much more sensible here, for example:

```
print(5/2)
```

This returns `2.5`, which is far more sensible. Be aware of versions and rounding results.

Floats

Floats are decimal numbers. For the most part, you can perform calculations with them similar to how we do with integers.

When we work with fioats, Python provides us with the useful function `round()` to manipulate and manage them.

```
print(round(42.12345, 2))
print(round(42.99912, 2))
```

The `round()` function does exactly what you'd expect: it rounds decimal numbers up or down, to a given preferred decimal place.

The sample code above will output:

```
42.12
43.0
```

We've asked for our fioats to be rounded to the nearest 2 decimal places. Notice in the second example, we're given `43.0` because with the rounding, to 2 decimal places we'd end up with `43.00`, but Python will remove the unnecessary second "0" in this case.

Type Conversion

Remember that each variable has its own datatype Since Python is dynamically typed the type of the variable depends on the data we assigned to it.

Sometimes you need to convert one type of data to another. For example, consider a case where our program asks the user to input a number, but we want to use this number to create an output that is a string.

If we do:

```
solved_cases = 14

print("Sarah has closed " + solved_cases + " cases this week!")
```

We'll get the following error:

```
Error(s), warning(s):
Traceback (most recent call last):
  File "program.py", line 3, in <module>
  print("Sarah has closed " + solved_cases + " cases this week!")
TypeError: cannot concatenate 'str' and 'int' objects
```

Concatenating only works with strings; we can't concatenate a string and an int together.

In order to get the output we want without triggering an error, we need to modify the variable type using the `str()` function like so:

```
solved_cases = 14
solved_cases_str = str(solved_cases)

print("Sarah has closed " + solved_cases_str + " cases this week!")
```

Running this gives us what we're looking for:

```
Sarah has closed 14 cases this week!
```

And just like we can change an integer to a string, we can also change a string to an integer (as long as the string only contains numbers).

```
morning_coffee = '3'
afternoon_coffee = 2
total_coffee = int(morning_coffee) + afternoon_coffee

print(total_coffee)
```

By using the `int()` function, we changed the type of `morning_coffee` which was a string (note the quotation marks) to an integer so that we could add the day's coffee totals together.

You can also convert to a float using the `float()` method:

```
pocket_money = 5
coffee_price = '3.50'
money_left = float(pocket_money) - float(coffee_price)

print(money_left)
```

The above code will turn both the integer `pocket_money` and the string `coffee_price` into floats that we can then do some arithmetic on.

Next, let's look at a more interesting type conversion, an integer to a boolean:

```
flag_found = 1
is_flag_found = bool(flag_found)

print(is_flag_found)
```

Notice this one is a bit different: when we print `is_flag_found`, which has been turned from the integer 1 into a boolean value, we get `True`. If we set `flag_found` to 0, the value of `is_flag_found` when printed would be `False`.

What happens if we use any number other than 1 or 0?

In Python, any number that isn't a 0 will convert to `True` using `bool()`. Even a negative number will convert to `True`.

Additionally, when we use `bool()` on a string value, like so:

```
flag_found = 'yes'
is_flag_found = bool(flag_found)

print(is_flag_found)
```

This will always print `True` except in a couple of notable circumstances. If we make `flag_found = None` or `flag_found = ''` then `bool()` will make these values `False` instead. Give it a try in the editor.

Lists and Tuples

A list is exactly what it sounds like; a list of things stored in a particular order. It could be anything: the numbers from 1 - 10; a list of animals; all the nail polish colours you could name... for example:

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print(fav_linux_distros)
```

If we print a list like we have above, we'll get the whole list returned to us like so:

```
['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
```

We probably won't find too much use for this list in this form: it will be more useful for us to be able to access individual items within this list using its index position within the list.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print(fav_linux_distros[0])
print(fav_linux_distros[1])
print(fav_linux_distros[2])
```

The above example will output:

```
Mint
Debian
Ubuntu
```

Note that when we want to access the very first item in the list, we use 0 instead of 1. This is true of most programming languages: whenever you're counting "things" in computer code, we always start with 0. So if we're looking for the fifth item in Mike's Linux distributions list, we'd use `fav_linux_distros[4]`.

What happens if we try to use an index outside the number of items in the list?

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print(fav_linux_distros[7])
```

Here we've asked for the item in the list at index 7... but we only have 6 items in the list. If nothing exists at the index we've requested, we'll get an `IndexError` error back instead.

```
$ python program.py
Traceback (most recent call last):
  File "program.py", line 3, in <module>
  print(fav_linux_distros[7])
IndexError: list index out of range
```

Python also gives us a shortcut way of getting the very last item in a list.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print(fav_linux_distros[-1])
```

By asking for an item at the index of -1, Python will always return the last item in the list. This is helpful because we don't always know how long our list will be. This syntax extends to other negative index methods as well, letting you count backward from the end of the list rather than forward from the front. Let's try it out.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print(fav_linux_distros[4])
print(fav_linux_distros[-2])
```

This code will output:

```
Fedora
Fedora
```

When we pluck an item from a list like this, we also get its individual item type. In the Linux distributions example we get a string, which we can then manipulate as a string using our various string manipulation tools.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
top_fav_distro = fav_linux_distros[0]

print("Mike's favourite Linux distro is " + top_fav_distro.upper() + "!")
```

This gives us a string where we've modified an item from our list to appear in all uppercase, to best communicate how much Mike loves Mint.

```
Mike's favourite Linux distro is MINT!
```

We can also get a quick count of the things in our list by using the `len()` function.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print(len(fav_linux_distros))
```

This will print out `6` which is indeed the number of items in Mike's list.

Mixed type lists

Python lets us create mixed-type lists, incorporating all different types of data into a single list.

```
misc = ['purple', 99, 3.14, False]

print(misc[0])
print(misc[1])
print(misc[2])
print(misc[3])
```

And the result:

```
purple
99
3.14
False
```

We can even put lists inside of other lists, like this:

```
misc = ['purple', 99, 3.14, False, ['apple', 'orange', 'pear']]

print(misc[0])
print(misc[1])
print(misc[2])
print(misc[3])
print(misc[4])
print(misc[4][0])
print(misc[4][1])
print(misc[4][2])
```

Which outputs as:

```
purple
99
3.14
False
['apple', 'orange', 'pear']
apple
orange
pear
```

Notice how we access the individual items "apple", "orange" and "pear" inside the inner list? The inner list is at index 4 of the primary list, and "apple" is at index 0 of the inner list. So to get the first item inside the inner list, we use `misc[4][0]`.

Modifying a list

What happens when Agent M wants to update the list of Linux distributions? How do we modify the list we have with new information?

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
print(fav_linux_distros)

fav_linux_distros[0] = 'Elementary'
print(fav_linux_distros)
```

We've created a list and saved it to the variable `fav_linux_distros`, with `Mint` as the first item in the list. Then we've changed the value of the first item to `Elementary`.

We've printed the list out before and after we've made the change so you can see what it looks like at both stages. Only the first item in the list has changed; everything else remains the same.

```
['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
['Elementary', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
```

We can also modify a list by applying a sort to it. Mike likes things to be nice and orderly, so let's sort this list of Linux distros alphabetically.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
fav_linux_distros.sort()

print(fav_linux_distros)
```

As we hoped, this gives us:

```
['Arch', 'Debian', 'Fedora', 'Manjaro', 'Mint', 'Ubuntu']
```

And afterwards, if we want to frustrate Agent M we can always reverse this list.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
fav_linux_distros.sort()

print(fav_linux_distros)

fav_linux_distros.reverse()
print(fav_linux_distros)
```

Our output from the above code would be:

```
['Arch', 'Debian', 'Fedora', 'Manjaro', 'Mint', 'Ubuntu']
['Ubuntu', 'Mint', 'Manjaro', 'Fedora', 'Debian', 'Arch']
```

Adding items to lists

There are a few different ways to add things to lists. Let's start by simply adding things onto the end of our list.

```
travel_bucket_list = ['Tokyo', 'Hawaii', 'London']
travel_bucket_list.append('New York')
travel_bucket_list.append('Berlin')

print(travel_bucket_list)
```

Here we use the`append()` method, which allows us to stick items onto the end of a list. When we print out our list after appending our new items, we can indeed see they've been added.

```
['Tokyo', 'Hawaii', 'London', 'New York', 'Berlin']
```

We can even start with an empty list.

```
travel_bucket_list = []
travel_bucket_list.append('Toronto')
```

```
travel_bucket_list.append('Barcelona')
travel_bucket_list.append('Dubai')

print(travel_bucket_list)
```

As expected, we've taken an empty list in line 1, and turned it into a list with 3 items, by appending them one by one.

```
['Toronto', 'Barcelona', 'Dubai']
```

We can also add items into an existing list at any position we want. Let's try adding a new item to the front of our travel bucket list.

```
travel_bucket_list = ['Tokyo', 'Hawaii', 'London']
travel_bucket_list.insert(0, 'New York')

print(travel_bucket_list)
```

Now we've put New York to the front of our list, and bumped everything else down one. Our output shows us our new list:

```
['New York', 'Tokyo', 'Hawaii', 'London']
```

Removing items from lists

Sometimes we want to remove an item from our list, for instance, if we travelled to one of the places on our travel bucket list. Let's remove 'London' from our list.

```
travel_bucket_list = ['Tokyo', 'Hawaii', 'London']
visited = travel_bucket_list.pop()

print(travel_bucket_list)
print(visited)
```

We can see how we've changed our list, and also that we've been able to "pop" London out of the list into the variable`visited` so that we can continue to use it later in our program if we want.

```
['Tokyo', 'Hawaii']
London
```

The `pop()` method has taken the last item from the list and "popped" it off. If we wanted to remove a different item from the list, we can use `pop()` with an index to specify which item we want popped out.

```
travel_bucket_list = ['Tokyo', 'Hawaii', 'London']
visited = travel_bucket_list.pop(1)

print("I recently went to " + visited)
print(travel_bucket_list)
```

This outputs:

```
I recently went to Hawaii
['Tokyo', 'London']
```

What if we don't know what the position is of the thing we want to remove? Perhaps we wrote our list a long time ago, and can't remember what index position "Hawaii" is at. How do we remove Hawaii from our list then? Voila:

```
travel_bucket_list = ['Tokyo', 'Hawaii', 'London']
travel_bucket_list.remove('Hawaii')

print(travel_bucket_list)
```

The `remove()` method will allow us to remove an item from our list without specifying an index. A very handy method to remember. As expected, the above code gets us the following output:

```
['Tokyo', 'London']
```

Tuples

Tuples are very similar to lists, except for 1 important quality: they are immutable. "Immutable" means the content items in the list can't be changed once they are set. This can sometimes be useful to a programmer if we want to ensure a piece of data can't be changed. The use of a tuple forces us to copy the information and modify that copy, leaving the original intact and unchanged.

Here's an example of how a tuple is declared and used.

```
stonehenge = ('51.1739726374', '-1.82237671048')
print('Stonehenge latitude: ' + stonehenge[0])
print('Stonehenge longitude: ' + stonehenge[1])
```

We've used a tuple for the coordinates of Stonehenge, since we don't want anyone to be able to change them: they're immutable, because Stonehenge isn't going anywhere (at least not anytime soon). Notice how we've used round bracket(s) instead of square brackets [ ] when we created our tuple? That's how Python knows it's a tuple we want instead of a list.

We can access the individual items in our tuple exactly the same way we access items in a list. The output of our code above is shown below.

```
Stonehenge latitude: 51.1739726374
Stonehenge longitude: -1.82237671048
```

What happens if we try to modify one of our values?

```
stonehenge = ('51.1739726374', '-1.82237671048')
stonehenge[0] = 'something else'
```

Here's the output we get when we try to run this code:

```
$ python program.py
Traceback (most recent call last):
  File "program.py", line 2, in <module>
    stonehenge[0] = 'something else'
TypeError: 'tuple' object does not support item assignment
```

Are lists just arrays?

If you're familiar with other high-level programming languages, you've probably played with a data structure called an "array" before. A list looks an awful lot like an array, and for most use cases it will more or less act like one. In this course, you can think of a "list" and an "array" as being the same thing.

However, there are some subtle differences in Python between a list and an actual array, which you can import and use. We'll leave you to Google for the difference if you're interested to dig into the details.

Dictionaries

A dictionary in Python is similar to a list in that it's a collection of "things" we can store together. Where it differs from a list is the format in which that information is stored and retrieved.

If we wanted to represent some closed and open case statistics, in a list we'd probably store it like this:

```
case_stats = [21, 12, 8]

print('Total cases: ' + str(case_stats[0]))
print('Solved cases: ' + str(case_stats[1]))
print('Unsolved cases: ' + str(case_stats[2]))
```

This code generates the following output:

```
Total cases: 21
Solved cases: 12
Unsolved cases: 8
```

This seems straightforward enough, but it's pretty fragile code. What if the list gets reordered or reversed? Or if a new stat gets added into the list? Then we'd have to update all our print functions that call on a specific index of our list for their data.

This is the kind of problem a dictionary is made for. It allows us to associate information together in helpful pairs called "key-value pairs", which helps in making our data more understandable as well as making it easier for us to get precisely the information we want from it without worrying about where it is in the list.

Using a dictionary, we'd represent the case statistics like this:

```
case_stats = {'total': 21, 'solved': 12, 'unsolved': 8}
```

Notice the difference between a list and a dictionary? The list uses square bracket notation `[]` whereas a dictionary uses curly bracket notation `{}`. The dictionary also uses a "key: value" notation: `'total'`, `'solved'`, and `'unsolved'` are all keys, and `21`, `12`, and `8` are all values.

The format for dictionaries is always the same: `key: value`. In the example above, since our key is a string, we've also wrapped our key in quotes (') while the values here are integers so they don't need quotes. Strings are always in quotes, even in dictionaries.

You can also use an integer as the key and a string as the value: this is a completely valid dictionary as well, though perhaps not as useful for us in this case (and we'll see why later in this section). But the below example won't cause an error. Keys and values are pretty fiexible parts of a dictionary.

```
case_stats = {21: 'total', 12: 'solved', 8: 'unsolved'}
```

So how do we get data from our dictionary into our output?

```
case_stats = {'total': 21, 'solved': 12, 'unsolved': 8}

print('Total cases: ' + str(case_stats['total']))
print('Solved cases: ' + str(case_stats['solved']))
print('Unsolved cases: ' + str(case_stats['unsolved']))
```

Instead of using the index number to find an item in our dictionary, we tell our output what keyto look for, and it will output thevalueit finds associated with that key.

This is great, because now it doesn't matter what order our items are stored in: as long as it has the same key, we'll always get the value we're looking for.

These kinds of data structures are used in a lot of programming languages, especially high-level ones. Sometimes they go by different names: in PHP these are called "associative arrays", in Ruby they're known as "hashes", and in Java they're known as "maps". But they all behave more or less the same, and follow a similar structure of key-value pairing in the syntax.

Lists inside dictionaries

Like with lists, we can store different types of data within a dictionary. Let's add some more information to our dictionary to describe Agent Q's case statistics and try out using a few more different data types.

```
case_stats = {'total': 21, 'solved': 12, 'unsolved': 8, 'month': 'June',
'percent_solved': 57.14, 'types': ['forensics', 'cryptography', 'web app']}

print('Month: ' + str(case_stats['month']))
print('Total cases: ' + str(case_stats['total']))
print('Solved cases: ' + str(case_stats['solved']))
print('Unsolved cases: ' + str(case_stats['unsolved']))
print('Percent solved: ' + str(case_stats['percent_solved']) + '%')
print('Types of cases: ')
print('\t' + str(case_stats['types'][0]))
print('\t' + str(case_stats['types'][1]))
print('\t' + str(case_stats['types'][2]))
```

The above code gives us the following output:

```
Month: June
Total cases: 21
Solved cases: 12
Unsolved cases: 8
Percent solved: 57.14%
Types of cases:
    forensics
    cryptography
    web app
```

Notice how we've embedded a list inside our dictionary linked to the key types? We can access the individual items in this list using the list index, which we've done here where we've called `case_stats['types'][0]`. This tells the computer to look for the variable called 'case_stats' which is a dictionary, and within that dictionary look for the key 'types', and within that list look for the item at index 0, which is 'forensics' in our example above.

Dictionaries inside dictionaries

Unsurprisingly, we can also put dictionaries inside our dictionaries, like so:

```
case_stats = {'month': 'June', 'stats': {'total': 21, 'solved': 12, 'unsolved': 8,
'percent_solved': 57.14}, 'types': ['forensics', 'cryptography', 'web app']}

print('Month: ' + str(case_stats['month']))
print('Total cases: ' + str(case_stats['stats']['total']))
print('Solved cases: ' + str(case_stats['stats']['solved']))
print('Unsolved cases: ' + str(case_stats['stats']['unsolved']))
print('Percent solved: ' + str(case_stats['stats']['percent_solved']) + '%')
print('Types of cases: ')
print('\t' + str(case_stats['types'][0]))
print('\t' + str(case_stats['types'][1]))
print('\t' + str(case_stats['types'][2]))
```

This code will give us the exact same output we saw in our previous example, but gives our dictionary more structure. We can start to see how powerful dictionaries can be when it comes to organizing our data. But we can also see how quickly dictionaries can get large and difficult to read. Since programming languages are for humans, let's use some formatting to make our code easier for us to read so we can see the structure of our dictionary data more clearly.

```
case_stats = {
        'month': 'June',
        'stats': {
                'total': 21,
                'solved': 12,
```

```
'percent_solved': 57.14
                },
                    'types': ['forensics', 'cryptography', 'web app']
        }

print('Month: ' + str(case_stats['month']))
print('Total cases: ' + str(case_stats['stats']['total']))
print('Solved cases: ' + str(case_stats['stats']['solved']))
print('Unsolved cases: ' + str(case_stats['stats']['unsolved']))
print('Percent solved: ' + str(case_stats['stats']['percent_solved']) + '%')
print('Types of cases: ')
print('\t' + str(case_stats['types'][0]))
print('\t' + str(case_stats['types'][1]))
print('\t' + str(case_stats['types'][2]))
```

There we go, that's aloteasier for us to read. Any time you're building a dictionary that's more than a few key-value pairs long, it's helpful to format it like this to better understand what you're building.

Dictionaries inside lists

We can also create a list of dictionaries. Let's create a list with some simple case stats for Bill, Susan, and James.

```
case_stats = [
        {'employee': 'Bill', 'solved': 12, 'unsolved': 8},
        {'employee': 'Susan', 'solved': 15, 'unsolved': 5},
        {'employee': 'James', 'solved': 8, 'unsolved': 3}
]

print('Bill:')
print('\t Solved cases:' + str(case_stats[0]['solved']))
print('\t Unsolved cases:' + str(case_stats[0]['unsolved']))

print('Susan:')
print('\t Solved cases:' + str(case_stats[1]['solved']))
print('\t Unsolved cases:' + str(case_stats[1]['unsolved']))

print('James:')
print('\t Solved cases:' + str(case_stats[2]['solved']))
print('\t Unsolved cases:' + str(case_stats[2]['unsolved']))
```

This code gives us the below output:

```
Bill:
        Solved cases:12
    Unsolved cases:8
Susan:
    Solved cases:15
        Unsolved cases:5
James:
        Solved cases:8
    Unsolved cases:3
```

Mixing dictionaries and lists in different ways is incredibly powerful for us. For now, let's concentrate on how we can modify and build dictionaries on the fiy.

Modifying dictionaries

So what happens when Bill solves another case? Now we need to update his stats. Let's modify our dictionary.

```
case_stats = {'month': 'June', 'total': 21, 'solved': 12, 'unsolved': 8}

print('Stats for ' + case_stats['month'] + ':')
print('\tTotal cases: ' + str(case_stats['total']))
print('\tSolved cases: ' + str(case_stats['solved']))
print('\tUnsolved cases: ' + str(case_stats['unsolved']))

case_stats['month'] = 'July'
case_stats['total'] = 22
case_stats['solved'] = 13

print('Stats for ' + case_stats['month'] + ':')
print('\tTotal cases: ' + str(case_stats['total']))
print('\tSolved cases: ' + str(case_stats['solved']))
print('\tUnsolved cases: ' + str(case_stats['unsolved']))
```

It's a new month, so we need to update a few pieces of information in Bill's stats. The first is, of course, the month itself. We do that by calling on the `month` key in `case_stats` with `case_stats['month']` and giving it a new value, `'July'`: `case_stats['month'] = 'July'`. Then we also update 2 more pieces of information in our dictionary following the same technique: the total case number needs to go up by 1, which makes it 22, and the total number of solved cases also needs to go up by 1, which is 13. Agent Q's unsolved cases number didn't change in July, so we can leave that one as it is.

The above code will output:

```
Stats for June:
    Total cases: 21
    Solved cases: 12
    Unsolved cases: 8
Stats for July:
    Total cases: 22
    Solved cases: 13
    Unsolved cases: 8
```

There's another way to update the numbers here that will prove very useful to us later, because it means we don't have to do the math ourselves. We might as well let the computer do the arithmetic, and sometimes we don't actually know what the original number was. We just know we want to increase whatever it was by a specific value. In the

example above, we want to increase Bill's total cases and solved cases both by 1. Let's modify our code to get the computer to do this addition for us.

```
case_stats = {'month': 'June', 'total': 21, 'solved': 12, 'unsolved': 8}

print('Stats for ' + case_stats['month'] + ':')
print('\tTotal cases: ' + str(case_stats['total']))
print('\tSolved cases: ' + str(case_stats['solved']))
print('\tUnsolved cases: ' + str(case_stats['unsolved']))

case_stats['month'] = 'July'
case_stats['total'] = case_stats['total'] + 1
case_stats['solved'] = case_stats['solved'] + 1

print('Stats for ' + case_stats['month'] + ':')
print('\tTotal cases: ' + str(case_stats['total']))
print('\tSolved cases: ' + str(case_stats['solved']))
print('\tUnsolved cases: ' + str(case_stats['unsolved']))
```

This will give us exactly the same output as before, but notice how we've modified lines 9 and 10. In line 9 we're setting `case_stats['total']` to itself plus 1. The computer will first retrieve the current value of `case_stats['total']`, which is 21, then it will add 1 to that value to get 22, and finally it will set the new value of `case_stats['total']` to be 22.

There's actually a useful short-hand way of specifying exactly this kind of 'self-incrementing' behaviour so we don't have to write out `case_stats['total'] + 1`, which is a bit wordy. We can use the `+=` notation in Python to help us out here.

```
case_stats = {'month': 'June', 'total': 21, 'solved': 12, 'unsolved': 8}

print('Stats for ' + case_stats['month'] + ':')
print('\tTotal cases: ' + str(case_stats['total']))
print('\tSolved cases: ' + str(case_stats['solved']))
print('\tUnsolved cases: ' + str(case_stats['unsolved']))

case_stats['month'] = 'July'
case_stats['total'] += 1
case_stats['solved'] += 1

print('Stats for ' + case_stats['month'] + ':')
print('\tTotal cases: ' + str(case_stats['total']))
print('\tSolved cases: ' + str(case_stats['solved']))
print('\tUnsolved cases: ' + str(case_stats['unsolved']))
```

We've changed lines 9 and 10 to use the `+=` notation, so now instead of the longer statement `case_stats['total'] = case_stats['total'] + 1` we can use our short hand `case_stats['total'] += 1` which does exactly the same thing.

Adding things to dictionaries

Like we did with lists before, we can add things to our dictionary and even start with an empty one if we want. Let's build Bill's stats dictionary from the ground up. We'll build up our dictionary slowly and output it at different states so we can watch how it's built.

```
case_stats = {}

case_stats['month'] = 'June'
print(case_stats)

case_stats['total'] = 21
print(case_stats)

case_stats['solved'] = 12
print(case_stats)

case_stats['unsolved'] = 8
print(case_stats)
```

The above code will output:

```
{'month': 'June'}
{'total': 21, 'month': 'June'}
{'solved': 12, 'total': 21, 'month': 'June'}
{'solved': 12, 'unsolved': 8, 'total': 21, 'month': 'June'}
```

Adding a new key-value pair to our dictionary is exactly the same as updating an existing key-value pair. If the computer finds the key in the dictionary, it will update the value. If it doesn't, it will add the key and value to the front of the dictionary. And since order doesn't matter in dictionaries (because we access everything we need with the key) we don't mind what order things are in.

Removing things from dictionaries

If you want to remove information from a dictionary, you can use `del`:

```
case_stats = {'month': 'June', 'total': 21, 'solved': 12, 'unsolved': 8}
print(case_stats)

del case_stats['unsolved']

print(case_stats)
```

In this example, we've printed out our `case_stats` dictionary before and after we've used `del` to remove the 'unsolved' key and its corresponding value, which you can see in the output:

```
{'solved': 12, 'unsolved': 8, 'total': 21, 'month': 'June'}
{'solved': 12, 'total': 21, 'month': 'June'}
```

Pretty straightforward, but remember: once you've deleted a key-value pair from a dictionary it's gone for good!

Dictionary Practice

Code Comments

Code comments are a prime example of how programming languages are mostly for us humans, as they are entirely for humans alone. Code comments are pieces of information you put in the code that the computer ignores. This allows us to add notes and narration into our code without having to worry about syntax, to help make it clear to ourselves - and to others - what our program is intended to do.

Here's an example of how to add a comment to your code.

```
# Say hello to the user
print("Hello User!")
```

You can also add a comment in line next to the greeting like this:

```
print("Hello User!")  # Say hello to the user
```

In the output of both the above examples, we see just the printed "Hello User!" statement and no indication of our commented line above it, or the commented text next to it.

```
Hello User!
```

Comments can be very useful to us when our code starts to get complicated, or during the "building" stages when we're still figuring out how to get our program to work.

If you need to add several lines of comments in a block, you can use the # notation before each new line.

```
# One comment line
# Second comment line
# Third comment line
print("Hello!")
```

When to use comments

Comments are particularly helpful in 3 situations:

When you're building a new program.

As we're trying to build out code to complete a series of tasks - especially if that task is a complicated one - using comments as a way to think through the logic can be extremely helpful. We can remind ourselves what the goal is of a particularly tricky bit of code, and if we have to come back to work on our program again later, comments can help remind us what we're building, and where we got to when last we worked on it.

Documenting your program for others.

Later on in this course we'll talk more about programming style and how to write clean, organised, well-structured code that shouldn't need lots of comments, but all programs benefit from an amount of inline documentation that code comments can provide. In particular, if you're working with other programmers, or if your program will need to be maintained by others in the future, code comments can help them understand what your program is doing, and the broader context for why you built the program the way you did if it's important.

When you want to temporarily "turn off" parts of your code.

Sometimes while building or debugging, it can be helpful to quickly turn on or off smaller chunks of our code. Instead of deleting the code, or copy-pasting it somewhere else, then having to re-write it or remember where in the program it was meant to go, we can simply comment the code out inline.

```
print("Hello Bill.")
# print("Hello Susan.")
print("Hello Mike.")
```

In the example above, we've commented out the `print("Hello Susan")` line, which means when we run this program, the computer will skip over that entire line and only print the "Hello Bill" and "Hello Mike" lines.

Bringing it Together

Use the skills from this module to correct the broken code. Debugging and methodically fixing, with trial and improvement, is a key skill in IT and cyber security.

Programming 2

Introduction

In this module we will be covering a plethora of new programming concepts in Python, and putting them to practice too. We will use them to start to write more resilient code, and to solve our problems more elegantly. Again, the goal of this course' programming section is not to train to be a staff developer or full time engineer, but to have the understanding and skills to create solution scripts in cyber security. These concepts are also invaluable in understanding how mistakes made during development can lead to security issues. In this module we will cover:

- Conditionals (if statements)
- Loops, including for loops and while loops
- Functions
- Getting and using user input
- Classes and objects
- Exceptions

Conditionals

The previous module gave us a solid foundation in how to manipulate data, create and manage different data types, and how to store those bits of data to variables where we can use this data elsewhere in our program. This module builds on that foundation and allows us to start making decisions about what to do with our data and when to change it in more complex ways.

A conditional allows us to check if certain conditions are being met before we run certain parts of the program. They're often referred to more casually as "if statements" or "if tests" and there's a very good reason for that: in most every programming language conditionals usually start out with an "if", as in "if coffee is available, then Agent J will have a coffee". Let's see our simple "if statement" coffee example in Python.

```
coffee_available = True

if coffee_available == True:
    print("Agent J will have coffee.")
```

In this example, since we've set the `coffee_available` variable to `True`, our test passes, so the program prints out "Agent J will have coffee.". If we change the `coffee_available` variable to `False` and run the program again, nothing will be printed to the screen: since the conditions of our if statement aren't met, the code inside the if statement never runs.

Notice here when we write our if statement, we use a double equal sign (==) instead of a single equals sign. The single one is reserved for assigning values to variables, like we did in line 1. So when we want to say X = Y in a conditional if statement, we use the double equals syntax X == Y.

Note also the indentation in our conditional if statement: the line `print("Agent J will have coffee.")` is indented from the line above it. This indentation structure is enforced by Python, and it's how Python knows which code below the if statement is within it and which code is outside it.

If we try to run this code without indentation, we get the following error:

```
Error(s), warning(s):
  File "program.py", line 4
    print("Agent J will have coffee.")
        ^
IndentationError: expected an indented block
```

We can see this inside vs outside structure more clearly when we run the following code, which includes 2 conditional statements one after the other, as well as some printed

output outside the conditionals.

```
print("Agent J arrives at HQ in the morning.")

coffee_available = True

if coffee_available == True:
    print("Agent J will have coffee.")

print("In the afternoon, Agent J goes to the HQ cafe.")

coffee_available = False

if coffee_available == False:
    print("Agent J is shocked to discover the cafe is out of coffee!")

print("Agent J goes home.")
```

In this example, when we run our program, we get the following output.

```
Agent J arrives at HQ in the morning.
Agent J will have coffee.
In the afternoon, Agent J goes to the HQ cafe.
Agent J is shocked to discover the cafe is out of coffee!
Agent J goes home.
```

But what happens when we switch our `coffee_available` assignments around the other way, so that the first one is `False` and the second one is `True`?

```
print("Agent J arrives at HQ in the morning.")

coffee_available = False

if coffee_available == True:
        print("Agent J will have coffee.")

print("In the afternoon, Agent J goes to the HQ cafe.")

coffee_available = True

if coffee_available == False:
    print("Agent J is shocked to discover the cafe is out of coffee!")

print("Agent J goes home.")
```

In this example, neither of the if statement conditions are met, so the print methods inside those statements never run. But the print statements outside the conditions aren't affected, so they still output to the screen as expected.

```
Agent J arrives at HQ in the morning.
In the afternoon, Agent J goes to the HQ cafe.
Agent J goes home.
```

At their heart, all if statements can be evaluated as either `True` or `False`. That doesn't mean we can only test a boolean variable, but it does mean that when the test is run it must result in either a `True` or `False` outcome.

So we can create tests to see if 2 strings are the same, for example.

```
drink_available = 'coffee'

if drink_available == 'coffee':
    print("Agent J will have coffee.")
```

This example checks to see if the `drink_available` variable contains a string that matches 'coffee'. If it is, the statement evaluates to `True` and the print method inside the statement is executed. If the `drink_available` variable doesn't match, everything inside the if statement is skipped over, so nothing would be printed.

It's important to remember that case matters to computers! We may think of 'coffee', 'Coffee', and 'COFFEE' as being all the same thing in everyday language, but to a computer all these strings are different. If we don't know what kind of letter case we'll get, but we want our conditional to consider 'coffee' and 'COFFEE' as a match, we need to tweak our code slightly.

```
drink_available = 'COFFEE'

if drink_available.lower() == 'coffee':
    print("Agent J will have coffee.")
```

Now we're transforming the value of `drink_available` to its lowercase version using the `lower()` method, which takes a string and changes any uppercase letters it finds within the string to their lowercase counterparts. Now our 2 strings match, so the statement evaluates to `True`, and our sentence will print out.

Let's look at another example.

```
coffee_available = 4
coffee_needed = 4

if coffee_needed == coffee_available:
   print("There is enough coffee.")
```

Here we're comparing the values of 2 different variables, `coffee_available` and `coffee_needed`. If they are the same, then we print the sentence `There is enough coffee.` and if there isn't, then we don't print anything. In this example, 4 and 4 match, so the statement evaluates as true, printing out our sentence.

But what happens here:

```
coffee_available = "4"
coffee_needed = 4

if coffee_needed == coffee_available:
    print("There is enough coffee.")
```

We've set `coffee_available` to a string value of 4, and `coffee_needed` as an integer value of 4. They're both 4 so we might expect the statement would evaluate to `True`, but when we run this program we find it doesn't, nothing gets printed out, telling us our if statement evaluated to be `False`.

When comparing values, an if statement takes the type of the variable into account. If they're different types, then they won't match in a conditional. It's important to keep this in mind when we're programming - in particular with numbers - because often numbers can be stored somewhere as string data, but will need to be evaluated as integers or floats. This is where the `int()` and `str()` functions we saw in the last module can come in very handy.

```
coffee_available = "4"
coffee_needed = 4

if coffee_needed == int(coffee_available):
        print("There is enough coffee.")
```

Now our conditional will evaluate to `True` and print our sentence for us, because we're turning our string `coffee_available` value into an integer, so now they'll match.

Comparison operators

There are several different ways of comparing things in conditionals. These are called comparison operators and they include:

- Equal to ==
- Not equal to !=
- Less than <
- Greater than >
- Less than or equal to <=

- Greater than or equal to `>=`

Let's run a few examples to get a feel for how these different operators work.

```
if "string 1" == "string 1":
    print("These strings are equal.")

if "string 1" != "string_2":
    print("String 1 does not equal string 2.")

if 2 < 4:
    print("The first number is less than the second number.")

if 4 > 2:
    print("The first number is greater than the second number.")

if 4 <= 4:
    print("The first number is less than or equal to the second number.")

if 4 >= 4:
    print("The first number is greater than or equal to the second number.")
```

If we run this code, all 6 statements will print out to the screen because all 6 statements will evaluate to `True`.

Conditionals and lists

Lists are an important kind of data while programming, and conditionals also allow us to check if a value is in a list, or not in a list.

```
available_drinks = ['coffee', 'tea', 'water', 'orange juice']

if 'coffee' in available_drinks:
        print('You may have that drink.')
```

Here, we're using `in` within our if statement to see if a string exists in our `available_drinks` list. If we can find an exact match somewhere in our list - in this example we can - then the statement evaluates to `True` and our sentence will print. If, on the other hand, it can't find our string, then the statement evaluates to `False`, the print method is skipped over, and nothing outputs to the screen.

We can also use `not in` to create a test that works in the opposite way.

```
available_drinks = ['coffee', 'tea', 'water', 'orange juice']

if 'apple juice' not in available_drinks:
    print('That drink is not available.')
```

Here, we're checking to see if `apple juice` isn't in our `available_drinks` list. If it's not found in the list, then the statement evaluates to `True` and our sentence "That drink is not available" will print to the screen.

Checking multiple conditions

Let's revisit our original example code again.

```
coffee_available = True

if coffee_available == True:
    print("Agent J will have coffee.")
```

It doesn't seem quite fair to force Agent J to have coffee just because it's available, he might not want one, or perhaps already has one.

We could do it like this, by nesting if statements:

```
coffee_available = True
has_coffee_already = False

if coffee_available == True:
    if has_coffee_already == False:
        print("Agent J will have coffee.")
```

In this example we've put one if statement inside the other. First, the `if coffee_available == True` check will run. If this evaluates to `True`, then the inner if statement `if has_coffee_already == False` will run. If this also evaluates to `True` then, finally, our "Agent J will have coffee" sentence will print to the screen.

This works... but it's a bit untidy, and if we had to check 3 or 4 things it quickly becomes very difficult for us humans to read and understand. There's a much nicer way to write this conditional without all that nesting.

```
coffee_available = True
has_coffee_already = False

if coffee_available == True and has_coffee_already == False:
    print("Agent J will have coffee.")
```

Here we're using the keyword `and` to connect our two conditions in a single if statement. If both `coffee_available` equals `True` and `has_coffee_already` equals `False`, then print our sentence. If neither check passes, or if only 1 of them does, don't print the sentence. Both must be true for our code inside the if statement to execute.

We can also use the `or` keyword, which allows us to check if at least one of our conditions are met. Let's do a slightly different version of our coffee example.

```
had_coffee_already = False
is_tired = False

if had_coffee_already == False or is_tired == True:
    print("Agent J will have coffee.")
```

Let's walk through this example. We have 2 tests we need to run before we can decide if Agent J is going to have coffee. The first thing to consider is whether or not he's already had his morning coffee: if he has, he probably doesn't want another one. But then again, if he had a late night working on a particularly tricky case and is feeling tired, he may want a second cup of coffee to help wake him up. So, in plain English: if Agent J has not had coffee already `or` if Agent J is very tired, then he will have coffee.

In the example above, `had_coffee_already` is set to `False`, and `is_tired` is also set to `False`. Our conditional only needs one of these to be set to `False` to succeed, so as soon as one of our conditions is met our if statement passes and the code inside it will run.

It's important to note that the code inside an if statement that contains an `or` like this will run as soon as one condition is met. It won't bother to check the others, because it doesn't matter what they are. This can leave our code with some hidden errors if we're not diligent. Let's see how.

Consider this example:

```
had_coffee_already = False
is_tired = False

if nope == True or had_coffee_already == False:
    print("Agent J will have coffee.")
```

What happens when we run this? We get an error.

```
Traceback (most recent call last):
  File "program.py", line 4, in <module>
    if nope == True or had_coffee_already == False:
NameError: name 'nope' is not defined
```

We get this error because we tried to check the value of a variable we haven't yet created, nope. Obviously we can't check the value of a variable if it doesn't exist.

But what happens if we reverse the order of the checks in this example? Let's find out.

```
had_coffee_already = False
is_tired = False

if had_coffee_already == False or nope == True:
    print("Agent J will have coffee.")
```

And, to our surprise, even though we still haven't created the variable `nope` anywhere, our program runs and we get:

```
Agent J will have coffee.
```

This is because the computer is trying to be efficient: it will do the absolute minimum work it needs to in order to get to the next step. In this case, if the first condition checks out, then it doesn't matter what the second one is because the condition only needs one to pass. So the computer immediately jumps from the first check passing to executing the code inside the if statement.

Our code still has an error in it, and if the first check fails, then our error message will return because the computer is forced to also test if `nope == True`.

Order of precedence

Now let's examine how we can use combinations of `and` and `or` in our conditionals to create more complex if statements. We'll keep building out our coffee example to try and consider all the relevant factors.

Let's examine the different checks we want to use to evaluate whether or not Agent J gets a coffee at the HQ cafe. Here's what we want our program to do:

- The cafe has to have coffee available for him to have. If there is no coffee, then it doesn't really matter what the other tests are, because even if they pass there's no coffee to provide.
- Agent J has either not already had his morning coffee or is tired enough to want a second coffee. Either being true is enough to convince Agent J he wants coffee.

Now that we understand what we want to create, let's write this up in code.

```
coffee_available = True
had_coffee_already = False
is_tired = False
```

```
if coffee_available == True and had_coffee_already == False or is_tired == True:
    print("Agent J will have coffee.")
```

When we run this code we get what we expect, the output "Agent J will have coffee." Either `had_coffee_already` is `False` or `is_tired` is `True`, while at the same time we must always have `coffee_available` be `True`. Our variables are set in a way that should allow this conditional the pass - which it does.

Let's do one more scenario. We're going to start by assuming HQ always has plenty of coffee and never runs out. But in addition to whether or not Agent J has already had coffee or is tired, he also needs to consider how much time he has to wait for coffee. The cafe is pretty popular, and sometimes the line is long: if he's in a hurry, he may not have time to wait no matter how much he wants coffee.

So in this situation, we need to consider:

- Combination A: the line must be short **or** Agent J must have lots of time to wait.
- Combination B: he must not yet have had coffee **or** he must be tired.

In order for our check to pass, we need one check from combination A **and** one check from combination B to pass.

```
# Combination A
line_is_short = True
in_hurry = True

# Combination B
had_coffee_already = False
is_tired = False

if line_is_short == True or in_hurry == False and had_coffee_already == False or
is_tired == True:
    print("Agent J will have coffee.")
```

This seems to work. We'd expect to get our sentence printed out in this case - `line_is_short` is `True` and `had_coffee_already` is `False`, which satisfies our needs. But it's always good to check our code with a variety of inputs to see if it works as we expect in all situations. What happens if we change `had_coffee_already` to `True`? We'd expect no sentence to be printed out in this case, because `is_tired` is `False` which means both checks in Combination B would fail, so we want our if statement to fail.

But that's not what happens!

```
# Combination A
line_is_short = True
in_hurry = True

# Combination B
```

```
had_coffee_already = True
is_tired = False

if line_is_short == True or in_hurry == False and had_coffee_already == False or
is_tired == True:
    print("Agent J will have coffee.")
```

This code will alsoprint out the sentence, even though we don't want it to. What's going on here?

All these and's and or's are confusing our computer. It's not executing them in the way we expect. In situations like this where there are and's and or's mixed together, there are rules about which are evaluated first. Aandis always executed before anor.

So in our check above, this is what's happening, in order:

- `in_hurry == False and had_coffee_already == False` is evaluated. This evaluates to `False`.
- `line_is_short == True or (False) or is_tired == True` is evaluated. Since we now only have 2 or's, only 1 has to pass before the entire check is considered to have passed. Since`line_is_short == True` passes, the checks immediately stop at that point and the computer moves on to execute what is inside the if statement.

But that's not what we want! We've found a bug in our program: now it's time to squash it.

We can use round brackets( ) to give the computer clearer instructions on the order in which we want it to execute the checks. This works the same way as round brackets in mathematics does: if we see them in an equation, we always do what's inside them first. In Python, round brackets have a higher precedence than aand and so anything inside them will be checked first.

Let's add some round brackets to our code.

```
# Combination A
line_is_short = True
in_hurry = True

# Combination B
had_coffee_already = True
is_tired = False

if (line_is_short == True or in_hurry == False) and (had_coffee_already == False or
is_tired == True):
    print("Agent J will have coffee.")
```

In this revised code, we're telling the computer in what order it should resolve our checks.

- First it checks `line_is_short == True or in_hurry == False` which evaluates to `True`.
- Second, it checks `had_coffee_already == False or is_tired == True` which evaluates to `False`.
- Last, it evaluates `True and False`, which is the result of the first check and the result of the second. In order for this last check to pass, both must be `True`. Since they aren't in this case, this last check fails.

Now our code is doing what we want it to in this situation: our sentence doesn't get printed because the check doesn't pass. And if we try different combinations of values in our variables, we can see we get what we expect in every combination. Success!

Expanding conditionals with 'else' and 'elif'

Currently, our Agent J coffee check program is printing out a sentence only if the if statement passes, and if it fails we don't get any feedback. What we really want is to get different feedback depending on whether or not our test passes or fails.

To do this, we can use an `if-else` block. This allows us to create a diverging path of instructions, depending on the outcome of our check.

Let's go back to one of our simplest examples to see how this works.

```
coffee_available = True

if coffee_available == True:
    print("Agent J will have coffee.")
else:
    print("Sorry Agent J, our coffee has run out!")
```

Here we've created a case for when the check passes, and a case for when it fails. In this example, where `coffee_available = True` then we'll get "Agent J will have coffee" printed to the screen. If we change line 1 to `coffee_available = False`, then our check fails and the computer moves to execute the code in the `else` portion of our code, which means "Sorry Agent J, our coffee has run out!" gets printed to the screen instead.

But what happens if we have more than 2 possible paths we want our program to be able to follow?

```
coffee_available = True
had_coffee_already = False
is_tired = False

if coffee_available == True and (had_coffee_already == False or is_tired == True):
        print("Agent J will have coffee.")
elif coffee_available == False:
        print("Sorry Agent J, our coffee has run out!")
```

```
else:
    print("Agent J doesn't want any more coffee.")
```

In this example, we have cases that cover any combination of values that might enter our conditional. We've done this using an `elif` check, which allows us to have a second set of checks we run if the first checks fail. And finally, we finish with an `else` that catches the other possible cases not caught by the `if` and `elif` above.

It's important to remember when we construct if-elif chains that it will execute the condition in order, and stop when it finds a condition it meets. So if the first `if` check is true, then we get the "Agent J will have coffee" statement printed, and the conditional is finished executing: it won't also then check the `elif`. Think of an if-elif chain as a valve: there can only be 1 path, and the computer will take the first path it finds. In complicated programs with if-elif chains that are a few elif's long, you can get bugs that have to do with how you order your conditions.

For Loops

In the previous module, we learned about lists and dictionaries, including how to add, remove, and generally modify the information in them. But so far we've done it very slowly, one item at a time. If we had a very long list we'd end up having to write a lot of repetitive code if we needed to update everything at once.

Loops are another tool that let us manage lists and dictionaries more efficiently, with just a few lines of code. We can run the same operation on multiple items in a list.

Looping through a list

Let's have another look at Agent M's favourite Linux distributions again.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print(fav_linux_distros[0])
print(fav_linux_distros[1])
print(fav_linux_distros[2])
```

In order to print out each item in the list before, we had to put in a new `print(fav_linux_distros[0])`. We'd need 6 in all to print out the entire list, and if Agent M added or removed one later we'd have to manually modify our code. This is a perfect use case for a for loop

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

for distro in fav_linux_distros:
    print(distro)
```

And our output:

```
Mint
Debian
Ubuntu
Manjaro
Fedora
Arch
```

Let's break our `for loop` down to make sure we understand how it's working.

First, we create our list of Linux distributions as normal in line 1. In line 3, we define our for loop: `for distro in fav_linux_distros:`. This line is asking for an item to be pulled from the list `fav_linux_distros` and assigned to the variable `distro` for use

within our for loop. On line 4, we're now *inside* the loop - notice the indention! - and we have asked for the item stored in the variable `distro` we created to be printed to the screen. The computer will go through the list, one by one, and follow the instructions we've given it for each item.

It might help to think of it like this, in plain English: for every distribution in the list of Linux distributions, print the distribution's name.

As you can see, loops allow us to work with lists in a powerful and very efficient way. They don't care how many items are in a list, or if you've added some or removed some. They take the list as it is, and do something over and over for each item until it runs out of items. Handy!

A for loop can contain as many lines of code as you like. As with our if statements, the level of indention is how Python understands what code belongs inside the for loop and what code belongs outside the for loop.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print('Here is Agent M\'s list of favourite Linux distributions.\n')

for distro in fav_linux_distros:
    print(distro)
    print('\t' + distro + ' is an excellent Linux distribution.')
    print('\tWouldn\'t you agree that ' + distro + ' is an excellent distro?')

print('\nThose are Agent M\'s top ' + str(len(fav_linux_distros)) + ' Linux
distributions.')
```

Here we have a multi-line for loop example, where some of our print statements are inside the loop - so they get repeated for each item in the list - while other print statements are outside the list, so they only print once.

Here's the output this code generates:

```
Here is Agent M's list of favourite Linux distributions.

Mint
    Mint is an excellent Linux distribution.
    Wouldn't you agree that Mint is an excellent distro?
Debian
    Debian is an excellent Linux distribution.
    Wouldn't you agree that Debian is an excellent distro?
Ubuntu
    Ubuntu is an excellent Linux distribution.
    Wouldn't you agree that Ubuntu is an excellent distro?
Manjaro
    Manjaro is an excellent Linux distribution.
    Wouldn't you agree that Manjaro is an excellent distro?
Fedora
    Fedora is an excellent Linux distribution.
    Wouldn't you agree that Fedora is an excellent distro?
```

```
Arch
    Arch is an excellent Linux distribution.
    Wouldn't you agree that Arch is an excellent distro?

Those are Agent M's top 6 Linux distributions.
```

The range() function in for loops

For loops can be useful to us even if we don't have a pre-existing list to loop through. The range( ) function allows us to create a temporary 'list' of numbers within a specified range, which we can then loop through similar to a normal for loop.

Let's take a look at a simple use of the range( ) function in a for loop by creating a simple counting program.

```
for counter in range(1,5):
    print(counter)
```

This code will create the following printed output.

```
1
2
3
4
```

Hmm... that seems strange. Our range( ) function looks like it should count from 1 to 5, but it stops at 4. Why is this?

The range( ) function starts counting at the first value you specify, in this case the 1. It will run the first loop, then when it hits the end it will increment that value by 1. If that newly incremented value is the second one specified in the range( ) function - in this case 5 - then it will stop without running the code within the loop.

So if we want our program to count to 5, we instead need to specify a range of 1 to 6, so that our loop will run through the 5th iteration and stop when it hits 6.

```
for counter in range(1,6):
    print(counter)
```

The range( ) function is useful for doing more than just counting numbers. Agent S likes to be able to visualize progress of the forensic team's solved case statistics. Let's build a little progress bar to help Agent S see how the team is doing.

```
total_cases = 10
total_solved = 7
total_unsolved = total_cases - total_solved

for x in range(1,total_unsolved+1):
    print('| |')

for x in range(1,total_solved+1):
    print('|X|')
```

This code will generate us a nice little thermometer-style progress bar so Agent S can see the team's progress at a glance. Notice we've had to add 1 to our `total_unsolved` and `total_solved` variable values in our `range()` function, like in our previous example where we wanted to count to 5 but had to use `range(1, 6)`.

```
| |
| |
| |
|X|
|X|
|X|
|X|
|X|
|X|
|X|
```

Any time we want to get an updated status thermometer, all we have to change is the values of the `total_cases` and `total_solved` variables. Nice!

Looping through a dictionary

We can also loop through a dictionary. Because a dictionary stores information in a variety of ways, we have several different techniques to loop through the content of a dictionary and its key-value pairs.

Let's start with the simplest loop: all we want to do is loop through every key-value pair in the dictionary and print out what we find, creating a nice list of everything we have stored in our dictionary.

```
user_profile = {
        'name': 'Agent M',
        'fav_distro': 'Mint',
        'fav_drink': 'Tea'
        }

for key, value in user_profile.items():
    print('Key: ' + key)
    print('Value: ' + value + '\n')
```

Our for loop has a few different components here than it did when we were looping through a list. In our `for key, value in user_profile.items():` opening statement, we create 2 variables, `key` and `value`, which the loop will assign the key and value to for each item it finds in our dictionary. We've named them `key` and `value` here in this example for clarity, but you can name them anything you want: 'label' and 'contents', 'x' and 'y', 'beeble' and 'brox', etc. We then give the loop the name of the dictionary we'd like it to loop through, in this case `user_profile`. Finally, we include the method `items()` with our dictionary, which tells the loop to return a list of key-value pairs.

The above code gets us the following output:

```
Key: fav_drink
Value: Tea

Key: name
Value: Agent M

Key: fav_distro
Value: Mint
```

Note our output isn't in the same order that we originally stored them in the dictionary. That's because dictionaries are by default orderless as far as Python is concerned: it doesn't care what order the stored items are in, because it finds information using the key.

Let's look at another quick example.

```
fav_distros = {
        'q': 'Mint',
        'j': 'Kali',
        'm': 'Ubuntu',
        's': 'Elementary'
        }

for agent, distro in fav_distros.items():
    print('Agent ' + agent.upper() + ': ' + distro)
```

And our output:

```
Agent Q: Mint
Agent S: Elementary
Agent J: Kali
Agent M: Ubuntu
```

We can modify our dictionary loop to tailor it to our particular needs. For example, maybe we just want to know which distributions have been called out as favourites in our dictionary, and we don't particularly care who each favourite belongs to.

```
fav_distros = {
        'q': 'Mint',
        'j': 'Kali',
        'm': 'Ubuntu',
        's': 'Elementary'
        }

for distro in fav_distros.values():
    print(distro)
```

Here we use the`values()` method in our for loop, so we only need to create 1 variable `distro` because we are only asking the computer to grab the values it finds for each key, and to not worry about what the key is.

We can also fiip this around and ask Python to give us just the keys. I bet you can guess what this method will look like now that you've seen how we ask for just the values.

```
fav_distros = {
        'q': 'Mint',
        'j': 'Kali',
        'm': 'Ubuntu',
        's': 'Elementary'
        }

for agent in fav_distros.keys():
    print(agent)
```

The output of the above code is below. Remember, dictionaries are orderless so the order our for loop runs through the dictionary isn't necessarily the order in which we originally defined it.

```
q
s
j
m
```

Nested loops

We can of course also get combinations of dictionaries and lists inside each other, so sometimes we'll want to loop through a list or dictionary, and if the values we get back are also sets of data, we might want to loop through those too.

We're not going to go through each of the variations one-by-one, but let's look at one quick example of how these loops can nest one inside the other in order to extract complex data efficiently.

```
# Create our 4 agents as their own dictionaries.
agent_m = {
        'name': 'M',
        'distro': 'Ubuntu',
        'drink': 'Earl Grey Tea'
        }

agent_j = {
        'name': 'J',
        'distro': 'Kali',
        'drink': 'Espresso'
        }

agent_s = {
        'name': 'S',
        'distro': 'Elementary',
        'drink': 'Coffee'
        }

agent_q = {
        'name': 'Q',
        'distro': 'Kali',
        'drink': 'Decaf Coffee'
        }

# Combine our agent dictionaries into a single list of agents.
agents = [agent_m, agent_j, agent_s, agent_q]

# Loop through the list of agents so that we get the
# dictionary belonging to each individual agent
for agent in agents:
    # For each individual agent, get each key-value pair and
        # print it to the screen
    for key, value in agent.items():
        print(key.title() + ': ' + value)

    # Add an extra return space for visual clarity:
        # note this is inside our first loop, but not the second one
    print('\n')
```

We've added some comments into this code so you can follow what it's doing. This code will produce the following output.

```
Drink: Earl Grey Tea
Name: M
Distro: Ubuntu

Drink: Espresso
Name: J
Distro: Kali

Drink: Coffee
```

```
Name: S
Distro: Elementary

Drink: Decaf Coffee
Name: Q
Distro: Kali
```

Combining loops and conditionals

Our example above is ok, but still a bit untidy. Ideally we'd like the name of the agent at the top of each list.

```
# Create our 4 agents as their own dictionaries.
agent_m = {
        'name': 'M',
        'distro': 'Ubuntu',
        'drink': 'Earl Grey Tea'
        }

agent_j = {
        'name': 'J',
        'distro': 'Kali',
        'drink': 'Espresso'
        }

agent_s = {
        'name': 'S',
        'distro': 'Elementary',
        'drink': 'Coffee'
        }

agent_q = {
        'name': 'Q',
        'distro': 'Kali',
        'drink': 'Decaf Coffee'
        }

# Combine our agent dictionaries into a single list of agents.
agents = [agent_m, agent_j, agent_s, agent_q]

# Loop through the list of agents so that we get the dictionary
# belonging to each individual agent
for agent in agents:
    # Get the agent's name and print that to the screen
    print('Agent ' + agent['name'] + ':')

    # Loop through that agent's dictionary items
    for key, value in agent.items():

            # If the key isn't the agent's name, print the key and value
        if key != 'name':
            print('\t' + key.title() + ': ' + value)

    # Add an extra return space after an agent's info
    print('\n')
```

Here we've continue to use our 2 nested loops, but in the first loop we've printed out the agent name specifically by calling for the key `agent['name']` which allows us to print it to the screen like a header for our list. In the second loop, because we've already printed the agent's name we don't need to see it again, so we use a conditional if statement within this loop to exclude this key-value pair from being printed.

The code above creates the below output.

```
Agent M:
    Drink: Earl Grey Tea
    Distro: Ubuntu

Agent J:
    Drink: Espresso
    Distro: Kali

Agent S:
    Drink: Coffee
    Distro: Elementary

Agent Q:
    Drink: Decaf Coffee
    Distro: Kali
```

As you can see, we're starting to be able to do more complex things by combining our tools together. Here we've incorporated lists, dictionaries, loops, and conditionals in just one way, but there are *lots* of ways that these tools can work together. It's when we start combining our tools together that the potential power of programming becomes more clear to us. With the right tools we can do basically anything!

While Loops

In the previous section, we saw the for loop, which takes a collection of items and loops through them one by one, executing the same block of code on each of those items, until it runs out of items.

But there's another kind of useful loop we can use: the while loop. Instead of giving the while loop a predetermined set of things to work through, we give it a condition: as long as that condition is true, the while loop will keep running.

The simplest kind of while loop usually contains some kind of counter, so let's have a look at this first.

```
counter = 1

while counter <= 3:
    print(counter)
    counter += 1

print('Loop complete!')
```

We start by creating the variable `counter` and assigning in the integer value of 1. Then we declare our while loop, and give it the condition `while counter <= 3`, which means the loop will run as long as the value of the variable `counter` is less than or equal to 3. Inside our while loop, we print the `counter` value for our reference, and then just before we leave our loop, we ask it to increment our counter variable by 1, which if you remember is the same as saying `counter = counter + 1`.

After this, we've reached the end of our while loop, so we start again by checking to see if the condition `counter <= 3` is still true. Our `counter` value is now 2, but since 2 is less than or equal to 3 our condition is still true so our while loop runs the code block again.

Once again, at the end of the block we increment our `counter` again, so now it's 3, and we start again by checking if our condition is still true. It is, because 3 = 3, so the code block within the loop runs a third time, incrementing our `counter` to 4 at the end.

Now when we check our condition for the forth time, because we've incremented our `counter` variable to 4, the condition `counter <= 3` evaluates to False. Our while loop will only continue to run if the condition is true, so now that it's false, our while loop closes and allows the computer to move on to the next block of code outside the while loop, which is our printed statement "Loop complete!".

So our final output would be:

```
1
2
```

```
3
Loop complete!
```

Using a flag to stop a while loop

The previous example is a simple one where we just want to run the while loop a certain number of times. But programs can be complicated, with lots of different branching logic. There might be many different conditions where we'd want our while loop to stop, depending on many different factors and considerations. In these situations, it's often helpful to use a flag to control your while loop.

Here's a slightly contrived example, but it gives you a reasonable idea how a flag can work.

```
active = True
countdown = 3

while active == True:
    if countdown == 0:
        countdown = 'Go!'
        active = False
    else:
        print(countdown)
        countdown -= 1

print(countdown)
```

Let's walk through this code.

First, we set and assign 2 variables: `active = True`, which we'll use as our flag to control the while loop, and `countdown = 3` which we'll use elsewhere in our code.

Next we create our while loop using `while active == True` which will force our while loop to keep running over and over until the `active` variable is set to something other than `True`.

Inside our while loop, we have a conditional if statement. Here's where our `countdown` variable comes into play. If our countdown variable is equal to 0, then we'll change its value to the string 'Go!' and set the `active` flag to False. Otherwise we follow the `else` pathway, where we'll print the current value of our `countdown` variable first before we reassign its value to itself - 1. (This works like the += we saw in the previous example, where we incremented the value of a variable by 1. This time we're decreasing the value of a variable by 1 instead.)

If the `countdown` variable is equal to anything except 0, the while loop will keep running over and over, printing the value of `countdown` then decreasing its value by 1 each time until it reaches 0. When it reaches 0, then we follow the other branch of our if statement,

which sets our `countdown` variable to the string 'Go!' and - most importantly - sets the `active` variable to `False`. This means, when the while loop checks again `active` does not equal `True`. Our while loop has finished running, and we move outside the loop to the last bit of code that needs to execute outside the loop, which is print(countdown), outputting the final end value of our `countdown` variable.

The output of this program is below.

```
3
2
1
Go!
```

Using break to exit a loop

The above example can be written slightly more efficiently using the break keyword. This keyword allows us to immediately break out of a loop without executing any more code inside it, or requiring us to check the condition. It's kind of like slamming down on the brakes in a car: stop this while loop immediately instead of when this particular loop is complete.

Here's our revised code using the break keyword.

```
countdown = 3

while True:
    if countdown == 0:
        countdown = 'Go!'
        break

    print(countdown)
    countdown -= 1

print(countdown)
```

Notice we don't use a flag in this code: we simply write `while True` as our while loop condition. `True` is always true, so this loop is set to run forever - a potential infinite loop, which we'll talk about in greater detail below. But in this loop, we use the `break` keyword in our `if countdown == 0` conditional. As soon as this if statement is true - as soon as `countdown == 0`, we set the `countdown` variable to 'Go!' and then immediately break out of the loop without executing the other code inside the loop or checking to see if the `while True` condition is still true (which it is). This allows us to escape our loop.

Infinite loops

What happens if we create a loop like the one below?

```
counter = 1

while True:
    print(counter)
    counter += 1
```

This is an infinite loop. Since there's no way to get out of this loop - no condition that can ever be anything except true, and no break to get us out of the loop - this while loop will run... forever! (Or until your computer crashes.) I tried running this on my machine, and before I cancelled it just a few seconds later my printed counter had counted all the way up to 1080308!

Every programmer writes an occasional accidental infinite loop once in a while. It's always good to know how to cancel your program's execution manually in case you find yourself trapped in an infinite loop. If you're executing Python via the command line, like in the editors we provide below, you can use control-C to cancel the execution. If you're using a local code editor to run your Python, you should know what your specific editor or tool uses to cancel code execution.

Manipulating lists with while loops

After you've been programming for a while you'll find that while loops are useful in lots of different ways - it would take us a long time to definitively go through all the many different ways we can use them.

But here are a couple of quick examples of ways we can use while loops to manipulate lists.

```
invited = ['Agent Q', 'Agent M', 'Agent J', 'Agent S', 'Agent M']

while 'Agent M' in invited:
    invited.remove('Agent M')

invited.append('Agent M')
```

Here we have a list of people invited to attend an important meeting, but we see Agent M has been accidentally added twice.

This while loop first checks to see if the string `Agent M` can be found inside the list `invited`. As soon as it finds one, it removes it, then starts the while loop again. This repeats twice - because Agent M is listed twice in this list - and then the third time the while loop runs the check, Agent M is no longer anywhere in the list, so the loop is finished.

Finally, since we do still want Agent M to come to our meeting, we add her back to the list, knowing now she'll only be on there once.

Here's an example of how we can move our agents from one list to another after the meeting has happened, taking them from the invited list and moving them automatically onto the attended list.

```
invited = ['Agent Q', 'Agent M', 'Agent J', 'Agent S']
attended = []

while invited:
    current_agent = invited.pop()
    print(current_agent + ' attended the meeting.')
    attended.append(current_agent)

print('Attended list: ' + str(attended))
```

Here, the `while invited` is checking to see if the `invited` list has anything in it. As long as it has at least 1 item in it, the while loop is true, so it will execute the code inside it. Once we've moved everyone from the invited list to the attended list using the `pop()` and `append()` methods, the `invited` list is empty and so our while loop check is false. This allows us to escape our loop and print out our final attendee list.

Functions

Our tool box is nearly full of the basic tools we need to start building complex programs. Now that we can use conditionals and loops as well as manipulate variables, lists and dictionaries we can start to see how nearly anything is possible.

Functions are another tool in our toolbox, and an important one. Functions allow us to split our code apart into smaller chunks that we can call on at any point in our program. This allows us to avoid repetition and make our code more human-readable.

Defining and calling functions

Let's start with a very simple function that says 'hello' to Agent J.

```
def greet_agent():
    print('Hello Agent J!')

greet_agent()
```

When we want to create a function we use the def keyword followed by what we want to name the function. In this case, we've named our function `greet_agent()`. Now we use indentation to create the body of our function: in this case, we want it to print the string "Hello Agent J!" when the function is called.

Defining the function itself doesn't actually print the text 'Hello Agent J' to the screen. In order to execute the code inside the function, we use a function call. To call a function, we write the name of the function, followed by any necessary information in round brackets. In this case there isn't any needed additional information, so we can just call the function with `greet_agent()`.

The code above will have the below output.

```
Hello Agent J!
```

It's important to remember that a function must be defined before it can be called. If we try to do things out of order and call a function before we define it, we'll get an error.

```
greet_agent()

def greet_agent():
    print('Hello Agent J!')
```

Running the above code will give us the following error:

```
$ python program.py
Traceback (most recent call last):
  File "program.py", line 1, in <module>
    greet_agent()
NameError: name 'greet_agent' is not defined
```

Function parameters

We can modify this function to allow us to greet any of our agents by using a function parameter. This lets us pass information into our function at the time we call it, then have the function use that information within it when it executes.

```
def greet_agent(letter):
    print('Hello Agent ' + letter + '!')

greet_agent('J')
greet_agent('M')
greet_agent('Q')
greet_agent('S')
```

In this example, we define our function again, but this time we give it a parameter, `letter`. Think of parameters like placeholders in our function: we use them in the body of our function where we'd like to be able to control data at the time we call the function.

Here we've used our placeholder parameter `letter` inside our print method, `print('Hello Agent ' + letter + '!')`.

Then, having declared and created our function, we call it 4 times: once each for each of our known agents. Here's how our output looks:

```
Hello Agent J!
Hello Agent M!
Hello Agent Q!
Hello Agent S!
```

When we call a function that uses a parameter, the information we pass at the time we call it is called an argument. So when we call `greet_agent('J')` the `'J'` is the argument. In this case, the argument `'J'` is passed into the function `greet_agent()` and assigned to the parameter `letter`. The terms parameter and argument are often used interchangeably: there are technical definitions, but if you're chatting casually with other programmers don't be surprised if you hear arguments referred to as parameters and parameters referred to as arguments.

We can declare as many parameters as we need when we define our function. For example, here's a more complex function that accepts 3 parameters and prints out a more complex output for our 4 agents.

```
def greet_agent(letter, total_cases, solved_cases):
    print('Hello Agent ' + letter + '!')
    print('\tYou have solved ' + str(solved_cases) + ' cases.')

    percent_solved = solved_cases * 100 / total_cases
    print('\tThat\'s ' + str(percent_solved) + '% of your total cases marked as solved,
great job!\n')

greet_agent('J', 11, 8)
greet_agent('M', 15, 12)
greet_agent('Q', 20, 12)
greet_agent('S', 20, 15)
```

This code gives us the below output.

```
Hello Agent J!
        You have solved 8 cases.
    That's 72% of your total cases marked as solved, great job!

Hello Agent M!
        You have solved 12 cases.
    That's 80% of your total cases marked as solved, great job!

Hello Agent Q!
        You have solved 12 cases.
    That's 60% of your total cases marked as solved, great job!

Hello Agent S!
        You have solved 15 cases.
    That's 75% of your total cases marked as solved, great job!
```

Notice here that when we call our function, we put our arguments in the same order as parameters were declared in our function definition: this is using positional arguments. If we are using positional arguments and don't get our order correct - say if we mix up the order of `total_cases` and `solved_cases` - then we get some pretty strange output.

```
def greet_agent(letter, total_cases, solved_cases):
    print('Hello Agent ' + letter + '!')
    print('\tYou have solved ' + str(solved_cases) + ' cases.')

    percent_solved = solved_cases * 100 / total_cases
    print('\tThat\'s ' + str(percent_solved) + '% of your total cases marked as solved,
great job!\n')

greet_agent('J', 8, 11)
```

This will give us an incorrectly inflated percent solved rate for Agent J in the below output.

```
Hello Agent J!
    You have solved 11 cases.
    That's 137% of your total cases marked as solved, great job!
```

We can also use keyword arguments in our function call, where we directly associate a value to a particular parameter. This lets us mix up our order because we're being clear about which value we're assigning to which parameter.

```
def greet_agent(letter, total_cases, solved_cases):
    print('Hello Agent ' + letter + '!')
        print('\tYou have solved ' + str(solved_cases) + ' cases.')

        percent_solved = solved_cases * 100 / total_cases
    print('\tThat\'s ' + str(percent_solved) + '% of your total cases marked as solved,
great job!\n')

greet_agent(letter='J', solved_cases=8, total_cases=11)
```

This will give us our expected output and the correct percent solved number for Agent J.

```
Hello Agent J!
    You have solved 8 cases.
    That's 72% of your total cases marked as solved, great job!
```

Default parameter values

Sometimes when defining functions, it's useful to set default values for some or all of our parameters. If there is a frequently used value as an argument, it can make sense to set it as a default assumed value to make your function calls simpler to use.

Both Agent S and Agent Q have 20 total cases: it turns out that's the largest number of cases any agent can get assigned in a month. Knowing that, and assuming enough agents regularly reach the 20 cases to make it worthwhile, we might want to modify our function to include a default value for the `total_cases` parameter.

```
def greet_agent(letter, solved_cases, total_cases=20):
    print('Hello Agent ' + letter + '!')
    print('\tYou have solved ' + str(solved_cases) + ' cases.')

    percent_solved = solved_cases * 100 / total_cases
    print('\tThat\'s ' + str(percent_solved) + '% of your total cases marked as solved,
great job!\n')
```

```
greet_agent('S', 15)
```

This code will generate the expected output below.

```
Hello Agent S!
    You have solved 15 cases.
    That's 75% of your total cases marked as solved, great job!
```

Notice in the function definition for `greet_agent()` in the above example that we swapped the `solved_cases` and `total_cases` parameters around? That's because if we have a mixture of parameters with default values and some without, you must have the parameters with default values at the end of the list.

Why? Because then we can use positional arguments when we call our function. If we didn't organize our parameters this way, then the computer wouldn't know when to use the default values and when not to.

Single Responsibility Principle

The most useful thing about functions is you can call them anywhere, even inside other functions. This allows us to break our code apart into smaller, more readable and reusable chunks.

A good rule of thumb for functions is they should only do 1 thing. This makes them much more reusable, and - when we get to testing our functions - much easier to test. This is called the Single Responsibility Principle

Let's revise our code to use a series of smaller functions called by one primary function. When we rewrite or revise our code to do the exact same thing in a different (and hopefully better) way, we call it refactoring our code. So let's refactor our previous code to break it up into smaller functions, each with a single responsibility.

```
def greet_agent(letter):
        print('Hello Agent ' + letter + '!')

def solved_case_rate(total_cases, solved_cases):
    percent_solved = solved_cases * 100 / total_cases
        print('\tYou have solved ' + str(solved_cases) + ' cases.')
    print('\tThat\'s ' + str(percent_solved) + '% of your total cases marked as solved,
great job!\n')

def agent_status(letter, total_cases, solved_cases):
    greet_agent(letter)
        solved_case_rate(total_cases, solved_cases)

agent_status(letter='J', total_cases=11, solved_cases=8)
agent_status(letter='M', total_cases=15, solved_cases=12)
```

```
agent_status(letter='Q', total_cases=20, solved_cases=12)
agent_status(letter='S', total_cases=20, solved_cases=15)
```

Here we have 3 functions. The first is `greet_agent()`, a function which takes a single parameter, `letter`, and prints out a nice greeting for our agent.

The second function is called `solved_case_rate()` and takes 2 parameters: `total_cases` and `solved_cases`. This function's responsibility is to print out some text about our agent's current case load status.

Finally we have agent_status which takes 3 parameters: `letter, total cases` and `solved cases`. This function does nothing but call our other 2 functions in the correct order.

So when we call `agent_status()` for each individual agent, for each call 2 more function calls are generated per agent, allow us to print out the information as before.

Return values

So far, we've had our functions use the print() method to create output directly, but we may not always want this. Fortunately, instead of generating output, we can instead use return values to send information from inside our function to the line where the function was called.

Here's a very simple example of how a return value works.

```
def addition(x, y):
    total = x + y
        return total

calculation = addition(31, 11)
print(calculation)
```

In the above example, we've created a function called `addition()` that takes 2 parameters. It adds these parameters together and saves that value in a variable called `total`, and then it returns the value assigned to `total`.

When we call the function later, passing in 31 and 11, our function dutifully does the arithmetic and returns the total - in this case 42 - which we store in the variable `calculation`. Finally, we print out the value stored in `calculation`.

We can streamline this code even more by removing some of the work of storing information in the variables `total` and `calculation`, which we don't really need here. If we want to be really concise, we can use the below code to do exactly the same thing.

```
def addition(x, y):
    return x + y

print(addition(31, 11))
```

When using a return value, it's important to know that it will immediately end the function execution, similar to how the `break` keyword works in a loop. For example, let's look at a variation of our code above.

```
def addition(x, y):
    return x + y
    print('You will never know I exist!')

print(addition(31, 11))
```

Here we've added a `print()` method to our function `addition()` but when we run it, it doesn't print out the line 'You will never know I exist!'. That's because we have a return value above it. The function finishes running before it gets to `print('You will never know I exist!')`, so this line will never execute.

This can be helpful for us if a function includes an if statement or a loop, as it allows us to leave the function execution early if certain conditions are met.

Let's create a program that determines whether or not Agent J has time to wait in line at the cafe for a coffee, or if he has to skip his morning coffee in order to get to his first meeting of the day. The answer depends how many people are in front of him, but also who is in front of him. Agent Q currently owes Agent J a favour, and she'll let him swap places with her in the coffee line if he's in a hurry just this one time.

```
# Determine if there is enough time to wait for coffee, given
# the current people in line in front of Agent J.
def can_wait_for_coffee(minutes_available, people_in_line):
    estimated_wait_time = len(people_in_line) * 2

    if estimated_wait_time < minutes_available:
        return 'Yes, plenty of time.'
    else:
        return can_swap_with_agent_q(minutes_available, people_in_line)

# Determine if Agent Q is in the line to swap with and, if so,
# if she's far enough ahead in line to make a swap worthwhile
# given the time available.
def can_swap_with_agent_q(minutes_available, people_in_line):
    if 'Agent Q' not in people_in_line:
        return 'Agent Q isn\'t in line... no coffee for Agent J today.'

    position_agent_q = people_in_line.index('Agent Q')
    wait_time_for_agent_q = (position_agent_q + 1) * 2
```

```
    if wait_time_for_agent_q < minutes_available:
        return 'Agent J can swap places with Agent Q and get his coffee.'
        else:
        return 'Not even Agent Q can save Agent J today... no coffee today.'

# Ask our question
print('Does Agent J have enough time to get coffee this morning?')

# Get our answer
people_in_line = ['Unknown Person A', 'Agent M', 'Agent Q', 'Unknown Person B']
print(can_wait_for_coffee(8, people_in_line))
```

We won't go through this example line by line - you should give it a try with the editor below with different values and orders of values in the `people_in_line` list as well as a different number of minutes Agent J can wait to make sure you understand what this program is doing and how it's doing it.

However, let's have a look at the `can_swap_with_agent_q()` function definition. Note here how we first do a check to see if Agent Q isn't in the line `if 'Agent Q' not in people_in_line`. If she's not found in the `people_in_line` list, then we immediately return out of the function. There's no need to set `position_agent_q` or `wait_time_for_agent_q` variables or run the following checks: she's not in the line, so the rest of the function's logic isn't needed, so we exit the function immediately.

Another thing we should look at in this example is in the `can_wait_for_coffee()` function definition. Here, we have an if statement that checks if the value of `estimated_wait_time` is less than the `minutes_available` value Agent J has available to wait before he's late for his meeting. If this evaluates to `True`, then we return a string. But if it doesn't we return another function! (Technically, we return the return value of another function. And if that function also returns a function, we return the return value of that function... and on and on like a game of pass the potato.)

The ability to have functions return other functions, which can then also return other functions, and on and on depending upon layers of logic gives you a glimpse of what's possible with programming. We take a series of small pieces of logic and slowly build up a complex system, piece by piece.

User Input Prompts

Computer programs are significantly more useful to us if they can take information provided by the user and incorporate that input into the program. We can already start to see where we clearly need user input in some of our previous examples, such as Agent J deciding whether or not he has time to wait for coffee. His user input would be things like the number of minutes he has on that particular day to wait, how badly he wants coffee that day, and how many people are currently in line ahead of him that he would have to wait behind.

So how do we capture this user input for use in our programs?

Before we get started on the how let's talk about how user input can be dangerous.

Being wary of user input

Providing users a way of putting input into a program is a key component of what makes a program useful, but it also represents one of the biggest security risks to our program. As soon as we open our program to user input, we give the user a little bit of control. That's often exactly what we want to do - give the user the ability to control bits of the program is probably what we've designed it to do - but it's very easy to give the user too much control in unintended ways, and this is where security vulnerabilities often hide.

We should always approach user input with a healthy heaping of suspicion. Allow it where your program needs it, but never fully trust it. Assume the user might be a malicious attacker trying to exploit the system, and take precautions with the ways you allow user data to enter and travel around your program.

Why you should always use raw_input() and never input()

In a lot of other courses and books that teach Python (either 2 or 3), you might have encountered the `input()` method, which allows us to incorporate user input into a program. Both Python 2 and Python 3 have this method, but depending on which version of Python you're using it behaves differently and - most importantly - in Python 2 the `input()` method is extremely insecure and should never be used

In Python 2 we should always use the `raw_input()` method. Always.

The `raw_input()` method will interpret everything the user feeds into it as a string, no matter what characters are typed. So even if the user provides a number - like 42 - this will be interpreted as the string '42'. The same goes if the user types 'False': it gets interpreted as the string 'False' rather than the boolean value `False`.

However, if we use the `input()` method, it tries to figure out the intended type of the user input: it wants to interpret 42 as the integer 42 and 'False' as the boolean value `False`. This seems like it should be helpful, but the way it does this behind the scenes is dangerous. If we dig into the particulars of how this method works, we see it uses the very dangerous method `eval()` to figure out what type the input should be.

Why is `eval()` dangerous? Because it will evaluate what is passed to it as code. So a user could hypothetically type a function definition and a function call into the space we've provided for user input, and since that user input gets run through `eval()` it would actually execute that function! As you can see, this is a terrible, terrible idea. This is why we always use `raw_input()` when writing Python 2 code: we never want to send anything a user controls to an `eval()` method.

This is an important thing to remember as a programmer: you should always know how user input methods work behind the scenes so you know exactly what they're doing and how they work, to prevent creating unintentional security vulnerabilities in your code. In general, you should try and identify the known dangerous methods like `eval()` and find out where they might be used in other, seemingly innocent methods.

Python 3 and input()

If you have already encountered Python 3, you may already know that `raw_input()` doesn't exist in Python 3. That's because in Python 3, `raw_input()` was renamed as `input()` and behaves exactly as `raw_input()` does in Python 2. The more dangerous user input method that uses `eval()` was removed entirely from Python 3, and for good reason. However, this has made things a bit more confusing if you're jumping from Python 3 to Python 2 or back again. When in doubt, default to using `raw_input()`. The system will give you an error if you're using Python 3, where this method doesn't exist.

User input with raw_input()

Let's give the `raw_input()` method a try by having our program ask us a simple question.

```
user_name = raw_input('Hello, what is your name? ')
print('Hi ' + user_name + ', nice to meet you!')
```

When we run this code from our terminal, we get the following prompt:

```
$ python program.py
Hello, what is your name?
```

The program pauses here until we give it some user input. We type our name into the terminal after this prompt, and hit the enter key when we're done. This allows the program to continue on to line 2, with our user input in hand.

Here is our final output, from start to finish, including where we typed our name in.

```
$ python program.py
Hello, what is your name? Agent L
Hi Agent L, nice to meet you!
```

We've successfully taken in some user input, and used it to print out a nice greeting for our user with their name in it.

Let's try another example, showing how `raw_input()` always interprets our user input as a string.

```
user_coffee_input = raw_input('How many cups of coffee have you had today? ')

if user_coffee_input > 2:
    print('Wow, that\'s a lot of coffee!')
elif user_coffee_input == 0:
    print('Should we go grab a coffee? I could use one too.')
else:
    print('Sounds like the right amount of coffee to start the day.')
```

What happens when we run this program? Things go ok until after we enter our user input, then we don't get quite what we expect.

```
$ python program.py
How many cups of coffee have you had today? 0
Wow, that's a lot of coffee!
```

But wait, our program should take the 0 and run through the `elif user_coffee_input == 0` branch of our conditional and print "Should we go grab a coffee? I could use one too." but that's not what it does.

The reason why is that our user input `"0"` is being interpreted as a string of "0" rather than an integer `0`. In Python 3 we'd get a TypeError in this case where we're trying to use > to compare a string value and an integer value, but Python 2 uses different rules and thus we get our confusing output.

We can fix this using our tried and tested `int()` function in our program.

```
user_coffee_input = raw_input('How many cups of coffee have you had today? ')

user_coffee_int = int(user_coffee_input)

if user_coffee_int > 2:
    print('Wow, that\'s a lot of coffee!')
elif user_coffee_int == 0:
    print('Should we go grab a coffee? I could use one too.')
else:
    print('Sounds like the right amount of coffee to start the day.')
```

In this revised code we've added a line that transforms our user input into an integer safely, so we can now use it in our program correctly.

The output we get is below.

```
$ python program.py
How many cups of coffee have you had today? 0
Should we go grab a coffee? I could use one too.
```

Except... now what happens when the user types a string like "none" instead of a number? We'd expect them to type a number, but an important lesson to learn when integrating user data into our programs is to expect the unexpected. So let's not assume the user will type a number.

```
$ python program.py
How many cups of coffee have you had today? None
Traceback (most recent call last):
  File "program.py", line 3, in <module>
    userCoffeeInt = int(userCoffeeInput)
ValueError: invalid literal for int() with base 10: 'None'
```

Oops! We thought we had fixed a bug previously, and we sort of did... but by doing so, we introduced a new one. This isn't an unusual occurrence for a programmer: often fixing one

bug reveals another. That's ok: let's keep squashing them.

Let's make sure we can deal with any type of input our user throws at us to make our program more resilient.

The `isdigit()` method can help us out here. This method allows us to check if the string the user provided can be turned into an integer. If it can't then we can provide a helpful error to our user. To do this, we're also going to refactor our code to use a couple of functions to make it easier to read and understand.

```
# Checks if the user's answer can be used by the determineReply()
# function, and if it cannot provides an error.
def get_reply(user_input):
        if user_input.isdigit():
        user_input_int = int(user_input)
                return determine_reply(user_input_int)
    else:
                return 'Sorry, I don\'t understand your answer. I was looking for a number, not
a string.'

# Determines the correct reply
def determine_reply(user_input_int):
    if user_input_int > 2:
                return 'Wow, that\'s a lot of coffee!'
    elif user_input_int == 0:
                return 'Should we go grab a coffee? I could use one too.'
    else:
                return 'Sounds like the right amount of coffee to start the day.'

# Ask for user input
user_coffee_input = raw_input('How many cups of coffee have you had today? ')

# Process the answer to get the right reply, and print that reply
reply = get_reply(user_coffee_input)
print(reply)
```

We've separated our logic into 2 function definitions. The first is called `get_reply()`, which determines if the user's input can be used in the way we want using the `isdigit()` method. If the user input can be transformed into an integer, `get_reply()` does this transform, then it returns another function with this transformed user input passed as an argument. Otherwise, if the user's string can't be turned into an integer, we return an error to be output to the screen.

The second function definition assumes an integer, and uses an if-elif-else conditional to return an appropriate reply string.

Now, after we ask for user input, we call the `get_reply()` function to sort out which of our 4 reply options we should use, and store that returned reply string to the variable `reply`, which we then print to the screen.

User Input via the Command Line

If you've ever used a security tool that's been built in Python, you may already be familiar with the idea that you can pass user input to a Python script at the point you choose to run it from the command line. If you're building a program that you mostly expect people to run from a terminal, it may make sense to capture your user input this way.

In order to do this, we'll first have to import a separate library.

Importing a library

Python comes with a standard library of additional components we can use in our code. They have been split out into smaller groups of functionality that we have to import in order to make our programs more efficient. If all the possible bits of functionality were always included, our programs would take longer to run. So the 'default' set of Python functionality includes only the core tools we need basically all the time, while other useful pieces of functionality are split up into useful libraries or modules that we can import and use at any time.

In order to allow user input to be passed via the command line at the program's run time, we need one of the Python Runtime Services libraries called "System-specific parameters and functions". This is the `sys` library, and to import and use it in our program we'll have to start our program off with `import sys`. You should always import the libraries or modules you intend to use in your program at the start: it's good practice to declare your dependencies upfront.

How command line arguments are passed to the program

Let's start by just passing in some user input and seeing what kind of format we get out the other side. When using a new function you've never used before, a good way to start understanding the ins and outs of how it works is by outputting what you get back to the screen. In order to access the arguments we pass in when we run our program via the command line, we use sys.argv.

```
import sys

arguments = sys.argv
print(arguments)
```

Let's start by running our program with some test arguments and see what we get as output when we print them. Then, we know what kind of data structure we're dealing with when we use `sys.argv`.

```
$ python program.py test1 test2 test3
['program.py', 'test1', 'test2', 'test3']
```

A few things we notice right away about our output:

fi. We've seen this structure before: those [ ] brackets indicate this is a list. The data we get is in a list form, so we can interact with it as a list.
fi. The first item in our list is the name of the python program we ran, in this case `program.py`.
fi. The arguments we pass through are added into the list, in the order that we run them.

Ok, that's some solid information gathering. Let's try a few more things and see what happens.

```
$ python program.py 1 false this='that' x=2
['program.py', '1', 'false', 'this=that', 'x=2']
```

We've given this another test run to see if everything we throw at it will be processed as a list, and to see if it will take all different data types and turn them into strings. As we can see in this example, that's exactly what it does: everything we pass in as an argument will become available to us in our program as a list of strings, with our program filename being the very first string in the list.

Using command line arguments in our program

Let us have fun with a contrived but useful scenario. We want to write a program that picks code names for certain secret projects a team is working on. We'll create a list of words suitable to be code names, and using a command line argument we'll specify the number of words we want to use from the list to generate our random code name.

```
import random

# Pick a random word from a provided list
def pick_random_word(list):
    return random.choice(list)

# Get a code name made up of the number of words specified
def get_code_name(list, num_words):
    code_name = ''

    for x in range(1,num_words+1):
            word = pick_random_word(list)
        code_name += word + ' '
```

```
# List of words to use
word_list = ['Aurora', 'Avalanche', 'Blizzard', 'Cyclone', 'Eagle', 'Edison', 'Frost',
'Hawk', 'Hexagon', 'Hornet', 'Medusa', 'Neptune', 'Orion', 'Osprey', 'Plato', 'Portal',
'Raven', 'Sand', 'Shadow', 'Storm', 'Sunset', 'Thunder', 'Vector', 'Vista', 'Vortex',
'Volcano']

# Create a code name and print it to the screen
code_name = get_code_name(word_list, 2)
print(code_name)
```

This simple little program allows us to generate code names using words picked at random from a list we provide, and provides us a way to specify how many words we want to use for our code name.

Now, let's add in the ability to specify the number of words from the command line, where we run the script. Don't forget the lesson we learned from the section on user input prompts, where we need to make sure we check the input we receive is usable by our program: in this case, we need an integer. We've already included a default parameter in our `get_code_name()` definition, so if we don't get an input we can use, we'll use our default number of words, which is 1.

```
import random
import sys

# Pick a random word from a provided list
def pick_random_word(list):
    return random.choice(list)

# Get a code name made up of the number of words specified
def get_code_name(list, num_words):
    if num_words.isdigit() == False:
            return 'Error: incorrect argument provided. You must provide an integer.'

        num_words = int(num_words)
    code_name = ''

    for x in range(1,num_words+1):
            word = pick_random_word(list)
        code_name += word + ' '

    return code_name.rstrip()

# List of words to use
word_list = ['Aurora', 'Avalanche', 'Blizzard', 'Cyclone', 'Eagle', 'Edison', 'Frost',
'Hawk', 'Hexagon', 'Hornet', 'Medusa', 'Neptune', 'Orion', 'Osprey', 'Plato', 'Portal',
'Raven', 'Sand', 'Shadow', 'Storm', 'Sunset', 'Thunder', 'Vector', 'Vista', 'Vortex',
'Volcano']

# Retrieve the command line argument
words_to_pick = sys.argv[1]

# Create a code name and print it to the screen
code_name = get_code_name(word_list, words_to_pick)
print(code_name)
```

We've only added a couple of things to our code here to get it working with command line arguments.

In the `get_code_name()` function definition, we've added a check to make sure our user input is something we can turn into an integer using our old friend the `isdigit()` method again. If it fails this check, we immediately return with an error that gets printed to the screen, and the rest of this function never runs. (Remember: when we return in a function it exits the function immediately.) If this conditional passes, the first thing we do next is change this string into an integer so we can use it in our loop as we did before.

The only other line we've added is just below where we create our list of words, which grabs the command line argument so we can pass it through to the `get_code_name()` function.

Now we can run our program via the command line and pass in our user data, which is the number of random words we want our code name to include. Here is an example from running the program three times, once requesting a code name with 2 words, once with 3 words, and once using a string which causes our error to appear.

```
$ python program.py 2
Sunset Volcano
$ python program.py 3
Shadow Blizzard Orion
$ python program.py test
Error: incorrect argument provided. You must provide an integer.
```

One last thing we need to take into account: what happens if the person running our program doesn't know they have to add an argument?

```
$ python program.py
Traceback (most recent call last):
  File "program.py", line 29, in <module>
  wordsToPick = sys.argv[1]
IndexError: list index out of range
```

Hmm... that's no good. Let's adjust our code one more time to provide a helpful error for this case as well.

```
import random
import sys

# Pick a random word from a provided list
def pick_random_word(list):
    return random.choice(list)

# Get a code name made up of the number of words specified
```

```
if num_words.isdigit() == False:
        return 'Error: incorrect argument provided. You must provide an integer.'

    num_words = int(num_words)
        code_name = ''

        for x in range(1,num_words+1):
        word = pick_random_word(list)
                code_name += word + ' '

        return code_name.rstrip()

# List of words to use
word_list = ['Aurora', 'Avalanche', 'Blizzard', 'Cyclone', 'Eagle', 'Edison', 'Frost',
'Hawk', 'Hexagon', 'Hornet', 'Medusa', 'Neptune', 'Orion', 'Osprey', 'Plato', 'Portal',
'Raven', 'Sand', 'Shadow', 'Storm', 'Sunset', 'Thunder', 'Vector', 'Vista', 'Vortex',
'Volcano']

if len(sys.argv) > 1:
    # Retrieve the command line argument
        words_to_pick = sys.argv[1]

        # Create a code name and print it to the screen
    code_name = get_code_name(word_list, sys.argv[1])
        print(code_name)
else:
        print('Error: You must provide the number of words as an argument.')
```

And let's test our different cases one more time on the command line to make sure we get what we expect in each situation.

```
$ python program.py 1
Hexagon
$ python program.py 3
Eagle Sand Cyclone
$ python program.py test
Error: incorrect argument provided. You must provide an integer.
$ python program.py
Error: You must provide the number of words as an argument.
```

Nice! We have succesfully handled some user input, and validated it. There are lots of examples of this type of code online, take some time to practice and explore others code.

CLI User Input Lab

Take some time to practice and enhance this program per the instructions provided in the lab steps. Don't be scared to experiment and try your own variations of this program too.

Classes and Objects

There is a style of programming called object-oriented that uses a construct called a class to create a model of a real-world thing, and then uses that as a kind of template to create objects from. When we create a class, we define the general way an object is constructed and how it behaves.

For example, if we think of a real world object like a car, there are some things all cars have in common. They all have 4 wheels, doors, an engine, and a steering wheel. There are also some ways in which all cars behave similarly: they can all move forwards and backwards, they can turn left and right, and they have horns that make a noise in an emergency. If we were programming a game that needed to have lots of different cars in it, an efficient way for us to do this would be to create a class to model all the necessary attributes of any given car, and any actions all cars should be able to take.

In this section we aren't going to talk too much about the style of object-oriented programming specifically, but for now let's make sure we know how to create a class, and how to create objects from them.

Creating an Agent class

Let's create a very simple class that models our agents. In previous sections and modules we've learned a lot about our team of agents, so let's use some of that information to create our first class.

```
class Agent():
  name = ''
    hot_drink = ''

    def speak(self, speech):
    print(self.name + ' says: "' + speech + '"')

  def drink(self):
        print(self.name + ' drinks a cup of ' + self.hot_drink + '.')
```

Let's go through our class line by line.

We first start by declaring our class and giving it a name: `class Agent():`. Everything that belongs to our class will be indented.

Next, we create 2 variables which our class will need to work properly, but we don't assign them any values: right now, both `name` and `hot_drink` are both set to be empty strings.

After that we create 2 functions, `speak()` and `drink()`. In a class, functions are called methods. The only difference between a function and a method is that a method is part of

a class, which means we need to call it in a slightly different way, which we'll see in the next example.

As it stands now, our `Agent()` class captures a model of a person who:

- Has a name
- Has a hot drink preference
- Can speak
- Can drink their hot beverage

Now that we've created our class, how do we use it?

```
class Agent():
  name = ''
  hot_drink = ''

  def speak(self, speech):
    print(self.name + ' says: "' + speech + '"')

  def drink(self):
    print(self.name + ' drinks a cup of ' + self.hot_drink + '.')

agent_q = Agent()
agent_q.name = 'Agent Q'
agent_q.speak("Hi, I'm Agent Q!")
```

After we've defined our class, once we want to start making use of it, the first thing we do is instantiate our class with `agent_q = Agent()`. Here, we create an object using our class as a model, and call that object `agent_q`. This object is a little self-contained "thing" that has all the attributes and methods associated to it that we defined in our class. As programmers, we would refer to this object as an instance of a class.

In the next line, we take our new object `agent_q` and set a value for the variable `name` that we left as an empty string when we created the class: `agent_q.name = 'Agent Q'`. Notice our notation here: we use dot notation to set the value of a class variable. Variables that are accessed on objects like this are called attributes.

Finally, we use one of the methods we created in our class - again, we use dot notation to access it because we are interacting with our object: `agent_q.speak("Hi, I'm Agent Q!")`.

Here is the output that's generated when we execute this code:

```
Agent Q says: "Hi, I'm Agent Q!"
```

Understanding self

Notice how we refer to the attributes we created in our class? When we want to use the `name` attribute within one of our methods, we use `self.name`. Why do we do this? The

`self` argument in the class refers to itself - the object the class creates. By calling `self.name` and `self.hot_drink` within our methods, we're telling the methods to look outside the method itself, but stay within the class when trying to find the variables `name` and `hot_drink`. It's a way of determining *scope* - it tells Python to stay within the scope of this class.

This is also why `self` is the first parameter of every method we define inside our class: within a class, every method needs to understand the entire class it's part of. In Python, we must declare `self` as the first parameter of the methods in our class, otherwise the methods are unaware of the other methods and attributes that are also available within the class. So `def speak(self, speech)` has 2 parameters: `self`, by default because all class methods have `self`, and a second one, `speech`, which we use within just this method. Notice that when we refer to the `speech` parameter within the `speak()` method that we don't use `self`? That's because the scope of this variable is inside the method. By not using `self` here, Python knows when running this code that it should only look for a variable or parameter called `speech` within this method.

Any time you need to reference something inside the class but outside the method, you'll need `self`.

When we call our methods `speak()` later on our `agent_q` object, notice that we skip right over the first `self` argument and only add the second, in this case the text we want to pass as our `speech` argument? Python is a bit picky: it wants us to always add `self` as a first parameter when we declare the method, but doesn't want us to add it as an argument: it does this for us automatically every time we call the method.

Creating multiple instances

The real power of classes and objects is the ability to create more than one object at a time. Each object is like a copy of the original empty class, and entirely independent of any other objects that we might have previously created using this class as our model.

```
class Agent():
  name = ''
  hot_drink = ''

  def speak(self, speech):
    print(self.name + ' says: "' + speech + '"')

  def drink(self):
    print(self.name + ' drinks a cup of ' + self.hot_drink + '.')

agent_q = Agent()
agent_q.name = 'Agent Q'
agent_q.hot_drink = 'decaf coffee'

agent_m = Agent()
agent_m.name = 'Agent M'
agent_m.hot_drink = 'Earl Grey tea'
```

```
agent_q.speak("Hi, I'm Agent Q!")
    agent_m.speak("Hi, nice to meet you. I'm Agent M.")
        agent_q.drink()
    agent_m.drink()
```

In this example, we've created 2 different objects using our class, 1 called `agent_q` and 1 called `agent_m`. We've given each one an appropriate `name` attribute.

Then we get them to speak to each other using the `speak()` method. Notice that each object still knows its own unique `name` attribute when the `speak()` method is run. Setting `agent_m.name = 'Agent M'` doesn't affect how we've set `agent_q.name`. They're modifying different objects.

```
Agent Q says: "Hi, I'm Agent Q!"
Agent M says: "Hi, nice to meet you. I'm Agent M."
Agent Q drinks a cup of decaf coffee.
Agent M drinks a cup of Earl Grey tea.
```

Object constructors

Each time we created a new instance of our class `Agent()` above, we had to first assign its attributes. This is a little bit tedious, so let's update our class and add an object constructor. This will let us both instantiate our class and assign our key attributes all on 1 line.

```
class Agent():
    name = ''
    hot_drink = ''

    def __init__(self, name, hot_drink):
        self.name = name
        self.hot_drink = hot_drink

    def speak(self, speech):
        print(self.name + ' says: "' + speech + '"')

    def drink(self):
        print(self.name + ' drinks a cup of ' + self.hot_drink + '.')

agent_q = Agent('Agent Q', 'decaf coffee')
agent_m = Agent('Agent M', 'Earl Grey tea')

agent_q.speak("Hi, I'm Agent Q!")
agent_m.speak("Hi, nice to meet you. I'm Agent M.")
agent_q.drink()
agent_m.drink()
```

We've added a new method to our class, the constructor method. This method has a special name, and if we use it, it should be the very first method we define in our class. As

its name "constructor" implies, it helps us "construct" the class more quickly.

The constructor method uses a strange looking name: `__init__()`. That's 2 underscores, followed by 'init', followed by 2 more underscores. This is to make sure this special method never confiicts with any other method you might have in your class.

We give our `__init__()` method 3 parameters here. The first is `self`... because `self` is always the first parameter of a method, and this one follows that rule. Then we also give it `name` and `hot_drink`.

Inside this method, we use these parameters to set the attributes of the object: `self.name = name` and `self.hot_drink = hot_drink`. Here's how we see some of our scope at work: `self.name` refers to the variable outside our constructor method but inside the class, while `name` in this method refers to the parameter. This is why `self` is important: it differentiates from the `name` inside the method and the `name` inside the class.

Now, when we create an object, we also pass the values we want as attributes into the object as parameters. The first thing that happens when we instantiate a class is that it is constructed, and so Python will run the `__init__()` method automatically every time we create a new object, using the arguments we specify at this time.

The output of this new revised code is exactly the same as the output of our previous version, but we've saved some lines and created much nicer looking code.

Modifying object attribute values

Let's add a few more attributes and methods to our class.

```
class Agent():
    name = ''
    hot_drink = ''
    cases_total = 0
    cases_solved = 0

    def __init__(self, name, hot_drink, cases_total, cases_solved):
        self.name = name
        self.hot_drink = hot_drink
        self.cases_total = cases_total
        self.cases_solved = cases_solved

    def speak(self, speech):
        print(self.name + ' says: "' + speech + '"')

    def drink(self):
        print(self.name + ' drinks a cup of ' + self.hot_drink + '.')

    def get_total_cases(self):
        print(self.name + ' has a total of ' + str(self.cases_total) + ' cases.')

    def add_new_case(self, number):
```

```
    if (number > 1):
            print(self.name + ' has been given ' + str(number) + ' new cases.')
                else:
            print(self.name + ' has been given ' + str(number) + ' new case.')

        def get_solved_cases(self):
                print(self.name + ' has solved ' + str(self.cases_solved) + ' cases.')

             def solve_case(self, number):
          self.cases_solved += number
                if (number > 1):
            print(self.name + ' has solved ' + str(number) + ' cases, wow!')
                else:
            print(self.name + ' has solved a case, great job!')

    agent_q = Agent('Agent Q', 'decaf coffee', 20, 12)

    agent_q.speak("Hi, I'm Agent Q!")

    agent_q.get_total_cases()
       agent_q.add_new_case(2)
    agent_q.get_total_cases()

    agent_q.get_solved_cases()
       agent_q.solve_case(1)
    agent_q.get_solved_cases()
```

In this example, we've added 2 new attributes`cases_total` and `cases_solved`. We've also updated our constructor to automatically set those attribute values when we construct the object.

We've also added a few new methods to work with these new attributes. We can now request some stats about Agent Q's total and solved cases using`get_total_cases()` and `get_solved_cases()`.

We've also created 2 more methods that let us update attributes of our object. Now when Agent Q gets assigned a new case, or solves a case, we can simply call`add_new_case()` or `solve_case()` on our`agent_q` object, and these attributes will get updated.

Class and Objects Lab

Classes and Objects are used on object oriented programming to increase the efficiency and speed of creating programs. Try you hand and using classes and objects.

Exceptions

As we've been working through learning Python, you've probably noticed a certain pattern in the way we get errors shown to us.

Let's cause an error on purpose and take a closer look at the format of our errors.

```
print(5/0)
```

This code will create an error, because we can't divide 5 by 0: mathematically, it doesn't make sense to divide a number by 0. That doesn't stop us from writing in code though, so what happens when we try to run this program?

```
$ python program.py
Traceback (most recent call last):
  File "program.py", line 1, in <module>
    print(5/0)
ZeroDivisionError: integer division or modulo by zero
```

This is a traceback error. These are helpful to us, because they give us additional information about what went wrong in our code. Looking at this error, here's what we know:

- The error happened in the "program.py" file, on line 1.
- It happened somewhere around the `print(5/0)` code execution.
- A particular exception was thrown, `ZeroDivisionError`, and a more friendly message follows this to explain what went wrong.

An exception is a special kind of object that Python uses as part of its error management system. Whenever an error happens, Python creates an exception object and, by default, halts the execution of the program as soon as the exception is generated. This is great when we're building code... but not so great when code is in production. We don't really want our users seeing our exceptions. Also, sometimes we don't necessarily want our code to just stop immediately: in some situations it's better to log the error, then allow the program to keep executing. Or sometimes the detection of an exception can be built into the program itself, using an exception as a decision point to move the user down a different logic path.

We can use a try-catch block to handle exceptions that we suspect might crop up in our code.

```
try:
    print(5/0)
```

```
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Now, instead of seeing the exception code and traceback, when we run this the error will still occur, but we'll see our custom message printed out.

```
$ python program.py
You can't divide by zero!
```

Let's try something more complex to show how useful catching exceptions can be, by building a program that takes 2 numbers from user input and divides them.

```
print('Please give me 2 numbers and I will divide them.')
print('Enter "q" to quit.\n')

while True:
    first_number = raw_input('First Number: ')
    if first_number == 'q':
        break

    second_number = raw_input('Second Number: ')
    if second_number == 'q':
        break

    answer = int(first_number) / int(second_number)
    print('Your answer is: ' + str(answer))
    print('Give me another!\n')

print('Ok, bye!')
```

This code accepts 2 numbers from the user, and then divides them and returns the answer. After it is successful, it will go back and run again, prompting the user to start over, until the user types "q" to quit.

So now, if we enter "5" as our first number and "0" as our second number, we once again get our exception and the program crashes. Let's add a try-catch block so our program can keep running in this situation.

```
print('Please give me 2 numbers and I will divide them.')
print('Enter "q" to quit.\n')

while True:
    first_number = input('First Number: ')
    if first_number == 'q':
        break

    second_number = input('Second Number: ')
    if second_number == 'q':
        break
```

```
try:
        answer = int(first_number) / int(second_number)
            except ZeroDivisionError:
        print('You can\'t divide by zero! Let\'s start again.')
            else:
        print('Your answer is: ' + str(answer))
                    print('Give me another!\n')

    print('Ok, bye!')
```

Here we've added a try-catch block. First we try to do our division. If it doesn't work and there's a ZeroDivisionError thrown, we catch that exception and print a nicer error message to the user, then invite them to start again. If we see this message, the while loop will then pick up from the beginning for a fresh run.

If the division is successful, then the user moves down to the `else` block, where their answer gets printed and we see the 'Give me another!' message, before the while loop restarts.

Here's how it looks to a user now if they run this program and try to divide 5 by 0.

```
$python program.py
Please give me 2 numbers and I will divide them.
Enter "q" to quit.

First Number: 5
Second Number: 0
You can't divide by zero! Let's start again.
First Number:
```

That's a lot nicer!

Exceptions Lab

Let's take some time to experiment with exceptions. Try the lab, but try creating some of your own exceptions too! This is a great general chance to practice programming.

Programming 3

Contents

Now we can build much more interesting programmes. We will use sockets to connect to the network, write files and allow multiple tasks to occur at once.

In this module, we will be covering:

- Reading and writing files
- Creating and using TCP and UDP sockets
- The concept of threads and how to make a program multi-threaded
- How to create a port scanner

Reading and Writing Files

An extremely useful feature of Python in particular is how easily it lets us work with other types of files, in particular files that contain text in various formats. Python is often the programming language of choice for scientists and researchers, in part because of how easy it is to work with other files.

Reading from a file

Here's a code sample where we print out text stored in a separate file, called file.txt. This file isn't a Python file (note the extension .txt instead of .py) but Python still lets us open up the file and manipulate the contents.

```
with open('file.txt') as file:
    content = file.read()
print(content)
```

This 3 lines of code shows how easy it is to read the content of a file. We create a with block: with open('file.txt') as file: which asks Python to open the file in the same directory as our program, which is called "file.txt", and create an object for that file which we have called - imaginatively - "file". We didn't have to call it that, we could have called it "unicorn" if we wanted, but since we like to use sensible names when programming "file" seemed appropriate.

Notice the indention for the next 2 lines? Because we have opened the file with a with statement, anything we want to do with the file has to be part of the indented block. As soon as we go outside our indention, we lose the file: it will get closed automatically once the indented block is finished executing.

Next, we use the read() method on the object file we created, which will go into the file and get all the content, which we assign to a variable called content.

Then, we print the value of content to the screen, which will be a copy of the content of the text file.

Here's what the output looks like.

```
$ python program.py
Hello, I'm some text inside a file.
```

Files as user input

In a previous module, we talked about 2 different ways to get user input in Python: using the input() function, and by passing command line arguments and using sys.argv.

As you can see, files are another way of incorporating user input into our program. As a result, we need to treat reading files with the same suspicion as we treat all forms of user input. As soon as we let it enter our program, we must be very careful to keep it well-controlled and assume the person who created the file might be an attacker. Always be cautious when allowing user input of any kind into your program - and that includes reading files.

Writing files

Python also lets us create new files and write data into them. If we wanted to create a file called "file.txt" that has the text "Hello, I'm some text inside a file", how would we do it? It's very simple, and follows the same pattern we used to read a file, except we use the write() method instead of the read() method, and need to pass one more argument to our open() function.

```
with open('newFile.txt', 'w') as file:
    file.write("Hello, I'm some text inside a file.")
```

Here, we've passed two arguments to open() function. The first, as before when we read the file, is a filename. Except this time, it's the name of the new file we want to write to. We also pass a w as a second argument. This tells Python we want to "write" to the file. More importantly, it tells Python we would like to overwrite the content of the file if it already exists. So if that "newFile.text" file already exists and it has important information in it... oops! It will be gone if we run this program, and overwritten with the text string we specified.

So what if we don't want to overwrite the file contents, but rather just add our text to the bottom? We can indicate that using a as our second argument instead of w, which stands for "append".

```
with open('file.txt') as file:
    content = file.read()
    print(content)

print('\n---\n')

with open('file.txt', 'a') as file:
    file.write("\nHello, I'm some text inside a file.")

with open('file.txt') as file:
    content = file.read()
    print(content)
```

Here, we've read the content of a file and printed it out, then printed out a divider so we can see the content of the file before and after we append to it.

Next, we append some content to the file, using a instead of w so we don't overwrite the content that's already in the file.

Finally, we read the file after we've added our text to it, and print this content to the screen.

```
$ python program.py
Don't overwrite me, I'm important!

---

Don't overwrite me, I'm important!
Hello, I'm some text inside a file.
```

There are a few other options when we're reading and writing files besides a and w. Here's the full list:

```
'r': Open the file for reading only.
'r+': Open the file for reading and writing. Any text written will overwrite the
contents of the file, starting from the beginning of the file.
'w': Create the file if it doesn't exist. If it exists overwrite it and open for
writing.
'w+': Create the file if it doesn't exist. If it exists overwrite it and open for
reading and writing.
'a': Open the file for writing only (create it if it doesn't already exist). Anything
written will be appended to the end of the file.
'a+': Open the file for reading and writing (create it if it doesn't already exist).
Anything written will be appended to the end of the file.
```

Sockets in Python

A socket allows us to make and receive network connections, which is a very useful thing for cyber security practitioners to be able to do. In particular, it's great to be able to quickly create a little socket that sends information or listens for specific information we can send to it.

TCP Client

Let's create a simple TCP connection using Python and the socket library.

```
import socket

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(('127.0.0.1', 1337))
client_socket.send(b"Do you want to play a game?\n")
received = client_socket.recv(1024)
print(received)
client_socket.close()
```

First, we import the socket library, which is part of Python's standard library.

Next, we create a socket object, which we've called client_socket here. When we construct the socket, we pass 2 arguments: socket.AF_INET and socket.SOCK_STREAM.

The argument socket.AF_INET means this socket is going to use IPv4 and not IPv6 or, say, Bluetooth. For IPv6 we would use socket.AF_INET6. If you're interested in the other types, you can have a look through the socket library documentation to learn more. For our uses, we typically just need an IPv4 socket, so we'll stick with that one.

The second argument we use is socket.SOCK_STREAM, which is how we tell the socket to use the TCP protocol and not the UDP protocol. If you wanted to make a UDP connection, you could use socket.SOCK_DGRAM instead.

After we've created the kind of socket object we want to use, we initiate a connection with our socket using the connect() method. Here we provide a tuple - remember tuples are similar to lists, but are immutable (notice the double round brackets) - which contains the IP address and a port number to create our connection.

Next, now that we have a connection, we use the send() method to send some data over the connection. In this case, we send the byte object "Do you want to play a game?" as well as the newline character \n. We do this because the send() method doesn't automatically add this newline character.

A byte object is just a binary representation of the string, this is what the b before the quotations comes in. It means we want to convert that string into a byte object.

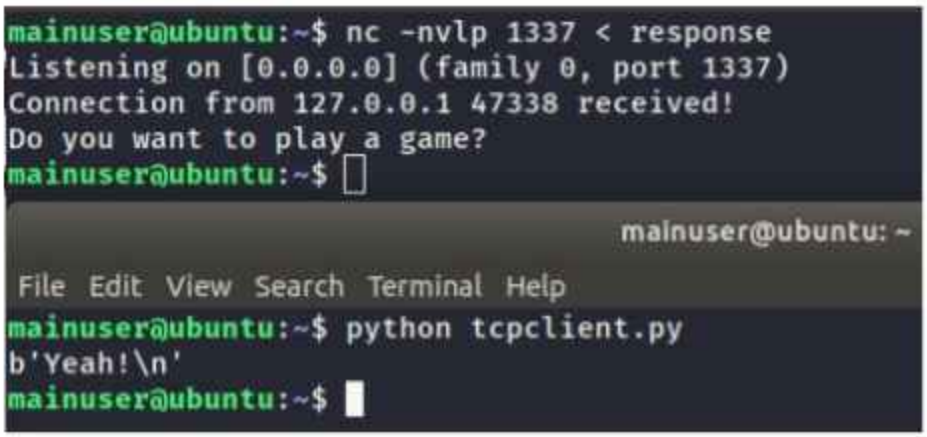
In order to receive information back from the connection, we use the recv() method, storing the return value of this in the variable received. Notice we use the argument 1024 in our recv() method? This is the maximum number of bytes we'll allow at once.

After receiving any incoming response data and storing it in the received variable, we then print this content out so we can see what it is.

Notice the response we receive is also a byte object. You'll be able to see the response would be for example:b"Yeah!"

Finished with our socket, we close the connection using the close() method on our socket object. This initiates the TCP teardown process. If we don't close it like we have done here, the socket will eventually close on its own due to a timeout, but this can take a long time. It's more efficient to remember to close the socket.

Let's look at one in action.



Here, we've opened two terminal windows. In one, we've used a Linux tool called 'netcat' to create a server listening on TCP port 1337: nc -l 1337. In the other window, we've run our python script above.

When the Python script runs, it makes the connection and sends "Do you want to play a game?", which we pick up with the netcat server we created in the top terminal. After seeing the message we were sent, in the netcat window we type "Yeah!", and this was sent back to the Python script, which was received and subsequently printed before the socket was closed.

TCP Server

Now that we know how to initiate a connection with TCP, let's look at receiving connections, which is essentially what a server does.

```
import socket
```

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(("0.0.0.0", 1337))
server_socket.listen(10)

while True:
    conn, addr = server_socket.accept()
        conn.send(b"Do you want to play a game?\n")
    received = conn.recv(1024)
        print(received)

server_socket.close()
```

Once again we need the socket library, and we also need to create a socket object: here we've called our object server_socket. This time, we use the bind() method instead of the connect() method, which allows our program to take ownership of the IP address and port number in the tuple we pass in as an argument, if it's not being used by any other program. Here, we use 0.0.0.0 as the IP address to listen on, which here is asking the program to listen on every IP address assigned to the computer it's running on.

Next, we use the listen() method, which does exactly what you expect: it allows the server to listen on the port it has bound to.

The next part is interesting: here we've used an infinite loop, on purpose. Here, it means that, once we have sent and received information, it will start over. The server will be available for the next connection and will never quit, unless we manually quit the server using the ctrl + c keys on our keyboard.

Within our infinite loop, we use the accept() method to establish a connection with a client. When we accept a new connection successfully, send the string "Do you want to play a game?\n" and then wait for a reply. Once again we see the recv() method being used to receive a response and store that response in the variable received which we then print to the screen.

After we've received that data and printed it out, the loop finishes and re-starts, allowing us to wait for the next connection request to come in.

Let's take a look at how this works in practice.

```
mainuser@ubuntu:~$ nc 127.0.0.1 1337
Do you want to play a game?
Nope!
```

```
                                    mainuser@ubuntu: ~
 File  Edit  View  Search  Terminal  Help
mainuser@ubuntu:~$ python tcpserver.py
b'Nope!\n'
```

Here again we have two terminal windows. This time, the bottom window is our TCP server, which we ran first. Then we used 'netcat' once again - this time to make a connection to our server instead of listening for a connection, notice our command is slightly different here: nc 127.0.0.1 1337.

Once the connection is made, the server immediately sends the query, "Do you want to play a game?" and waits for our response. We type 'Nope!' and this is sent back to the server, which prints it out.

UDP Client

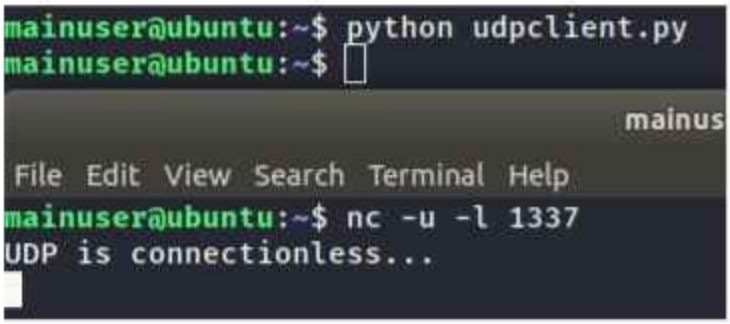
Creating a UDP socket is similar to the TCP socket we created in the previous section.

```
import socket

client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client_socket.sendto(b"UDP is connectionless...\n", ("127.0.0.1", 1337))
```

Once again, we need the socket library, and the socket object needs to be created, here called client_socket. This time, we are using socket.SOCK_DGRAM as our second argument, which we mentioned previously is for UDP.

Notice there is no connect() method: this is because UDP doesn't have connections. With this protocol, we just send data and hope it gets to the other side. This is also why we use a different method for sending, sendto(), which forces us to send the tuple containing the IP address and port number along with every message. Because there is no connection to re-use, we have to specify the destination every time.

Here's how it looks in practice.



In the terminal window above, we've used netcat to set up a server as before, but with the addition of -u parameter in the command: nc -u -l 1337 because we want a UDP server. In the bottom terminal window, we've run our example script, and we see the server gets the data we sent.

UDP Server

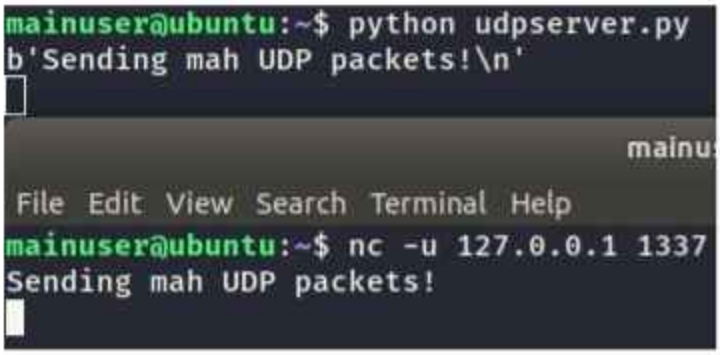
Setting up a UDP server is comparatively easier than setting up a TCP server.

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind(("0.0.0.0", 1337))

while True:
        data, addr = server_socket.recvfrom(1024)
                print(data)
```

We create the UDP socket object, here called server_socket. Next, we bind to the UDP port 1337, listening on any IP address assigned to this computer - it looks very similar to how we created the TCP server.

Again, we have an infinite loop so we can keep sending stuff to the server and it will keep printing it. Then we receive using the recvfrom() method, which takes a maximum number of bytes allowed to be sent: in this case, 1024 bytes.

Notice we have passed two arguments to recvfrom() here? That's because it returns two values: data which is the contents of the UDP packet, and addr which is the address the packet came from. So if you want to fire some packets back, you know where they should go.



Our top terminal window is our UDP server, in the example above. The bottom window is using netcat to send UDP packets: nc -u 127.0.0.1 1337. We typed into netcat "Sending mah UDP packets!" and it was received by the server, which printed it.

Threads

Normally when you run a program a single process is created, which is the code running in memory. The problem is, by default, the program can then only do one thing at a time.

Let's revisit our TCP socket server example from a previous section.

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(("0.0.0.0", 1337))
server_socket.listen(10)

while True:
        conn, addr = server_socket.accept()
                conn.send(b"Do you want to play a game?\n")
        received = conn.recv(1024)
                print(received)

server_socket.close()
```

Before anyone connects to the server, the code will be waiting to accept a connection on line 8, conn, addr = server_socket.accept(). Once anyone connects to it, the code will be waiting on line 10 at received = conn.recv(1024) to accept input from the computer that connected to it. The recv() method is blocking, which means that the program halts at this point and won't proceed until it receives data and is able to execute this step.

If connection A has been made, but no response has been received, and meanwhile connection B tries to initiate a connect... connection B won't be able to proceed, because our code is executing on 10 with connection A. Our while loop can't restart because it hasn't completed.

Check it out in the example below.

```
{ ~ }  » python tcpserver.py
```

```
{ ~ }  » nc 127.0.0.1 1337
Do you want to play a game?
```

```
{ ~ }  » nc 127.0.0.1 1337
```

In the first terminal, we run our TCP server.

In the second terminal, we connect to the TCP server on port 1337, and we get the message "Do you want to play a game?" as usual, but we don't type anything. So the server is now executing line 10, waiting for the first user to type something.

In the third terminal, we try to connect again to the TCP server on port 1337, but we can't because the TCP server is still stuck on line 10, waiting for a response from the second terminal window user before the loop can complete and restart, allowing it to receive another connection.

Making programs multi-threaded

This is the kind of problem that threads can solve. If we use threads in our program, we are making our program multi-threaded. By using threads, we can tell our program to create a separate process for a chunk of code that the processor will execute independently from the main body of code.

Let's fix our TCP server to make it multi-threaded.

```
import socket
import _thread

# Thread handler
def handler(client_sock, address):
    client_sock.send(b"Do you want to play a game?\n")
    data = client_sock.recv(1024)
    print(repr(address) + " said: " + data.decode())
    client_sock.close()
    print(repr(address) + " connection ended.")

# Set up our server
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
server_socket.bind(("0.0.0.0", 1337))
server_socket.listen(10)

# Run the server with threads
while True:
    print("Server listening for connections...")

    client_sock, address = server_socket.accept()
        print("Connection from: " + repr(address))

        _thread.start_new_thread(handler, (client_sock, address))
```

Let's go through this line by line.

In addition to importing our socket module, we also need to import the thread module, which is also part of the Python standard library.

Next, we create the function handler() which accepts a socket object and an address as parameters. We create this as a separate function because it needs to be able to execute separate from the main program in order to be threaded.

Inside this function definition we have the code that sends our message "Do you want to play a game?" and waits to receive a response. Notice this time we are also using the address variable to print out who we're communicating with. We use the repr() function with the address variable, because the information stored in address at this time will be a tuple. In order to print out the content of a tuple, we first have to transform it into a string, which we can do with repr().

Now to the main body of the program.

We create a socket object as before and hold it in our server_socket variable, bind it to 0.0.0.0 on port 1337, then listen for connections. This is exactly the same as when we previously created a TCP server.

Next is our infinite loop where we can accept a new connection. Here, as soon as a connection comes in, we create a new thread using thread.start_new_thread(). This function accepts 2 parameters: our handler() function name, and also a tuple, which contains the 2 variables we need to pass to our handler, client_sock and address, which we received from accepting the connection.

That thread can now work independently from the rest of the program. With that thread created and the connection "handed off" to our handler() function in its own thread, our while loop can now finish, and immediately re-start, awaiting the next connection.

Let's see it in action!

After running our example code in the top terminal window, the server was listening for new connections. In the second window, we connected and received the message "Do you want to play a game?" and the server was waiting for a response. Without providing a response, we connected in the bottom terminal window and also received the message "Do you want to play a game?", which is already better than what we had before.

We then went to the second terminal window and sent back 'yes' to complete the connection, and in the bottom terminal window, we typed 'no' to complete the connection.

Finally, we re-connected from both terminal windows at the same time and received the "Do you want to play a game?" message on both without typing anything in response.

Our TCP server is now multi-threaded and can allow multiple people to connect to it at the same time.

This is just scratching the surface of threads. There are many more applications and many more restrictions on threads, but it is an immense topic that even many university students struggle with. Unless you are writing specialised or very advanced software, it is unlikely that you'll need to go deeper into threading than this.

Create a Portscanner

We aren't done with Python yet, but we're far enough into it that you should now be able to create simple programs.

In cyber security, we often need to know what ports are listening on a target system. To do that we use a tool called a port scanner. At the most basic level, a port scanner is a tool that will try to connect to ports on a target system and report back about which ones are open or not.

Go ahead and try to make one now. You will need to do a little research using a search engine to learn about the connect_ex() method available for socket objects in the socket module. We haven't used yet anywhere in this course, but it will likely be very helpful as you try to create your own port scanner.

You can test your code by scanning the IP address 127.0.0.1. You should notice some open ports.

Try to create your program without looking at the example solution below and see how far you get!

Example solution:

```
import socket

print("Please enter an IP Address to scan.")
target = input("> ")

print("*" * 40)
print("* Scanning: " + target + " *")
print("*" * 40)

for port in range(1, 1025):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    result = s.connect_ex((target, port))
        if result == 0:
        print("Port: " + str(port) + " Open")
        s.close()
```

Taking it further

If you fiew through that problem and you want a tougher challenge, here's an extra stretch goal for you. Don't feel obliged to do it if you struggled: if we were keeping score, this would be for bonus points!

You may have noticed your portscanner is really slow. Why is this?

If a port is closed, your portscanner has to wait for the maximum timeout on the connection before it can say for sure that it isn't open. The problem here is that it can only really do one port at a time, so the more open ports, the slower the whole process.

To speed things up, try using what we learned about threading. You may want to do some research into queues and multiprocessing to solve some common threading pitfalls if you run into problems.

Good luck!

Programming 4

Contents

Now that you can build more powerful programmes we will explore conventions and strategies to build effective, maintainable and clear programmes.

In this module, we will be covering:

- Python's documentation and how to use it
- The importance of style and Python's PEP 8 style guide
- Defensive programming
- Procedural vs object-oriented programming paradigms
- Programming tips to get you started writing your own programs, to help you debug, and to help you recognise common "code smells".

Using Python Documentation

So far we've been highlighting which functions to use in our example programs, but as you start to write your own programs, you'll need more than what we've covered in this course to solve some of the problems you run into. Knowing which standard Python functions exist, which to use when, and how they work is something that comes from a combination of practice, research, and knowing where to find the common reference documents.

Python has an extensive documentation, which lists out the standard provided objects, functions, and methods, as well as the parameters they accept and the values they return. This documentation is very useful: experienced programmers have to look up information all the time, in particular when they need to use some of the less-frequently used Python tools in their work.

The first step is to look up the documentation for the version of Python you have installed. We can find out which version we have by bringing up the Python interactive console, accessed by running python on its own in the terminal.

```
mainuser@ubuntu:~$ python
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
mainuser@ubuntu:~$
```

In the above example we are running Python 3.6.9. To exit out of the Python interactive console, you need to use the quit() function, which is also pictured above.

So now let's find the documentation for Python 3.6 with a Google search. We quickly get a result of: https://docs.python.org/3/

As an example, let's take a function from the math library and see if we can find out how to use it from the documentation alone. Let's say we want to do a calculation such as 2 ^ 18 (2 to the power of 18).

We typed in 'power' into the quick search in the documentation, and the third result from the top was "math - Mathematical functions". That sounds promising!

Clicking on it takes us to the page with a bunch of mathematical functions in it. Scrolling down, or using your browser's find tool to find "power" on the page reveals the below text.

```
math.pow(x, y)

Return x raised to the power y. Exceptional cases follow Annex 'F' of the C99 standard
as far as possible. In particular, pow(1.0, x) and pow(x, 0.0) always return 1.0, even
when x is a zero or a NaN. If both x and y are finite, x is negative, and y is not an
```

```
Unlike the built-in ** operator, math.pow() converts both its arguments to type float.
Use ** or the built-in pow() function for computing exact integer powers.
```

That is quite a lot of information. First of all, we can see that there is a built-in ** operator that we could use instead of this function.

E.g. 2 ** 18

```
mainuser@ubuntu:~$ python
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2**18
262144
>>>
```

That could work. What other options do we have?

There is also a built-in pow() function, which could also work.

```
mainuser@ubuntu:~$ python
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> pow(2, 18)
262144
>>>
```

And finally, there is this math.pow() function, which is similar to pow() except it converts both of the arguments it receives to floats (remember, these are numbers with a decimal point).

```
mainuser@ubuntu:~$ python
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import math
>>> math.pow(2, 18)
262144.0
>>>
```

Notice the decimal point at the end of the result in this third example. It's a small, but subtle difference and one we would likely never have known about without reading the documentation.

The PEP8 Style Guide

Different programmers often have their own styles of programming. How we decide to name our variables, how we use comments, how long we let our lines get before we start adding line breaks, even the epic space vs tab debates you may have seen between programmers are all about style.

If we all only ever worked along on our own individual projects, we probably wouldn't argue so much about style because who would care? If the only code you ever saw was your own, and your style works for you, great! The problem is that, almost inevitably, programs we write will be read and worked on by more than 1 person. This is when style becomes an important - and often fought over - point.

If two programmers have a very different style of writing code, including different ways of naming things or constructing things, and they work on the same project... the code can quickly become a nightmare to work with. Remember: code is for humans. If the code in your program isn't easy for humans to read, then it's not doing an important part of what it was designed to do. The reason we don't tell computers what to do in binary or assembly is because these languages are difficult for humans to read and understand. The computer doesn't care what you name a variable or whether you use 2 spaces or 4 spaces. But humans do, because humans are the ones trying to read and understand what the program does.

PEP 8 Python Style Guide

Many programming languages have preferred style guides. Often, these style guides fiow from the programmers who created the language, and suggest conventions that are consistent with how the language works behind the scenes.

Python has such a style guide: the current version is the PEP 8 Python Style Guide **which you can find at https://www.python.org/dev/peps/pep-0008

This style guide is frequently used by programmers in order to make their code accessible and readable to a large number of other programmers. It contains conventions and style rules that help make Python code generally more readable to humans if followed.

For example, here are some of the style conventions in the PEP 8 style guide:

- Indention should use 4 spaces
- Lines should be limited to a length of 79 characters
- Use of whitespace in expressions and statements
- When and where to use block comments vs inline comments vs documentation strings
- Variables and functions should be named in lowercase with words separated by underscores
- Classes should be named with CapWords convention.

These are just some of the style rules called out in the Python PEP 8 style guide.

Consistency is important

Whenever we're programming, whether it's something only we are working on alone or a larger project we're working on with other programmers, we should always make our style as consistent as possible. Pick a naming convention for your variables, functions, methods, and objects and stick to it. The most important thing about style is not whether you use camel case or underscores or tabs or spaces, it's that the program has a consistent style. Consistent style makes the code easier to read and understand.

This is important even when we build code that will only ever be used by ourselves. It's likely we'll write a program, and then put it away for months or even years before we have to take it out to debug a problem or modify it for a new purpose. Months or years later you're practically a different person: who knows what past you was thinking! Future you will thank past you for creating code with a consistent style, making it easier to read, understand, and modify later.

Sometimes, consistency means we have to bend or break our own personal style rules when we work on someone else's code or work with a team. If we follow PEP 8 rules in our own code, but join a team where they follow a different set of style guidelines when they write Python code, we should always remain as consistent as possible with the existing style. We might think PEP 8 is better than what's there, but what is always worse is mixing multiple styles together. If we refuse to break our "4 spaces forever" rule when we join a "2 spaces" style team, we make the code worse and more difficult to read for everyone.

Defensive Programming

Defensive programming isn't really a paradigm like procedural or object-oriented, but it is a way of programming that tries to expect the unexpected, in order to prevent bugs and unexpected behaviour from occurring.

Assertions

For example, let's consider a function which takes an integer value and does something with it. When we wrote the function, we expected only positive numbers would be passed in.

If later on we forgot about this assumption and pass in a negative integer, the function may not break, but it might cause a subtle error in your program further down the line, which could take a long time to notice and sort out.

Wouldn't it be better to check within your function if the value passed in was negative, and throw an error with a useful error message at that time? The program would stop working, but we'd know clearly why it stopped working. This is often better than introducing a subtle bug that is difficult to spot.

```
def no_negatives(number):
    assert number >= 0, 'negative value passed to the no_negatives() function!'
    print(number)

no_negatives(5)
no_negatives(-2)
```

And here's the output when we run this code:

```
$ python program.py
5
Traceback (most recent call last):
  File "program.py", line 7, in <module>
    no_negatives(-2)
  File "program.py", line 2, in no_negatives
    assert number >= 0, 'negative value passed to the no_negatives() function!'
AssertionError: negative value passed to the no_negatives() function!
```

Notice we have a useful error message here that tells us exactly why the program stopped working. The assert function will cause the program to crash with the error message we provided if a condition is not met. In this case, if the integer passed into the function is not 0 or more, then the program will crash.

We can also do multiple assertions like so:

```
def no_negatives(number):
    assert isinstance(number, int), 'non-integer value passed to noNegatives()
function!'
    assert number >= 0, 'negative value passed to the noNegatives() function!'
    print(number)

no_negatives(5)
no_negatives("hello")
no_negatives("-2")
```

Here, we are making sure that:

- The data type passed into the function is an integer.
- The data is a positive number.

Tests

So far, we've only been making sure variables within a function are appropriate values. How about testing if a function is working as expected overall? We can write tests for a function to check their behaviour.

Take this function as an example:

```
def mult(x, y):
    a = abs(x)
    b = abs(y)
    return a * b
```

This function is supposed to take two parameters and multiply them together, but we've made an error here, where any negative numbers passed into the function are positive.

If you test the function like this:

```
print(mult(5, 5))
```

We'll get the expected result of 25, so we might miss this subtle bug.

Let's write a test for this function.

```
def test_mult():
    assert mult(5, 5) == 25, 'mult test failed 5 * 5'
    assert mult(5, -5) == -25, 'mult test failed 5 * -5'
    assert mult(-5, -5) == 25, 'mult test failed -5 * -5'
```

When we run this test function, it will try a variety of possible combinations to make sure it's getting the expected result. We can have these tests run every time you run the program and so we can be alerted if someone makes a change in the code that breaks some functionality down the line.

These kinds of tests can be very useful, but only as much as they are written well and cover all possible test cases and pathways through the code. If we make a mistake in our test, then the test is going to be useless.

Unit Tests in Python

James Lyne talks about the importance of unit tests when developing applications.

Programming Paradigms

When we talk about programming paradigms we're talking about the different approaches we can take in general toward how we create programs. If we look up the word "paradigm" in the Oxford English Dictionary, the first two definitions are:

> A typical example or pattern of something; a pattern or model.
> A world view underlying the theories and methodology of a particular subject.

So when we think about programming paradigms, what we're considering is the larger way in which we structure and organize our code, especially when trying to build complex programs.

There are several different programming paradigms, but in this course we're only going to talk about the 2 most common ones you're likely to hear about when writing Python programs: procedural programming and object-oriented programming.

Procedural programming

Procedural programming is the paradigm that new programmers tend to gravitate toward, and is most often represented in beginner tutorials and guides. We've regularly seen procedural programming in most of the Python code examples found in this very course.

When we write procedural code, we break down tasks into a series of steps. Frequently repeated tasks are split apart into reusable functions that can be called at any time, often by any other function. Both humans and the computer run through the program step by step.

Another quality of procedural code is that the logic and the data are often quite separated. Each step gets handed the specific data it needs to complete that step, and then hands off some data to the next step.

Procedural programming is where almost every programmer starts. That's because it's very good for creating small to medium sized programs with minimal to moderate amounts of complexity. If you need a program to parse a CSV file and transform it into a slightly different CSV file, or build a port scanner, or create a rock-paper-scissors game, procedural programming will do the job very well for you.

Object-oriented programming

Object-oriented programming uses classes to model objects, capturing both their attributes - the data that describes them - and their methods - the behaviours they have. By thinking about programs this way, we tie together data with the functions that create and modify that data.

This style of programming thinks about most things as objects, and tries to model them as such. If we were writing code to handle a registration system, we'd likely have a User() object. That object would know everything about the user, such as their name, their email address, their account creation date and time, and their password. But it would also know how to modify that data as well, using methods such as change_user_name() and change_user_email().

What's useful about this is we can now move this object through our program and use it in lots of different ways. Registering a new user is about populating all the important attributes of the object before the data is saved. Creating a sign in form for existing users can also make use of this same user object, accessing information about the user's password and username. And a profile page that displays the user's information to others can also use this same object. The logic we build in our program centres around creating, manipulating, and passing around objects.

Object-oriented programming tends to require more abstract thinking, and a strong ability to spot reusable patterns that can be reasonably grouped together into objects. But it can be very powerful, and make large, complex programs much easier to understand and maintain. Which paradigm should you use?

Well... it depends.

Some languages don't support object-oriented programming. C, for example, doesn't have a concept of objects or classes. But many languages do allow for the object-oriented paradigm, and it is very popular among experienced programmers.

It also depends on your experience. If you're brand new to writing code, procedural programming is a good way to get better at understanding the basic concepts of how to program without having to also juggle the extra abstraction that come with building things as objects.

Object-oriented programming is also quite a lot of setup. It can pay off big when we create large, complex programs, but for smaller one-task problems that need solving, it's often too much overhead given a small set of functionality. Simple programs are generally better suited to the procedural programming style.

Programming Tips

One of the most difficult transitions for a new programmer to make is switching from following a tutorial or series of examples - like you've seen in this course - to creating your own program. When you look at code someone else has written, it seems very straightforward and easy as long as you know the syntax. But as soon as you have a blank code editor open, it can be difficult to know how to get started.

Be specific about the problem you are trying to solve

Programming is mostly about solving problems. If you want to build a program to do something, the first step is to be as specific as you can about exactly what you want your program to do.

Perhaps we have a website that we want to ensure is always running, and if for any reason it goes down we want to be notified.

This is definitely a problem that code can help us solve! But before we can start writing any code, we have to make some important decisions. For example, before we can write a single line of code, we have to decide:

> How do we want to be notified? Via email? Maybe via text message?
>
> Do we want to be notified if our website is only down for 1 minute? By the time we get to a computer to look at the problem, maybe it will have already come back online.
>
> Maybe we only want it to alert us if the site is down for at least 10 minutes.
>
> If we can't get to a computer to look at the problem right away - perhaps it will take us an hour to drive home to get to our computer - do we want to keep getting an email or text message every 10 minutes?

It's helpful to start by writing down exactly what you want the program to be able to do. In this example, we might decide on the following pieces of functionality we want our program to have:

> The program should check our site every 10 minutes. If it is down for at least 2 checks in a row - a minimum of 10 minutes - it should send an alert.
>
> The program should alert us via a text message.
>
> The text message should include the date and time at which our program first detected our site was down.
>
> No further text message should be sent after the first indicating our site is down, but the program should keep checking it.
>
> If the site comes back up after we have received a text message indicating it was down,

```
our program should send a text message letting us know, and include the date and time
when the program detected our site was back up.
```

These 5 points give us a very clear idea of what we need to build, and exactly what we want it to do when it's finished. It always helps to think through what we want to achieve before we start writing any code. Break down large problems into smaller ones

Ok, now we know what we want to build... but how do we know where to start?

Next, we should start breaking down our functionality into a smaller series of problems we have to solve. A good way to do this is by writing pseudocode. Pseudocode is a way of starting to think like a programmer but without worrying about syntax or the details around making real code work. It's all about wrapping our head around the logic of what has to happen in order for our program to do what we need it to.

Let's create a little bit of pseudocode to help us understand the logic we'll need to implement for the monitoring program we want to build.

```
EVERY 10 MINUTES, check if my website is still up.

IF website check fails:

    IF it is the first time in a row a check has failed:
        THEN remember one check has failed and store the date and time.

    ELSE IF it is the second time in a row a check has failed:
        THEN remember two checks have failed,
            AND send me a text message containing the date and time
            of the first check to fail.

    ELSE is more than the second time in a row a check has failed:
        THEN remember the number of times in a row the check has failed.

ELSE the check has passed:

    IF this check has passed after previously failing a min of 2 checks in a row:
        THEN send me a text message containing the date and time of this check.

    ELSE this check has previously failed only 1 check in a row,
        OR has not failed the previous check:
        THEN do nothing
```

You don't have to use a specific style of pseudocode or special syntax. The idea of pseudocode is to help us understand in our own human brains what the broad structure of logic is, and to get an idea of what order we should start solving our problems in.

This pseudocode gives us some idea of how we probably want to tackle this project. Here are some things we can start to understand about our unwritten program looking at this pseudocode:

We'll need some kind of loop, which we can tell because we want something to happen every 10 minutes. In this case, it's probably going to be an infinite loop, because we want to just keep running until we shut it down manually.
We need to be able to count and remember how many times our check has failed. We have a very specific action we need to take at a specific count - when the check has failed 2 times in a row - so we need to keep track of this.
We also need to save and keep track of the date and time of the first failed check. If the site fails twice in a row, we want to send the date and time of the first failed check rather than the second.
If a check passes after having previously failing 2 checks in a row, we need to send another text message and also reset our failed check counter back to 0, since in this program we only care about failed checks if they happen one after the other.

These are all smaller, self-contained little problems we can solve one-by-one. Work on 1 small piece at a time

Many new programmers try to write an entire program at once, but experienced programmers know that a better strategy is to work on 1 piece at a time, and slowly build a program up piece by piece.

We know in this case we need a loop that will check if our website is up and responding every 10 minutes. Forget about everything else - all the functionality around what to do if the check succeeds or fails - and start with solving just that problem. Create a little piece of code that just checks a website every X minutes and returns a pass or fail if that website is up. Create a simple program that just prints PASS or FAIL at that point.

After we get that working nicely, what's next?

There are 2 paths to follow here, PASS or FAIL. Let's pick one and do the next piece of that path. In this case, it's probably best to pick the FAIL path first. Why? Because of this bit of pseudocode we wrote:

```
ELSE this check has previously failed only 1 check in a row,
    OR has not failed the previous check:
    THEN do nothing
```

In order to go down the PASS path, at some point we need information from the FAIL path according to our pseudocode logic. That means it makes more sense to work on the FAIL path first in this particular case.

In the FAIL path, all our options from here require us to be able to count the number of times a check has failed in a row. So the next problem we can solve is to add a "fail counter" to our program that tracks the number of times in a row our program fails, and if it passes after previously failing, we reset the counter.

After we get that bit working, we have a program that does only the following:

- It checks to see if a website is up every 10 minutes.
- If the website check fails, it increments a failed check counter, prints out the number of failed checks, and prints the string FAIL.
- If the website check passes, it resets the failed check counter to 0, prints out the number of failed checks, and prints the string PASS.

Good progress!

After this, we would pick another small piece of the needed functionality and add that in. Eventually, after we add little features one by one, we'll have our entire program.

Programming is best done iteratively. We isolate one small piece of functionality and write code to solve just that problem, and only that problem, creating a very simple program. Then we tackle the next small piece of functionality and add that into our program so it is slightly more complicated. We do this again and again, slowly building up our program piece by piece and problem by problem.

Sometimes when we add a new piece of functionality we have to modify previous things we've built. That's ok! Refactoring is an important part of making progress in programming.

Test your code frequently

It's tempting to try and write an entire program or feature from start to finish in code before testing it... but this is rarely a good strategy. It's better to make small changes, then run and test your code and see if that change is doing what you expect.

The problem with making many changes at once before testing is it becomes more difficult to understand which change caused the error. If you test after making a small change, you know right away that the change you made is probably the reason for the error, so it becomes much easier to debug.

Quite a lot of the time programmers spend writing code is spent printing out variables, arrays, and object attributes to the screen at various steps in the program so we can understand what's going on, and if it matches what we're trying to do. It's a very rare programmer who can write 1000 lines of code and have it work perfectly the first time... and if it did happen most people would consider it a fiuke. Build in small pieces, and test frequently.

Error messages are useful

Programmers love error messages - seriously. Error messages and exceptions are great helpers! They give us helpful information about where our program has gone wrong so we can fix the problem more quickly. And often they expose cases we haven't considered yet: such as when a user inputs a string like "zero" instead of the number 0. When you see an error it doesn't mean failure, it means progress.

Always read your exception and error messages closely: they have helpful information to guide you on how to fix them. They'll usually come with a traceback to show you where in the code the error occurred, and give you some idea of what type of error it is, such as a syntax error or a divide by 0 error. If you haven't seen the error message before, Google it! You'll almost certainly find someone who has had a similar problem before, seen that error, and has figured out how to solve it. Revise, rebuild, refactor

As the saying goes, "Rome wasn't built in a day". That applies to programming too. Programming takes time, and quite often requires a lot of revisions. Sometimes we have to start writing code that doesn't quite work in order to understand how to build code that does work. This is entirely normal and every programmer - from the beginner to the expert - has to regularly rewrite and refactor their code. If anything, experts probably refactor their code more often than beginners do.

Sometimes we have to revise larger portions of our code when we add in new features. That's ok, and also entirely normal! Adding new features often doesn't just add new functionality, it often changes existing functionality, sometimes in subtle ways.

Because programming is iterative, we revise the same bit of code over and over again.

Code Smells

If you're familiar with programming and have been writing code for a while, even as a beginner, you may have heard the term "code smell" before. This is a term programmers generally use to refer to visual symptoms that code has deeper underlying problems. Often, code that "smells" is confusing to understand. And since code is for humans, that often spells trouble. Code that is difficult to understand is hard to maintain, and difficult to ensure is working properly, with each possible outcome fully tested.

This section will review some common code smells and why we try and avoid them.

Too much nesting

Let's consider this code:

```
for item in items:
    if a:
        if e or f:
            for thing in item:
                if g and h:
                    if j:
                        # Do this thing
                    else:
                        # Do this thing slightly differently
                else i or j:
                    # Do something else
                else
                    # Do something different
        else
            for other_thing in item:
                # Do another different thing
    elif b:
        for thing in item:
            if i or j:
                # Do another thing
            elif g:
                if h:
                    # Do a thing
                else:
                    # Do a different thing
else:
        for other_thing in item:
            if g and h:
                # Do something
            if g or i:
                # Do another thing
            else
                # Do something different
```

Look at all that nesting! It gets 6 layers deep in some places. This kind of nesting of loops and if statements inside each other generates a lot of indentation, which makes code difficult to read.

But - even more importantly - code that has been heavily nested makes it very difficult to understand how many different paths your code has to follow, and difficult to tell if your code covers all the cases it needs to cover.

Heavily nested code can almost always be simplified and made easier to understand and test by breaking apart the logic into smaller functions. If your code starts to look like the sample above, it's time to pick it apart carefully and break your code down into smaller, easier to follow pieces.

Keep your functions small

Typically, the longer a function is, the less readable and less maintainable it is. This rule is similar to the rule about nesting, and the 2 rules often go hand-in-hand: quite often one of the reasons functions get long is because we have too many nested loops or conditionals, or are trying to have one function check too many different cases. When you create functions, try to think about them as individual bricks. If bricks were huge and had a complicated shape that only fit together in one way that would make them a lot less useful.

In a previous section, we wrote a little program to help James decide if he had time to get coffee. In that example, we broke down the code into smaller functions. Here's the same code in one larger function:

```
def can_agent_j_get_coffee(minutes_available):
    print('Does Agent J have enough time to get coffee this morning?')

    people_in_line = ['Unknown Person A', 'Agent M', 'Agent Q', 'Unknown Person B']
    estimated_wait_time = len(people_in_line) * 2

    if estimated_wait_time < minutes_available:
        print('Yes, plenty of time.')
    else:
        if 'Agent Q' not in people_in_line:
            print('Agent Q isn\'t in line... no coffee for Agent J today.')

        position_agent_q = people_in_line.index('Agent Q')
        wait_time_for_agent_q = (position_agent_q + 1) * 2

        if wait_time_for_agent_q < minutes_available:
            print('Agent J can swap places with Agent Q and get his coffee.')
        else:
            print('Not even Agent Q can save Agent J today... no coffee today.')

can_agent_j_get_coffee(8)
```

There's a lot going on in the can_agent_j_get_coffee() function, and if you have to work your way through it to modify it, it will take extra time. Additionally, if you're trying to add a feature, you may find it difficult to add. For example, what if we needed to add in some additional checks to see how badly J wanted coffee? Or checks to see how important it

was to be on time for his meeting? Adding these pieces of functionality into this one function would really start to make things difficult to manage.

Here's the code broken apart into smaller bits of functionality:

```
def can_wait_for_coffee(minutes_available, people_in_line):
    estimated_wait_time = len(people_in_line) * 2

    if estimated_wait_time < minutes_available:
        return 'Yes, plenty of time.'
    else:
        return can_swap_with_agent_q(minutes_available, people_in_line)

def can_swap_with_agent_q(minutes_available, people_in_line):
    if 'Agent Q' not in people_in_line:
        return 'Agent Q isn\'t in line... no coffee for Agent J today.'

    position_agent_q = people_in_line.index('Agent Q')
    wait_time_for_agent_q = (position_agent_q + 1) * 2

    if wait_time_for_agent_q < minutes_available:
        return 'Agent J can swap places with Agent Q and get his coffee.'
    else:
        return 'Not even Agent Q can save Agent J today... no coffee today.'

# Ask our question
print('Does Agent J have enough time to get coffee this morning?')

# Get our answer
people_in_line = ['Unknown Person A', 'Agent M', 'Agent Q', 'Unknown Person B']
print(can_wait_for_coffee(8, people_in_line))
```

This is already an improvement over having one large function. And even here we can spot another place where we can further refactor to break apart another function that checks if Agent Q is in the line separate from the logic that checks if she's far enough ahead in line to make the swap worth it. That would make our code even more modular, and would later allow us to more easily allow J to check for other agents in line who owe him a favour, not just Agent Q.

As a general rule, if your function starts to get more than 40 lines long then it's definitely time to roll up your sleeves and start to do some refactoring, but as we can see above sometimes even a 10 line function can be broken into smaller pieces. Break your big functions down into smaller ones that can work together to get the job done, and can be useful to you in a larger variety of situations.

Avoid code duplication

Let's keep looking at the above example. What if both Agent Q and Agent S owe Agent J a favour? We have a function can_swap_with_agent_q() that checks for Agent Q in the line, but not for anyone else. It's tempting to just copy-paste this function and call it can_swap_with_agent_s(), change a few lines, and incorporate it into the program too.

Ok... but what about other agents that might owe Agent J a favour in the future? What if 20 agents owe Agent J a favour? We might have 20 different can_swap_with_agent_x() functions in our code, all basically the same. And then what happens if we find a bug in that code and have to fix it? We'll have to change all 20 functions! What a terrible, tedious task!

If you find yourself tempted to copy and paste a piece of code in order to solve a problem, stop yourself if you can. The better solution is to ask: what is the common task we need both of these pieces of code to do, and how do we refactor our code so that if there's a bug or a change needed later, we only have to fix it in 1 place?

Debugging

If there's one thing that's inevitable when you're a programmer, it's bugs. There are always more bugs to find and fix, in particular as your program starts to get more complex. And where there are bugs... we must debug!

Debugging is a skill like any other skill, and we get better at it with experience and practice. It's one of the areas of programming that beginners can struggle with the most, and it's easy to understand why: debugging can be hard! Sometimes it's a tricky combination of detective work and research to get to the bottom of a particularly subtle bug.

So when something in our code is going wrong, how do we begin?

Step One: find a way to replicate the issue reliably

The first step is to figure out how to cause the bug to happen. If you can do this reliably - if you know the exact sequence of steps that will get the bug to appear every time - that's the first big hurdle crossed.

Until we can reliably replicate a bug, it is very, very difficult to fix it. It's like trying to find a blue dot on a wall in a dark room: if we can't see, how are we ever going to find it?

Some bugs are easy to replicate. They're usually the ones that are easy to find and easy to troubleshoot and fix. The sneaky, intermittent bugs are more difficult. Sometimes programmers get reports of weird bugs from other people, but if those people can't give us a way to replicate it and see the bug in action for ourselves, then there isn't much we can do to fix it.

Step Two: figure out where the bug is in the code

The next step when we can reliably recreate the bug is to start tracing it to the source. At this point we're not trying to figure out why it's happening, we're more concerned with understanding where in the code the bug is being created.

Sometimes this is straightforward: a traceback or exception may direct you to the exact file and line number where the bug is. But sometimes it can be quite tricky to track a bug to its source.

What makes this extra tricky is when the error or exception getting thrown is at line 200, but it's not anything in this code block that's actually causing the bug. The bug may be caused at line 50, and not revealing itself until 150 lines later.

Consider the example of a ZeroDivisionError exception. This is caused if the program is asked to divide a number by 0. The exception is thrown at line 200, but it's likely the 0 was created farther back in the stack. The bug could be 10 steps before this, when a 0

value was assigned to a variable in an unexpected way, and then this value was passed across several functions without causing a problem until it hit this one spot in the code, where it was stopped in its tracks.

So how do we figure out where a bug is happening in our code? It can take some detective work, but here are some tips to help you track it down.

- Step backward through your code. Start at the point in the code where the bug was exposed, then slowly step backward through your program's logic, re-reading your code and reviewing it carefully.
- Print out variable, list, and dictionary values at each step. As you walk back through your code, print out the values of variables, lists and dictionaries at every point they are passed from function to function, or modified by a function, to make sure the values are as you expect at that point.
- Comment out code to focus on and run only specific sections of your program. The beauty of comments is that we can comment out anything: even code! We can turn off bits of functionality without having to delete and re-write something we've already done. By commenting out sections of code, we can "backtrack" and turn off functionality selectively to see how our program runs without it. If our bug disappears after commenting out 1 particular line or a specific function... bingo! There's a good chance that's the culprit.

Step Three: understand why the bug occurred

Once we've isolated exactly where the bug is happening, the next step is understanding why the bug is happening. What's causing it, exactly? Sometimes this is very easy to see once you find where the bug is happening, and other times the issue can be more complicated, or related to misunderstanding how a library or function you're using works.

This is the step where you might end up having to spend a lot of time researching and searching on Google to understand why the bug is happening. And it's important to understand why, because if we don't then we might not fix it correctly.

Sometimes we think we're at this step, and then after some research we realize we haven't actually found the bug yet, and it's still farther back in the stack. For tricky bugs, we may have to bounce between step 2 and step 3 and back again a few times until we get to the heart of the problem.

In some cases we can eliminate the bug by guessing what the fix is and making it. And if the bug appears to go away with this fix then the problem is solved... maybe. If we don't fully understand what went wrong in the first place, we can't really be sure the bug is gone.

Step Four: refactor to eliminate the bug

Once we know exactly where the bug is and why it's happening, all that's left is to make revisions to our code to eliminate the source of the bug.

Hopefully the bug you find is a relatively easy fix... but sometimes, a bug can be a bit of a nightmare to fix, in particular if it's baked in at the most fundamental level of how your program works.

In some situations, it may be impossible to eliminate the bug entirely in the time you have to work on the problem. If this is a situation you find yourself in - and all programmers do from time to time - then do what you can to minimize the bug or create useful error messages to get the user back on track after an error as smoothly as possible.

It's always better to fix a bug if you can, even if it means rewriting and reworking a large chunk of your code. Otherwise, it will continue to be a thorn in your side, and will likely cause more problems down the road. Take a break... but don't give up!

Debugging can be a frustrating exercise. Sometimes it can feel like an impossible task, and all we want to do is throw our computer out a window and never program again.

Every developer feels this way from time to time when they hit a particularly troublesome bug. Sometimes the best thing to do is to walk away from the problem for a little while. Do something else, and come back to the problem the next day with a fresh outlook. Often a debugging problem that seemed unsolvable yesterday is much easier to sort out when we come back refreshed and relaxed.

Programming 5

Contents

Python is a powerful and intuitive language, but sometimes you need lower level control. In these two modules, we will introduce C and some of the fundamentals of this extremely important language.

In this module, we will be covering:

- What is C and why is it important?
- Running C programs
- Printing in C
- Variables in C
- Maths in C
- Functions in C

What is C

C is a fairly low-level programming language, it was created in 1972, and it was based on a programming language called B. Despite being so old, C is still widely used today. In fact, many higher-level programming languages are written in C. Python is one of these, the Python interpreter (the bit that takes the Python you write and turns it into machine code that the CPU can understand) is a program that was written in C. The list of languages that are built on top of C is extensive and includes JavaScript, Java, PHP and Perl amongst others.

C is popular for a few reasons. First of all, C is available on pretty much any system. It's also portable, if you write it well, which means if you write a piece of C code, you can generally compile it on any other system and have it work the same way. C is fast; if you write a piece of C code and compile, the speed at which it runs is much faster than a language such as Python. Finally, C is a mature language. It has been around a long time, and major changes to the standard are few and far between, unlike other languages such as Python where new versions are released regularly, and they aren't always backwards compatible with older versions.

C does have its drawbacks though. It's a dangerous language to use for casual programmers because there is no safety net. C will always do exactly what you tell it to do and nothing more, and this can cause unintended bugs which can lead to exploitable software.

So, why are we teaching you C? Well, in no way are you going to be able to write fiuent C at the end of this course, but you should be able to at least read it to some extent. That is going to be useful later in the course when we start looking at software fiaws and how to take advantage of them, because most of those fiaws will come as a result of badly written C. Additionally, because C is a lower-level programming language than Python, it has some features which Python doesn't have, so we will be taking this opportunity to explore those aspects of programming also.

Running C Programs

C is a compiled language, so to run a C program you first have to compile it into an executable file. The process of compilation by default generates an executable file that is suitable for the processor architecture you are compiling the code on. If you need to compile the program to run on a different processor architecture, you must either compile the program on a system with that processor architecture or you need to perform a task known as cross-compilation, where you tell the compiler about the target architecture you are building the executable for.

There are several different compilers out there for C. On Linux the most common one is GCC so we will be using that one in our examples.

For this example, we are going to be using the "Hello, World" example from the previous section. Our code is in a .c file called 'hello.c'. To compile it we use gcc from the command line like so:

```
gcc -o hello hello.c
```

The -o parameter specifies the output file (the filename of the executable to generate). In this case, we are creating an executable file called 'hello'. After that, we specify the input file, where our C code resides.

```
user@SANS:~/Desktop$ gcc -o hello hello.c
user@SANS:~/Desktop$ ./hello
Hello, World!
user@SANS:~/Desktop$
```

Remember in Linux the './' means execute the file in the current directory and 'hello' is the file name.

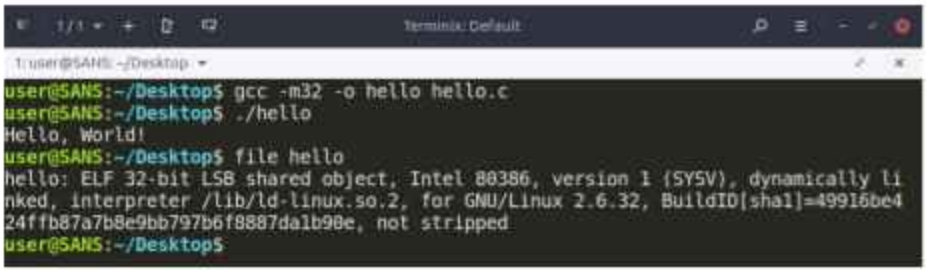
If we run the Linux tool 'file' on our executable, we can get some information about it:

```
hello: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=88cd59f387301bdc8abe58360ea55b641135b5c7, not stripped
```

So this is an ELF file, which isn't to say it has pointy ears... ELF stands for Executable and Linkable Format. It's basically the Linux equivalent of .EXE on Windows. We can also see that this is a 64-bit executable, which means it isn't going to run on a 32-bit processor. Remember, a 64-bit processor can run 32-bit software, but a 32-bit processor can't run 64-

bit software. So we know that this executable file won't run on Windows, and it won't run on a 32-bit processor even if it is running Linux.

What if we want to send our friend a program, and we're on a 64-bit processor, and he is on a 32-bit processor? We can just add the -m32 parameter to gcc:



Now, to get a program that is compiled for Windows is a different story altogether, and we actually need to use a cross-compiler such as 'MinGW'. However, that is beyond the scope of this course.

Printing

Okay, let's get started on our very first C program:

```
#include <stdio.h>

int main() {
    puts("Hello, World!");
    return(0);
}
```

So, there are a few differences from Python. First of all, the include is similar to Python's import. It's a way of including a library. In this case, the "stdio" library is a standard C library for dealing with input and output. The name stands for "standard input output".

The main function is the program's entry point. In Python, you could just write your code, but in C everything has to be within a function. The function that is run first when the program starts is called the program's entry point, and it is the main function.

C has a slightly different way of defining functions; int main() tells us that the main function takes no parameters and returns an integer value. C is a strictly typed language, unlike Python which is dynamically typed. In Python, the value determines the type of the variable, but in C you have to determine the type of the variable first and make sure whatever value you are storing in that variable conforms.

The curly braces are a way of determining a block of code. Remember in Python where we used the indent to show which block of code belongs where? In C you don't use indents, you use curly braces { }. It is good practice to indent your code anyway, because it makes it more readable, but you don't have to. We could just as well have written our code like this:

```
#include <stdio.h>

int main() { puts("Hello, World!"); return(0); }
```

The code above is perfectly valid in C, but it's also not a good habit to get into writing it that way. The indent is not required because curly braces show that the block of code within belongs to the main function.

You'll also notice that we put multiple statements on one line. In Python, every statement goes on its own line, but in C the semi-colon (;) is used to show when a statement ends so you can actually string multiple statements on one line (but you shouldn't, for readability reasons).

To output text, we use the puts() function. "puts" stands for "put string", and it takes just one parameter: the string to output.

Finally, we return 0. Remember, the main function we defined returns an integer value, so in this case, we are returning a 0. Why? It's common practice to make the main function return 0 if the program finished executing successfully, and return a 1 if the program ran unsuccessfully.

You can really see the difference between Python and C using this example. The equivalent code in Python is just:

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

That's five lines of code in C vs one line in Python. This is the difference between a low/mid level programming language and a high-level programming language.

Your First Variable

The next section will cover variables and data types much more, but it is timely here to introduce the idea of a char array. This is a variable with a type that can store 'strings'. This means we can use something stored in memory once multiple times. Let us store a string that contains 'Hello my first variable'.

```
char variable_1[] = "Hello my first variable";
```

Note the []? This is normally where a size would be supplied, as in maximum length of string we can store. However in this instance [] and a string means the compiler will calculate the required size and set it for us.

We can use this with puts if we want, as follows:

```
char variable_1[] = "Hello my first variable";
puts(variable_1);
puts("and a space ");
puts(variable_1);
```

String Handling and printf()

In the last section we used puts() to print out contents. Puts is a relatively basic tool, but let's take a moment to use a more advanced function printf().

printf() has the ability to handle multiple output streams, different data types and even convert a given type to a different format.

Let's take a look at printing a simple string with printf(). We will insert it in to another string to show how it is more powerful.

```
#include <stdio.h>

int main() {
   char username[] = "jameslyne";
   printf("Your username is %s \n", username);
   return(0);
}
```

The `char username[]` declares a char array that stores the string 'jameslyne'. Printf is asked to print the string, which is mostly self explanatory. There is however a special couple of components to this: - '%s' says please insert string here. printf() will grab the first variable passed in to it and substitute whatever it finds at that location. If this is a string it will print until a NULL byte, that is a the standard way of saying 'this string is over'. That is simply put a 0 value, which as we know from our ASCII module is an actual 0 byte integer. The '0' in ASCII is actually 0x30 and they are not to be confused. - '\n' This says let's print a new line return, which makes sure there is a return before the program exits. Otherwise the prompt would appear straight after the text, which looks odd. Try it without!

We can also bring in other types of data and format them accordingly. For example we can declare an age value and print this too:

```
#include <stdio.h>

int main() {
   char username[] = "jameslyne";
   int age = 934;
   printf("Your username is %s and you are %d years old\n", username, age);
   return(0);
}
```

This outputs:

```
Your username is jameslyne and you are 934 years old
```

Notice how the %d was translated to a number? This is because %d says grab a number from this location, where we supplied an integer 934.

Welcome to the Danger Zone

Here we are accessing variables in memory and printing them, and we as the developer get to specify how they should work. But mistakes can be a real issue and lead to functional or security bugs! For example, what if we confused a number with a %s?

```
#include <stdio.h>

int main() {
    char username[] = "jameslyne";
        int age = 934;
    printf("Your username is %s and you are %s years old\n", username, age);
        return(0);
}
```

The compiler when we try to build this will warn us we have done something potentially stupid:

```
output.c: In function 'main':
output.c:6:45: warning: format '%s' expects argument of type 'char *', but argument 3
has type 'int' [-Wformat=]
```

It does however let us continue and compile anyway. We get to do what we want in C whether it is a good idea or not!

Run this program and it will either outright crash, or print bizarre text to the screen until it finds a null. You said to print a string and it will try to do so whether there is one there or not. It might even leak information that is useful for security purposes! Welcome to the subtle world of careful programming in a low level language.

Variables

We mentioned this briefiy in the last section, but C is a strictly typed language. That means you need to tell it the 'type' of data to expect in a variable. Here are the data types available in C:

Integer Types

```
int: An integer value
short: A short integer value
long: A long integer value
```

The above integer types could either be signed or unsigned. A signed integer is capable of being a negative value, but the most significant bit is used to hold the sign, and therefore the maximum size of the integer is reduced.

So, if you take an 8 bit signed integer:

10010001

The most significant bit is the one furthest on the left. That is the one that is used to indicate a negative value, if it is a 1 then the number is negative. If it is 0 then the number is positive. This means because one of the bits is used to hold the sign (and it's the one with the highest value then the maximum number that the integer can represent is reduced quite significantly).

For example:

A signed (16 bit) integer is capable of representing numbers from (at least) -32,767 to +32,767. An unsigned integer is capable of representing numbers from (at least) 0 to 65,535. It is therefore 16 bits in size (2 ^ 16 = 65,536, and remember we count from 0 so the maximum size is correctly 65,535).

A signed short is capable of representing numbers from (at least) -32,767 to +32,767. An unsigned short is capable of representing numbers from (at least) 0 to 65535. (Yes this is the same as an integer.) It is therefore 16 bits in size (2 ^ 16 = 65,536, and remember we count from 0 so the maximum size is correctly 65,535).

A signed long is capable of representing numbers from (at least) -2,147,483,647 to +2,147,483,647. An unsigned long is capable of representing numbers from (at least) 0 to 4,294,967,295. It is therefore 32 bits in size (2 ^ 32 = 4,294,967,296, and remember we count from 0 so the maximum size is correctly 4,294,967,295).

Take the values above with a pinch of salt; the actual values vary from processor architecture to processor architecture, which is why we wrote: "at least". These are the

minimum values that these types must be able to store. A good way to check them is to simply compile a program and ask it to output the size!

Floating Point Types

> float: A floating point value (number with a decimal point). This one is a single-precision floating point value. e.g. 10.327000. Typically 32-bits.

Note a fioat is smaller than a double so can represent less overall decimal accuracy.

> double: A floating point value (number with a decimal point). This one is a double-precision floating point value. e.g. 7.23847298343 (and more!).

Note that a double will often be 64-bits these days, and can represent a vast number of significant digits past the decimal point.

Again, the implementation of these varies between processor architectures. As a general rule, always use a double in C, unless you are dealing with thousands of fioating point numbers, in which case use fioats.

Character Types

> char: A character type. This is actually technically an integer type that is always 1 byte large. Practically speaking, it is used for holding characters (remember they are just numbers that are displayed using an encoding scheme), but it could be used for other types of numerical data if you wanted. An example would be 'A' which is the hexadecimal literal value of 0x41. This would fill a standard single char byte. Char bytes can be chained to store strings, or a series of numbers.

And that's it...

Hey, wait, what about strings?!

There is no string data type in C. If you want a string, you have to make an array of characters.

Now let's get to writing some code. First, let's take a look at our integer types:

```
#include <stdio.h>

int main() {
   int anInt = 42;
      long aLong = 42;
   short aShort = 42;
```

```
unsigned int anUnsignedInt = 42;
    unsigned long anUnsignedLong = 42;
        unsigned short anUnsignedShort = 42;

        printf("anInt contains: %d and is size: %lu\n", anInt, sizeof(anInt));
    printf("aLong contains: %ld and is size: %lu\n", aLong, sizeof(aLong));
        printf("aShort contains: %hd and is size: %lu\n", aShort, sizeof(aShort));

        printf("anUnsignedInt contains: %u and is size: %lu\n", anUnsignedInt,
sizeof(anUnsignedInt));
        printf("anUnsignedLong contains: %lu and is size: %lu\n", anUnsignedLong,
sizeof(anUnsignedLong));
        printf("anUnsignedShort contains: %hu and is size: %lu\n", anUnsignedShort,
sizeof(anUnsignedShort));

    return(0);
}
```

Here we are using printf to look at our values in detail.

Notice the format string specifiers are very specific.

```
%d is for an int
%ld is for a long
%hd is for a short
%u is for an unsigned int
%lu is for an unsigned long
%hu is for an unsigned short
```

The results on my 64-bit Linux system (your results may vary by architecture) are:

```
user@SANS:~/Desktop$ ./vars
anInt contains: 42 and is size: 4
aLong contains: 42 and is size: 8
aShort contains: 42 and is size: 2
anUnsignedInt contains: 42 and is size: 4
anUnsignedLong contains: 42 and is size: 8
anUnsignedShort contains: 42 and is size: 2
user@SANS:~/Desktop$
```

Now let's look at our fioating point types:

```
#include <stdio.h>

int main() {
    float aFloat = 0.1337133713371337;
        double aDouble = 0.1337133713371337;

        printf("aFloat is value: %20.18f\n", aFloat);
    printf("aDouble is value: %20.18f\n", aDouble);
        return(0);
}
```

Here we have a slightly different way of using format string specifiers with printf. Here we are telling printf() that we want to print the fioating point values with 20 digits before the decimal point (if there are 20 digits) and 18 digits after the decimal point (if there are 18 digits). The result is somewhat unusual:

```
user@SANS:~/Desktop$ ./vars
aFloat is value 0.133713364601135254
aDouble is value: 0.133713371337133707
user@SANS:~/Desktop$
```

We stored the same decimal value in each variable, so why are they different when we print them? It's a case of accuracy, a double is more accurate than a fioat, and so you can see that the fioat is accurate to about 7 decimal places in this example, while the double is accurate to 16. The only reason you'd ever need to use a fioat instead of a double is if you were calculating thousands of fioating point values, because a fioat is quicker to calculate than a double, at the expense of accuracy.

Finally, let's look at our lonely character data type:

```
#include <stdio.h>

int main() {
   char character = 'A';
   printf("character is: %c\n", character);
   printf("character is also decimal: %d\n", character);
   printf("character is also hexadecimal: %x\n", character);

   char string[] = "Hello, World!";
   printf("A string is just an array of characters, for example: %s\n", string);

   return(0);
}
```

So we have more format string specifiers here:

```
%c is for a character
%x is for hexadecimal
%s is for a string (it expects an array of characters that is terminated with a null
byte which is hexadecimal 0x00)
```

WARNING: If you are saving a single character into a char variable, you MUST use single quotes in C. Double quotes are for saving data into an array of characters. This is an easy mistake to make, so don't fall into this trap.

When we run this code, we see:

```
user@SANS:~/Desktop$ ./vars
character is: A
character is also decimal: 65
character is also hexadecimal: 41
A string is just an array of characters, for example: Hello, World!
user@SANS:~/Desktop$
```

From this, we can see that the character data type is actually just an integer. Have a look at an ASCII table, and you'll see that decimal 65 and hexadecimal 41 corresponds to the character 'A'. If this is unfamiliar you can jump back to ASCII earlier in the course and how data is stored!

Maths

This section is going to be a short one because mathematical operators are pretty much the same as in Python with just a couple of additions.

You can increment (add 1 to) an integer variable using the '++' operator.

You can also decrement (subtract 1 from) an integer variable using the '--' operator.

They are used like this:

```
#include <stdio.h>

int main() {
        int value = 42;

        value++;
    printf("Value is now: %d\n", value);

    value--;
        printf("Value is now: %d\n", value);
    return(0);
}
```

And the result:

```
user@SANS:~/Desktop$ ./maths
Value is now: 43
Value is now: 42
user@SANS:~/Desktop$
```

Functions

We already touched on some of the differences in function definitions in C, but let's look at them a bit more closely now.

The differences stem from C being a strictly typed language, so we need to specify what types of data we are passing into a function and what types of data we expect to get back from the function. Let's show this in code:

```
#include <stdio.h>

int addStuff(int valueA, int valueB) {
        return(valueA + valueB);
}

int main() {
        int result = addStuff(5, 10);
    printf("The answer is: %d\n", result);
}
```

First, we define a function called addStuff. It's going to take two integers and assign them to the variables valueA and valueB, which can be accessed from within the addStuff function. It's also going to return an integer. Within the function body, we just add valueA to valueB and return the result.

In the main function (remember this is the entry point of the program) we call the addStuff function which we already defined and pass in 5 and 10. The result is saved in the result variable and is then printed.

We aren't going to show you how to pass strings into functions just yet. In C you have to deal with pointers and memory allocation if you want to pass and return strings from functions. That is a whole other topic, which we will be covering in the next programming module.

Programming 6

Contents

In this module, we will be covering:

- Comments in C
- Conditionals in C
- Loops in C
- Arrays in C
- User Input in C
- Pointers & Memory in C
- Object-Oriented Programming in C

Comments

There are two types of code comments in C. The first is the double forward-slash `//`:

```
#include <stdio.h>

int main() {
   // This is a standard C-style code comment. The compiler will ignore it.
   puts("Hello, World!");
   return(0);
}
```

And the result, as expected:

```
user@SANS:~/Desktop$ ./comments
Hello, World!
user@SANS:~/Desktop$
```

The second type of comment is the multi-line comment `/* */`.

```
#include <stdio.h>

int main() {
      /* This is a multi-line comment. The comment continues until
   it sees the closing
      tag. */
   puts("Hello, World!");
      return(0);
}
```

Once again, the result:

```
user@SANS:~/Desktop$ ./comments
Hello, World!
user@SANS:~/Desktop$
```

Conditionals

Conditionals in C are almost exactly the same as in Python:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
        srand(time(NULL));
    int r = rand() % 20;

    printf("The value of r is: %d\n", r);

    if (r < 5) {
            puts("r is less than 5");
    } else if (r > 15) {
            puts("r is greater than 15");
    } else {
            puts("r is not less than 5 or greater than 15.");
    }
        return(0);
}
```

And the result:

```
user@SANS:~/Desktop$ ./conditionals
The value of r is: 2
r is less than 5
user@SANS:~/Desktop$ ./conditionals
The value of r is: 0
r is less than 5
user@SANS:~/Desktop$ ./conditionals
The value of r is: 16
r is greater than 15
user@SANS:~/Desktop$ ./conditionals
The value of r is: 13
r is not less than 5 or greater than 15.
```

Loops

Loops in C are more traditional than in Python. You'll see this pattern in most other programming languages. Let's take a look:

```
#include <stdio.h>

int main() {
   // While Loop
   puts("While Loop:");
   int a = 0;
   while (a <= 5) {
      printf("a is %d\n", a);
      a++;
   }

   // For Loop
   puts("\nFor Loop:");
   int x;
   for (x = 0; x <= 5; x++) {
      printf("x is %d\n", x);
   }

   // Nested Loop
   puts("\nNested Loop:");
   int i;
   int j;
   for (i = 0; i <= 5; i++) {
      for(j = 0; j <= 5; j++) {
         printf("When i is %d, j is %d\n", i, j);
      }
   }
   return(0);
}
```

The main difference here is the 'for' loop.

First of all, you need to initialise the variable outside of the loop. We did int x; before the loop to initialise x.

Then you need to construct the loop, so you give a starting value for x. In our example, x starts at 0. Then you provide the condition that would prevent the loop from running. In our example, as long as x is less than or equal to 5, the loop will continue to run. Finally, we tell the loop what happens after each iteration of the loop; in this case, x is incremented by 1.

Also, take a look at this nested loop in our example. It's already quite difficult to predict what the values of i and j will be at each stage of the loop. Can you imagine how much more difficult it would be if we nested it some more? This is why you should avoid using nested loops in your code if at all possible. Sometimes you just have to use them, though!

Here is the output. The nested loop ran too long to fit in the screenshot, but you get the idea:

```
user@SANS:~/Desktop$ ./loop
While Loop:
a is 0
a is 1
a is 2
a is 3
a is 4
a is 5

For Loop:
x is 0
x is 1
x is 2
x is 3
x is 4
x is 5

Nested Loop:
When i is 0, j is 0
When i is 0, j is 1
When i is 0, j is 2
When i is 0, j is 3
When i is 0, j is 4
When i is 0, j is 5
When i is 1, j is 0
When i is 1, j is 1
When i is 1, j is 2
When i is 1, j is 3
When i is 1, j is 4
When i is 1, j is 5
When i is 2, j is 0
When i is 2, j is 1
When i is 2, j is 2
When i is 2, j is 3
When i is 2, j is 4
When i is 2, j is 5
When i is 3, j is 0
When i is 3, j is 1
When i is 3, j is 2
When i is 3, j is 3
When i is 3, j is 4
When i is 3, j is 5
When i is 4, j is 0
When i is 4, j is 1
When i is 4, j is 2
When i is 4, j is 3
When i is 4, j is 4
When i is 4, j is 5
When i is 5, j is 0
When i is 5, j is 1
When i is 5, j is 2
When i is 5, j is 3
When i is 5, j is 4
When i is 5, j is 5
```

Arrays

Arrays in C are pretty easy; we've been using them to hold characters so far.

```
#include <stdio.h>

int main() {
        char array[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd'};

        puts(array);

        array[0] = 'A';
    array[1] = 'r';
        array[2] = 'r';
    array[3] = 'r';
        array[4] = 'r';

        puts(array);

        return(0);
}
```

You can access and modify data at array positions using the square bracket notation. Again, remember the first position in the array is at position 0 because we count from 0 onwards.

Here is our output:

```
user@SANS:~/Desktop$ ./arrays
Hello World
Arrrr World
user@SANS:~/Desktop$
```

I originally wanted to use this opportunity to make a pirate joke, but frankly, I couldn't be bothered to change each element of the array one by one beyond changing the first word.

There are some slight differences with arrays in C compared to Python. First of all, you can't append to an array in C as you can in Python. An array is always the same size as when you created it. You can access data at existing positions in the array, but you can't add more data or change the size of the array. If you need more space, your only option is to create a new array, copy the data from the old array into the new array and then add your new data using the square brackets notation.

The other difference is, of course, the type. In Python, you can store different types of data in a single array. In C, you have to specify the type of the array when you create it, and it can only ever hold that type of data. You can't mix and match within the same array.

User Input

Taking user input in C is the source of a great many vulnerabilities. It is also a great demonstration of how C doesn't hold your hand.

Here is an example of a simple C program which takes user inpu**WARNING: This is not a safe method of accepting user input, do not use it in practice. This is for the purposes of demonstration only.**

```
#include <stdio.h>

int main() {
        char data[20];
    puts("Enter some data here: ");
        scanf("%s", &data);
    printf("You entered: \n%s\n", data);
        return(0);
}
```

If you look at this code you can see we've defined an area of memory on the stack which can contain 20 bytes of data. We then get some input from the user using the 'scanf()' function. The scanf function takes two parameters, the format of the data represented in our example by '%s' which indicates the input will be a string and a pointer to the memory address where the data will be stored. Then we print what the user typed.

```
mainuser@ubuntu:$ ./program
Enter some data here:
hello
You entered:
hello
mainuser@ubuntu:$
```

There is one major problem with this code. Did you spot it? At no point did we check that what the user types will fit within 20 bytes. C will just put the data in that memory address even if there isn't enough space. This will cause the program to crash and in the worst case introduces an exploit known as a buffer overflow. You'll learn how to exploit it later in the course, but for now let's see how the program can crash as a result of our data overwriting adjacent memory addresses.

```
mainuser@ubuntu:$ ./program
Enter some data here:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
You entered:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
mainuser@ubuntu:$
```

Let's fix this problem by using the 'fgets' function instead.

```
#include <stdio.h>

int main() {
   char data[20];
   puts("Enter some data here: ");
   fgets(data, sizeof data, stdin);
   printf("You entered: \n%s", data);
   return(0);
}
```

Here the 'fgets' function takes the variable to store the data in, the maximum number of bytes to read (in our case we did the size of the data variable), and the place to read the input from which in this case is standard input. Also notice a slight change in the printf function. We removed the trailing '\n' because the fgets() function automatically adds a newline.

The result is:

```
mainuser@ubuntu:$ ./program
Enter some data here:
hello
You entered:
hello
mainuser@ubuntu:$
```

And if we try to break it we get:

```
mainuser@ubuntu:$ ./program
Enter some data here:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
You entered:
AAAAAAAAAAAAAAAAAAmainuser@ubuntu:$
```

But notice there is now another problem here. When we enter too much data the program doesn't crash anymore but the newline character at the end has been cut off. We should add it back in for consistency.

```
#include <stdio.h>
#include <string.h>

int main() {
   char data[20];
   puts("Enter some data here: ");
   fgets(data, sizeof data, stdin);
```

```
if (data[strlen(data) - 1] != '\n') {
        data[strlen(data) - 1] = '\n';
            }
   }
      printf("You entered: \n%s", data);
   return(0);
}
```

So here we are checking first that the user has entered some data at all. Then we go on to check if the second to last byte in the memory address is a '\n' newline. This is because the last byte will be a null byte to signify the end of the string. So the second to last byte should be the newline. If it isn't, then we add it in.

You may have noticed that if we don't max out the input then we will still be adding a newline character to the second to last byte of the memory space, but it doesn't matter because it comes after the null byte so it will never be reached. When you submit a smaller amount of data using fgets then the function will add a newline and a null byte. Once the null byte is in the data, that is the end of the string and the rest of the memory allocated is never accessed.

```
mainuser@ubuntu:$ ./program
Enter some data here:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
You entered:
AAAAAAAAAAAAAAAAAA
mainuser@ubuntu:$
```

So you can see how much we have to account for in just a simple C program to read input from the user and how easy it is to go wrong. You can imagine the difficulty with more complex programs which also have to read data from configuration files, etc.

Arguments

In C we can also read in arguments from the user over the command line.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
      int i;
   for (i = 0; i < argc; i++) {
            printf("Arg %d: %s\n", i, argv[i]);
   }
      return(0);
}
```

Here we can see the main function has been changed to add two parameters. We have 'argc' and 'argv'. The 'argc' variable is an integer which contains the number of arguments passed to the program when it was executed. The 'argv' array contains the data passed in.

The first element of the array is the program name, and after that are the arguments the user typed.

```
mainuser@ubuntu:$ ./program Hello
Arg 0: ./program
Arg 1: Hello
mainuser@ubuntu:$
```

Files

When it comes to reading files, it's almost identical to reading from the command line over stdin. The reason is, stdin is actually treated as a file when the program runs.

```
#include <stdio.h>
#include <string.h>

int main() {
        char data[20];
    FILE *file = fopen("test.txt", "r");

    fgets(data, sizeof data, file);
        if (strlen(data) > 0){
        if (data[strlen(data) - 1] != '\n') {
                        data[strlen(data) - 1] = '\n';
        }
        }
    printf("The file contains: \n%s", data);
                fclose(file);
    return(0);
}
```

There are only three changes here. First we open a file handle to the file we wish to read using 'fopen'. The 'fopen' function takes the path to the file and a character which indicates what permissions to open the file with. In this case we are opening 'test.txt' in the same directory as our executable with read permissions.

Once we have the file handle, we use fgets to read from it as we did before with 'stdin'. In fact 'stdin' is treated as a file, so 'fgets' can be used to read from files as well as from the command line. Finally, since we opened a file handle, we have to close it when we are done with it with 'fclose'.

The result is:

```
mainuser@ubuntu:$ echo "test file" > test.txt
mainuser@ubuntu:$ ./program
The file contains:
test file
mainuser@ubuntu:$
```

Pointers and Memory

Many people find pointers in C to be troublesome, but generally, those people are self-taught programmers with no formal training in computer science. The reason they find it difficult is because they don't know how computer memory works. We do know how computer memory works so this subject isn't going to be nearly as tricky as you imagine.

There are two areas of computer memory; the stack and the heap.

The stack is a very structured and orderly section of memory. When you launch a program, the instructions for that function are loaded onto the stack, and then each function is assigned an area of memory called a 'stack frame'. The stack frame contains the local variables that are used by the function.The heap is much simpler. It's an unstructured area of memory that can be used to store data. The heap is somewhat slower to access than the stack, but the benefit is you can store whatever you like on there without knowing beforehand what the size of the data will be. By contrast, the stack is very structured, and you need to know how much data to reserve on the stack when you write the program. Stacks and Heaps are discussed in more detail in a later section, Advance Computer Hardware.

Normally when a variable is declared, that memory will exist on the stack. The downside to using stack memory is that the program needs to know exactly how much memory to allocate at the time the program is compiled. Say you wanted to allow a user to enter their name, if you were to store that information on the stack you would need to have a maximum number of characters the user could enter. On the other hand, memory on the heap can be allocated dynamically during runtime, so if you were to store the user's name on the heap you could allow the user to type any number of characters.

Pointers

A pointer is just a memory address that points to the contents of a variable. Think about what happens when you save a variable in your program. The program asks the kernel for some space in memory, the kernel provides a memory address, and then the data is stored into memory at that location. The variable references that memory address so it contains the data at that memory address.

It's easier to explain through code:

```
#include <stdio.h>

int main() {
   int stuff = 42;
   printf("The stuff variable contains %d and is saved in memory at address: %p\n",
stuff, &stuff);
   return(0);
}
```

First we declare a variable called 'stuff' and we assign it to the data '42'. Then we are printing both the contents and the memory address of the variable. The '&stuff' means 'address-of stuff'.

And the result is:

```
user@SANS:~/Desktop$ ./pointers
The stuff variable contains 42 and is saved in memory at address: 0x7ffcf3fbd264
user@SANS:~/Desktop$
```

When a variable is declared in code as we have done with the 'stuff' variable, the variable will be placed on the stack (remember, RAM has two sections, the stack and the heap with the stack being structured and the heap unstructured). Remember, each function in a program has its own stack frame, so normally these kinds of variables cannot be accessed from outside of the function they were declared in. As an example:

```
#include <stdio.h>

void doStuff(int stuff) {
   stuff = 1337;
   printf("Within the doStuff() function, the variable stuff has the value: %d\n\n",
stuff);
}

int main() {
   int stuff = 42;
   printf("Within the main function, before doStuff is called, the variable stuff has
the value: %d\n\n", stuff);
   doStuff(stuff);
   printf("Within the main function, after doStuff is called, the variable stuff has
the value %d\n\n", stuff);
   return(0);
}
```

When it runs, stuff is 42. Then it passed 'stuff' into the doStuff() function, but what is passed is a copy of the data at the memory address of the stuff variable. So a new memory address is created in the doStuff() stack frame with 42 in it, which the doStuff() function then modifies. That is why after doStuff() runs, the stuff variable in the main() function is still 42.

```
user@SANS:~/Desktop$ ./pointers
Within the main function, before doStuff is called, the                variable stuff has the value:
42

Within the doStuff() function, the variable stuff has the value: 1337

Within the main function, after doStuff is called, the variable stuff has the value 42

user@SANS:~/Desktop$
```

So, by passing variables to other functions, what we are actually passing is just a copy of the data at the time the function was called. We aren't using the same memory address in both functions.

Let's modify our code to use pointers now:

```
#include <stdio.h>

void doStuff(int * stuff) {
   *stuff = 1337;
      printf("Within the doStuff() function, the variable stuff has the value: %d\n\n",
*stuff);
}

int main() {
   int stuff = 42;
      printf("Within the main function, before doStuff is called, the variable stuff has
the value: %d\n\n", stuff);
      doStuff(&stuff);
   printf("Within the main function, after doStuff is called, the variable stuff has
the value %d\n\n", stuff);
   return(0);
}
```

This code looks pretty similar, right? It's almost the same with a few key differences.

```
   doStuff(int * stuff): This means the doStuff function expects a pointer to an
integer value. A pointer is a memory address, so it expects a memory address that is
storing an integer.
   *stuff = 1337;: This means the value at the memory address 'stuff' is 1337. It's a
way of accessing data at a pointer.
   doStuff(&stuff);: This is calling the doStuff function, and passing in the memory
address of the stuff variable in the main function.
```

Now look at what happens when our program runs:

```
user@SANS:~/Desktop$ ./pointers
Within the main function, before doStuff is called, the                variable stuff has the value:
42

Within the doStuff() function, the variable stuff has the value: 1337

Within the main function, after doStuff is called, the variable stuff has the value
1337

user@SANS:~/Desktop$
```

Notice how the 'stuff' variable is changed in the main function after it is changed in the doStuff() function? That is because the doStuff() function modified the same piece of

computer memory that the stuff variable in the main() function was referencing. The doStuff() function was no longer working on a copy of the data in a new memory address; it was working on the original memory address instead.

Don't use pointers where you don't need to use them, though. The code above is just an example to illustrate how pointers work, but it isn't how pointers should be used. It would be far quicker to write the same program without using pointers at all like so:

```
#include <stdio.h>

int doStuff() {
        return(1337);
}

int main() {
        int stuff = 42;
    printf("Within the main function, before doStuff is called, the variable stuff has
the value: %d\n\n", stuff);
    stuff = doStuff();
        printf("Within the main function, after doStuff is called, the variable stuff has
the value %d\n\n", stuff);
        return(0);
}
```

The effect is the same, but we didn't need to use pointers at all.

```
user@SANS:~/Desktop$ ./pointers
Within the main function, before doStuff is called, the  variable stuff has the value:
42

Within the doStuff() function, the variable stuff has the value: 1337

user@SANS:~/Desktop$
```

Malloc

Remember we said that variables declared in code are saved on the stack? Well, we can also put things on the heap if we need to. The benefit to using the heap is that we don't need to know how much data is going to be saved there when the program compiles.

Let's see an example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main() {
   // This program generates a random number between 0 and 19.
```

```
// that random number of characters.

    // Generate a random number between 0 and 19
  srand (time(NULL));
    int r = rand() % 20;
  printf("Going to save %d 'A's into memory this time!\n", r);

  // Create space for 'chars' number of characters + 1 byte on the heap
    // The extra byte is for the string line terminator which is a 0x00 or a null byte
  char *string = malloc((sizeof(char) * r) + 1);

  // For loop to add 'r' number of A's to the address at the string variable on the
heap
  int i;
    for (i = 0; i < r; i++) {
    // The strcat function concatenates two strings together.
        // The first parameter is the destination where the result will go
    // The second parameter is the string to add to the destination.
        strcat(string, "A");
  }

  // Print the result to prove it worked.
    puts(string);
  free(string);
    return(0);
}
```

We've commented the code quite heavily this time because there are things in here that you haven't seen before. Every time this program runs, it is going to generate a random number between 0 and 19. It is then going to save that number of 'A's into memory and print it.

The problem is, we don't know what the random number is going to be when the program compiles, since it changes every time the program runs. In this case, we are using malloc() to dynamically allocate memory at program execution instead of at compile time.

Now, this next bit is really important. Whenever you use malloc to create memory on the heap, you have to be responsible for clearing that memory when you no longer need it. That is what free() does. If you fail to call free on memory you allocate, your program will have a 'memory leak', where the memory it uses keeps going up until eventually it runs out of memory and crashes.

Let's see it in action:

```
user@SANS:~/Desktop$ ./malloc
Going to save 6 'A's into memory this time!
AAAAAA
user@SANS:~/Desktop$ ./malloc
Going to save 5 'A's into memory this time!
AAAAA
user@SANS:~/Desktop$ ./malloc
Going to save 15 'A's into memory this time!
AAAAAAAAAAAAAAA
user@SANS:~/Desktop$ ./malloc
Going to save 0 'A's into memory this time!
```

```
user@SANS:~/Desktop$ ./malloc
Going to save 0 'A's into memory this time!
Going to save 15 'A's into memory this time!
AAAAAAAAAAAAAAA
Going to save 13 'A's into memory this time!
AAAAAAAAAAAAA
user@SANS:~/Desktop$
```

Remember in the programming-5 module when we said returning a string in C involves manipulating memory? Well, here is an example showing how you would return a string.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

char *getStr(int chars) {
   // Create space for 'chars' number of characters + 1 byte on the heap
      // The extra byte is for the string line terminator which is a 0x00 or a null byte
   char *string = malloc((sizeof(char) * chars) + 1);
      // For loop to add 'chars' number of A's to the address at the string variable on
the heap
      int i;
   for (i = 0; i < chars; i++) {
            strcat(string, "A");
   }
      return(string);
}

int main() {
      // This program generates a random number between 0 and 19.
   // Then it creates a character array in memory on the heap big enough for
      // That random number of characters.

      // Generate a random number between 0 and 19
   srand (time(NULL));
      int r = rand() % 20;
   printf("Going to save %d 'A's into memory this time!\n", r);

   char *res = getStr(r);

   puts(res);
      free(res);
   return(0);
}
```

There is actually another benefit to using the heap. Data on the heap exists until free() is called on it, while data on the stack only exists before that function returns.

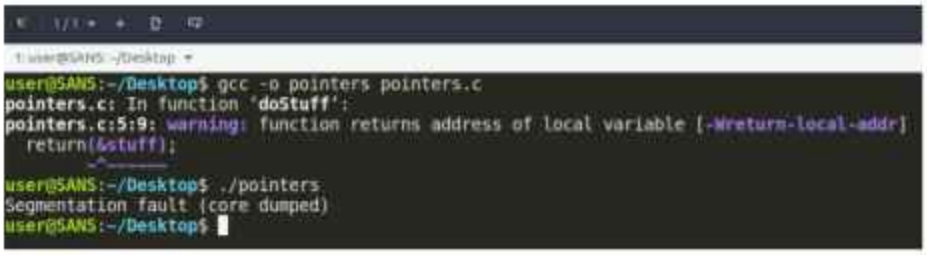
Let's illustrate:

```
#include <stdio.h>

int *doStuff() {
```

```
return(&stuff);
}

int main() {
        int *stuff = doStuff();
    printf("The stuff variable contains: %d and is at memory address: %p", *stuff,
stuff);
    return(0);
}
```

This code is not going to work properly because the stuff variable in doStuff() only exists while the doStuff() function hasn't returned yet, so it's returning a pointer to a memory address that doesn't exist after it returns.

The compiler is kind enough to warn us about it, but it compiles the program anyway, and bad stuff happens:



To solve this, you could do it one of two ways.

You could create a variable in the main function and pass a pointer to the variable into the doStuff function, which will then modify the data at that memory address and then return. This would work because the variable was created in the main() function, which hasn't returned yet.

You could allocate memory on the heap, and return a pointer to that memory address. Memory on the heap exists until you call free() on it, no matter where it was created, so even if you allocate the memory inside the doStuff() function it will still exist even when the doStuff() function returns.

Object Oriented Programming in C

After the last few sections, you can finally let your brain cool off because there is no such thing as object oriented programming in C. C actually predates the whole concept of object oriented programming. That is the reason C++ exists. C++ is basically just a patch on top of C to make it work with the OOP paradigm. We aren't going to teach you any C++ in this course, so that's all there is to this chapter.

Programming: In Practice

James Lyne shares some real-world examples of how a good knowledge of programming is invaluable in cyber security work.

Introduction to SQL

An Introduction to SQL

What is a database?

A database is very simply a store of data. There are two kinds of databases, relational databases and non-relational databases. We will cover each in more depth later in this module. However for now you can simply say a relational database is an organised store of data. The data is stored in a structured way, in tables with data types and fixed maximum lengths.

What is SQL

SQL or Structured Query Language (pronounced sequel or ess-que-ell) is a language designed for managing relational databases. In a way you can think of it as a kind of programming language for accessing relational databases.

There are many kinds of SQL database engines produced by a multitude of vendors and while they all utilise SQL for database management, each database engine speaks a slightly different dialect of SQL. The result is that not all SQL works on every SQL database. Mostly if you learn SQL for one database engine, you can easily write SQL for any other if you simply learn the minor differences. The core of SQL remains the same.

Database Management Systems

A database management system (DBMS) is the software that (as the name would suggest) manages databases. It sits between the user and the database and the user interacts with the DBMS using SQL. The DBMS is responsible for managing access to the databases, and executing SQL queries. Often the terms DBMS and database are used interchangeably.

SQLite vs MySQL vs MariaDB

SQLite is a lightweight DBMS that stores data in a single file. It is often used in embedded systems where a small database is required but where processing power is limited.

MySQL is one of the most well known and most used open source database management systems. It is designed as a production DBMS to run at scale.

In 2010 Oracle acquired MySQL. This event was quite controversial in the open source community and so MySQL was forked into MariaDB. MariaDB is intended to be a drop-in replacement for MySQL, indeed even the MariaDB command line client is still called `mysql`.

In this course we will be using MariaDB for all of our SQL examples, however we will use the terms MariaDB and MySQL interchangeably given the nature of MariaDB.

Relational vs Non-Relational Databases

Relational Databases

We talked briefiy about how relational databases are a structured store of data. In a relational database, data is stored in tables which can hold fixed types of data of a fixed maximum size.

The name relational comes from the fact that tables in a relational database can be linked together or related to each other. This comes in the form of primary keys and foreign keys. Each table in a relational database**must** have a primary key. A primary key is a value in a row that is unique to that row. At its simplest this could be a numerical and incrementing ID for each row.

The primary keys of data in one table can be stored in another table as a foreign key. This would essentially create a relationship between the two tables.

As an example here are two tables:

customers:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom

9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

orders:

orderId	date	currency	total	customerId
1	2020-11-14	$	111	6
2	2020-07-07	£	958	4
3	2021-02-18	£	721	2
4	2020-05-25	$	834	4
5	2020-07-10	£	47	1
6	2021-02-27	£	587	4
7	2021-03-04	£	198	10
8	2020-09-03	$	200	3
9	2020-11-17	£	726	3
10	2020-12-29	$	200	5

Going by these two examples the primary key for the customers table is the customerId column. The primary key for the orders table is the orderId column.

In the orders table, customerId is the foreign key that relates orders to customers. In this case we would say an order has a single customer and a customer can have many orders.

Non-relational databases

Not all databases are relational. Non-relational databases are designed to hold unstructured data where the designer may not know exactly what kind of data needs to be stored or if the type of data that needs to be stored is constantly changing. As the name suggests there are no relationships between data in non-relational databases.

Non-relational databases are sometimes called NoSQL databases because you do not use SQL to query them. One of the more common types of NoSQL database is the document data store. In this case a document would be associated with a key for example:

```
Key: 3
Document:
{
   "firstName": "Orli",
      "lastName": "Klein",
   "address": "4981 Gravida St.",
      "city": "Barrow-in-Furness",
   "country": "United Kingdom",
      "orders": [{
      "date": "2020-09-03",
            "currency": "$",
      "total": 200,
      },
   {
            "date": "2020-11-17",
      "currency": "£",
            "total": 726,
   }]
}
```

This is a somewhat poor example because this kind of data is far more suited to a relational database. A good example of the type of data which would be well suited to NoSQL databases is log messages. Log messages vary from system to system, with varying types and sizes of messages, and there are times when new log sources need to be added quickly.

Which one is better?

Both types of databases have their uses, in fact it is not uncommon for applications to use both types of databases at once. It is just a case of picking the right tool for the job. You should know enough about their differences to be able to tell which is better for your use case.

In this module we will be looking strictly at relational databases going forward.

Installing MariaDB

Now we will cover a simple installation and configuration for MariaDB on an Ubuntu Linux system.

Open the command line to start.

As usual we should make sure our package manager is up to date before we proceed with the installation:

```
sudo apt update
sudo apt install mariadb-server
```

Then we will configure the DBMS:

```
sudo mysql_secure_installation
```

You will be prompted to enter a password for the root user and then you will be prompted to answer yes or no to several options - enter y at all the prompts. This is a secure default configuration for MariaDB.

Connecting to MariaDB

Now that we have MariaDB installed we need to create a new user for ourselves.

```
sudo mysql -u root -p
```

You should be greeted with a prompt:

```
MariaDB [(none)]>
```

Enter the following SQL statements replacing the <password> with your own password:

```
CREATE USER 'foundations'@'localhost' IDENTIFIED BY '<password>';
GRANT ALL PRIVILEGES ON *.* TO 'foundations'@'localhost';
FLUSH PRIVILEGES;
exit
```

NOTE: The syntax is critical. Special characters, such as semi-colons and apostrophes, have specific meaning in SQL. The SQL commands are not case sensitive, they are capitalized for readability purposes.

With our new user setup (called foundations) we can now login to MariaDB without using sudo:

```
mysql -u foundations -p
```

If you've completed the setup correctly you should be once again greeted with the prompt:

```
MariaDB [(none)]>
```

MySQL Basic Statements

Introduction to MySQL

In this module we will cover the following SQL commands to help you build your first SQL statements:

- SELECT
- ORDER BY
- WHERE
- DISTINCT
- ALIAS

Using a database

When you first connect to MySQL you will by default have no database selected. A DBMS can have multiple databases within it, so before you can run any queries on any tables you must pick your database.

You can do this with `USE`:

```
USE databasename;
```

Once you have done that your prompt should change from:

```
MariaDB [none]>
```

to

```
MariaDB [databasename]>
```

Structure of a SQL statement

SQL has a syntax much like programming languages do.

Without knowing what this simple statement does we can observe some requirements and best practices:

```
SELECT orderId, currency FROM orders;
```

The most important thing to note here is the ; at the end of the statement. If you forget the semi-colon then the statement will not be executed because SQL will assume there is more to come. The semi-colon is the character that indicates the statement has ended.

The next thing we can note is that SQL commands are capitalised. Although this is not a strict requirement it makes the statement much more readable.

While the SQL commands are **not** case sensitive, the names of columns and tables **are** case sensitive. Therefore if we were to write:

```
SELECT orderId, currency FROM ORDERS;
```

This would be invalid unless the table was called `ORDERS` rather than `orders`.

We can see also that where a command accepts multiple parameters when they are separated by a comma. A space after the comma is not strictly necessary but is recommended for readability.

This is the structure of a very basic SQL statement.

SELECT and FROM

The SELECT statement is used to retrieve data from one or more tables in a database. The FROM command identifies which table to retrieve the data from.

The most basic syntax for a SELECT statement is:

```
SELECT comma, separated, fields
FROM tablename;
```

Take this customers table as an example:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

If we wanted to get the `firstName` and `lastName` of every customer in the table we could do:

```
SELECT firstName, lastName FROM customers;
```

The result would be:

firstName	lastName
Ursa	Vasquez
Quyn	Meyer
Orli	Klein
Tallulah	Hines
Joel	Ross
Charlotte	Ramos
Dennis	Avery
Igor	Malone
Connor	Witt
Karen	Marquez

Select ALL

If we wanted to extract all the data from a table without having to specify the fields we could use the wildcard operator. In SQL the wildcard operator is the asterisk (`*`). To retrieve all the data from the `customers` table we would do:

```
SELECT * FROM customers;
```

Producing:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416		

Nullam St.	Worcester	United States			
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

ORDER BY

When you use the SELECT command the data returned is not ordered. Or more accurately it is ordered however the SQL engine orders data by default. This can change between SQL engines and also between different versions of SQL engines so it is important to never rely on the default sort order of data returned by the SELECT command.

This is where the ORDER BY clause comes in. ORDER BY is used to sort data in either ascending or descending order based on one or more fields.

Here is the syntax for the ORDER BY clause:

```
SELECT fields FROM tablename ORDER BY field DESC
```

or:

```
SELECT fields FROM tablename ORDER BY field ASC
```

depending on if you wish to sort the data in descending or ascending order.

Using the following customers table as an example:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States

Molestie Avenue	Matlock	United Kingdom			
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

Let's retrieve the customer first and last names in ascending order of last names.

```
SELECT firstName, lastName FROM `customers` ORDER BY lastName ASC;
```

The result is:

firstName	lastName
Dennis	Avery
Tallulah	Hines
Orli	Klein
Igor	Malone
Karen	Marquez
Quyn	Meyer
Charlotte	Ramos
Joel	Ross
Ursa	Vasquez
Connor	Witt

Notice the last names are now sorted in alphabetical order.

WHERE

So far when we have been retrieving data from a table we have been retrieving every row if not always every field. The `WHERE` clause is where we start to change that. The `WHERE` clause helps us to filter records to only the ones that match a certain condition.

Here is the syntax for using the `WHERE` clause:

```
SELECT fields FROM tablename WHERE condition;
```

Take this example `orders` table:

orderId	date	currency	total	customerId
1	2020-11-14	$	111	6
2	2020-07-07	£	958	4
3	2021-02-18	£	721	2
4	2020-05-25	$	834	4
5	2020-07-10	£	47	1
6	2021-02-27	£	587	4
7	2021-03-04	£	198	10
8	2020-09-03	$	200	3
9	2020-11-17	£	726	3
10	2020-12-29	$	200	5

If we wanted to retrieve only the orders which were fulfilled in US Dollars ($) we could write a query like so:

```
SELECT * FROM orders WHERE currency = "$";
```

This would result in:

orderId	date	currency	total	customerId

1	2020-11-14	$	111	6
4	2020-05-25	$	834	4
8	2020-09-03	$	200	3
10	2020-12-29	$	200	5

Going back to the previous section we could include an `ORDER BY` to sort the table into ascending order by the amount paid:

```
SELECT * FROM orders WHERE currency = "$" ORDER BY total ASC;
```

Producing:

orderId	date	currency	total	customerId
1	2020-11-14	$	111	6
8	2020-09-03	$	200	3
10	2020-12-29	$	200	5
4	2020-05-25	$	834	4

DISTINCT

Sometimes we need to return rows of data without duplicates in them. This is where the `DISTINCT` clause comes in.

Take this `customers` table as an example:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

If you were wanting to return all the countries your customers are based in you might try something like this:

```
SELECT country FROM customers;
```

However that would get you this result:

country
United States
Canada
United Kingdom
United States
United Kingdom
United States
United Kingdom
United Kingdom
United Kingdom
Canada

The duplicates occur because every row from the country column has been selected.

If we would like a neat table with only unique countries listed, we must use the `DISTINCT` clause like so:

```
SELECT DISTINCT country FROM customers;
```

Resulting in:

country
United States
Canada
United Kingdom

AS

The AS keyword is used to assign an alias to a table or field. An alias is simply a temporary name which only exists for the duration of the SQL query as it is executed. Primarily they are used to make long SQL statements more readable, more descriptive or set the table headings for a query to make the results more descriptive.

Aliases for Columns

The syntax for an alias by column is:

```
SELECT field AS aliasname FROM tablename;
```

Let's use the customers table for our example:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom

10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

We can write some SQL to return the customers first and last names combined and alias that as fullname:

```
SELECT customerId, CONCAT_WS(' ', firstName, lastName) AS fullName FROM customers;
```

Which provides us with:

customerId	fullName
1	Ursa Vasquez
2	Quyn Meyer
3	Orli Klein
4	Tallulah Hines
5	Joel Ross
6	Charlotte Ramos
7	Dennis Avery
8	Igor Malone
9	Connor Witt
10	Karen Marquez

Don't worry too much about `CONCAT_WS` for now although if you are interested this is a function that can concatenate two fields together with a separator of our choosing (in this case we chose a space as the separator). The `WS` in `CONCAT_WS` actually stands for with separator.

This is an example of where you would use an alias.

Aliases for Tables

The other type of aliases you can create are aliases for tables. Let's say we want to get a list of customers and their respective orders. We are now working with both tables at once for the first time ever.

`customers`:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

`orders`:

orderId	date	currency	total	customerId
1	2020-11-14	$	111	6
2	2020-07-07	£	958	4
3	2021-02-18	£	721	2
4	2020-05-25	$	834	4
5	2020-07-10	£	47	1
6	2021-02-27	£	587	4

7	2021-03-04	£	198	10
8	2020-09-03	$	200	3
9	2020-11-17	£	726	3
10	2020-12-29	$	200	5

We could write a query like so:

```
SELECT orders.orderId, customers.firstName, customers.lastName, orders.currency,
orders.total
FROM orders, customers
WHERE orders.customerId = customers.customerId;
```

We end up with:

orderId	firstName	lastName	currency	total
1	Charlotte	Ramos	$	111
2	Tallulah	Hines	£	958
3	Quyn	Meyer	£	721
4	Tallulah	Hines	$	834
5	Ursa	Vasquez	£	47
6	Tallulah	Hines	£	587
7	Karen	Marquez	£	198
8	Orli	Klein	$	200
9	Orli	Klein	£	726
10	Joel	Ross	$	200

Notice how we linked the two tables based on the `customerId` field? This is the power of a relational database. Since every order has a customer we can look up the name in the customers table based on the `customerId` stored in the orders table.

This query is kind of long though right? We can use aliases to shorten it a little bit:

```
SELECT o.orderId, c.firstName, c.lastName, o.currency, o.total
FROM orders AS o, customers AS c
WHERE o.customerId = c.customerId;
```

Notice here how we aliased the `orders` table to `o` and the `customers` table to `c`?

MySQL Joins

JOIN

Joins are one of the topics people tend to struggle with in SQL but they are actually quite simple. As the name suggests a `JOIN` clause combines (or you might say joins) results from two or more tables.

The complexity comes with the different types of joins and exactly how and what they do. We'll get into that in the next sections of this module but for now let's look at a basic `JOIN`.

Once again we're going to need both example tables for this:

`customers`:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

orders:

orderId	date	currency	total	customerId
1	2020-11-14	$	111	6
2	2020-07-07	£	958	4
3	2021-02-18	£	721	2
4	2020-05-25	$	834	4
5	2020-07-10	£	47	1
6	2021-02-27	£	587	4
7	2021-03-04	£	198	10
8	2020-09-03	$	200	3
9	2020-11-17	£	726	3
10	2020-12-29	$	200	5

If you recall we previously linked both tables based on their `customerId` like so:

```
SELECT orders.orderId, customers.firstName, customers.lastName, orders.currency,
orders.total
FROM orders, customers
WHERE orders.customerId = customers.customerId;
```

Which produced:

orderId	firstName	lastName	currency	total
1	Charlotte	Ramos	$	111
2	Tallulah	Hines	£	958
3	Quyn	Meyer	£	721
4	Tallulah	Hines	$	834
5	Ursa	Vasquez	£	47

7	Karen	Marquez	£	198
8	Orli	Klein	$	200
9	Orli	Klein	£	726
10	Joel	Ross	$	200

We can look a bit more at how this works by removing the WHERE clause:

```
SELECT orders.orderId, customers.firstName, customers.lastName, orders.currency,
orders.total
FROM orders, customers;
```

The partial result is:

orderId	firstName	lastName	currency	total
1	Ursa	Vasquez	$	111
2	Ursa	Vasquez	£	958
3	Ursa	Vasquez	£	721
4	Ursa	Vasquez	$	834
5	Ursa	Vasquez	£	47
6	Ursa	Vasquez	£	587
7	Ursa	Vasquez	£	198
8	Ursa	Vasquez	$	200
9	Ursa	Vasquez	£	726
10	Ursa	Vasquez	$	200
1	Quyn	Meyer	$	111
2	Quyn	Meyer	£	958
3	Quyn	Meyer	£	721
4	Quyn	Meyer	$	834

5	Quyn	Meyer	£	47
6	Quyn	Meyer	£	587
7	Quyn	Meyer	£	198
8	Quyn	Meyer	$	200
9	Quyn	Meyer	£	726
10	Quyn	Meyer	$	200

...

Note: We have not printed the entire table of results, however every customer has an entry for every orderId.

Notice how for every `orderId` we have every combination of customer from the `customers` table. This gives us 10 * 10 = 100 results.

Only one of those results is accurate however. We only want the results where the `customerId` in the `orders` table matches the `customerId` in the `customers` table. That is the genuine customer that belongs to that order.

```
SELECT orders.orderId, customers.firstName, customers.lastName, orders.currency,
orders.total
FROM orders, customers
WHERE orders.customerId = customers.customerId;
```

orderId	firstName	lastName	currency	total
1	Charlotte	Ramos	$	111
2	Tallulah	Hines	£	958
3	Quyn	Meyer	£	721
4	Tallulah	Hines	$	834
5	Ursa	Vasquez	£	47
6	Tallulah	Hines	£	587
7	Karen	Marquez	£	198
8	Orli	Klein	$	200

9	Orli	Klein	£	726
10	Joel	Ross	$	200

When we match on the `customerId` you can see that we are left with an accurate list of which customer is related to which order. This is similar to how `JOIN` works without actually using a `JOIN`. Let's now produce the same result with a `JOIN` clause:

```
SELECT orders.orderId, customers.firstName, customers.lastName, orders.currency,
orders.total
FROM orders
INNER JOIN customers ON orders.customerId = customers.customerId;
```

Producing:

orderId	firstName	lastName	currency	total
1	Charlotte	Ramos	$	111
2	Tallulah	Hines	£	958
3	Quyn	Meyer	£	721
4	Tallulah	Hines	$	834
5	Ursa	Vasquez	£	47
6	Tallulah	Hines	£	587
7	Karen	Marquez	£	198
8	Orli	Klein	$	200
9	Orli	Klein	£	726
10	Joel	Ross	$	200

```
SELECT orders.orderId, customers.firstName, customers.lastName, orders.currency,
orders.total
FROM orders
INNER JOIN customers ON orders.customerId = customers.customerId;
```

There are a few important points here. Notice first that in the `FROM` clause we only mention one of the tables. The second table to be joined is mentioned in the `INNER`

`JOIN` clause. Then after the `ON` clause we specify the condition for the join. In this case we want all entries where the `customerId` in the `orders` table matches the `customerId` in the `customers` table.

This is an example of a `JOIN`. However remember there are many different types of joins with slightly different behaviour for each which we will cover in this module.

INNER JOIN

We saw an example of an `INNER JOIN` in the previous section. For a quick refresher here are the two tables:

`customers`:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

`orders`:

orderId	date	currency	total	customerId
1	2020-11-14	$	111	6

3	2021-02-18	£	721	2
4	2020-05-25	$	834	4
5	2020-07-10	£	47	1
6	2021-02-27	£	587	4
7	2021-03-04	£	198	10
8	2020-09-03	$	200	3
9	2020-11-17	£	726	3
10	2020-12-29	$	200	5

And the query:

```
SELECT orders.orderId, customers.firstName, customers.lastName, orders.currency,
orders.total
FROM orders
INNER JOIN customers ON orders.customerId = customers.customerId;
```

Which produces:

orderId	firstName	lastName	currency	total
1	Charlotte	Ramos	$	111
2	Tallulah	Hines	£	958
3	Quyn	Meyer	£	721
4	Tallulah	Hines	$	834
5	Ursa	Vasquez	£	47
6	Tallulah	Hines	£	587
7	Karen	Marquez	£	198
8	Orli	Klein	$	200
9	Orli	Klein	£	726
10	Joel	Ross	$	200

An `INNER JOIN` is used for retrieving rows that meet a condition in **both** tables.

Let's try a different query using the same tables:

```
SELECT customers.customerId, customers.firstName, customers.lastName, orders.orderId,
orders.currency, orders.total
FROM customers
INNER JOIN orders ON customers.customerId = orders.customerId;
```

Notice how not every customer has an order:

customerId	firstName	lastName	orderId	currency	total
6	Charlotte	Ramos	1	$	111
4	Tallulah	Hines	2	£	958
2	Quyn	Meyer	3	£	721
4	Tallulah	Hines	4	$	834
1	Ursa	Vasquez	5	£	47
4	Tallulah	Hines	6	£	587
10	Karen	Marquez	7	£	198
3	Orli	Klein	8	$	200
3	Orli	Klein	9	£	726
5	Joel	Ross	10	$	200

An `INNER JOIN` requires the condition to be true for both tables, in other words any customer that doesn't have a `customerId` in the `orders` table isn't represented in the results at all.

LEFT JOIN

A `LEFT JOIN` returns all the results from the first table and matching results from the second table.

To demonstrate we'll use these example tables once again:

`customers`:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

`orders`:

orderId	date	currency	total	customerId

2	2020-07-07	£	958	4
3	2021-02-18	£	721	2
4	2020-05-25	$	834	4
5	2020-07-10	£	47	1
6	2021-02-27	£	587	4
7	2021-03-04	£	198	10
8	2020-09-03	$	200	3
9	2020-11-17	£	726	3
10	2020-12-29	$	200	5

And let's look at our last query from the previous section again:

```
SELECT customers.customerId, customers.firstName, customers.lastName, orders.orderId,
orders.currency, orders.total
FROM customers
INNER JOIN orders ON customers.customerId = orders.customerId;
```

This produced:

customerId	firstName	lastName	orderId	currency	total
6	Charlotte	Ramos	1	$	111
4	Tallulah	Hines	2	£	958
2	Quyn	Meyer	3	£	721
4	Tallulah	Hines	4	$	834
1	Ursa	Vasquez	5	£	47
4	Tallulah	Hines	6	£	587
10	Karen	Marquez	7	£	198
3	Orli	Klein	8	$	200
3	Orli	Klein	9	£	726

5	Joel	Ross	10	$	200

Which meant that some customers were not represented in the results at all because they had never placed an order.

Let's say instead we want a list of every single customer and if they placed an order the details of that order. We could use a `LEFT JOIN` for this:

```
SELECT customers.customerId, customers.firstName, customers.lastName, orders.orderId,
orders.currency, orders.total
FROM customers
LEFT JOIN orders ON customers.customerId = orders.customerId
ORDER BY customers.customerId ASC;
```

Notice how the `INNER JOIN` became a `LEFT JOIN`. We also added an `ORDER BY` clause to make it easier to see that all the customers are now represented:

customerId	firstName	lastName	orderId	currency	total
1	Ursa	Vasquez	5	£	47
2	Quyn	Meyer	3	£	721
3	Orli	Klein	8	$	200
3	Orli	Klein	9	£	726
4	Tallulah	Hines	2	£	958
4	Tallulah	Hines	4	$	834
4	Tallulah	Hines	6	£	587
5	Joel	Ross	10	$	200
6	Charlotte	Ramos	1	$	111
7	Dennis	Avery	NULL	NULL	NULL
8	Igor	Malone	NULL	NULL	NULL
9	Connor	Witt	NULL	NULL	NULL
10	Karen	Marquez	7	£	198

An important thing to note is that since some customers have never placed an order the values for `orderId`, `currency` and `total` are `NULL`. `NULL` is a keyword in SQL which indicates some piece of information does not exist.

The first table in a `LEFT JOIN` will always have all of its records returned in the query no matter what the condition of the join is. The second table only has records returned if the condition matches the join.

RIGHT JOIN

A `RIGHT JOIN` is the direct opposite of a `LEFT JOIN`. While a `LEFT JOIN` includes all results from the first table and matching results from the second table, a `RIGHT JOIN` includes matching results from the first table and all results from the second table.

`customers`:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

`orders`:

orderId	date	currency	total	customerId
1	2020-11-14	$	111	6

2	2020-07-07	£	958	4
3	2021-02-18	£	721	2
4	2020-05-25	$	834	4
5	2020-07-10	£	47	1
6	2021-02-27	£	587	4
7	2021-03-04	£	198	10
8	2020-09-03	$	200	3
9	2020-11-17	£	726	3
10	2020-12-29	$	200	5

This is the same query from the previous section but with a `RIGHT JOIN` instead of a `LEFT JOIN`.

```
SELECT customers.customerId, customers.firstName, customers.lastName, orders.orderId,
orders.currency, orders.total
FROM customers
RIGHT JOIN orders ON customers.customerId = orders.customerId
ORDER BY customers.customerId ASC;
```

Produces:

customerId	firstName	lastName	orderId	currency	total
1	Ursa	Vasquez	5	£	47
2	Quyn	Meyer	3	£	721
3	Orli	Klein	8	$	200
3	Orli	Klein	9	£	726
4	Tallulah	Hines	2	£	958
4	Tallulah	Hines	4	$	834
4	Tallulah	Hines	6	£	587
5	Joel	Ross	10	$	200

6	Charlotte	Ramos	1	$	111
10	Karen	Marquez	7	£	198

All records from the `orders` table are represented and any customer with a `customerId` in the `orders` table is also represented in this query.

If we wanted the same results as the `LEFT JOIN` from the previous section we could swap the order of the tables in the query:

```
SELECT customers.customerId, customers.firstName, customers.lastName, orders.orderId,
orders.currency, orders.total
FROM orders
RIGHT JOIN customers ON customers.customerId = orders.customerId
ORDER BY customers.customerId ASC;
```

Producing:

customerId	firstName	lastName	orderId	currency	total
1	Ursa	Vasquez	5	£	47
2	Quyn	Meyer	3	£	721
3	Orli	Klein	8	$	200
3	Orli	Klein	9	£	726
4	Tallulah	Hines	2	£	958
4	Tallulah	Hines	4	$	834
4	Tallulah	Hines	6	£	587
5	Joel	Ross	10	$	200
6	Charlotte	Ramos	1	$	111
7	Dennis	Avery	NULL	NULL	NULL
8	Igor	Malone	NULL	NULL	NULL
9	Connor	Witt	NULL	NULL	NULL
10	Karen	Marquez	7	£	198

Because we swapped the order of the tables in the query the `customers` table became the second table and so every customer is once again represented in the results no matter if they have an order or not.

So why use a `RIGHT JOIN` when you could use a `LEFT JOIN` and just swap the order of the tables? Well when you have larger existing queries that you wish to add a join to it can be inconvenient to swap the order of the tables.

MySQL Operators

Operators

In a similar way to programming languages SQL also has the concept of operators.

Arithmetic operators are operators such as:

- add: +
- subtract: -
- multiply: *
- divide: /
- modulus: %

Comparison operators are operators such as:

- less than: <
- greater than: >
- equals to: =
- less than or equals to: <=
- greater than or equals to: >=
- not equal to <>

Logical operators are operators such as:

- ALL
- AND
- ANY
- BETWEEN
- EXISTS
- UNION
- IN
- LIKE
- NOT
- OR
- SOME

Subquery

You can use subqueries as a query within a query. When you execute a query that contains a subquery, the subquery is evaluated first and then the result of that query is used in the enclosing query.

It is easier to explain with an example so we'll use our two example tables:

`customers`:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

`orders`:

orderId	date	currency	total	customerId

1	2020-11-14	$	111	6
2	2020-07-07	£	958	4
3	2021-02-18	£	721	2
4	2020-05-25	$	834	4
5	2020-07-10	£	47	1
6	2021-02-27	£	587	4
7	2021-03-04	£	198	10
8	2020-09-03	$	200	3
9	2020-11-17	£	726	3
10	2020-12-29	$	200	5

Here is an example of a query that contains a subquery:

```
SELECT firstName, lastName
FROM customers
WHERE customerId IN
   (SELECT customerId
      FROM orders
   WHERE total > 300);
```

This query is designed to produce a list of customers who have spent more than 300 units of currency:

firstName	lastName
Quyn	Meyer
Orli	Klein
Tallulah	Hines

Now let's break down this query. The subquery is executed first so let's see what happens when we run this manually:

```
SELECT customerId FROM orders WHERE total > 300;
```

customerId
4
2
4
4
3

These are the `customerIds` from the orders table for any orders greater than 300. This is the data that will be passed into the parent query.

```
SELECT firstName, lastName
FROM customers
WHERE customerId IN
   (One of the Ids in the table above);
```

Since there are only three unique Ids then there are only three customers that have spent more than 300 units of currency in the past.

In the `WHERE` clause we are using the `IN` operator. This query will go through the `customers` table and check if the `customerId` for that row is in the results of the subquery. If it exists then it returns the first and last name for that row.

EXISTS

The `EXISTS` operator checks for the existence of records in a subquery.

Let's take a look at an example using our example tables:

`customers`:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

`orders`:

orderId	date	currency	total	customerId
1	2020-11-14	$	111	6

2	2020-07-07	£	958	4
3	2021-02-18	£	721	2
4	2020-05-25	$	834	4
5	2020-07-10	£	47	1
6	2021-02-27	£	587	4
7	2021-03-04	£	198	10
8	2020-09-03	$	200	3
9	2020-11-17	£	726	3
10	2020-12-29	$	200	5

And here is a query:

```
SELECT firstName, lastName
FROM customers
WHERE EXISTS
   (SELECT customerId
      FROM orders
   WHERE customerId = customers.customerId);
```

This query is designed to return the first and last names of any customers who have placed orders. Here is the result:

firstName	lastName
Ursa	Vasquez
Quyn	Meyer
Orli	Klein
Tallulah	Hines
Joel	Ross
Charlotte	Ramos
Karen	Marquez

There are seven records which means there are three customers who have never placed an order.

The `EXISTS` operator is a little unusual because instead of evaluating the subquery first, the subquery is evaluated for every record returned from the parent query. Let's take a look at the subquery first anyway:

```
SELECT customerId FROM orders WHERE customerId = customers.customerId;
```

This query doesn't work on its own because this subquery references the `customers` table which was a part of the parent query. This is known as a correlated query. To simulate how the `EXISTS` operator works we need to tweak the query a little bit:

```
SELECT customerId FROM orders WHERE customerId = 1;
```

This query will return one or more rows for any customers that exist in the orders table with an id of one. If no rows are returned then the customer with the id of one (1) doesn't exist in the `orders` table and therefore never placed an order.

Since the `EXISTS` operator is evaluated for every record in the `customers` table then the above query is re-run with the one changing to the `customerId` of the current record that is being evaluated.

So:

```
SELECT customerId FROM orders WHERE customerId = 1;
```

Then:

```
SELECT customerId FROM orders WHERE customerId = 2;
```

Then:

```
SELECT customerId FROM orders WHERE customerId = 3;
```

and so forth...

Every time the subquery returns one or more rows, then the parent query selects the first and last names of the row that is currently evaluating the `EXISTS` operator.

If the subquery returns no rows then the query moves on to the next row in the customers table.

```
SELECT firstName, lastName
FROM customers
WHERE EXISTS
   (SELECT customerId
   FROM orders
   WHERE customerId = customers.customerId);
```

From this we end up with a list of customers that have placed orders before:

firstName	lastName
Ursa	Vasquez
Quyn	Meyer
Orli	Klein
Tallulah	Hines
Joel	Ross
Charlotte	Ramos
Karen	Marquez

Note: It is worth remembering that the `EXISTS` operator is quite inefficient because the subquery has to be executed once for each record in the parent table. If your datasets are large then this could become problematic.

UNION

The `UNION` operator allows you to combine the results of two select statements into one.

Let's use our example tables again:

`customers`:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

`orders`:

orderId	date	currency	total	customerId
1	2020-11-14	$	111	6

2	2020-07-07	£	958	4
3	2021-02-18	£	721	2
4	2020-05-25	$	834	4
5	2020-07-10	£	47	1
6	2021-02-27	£	587	4
7	2021-03-04	£	198	10
8	2020-09-03	$	200	3
9	2020-11-17	£	726	3
10	2020-12-29	$	200	5

Let's say we wanted to have a list of unique countries and currencies that our customers use in one column, we could do something like:

```
SELECT DISTINCT country AS "countries & currencies" FROM customers
UNION
SELECT DISTINCT currency AS "countries & currencies" FROM orders;
```

Which results in:

countries & currencies
United States
Canada
United Kingdom
$
£

FULL JOIN

There is one kind of `JOIN` we haven't covered yet. The `FULL JOIN` is not supported by MySQL or MariaDB, however we can emulate it with the help of `UNION`.

If you recall from the previous module an `INNER JOIN` produces results where both tables meet the condition. The `LEFT JOIN` and `RIGHT JOIN` return all the results of one table and results from a second table which meets the condition of the `JOIN`.

A `FULL JOIN` returns results from both tables. Let's look at our example tables again this time with a slight modification:

`customers`:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

`orders`:

orderId	date	currency	total	customerId
1	2020-11-14	$	111	NULL

2	2020-07-07	£	958	4
3	2021-02-18	£	721	2
4	2020-05-25	$	834	4
5	2020-07-10	£	47	1
6	2021-02-27	£	587	4
7	2021-03-04	£	198	10
8	2020-09-03	$	200	3
9	2020-11-17	£	726	3
10	2020-12-29	$	200	5

There is one very important change in the orders table here. The first order has been corrupted and is no longer associated with a customer (note this would be extremely unusual if it were to happen but we need this for our example).

Now we have a situation where not every customer is associated with an order (not every customer has placed an order - this would be normal) and ALSO a situation where not all orders are associated with a customer (this would be much more unusual of course).

Remember our left join from the previous module:

```
SELECT customers.customerId, customers.firstName, customers.lastName, orders.orderId,
orders.currency, orders.total
FROM customers
LEFT JOIN orders ON customers.customerId = orders.customerId
ORDER BY customers.customerId ASC;
```

This now produces:

customerId	firstName	lastName	orderId	currency	total
1	Ursa	Vasquez	5	£	47
2	Quyn	Meyer	3	£	721
3	Orli	Klein	8	$	200
3	Orli	Klein	9	£	726

4	Tallulah	Hines	4	$	834
4	Tallulah	Hines	6	£	587
5	Joel	Ross	10	$	200
6	Charlotte	Ramos	NULL	NULL	NULL
7	Dennis	Avery	NULL	NULL	NULL
8	Igor	Malone	NULL	NULL	NULL
9	Connor	Witt	NULL	NULL	NULL
10	Karen	Marquez	7	£	198

And here is our right join from the previous module:

```
SELECT customers.customerId, customers.firstName, customers.lastName, orders.orderId,
orders.currency, orders.total
FROM customers
RIGHT JOIN orders ON customers.customerId = orders.customerId
ORDER BY customers.customerId ASC;
```

This now produces:

customerId	firstName	lastName	orderId	currency	total
NULL	NULL	NULL	1	$	111
1	Ursa	Vasquez	5	£	47
2	Quyn	Meyer	3	£	721
3	Orli	Klein	8	$	200
3	Orli	Klein	9	£	726
4	Tallulah	Hines	2	£	958
4	Tallulah	Hines	4	$	834
4	Tallulah	Hines	6	£	587
5	Joel	Ross	10	$	200

To get the `FULL JOIN` we need the `UNION` of the `LEFT JOIN` and the `RIGHT JOIN`:

```
SELECT customers.customerId, customers.firstName, customers.lastName, orders.orderId,
orders.currency, orders.total
FROM customers
LEFT JOIN orders ON customers.customerId = orders.customerId
UNION
SELECT customers.customerId, customers.firstName, customers.lastName, orders.orderId,
orders.currency, orders.total
FROM customers
RIGHT JOIN orders ON customers.customerId = orders.customerId
ORDER BY customerId;
```

And the result:

customerId	firstName	lastName	orderId	currency	total
NULL	NULL	NULL	1	$	111
1	Ursa	Vasquez	5	£	47
2	Quyn	Meyer	3	£	721
3	Orli	Klein	8	$	200
3	Orli	Klein	9	£	726
4	Tallulah	Hines	2	£	958
4	Tallulah	Hines	4	$	834
4	Tallulah	Hines	6	£	587
5	Joel	Ross	10	$	200
6	Charlotte	Ramos	NULL	NULL	NULL
7	Dennis	Avery	NULL	NULL	NULL
8	Igor	Malone	NULL	NULL	NULL
9	Connor	Witt	NULL	NULL	NULL
10	Karen	Marquez	7	£	198

You can see that every order has been listed even if it leads to a `NULL` in the `customers` table and also every customer has been listed even if it leads to a `NULL` in the `orders`

table. This is exactly the behavior of a `FULL JOIN`. At the end of the day it was simply the `UNION` of a `LEFT JOIN` and a `RIGHT JOIN`.

MySQL Database Admin

Setting up a database

So far you've learned how to interact with a database that already exists but how about setting up your own simple database? In this module we'll go through what it takes to setup a database using SQL.

So far in this course we've been using one database consisting of two tables. Here are those tables again:

`customers`:

customerId	firstName	lastName	address	city	country
1	Ursa	Vasquez	P.O. Box 878, 8416 Nullam St.	Worcester	United States
2	Quyn	Meyer	P.O. Box 670, 7155 Tincidunt St.	Price	Canada
3	Orli	Klein	4981 Gravida St.	Barrow-in-Furness	United Kingdom
4	Tallulah	Hines	6279 Pellentesque Street	Omaha	United States
5	Joel	Ross	P.O. Box 842, 4634 Egestas Avenue	Clovenfords	United Kingdom
6	Charlotte	Ramos	794-1654 A Rd.	Akron	United States
7	Dennis	Avery	P.O. Box 506, 4804 Molestie Avenue	Matlock	United Kingdom
8	Igor	Malone	6627 Porttitor Rd.	Irvine	United Kingdom
9	Connor	Witt	5979 Vel St.	Tain	United Kingdom
10	Karen	Marquez	Ap #524-1173 Metus. Road	Annapolis Royal	Canada

`orders`:

1	2020-11-14	$	111	6
2	2020-07-07	£	958	4
3	2021-02-18	£	721	2
4	2020-05-25	$	834	4
5	2020-07-10	£	47	1
6	2021-02-27	£	587	4
7	2021-03-04	£	198	10
8	2020-09-03	$	200	3
9	2020-11-17	£	726	3
10	2020-12-29	$	200	5

We're going to go through the SQL that was required to setup these two tables. The first step is to connect to the DBMS and create the database itself:

```
mysql -u foundations -p
```

```
CREATE DATABASE foundations;
USE foundations;
```

This has created a database called 'foundations' then selected the foundations database as the active database. Keep in mind that each database on the DBMS has to have a unique name.

Datatypes

Now that our database is setup we must create the tables:

```
CREATE TABLE `customers` (
    `customerId` mediumint(8) unsigned NOT NULL auto_increment,
  `firstName` varchar(255) default NULL,
    `lastName` varchar(255) default NULL,
  `address` varchar(255) default NULL,
    `city` varchar(255),
  `country` varchar(100) default NULL,
    PRIMARY KEY (`customerId`)
) AUTO_INCREMENT=1;

CREATE TABLE `orders` (
    `orderId` mediumint(8) unsigned NOT NULL auto_increment,
  `date` varchar(255),
    `currency` varchar(255) default NULL,
  `total` mediumint default NULL,
    `customerId` mediumint unsigned NOT NULL,
  PRIMARY KEY (`orderId`),
    FOREIGN KEY (`customerId`) REFERENCES customers(`customerId`)
) AUTO_INCREMENT=1;
```

Don't panic and we'll go through this line by line.

The first step is to specify the name of the table, in our case it is 'customers'. Then we need to specify the fields.

For each field we must specify the name of the field and then a datatype. Remember SQL is a structured database so each field has a static type. The types are very similar to types in programming.

For integers you have the int(x) datatype: an integer that can be signed or unsigned. For a signed integer the allowed range is from -2147483648 to 2147483647. If the integer is unsigned the range is from 0 to 4294967295. You can specify the number of digits to display up to 11 (this is the number in brackets after the datatype represented by 'x').

There are several variations on the int datatype including: - tinyint(x) - smallint(x) - mediumint(x) - bigint(x)

These are all similar to the int datatype except they have a different allowed range for the numbers held in the fields. It is much more efficient to use a tinyint if you only need to store small numbers in the field for example.

For floats you have the float(x, y) datatype: a float is a floating point number (a number with a decimal point in it). The x in brackets is the number of digits to display and the y in brackets is the number of decimals allowed. The maximum number of decimals allowed for a float datatype is 24 decimal places.

There are a few other variations on a fioat: - double(x, y) - decimal(x, y)

For dates you can use the date datatype: A date is in international format of YYYY-mm-dd for example 2021-04-20. There are other variations such as:

- datetime
- timestamp
- time
- year

Finally we have datatypes designed for storing strings. Here we use the varchar(s) datatype. A varchar(s) is a variable length string with s being the length which must be between 1 and 255. There are also variations of this:

- char(s)
- blob
- tinyblob
- mediumblob
- longblob
- enum

For a production database it is always worth making sure you know the best datatype and size to use for each field. Being too generous with assigning sizes can lead to problems at scale but also being too conservative with field sizes can also cause problems at scale.

Constraints

We'll continue going through the SQL statements that create our tables:

```
CREATE TABLE `customers` (
  `customerId` mediumint(8) unsigned NOT NULL auto_increment,
  `firstName` varchar(255) default NULL,
  `lastName` varchar(255) default NULL,
  `address` varchar(255) default NULL,
  `city` varchar(255),
  `country` varchar(100) default NULL,
  PRIMARY KEY (`customerId`)
) AUTO_INCREMENT=1;

CREATE TABLE `orders` (
  `orderId` mediumint(8) unsigned NOT NULL auto_increment,
  `date` varchar(255),
  `currency` varchar(255) default NULL,
  `total` mediumint default NULL,
  `customerId` mediumint unsigned NOT NULL,
  PRIMARY KEY (`orderId`),
  FOREIGN KEY (`customerId`) REFERENCES customers(`customerId`)
) AUTO_INCREMENT=1;
```

We have already gone over how to assign the table names and field names and data types.

If we look at the `firstName` field in the customers table we have `default NULL` after the datatype. This is known as a constraint. A constraint defines the rules for that field in the table.

The `default NULL` constraint means that if no value is entered into this field for a row then the value defaults to `NULL`.

On the other hand if we look at the `customerId` field in the customers table we have `NOT NULL auto_increment`. In this case we are saying the `customerId` field always must have a value in it (can never be `NULL`) and that it should autoincrement. This means basically that the first row you enter will have a `customerId` of 1 and the next row will have a `customerId` of 2 and so on. This is important because the `customerId` is the primary key for this table and therefore must always be unique. You cannot have two rows in the customers table with the same `customerId`.

After we have specified our datatypes we have the primary key constraint:

```
PRIMARY KEY (`customerId`)
```

This indicates that our primary key for this table is the `customerId` field.

Finally we end the statement with `AUTO_INCREMENT=1` which simply tells us to start the auto_increment of ids for this table from 1.

We can see very much the same thing with the orders table, however there is one extra constraint here:

```
FOREIGN KEY (`customerId`) REFERENCES customers(`customerId`)
```

This is a foreign key constraint which enforces the relationship on the `customerId` field in the `orders` table and the `customerId` field in the `customers` table. This constraint will prevent these links from being broken - for example you couldn't add a row into the `orders` table with a `customerId` that doesn't exist in the customers table.

Inserting Data

The next step in building our database is inserting data into our newly formed tables. Here is what we've done:

```
INSERT INTO `customers` (`firstName`,`lastName`,`address`,`city`,`country`) VALUES
("Ursa","Vasquez","P.O. Box 878, 8416 Nullam St.","Worcester","United States"),
("Quyn","Meyer","P.O. Box 670, 7155 Tincidunt St.","Price","Canada"),
("Orli","Klein","4981 Gravida St.","Barrow-in-Furness","United Kingdom"),
("Tallulah","Hines","6279 Pellentesque Street","Omaha","United States"),
("Joel","Ross","P.O. Box 842, 4634 Egestas Avenue","Clovenfords","United Kingdom"),
("Charlotte","Ramos","794-1654 A Rd.","Akron","United States"),("Dennis","Avery","P.O.
Box 506, 4804 Molestie Avenue","Matlock","United Kingdom"),("Igor","Malone","6627
Porttitor Rd.","Irvine","United Kingdom"),("Connor","Witt","5979 Vel
St.","Tain","United Kingdom"),("Karen","Marquez","Ap #524-1173 Metus. Road","Annapolis
Royal","Canada");

INSERT INTO `orders` (`date`,`currency`,`total`,`customerId`) VALUES ("2020-11-
14","$",111,6),("2020-07-07","£",958,4),("2021-02-18","£",721,2),("2020-05-
25","$",834,4),("2020-07-10","£",47,1),("2021-02-27","£",587,4),("2021-03-
04","£",198,10),("2020-09-03","$",200,3),("2020-11-17","£",726,3),("2020-12-
29","$",200,5);
```

The syntax here is very simple compared to creating the tables. First we specify the table to insert the data into, then we list the names of the fields to insert the data into.

Finally we specify the values to enter into those fields in the same order we listed the field names.

Scripting Table Creation

You can combine all of this into a single script to automate the creation and insertion of the tables. Here is the file we created for the example tables:

```
USE foundations;

DROP TABLE IF EXISTS `orders`;
DROP TABLE IF EXISTS `customers`;

CREATE TABLE `customers` (
  `customerId` mediumint(8) unsigned NOT NULL auto_increment,
  `firstName` varchar(255) default NULL,
  `lastName` varchar(255) default NULL,
  `address` varchar(255) default NULL,
  `city` varchar(255),
  `country` varchar(100) default NULL,
  PRIMARY KEY (`customerId`)
) AUTO_INCREMENT=1;

INSERT INTO `customers` (`firstName`,`lastName`,`address`,`city`,`country`) VALUES
("Ursa","Vasquez","P.O. Box 878, 8416 Nullam St.","Worcester","United States"),
("Quyn","Meyer","P.O. Box 670, 7155 Tincidunt St.","Price","Canada"),
("Orli","Klein","4981 Gravida St.","Barrow-in-Furness","United Kingdom"),
("Tallulah","Hines","6279 Pellentesque Street","Omaha","United States"),
("Joel","Ross","P.O. Box 842, 4634 Egestas Avenue","Clovenfords","United Kingdom"),
("Charlotte","Ramos","794-1654 A Rd.","Akron","United States"),("Dennis","Avery","P.O.
Box 506, 4804 Molestie Avenue","Matlock","United Kingdom"),("Igor","Malone","6627
Porttitor Rd.","Irvine","United Kingdom"),("Connor","Witt","5979 Vel
St.","Tain","United Kingdom"),("Karen","Marquez","Ap #524-1173 Metus. Road","Annapolis
Royal","Canada");

CREATE TABLE `orders` (
  `orderId` mediumint(8) unsigned NOT NULL auto_increment,
  `date` varchar(255),
  `currency` varchar(255) default NULL,
  `total` mediumint default NULL,
  `customerId` mediumint unsigned NOT NULL,
  PRIMARY KEY (`orderId`),
  FOREIGN KEY (`customerId`) REFERENCES customers(`customerId`)
) AUTO_INCREMENT=1;

INSERT INTO `orders` (`date`,`currency`,`total`,`customerId`) VALUES ("2020-11-
14","$",111,6),("2020-07-07","£",958,4),("2021-02-18","£",721,2),("2020-05-
25","$",834,4),("2020-07-10","£",47,1),("2021-02-27","£",587,4),("2021-03-
04","£",198,10),("2020-09-03","$",200,3),("2020-11-17","£",726,3),("2020-12-
29","$",200,5);
```

A few things to note here - this script assumes that a database called 'foundations' already exists in the DBMS.

Notice the `DROP TABLE IF EXISTS` - this will simply delete the table if a table by that name already exists. This is necessary because a simple `CREATE TABLE` will fail if a table by that name already exists. This helps us to refresh the data by simply running the script again, if we run some queries that mess up the data.

To import the data you would simply save the script into a text file and then you can run:

```
mysql -u foundations -p < db.sql
```

You will be prompted to enter the password for your user and once you do the data will be in the database.

Deleting a Table

Now that we have a database setup, what do we do if we want to delete a table?

We need to be able to delete tables.

To delete a table we can use:`DROP TABLE tablename;` for example:

```
DROP TABLE customers;
```

If we run this however we will get an error:

```
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint
fails
```

Remember when we setup a foreign key constraint on the orders table?

Here is a reminder for our table definition:

```
CREATE TABLE `customers` (
  `customerId` mediumint(8) unsigned NOT NULL auto_increment,
  `firstName` varchar(255) default NULL,
  `lastName` varchar(255) default NULL,
  `address` varchar(255) default NULL,
  `city` varchar(255),
  `country` varchar(100) default NULL,
  PRIMARY KEY (`customerId`)
) AUTO_INCREMENT=1;

CREATE TABLE `orders` (
  `orderId` mediumint(8) unsigned NOT NULL auto_increment,
  `date` varchar(255),
  `currency` varchar(255) default NULL,
  `total` mediumint default NULL,
  `customerId` mediumint unsigned NOT NULL,
  PRIMARY KEY (`orderId`),
  FOREIGN KEY (`customerId`) REFERENCES customers(`customerId`)
) AUTO_INCREMENT=1;
```

This foreign key constaint will prevent us from breaking the relation between the `orders` table and the `customers` table by deleting the `customers` table. This means we actually first have to delete the `orders` table:

```
DROP TABLE orders;
```

This should work and now that there is no more foreign key constraint, we should be free to delete the customers table:

```
DROP TABLE customers;
```

Windows Overview

Learning Objectives

After completing this module, you should be able to:

- Know the differences between Windows versions, and their uses in desktop, server and mobile environments.
- Install Windows 10.
- Set up networking on Windows.
- Configure Windows Defender.
- Change settings in the registry.
- Examine the log files that Windows generates.

Module Content

This module will provide a basic introduction to Windows, focusing on the desktop offering.

We will be covering the following components:

- What is Windows?
- Networking
- Windows Defender
- The Registry
- Log Files

What is Windows?

Microsoft Windows (usually referred to as simply 'Windows') is an operating system developed by Microsoft. Microsoft Windows is a graphical operating system, meaning it has a GUI (Graphical User Interface). The earliest version of Microsoft Windows was simply a graphical shell added on top of the existing Microsoft DOS operating system, which up until then had been purely text based.

Windows Desktop

As of the time of writing, the most current version of Windows is known as Windows 10. This is the version of Windows we will be focusing on in this module. Windows on the desktop enjoys a majority market share, due mainly to its ease of use, particularly in the earlier versions of the operating system. In addition, ubiquitous use in business settings means around 90% of all desktop computers run one version of Microsoft Windows or another.

Windows Server

Although Microsoft Windows is best known as a desktop operating system, there are many different versions which are specialised for different uses. Microsoft Windows Server is, as the name suggests, an operating system designed to run on servers. It is set up to allow administrators to easily set up file sharing, email and other such functionality. The reason Windows Server is so common in enterprise settings, however, is mostly down to Active Directory. Active Directory is a system that allows the server to communicate with Windows desktops, allowing the administrator to easily configure all the connected desktops from one location, amongst other uses which we will cover in future modules.

Windows Server does also have a GUI (Graphical User Interface), however there is a version that strips out most of the interface. This has only been possible in the most recent versions of Windows Server because of PowerShell, an incredibly powerful command line (text based) interface. This version of Windows Server is known as Windows Server Core.

Windows IoT (Embedded)

Windows IoT (Internet of Things), formerly known as Windows Embedded is a version of Windows designed to be run on low power computers, such as those found in point of sale systems (tills), digital billboards, cash points, and so on.

Windows on Mobile Devices

Windows can even be found on mobile devices. Although the old Windows Phone operating system has now been retired, the current operating system for mobile devices such as phones and tablets is a version of Windows 10, albeit one that has been customised for use on mobile devices.

XBox

You might be surprised to discover that the Xbox One games console also runs a version of Windows 10. Although this version of Windows 10 is heavily customised, its core remains the same as the operating system you use on desktop computers.

Windows 10 Install Walkthrough

In this section we will run through the standard procedure to install Windows 10. In this instance, we will use a virtual machine but the process is highly similar to a standard hardware build. One of the big differences is that virtual devices and virtual setup almost always 'just work' where physical hardware systems often require special drivers, or tweaks. We cover this separately in the course.

Virtualization Tools

In this video we also cover the installation of VMware tools, though the process is similar for other virtualization providers. These tools provide drivers for the virtually presented devices that the system interacts with. Installing these increases the functionality and smooth operating of the guest system, though it makes it abundantly obvious it is a virtual system to any processes running inside the system. These drivers provide better graphics capabilities, smoother file transfer and host to guest integration.

Patching

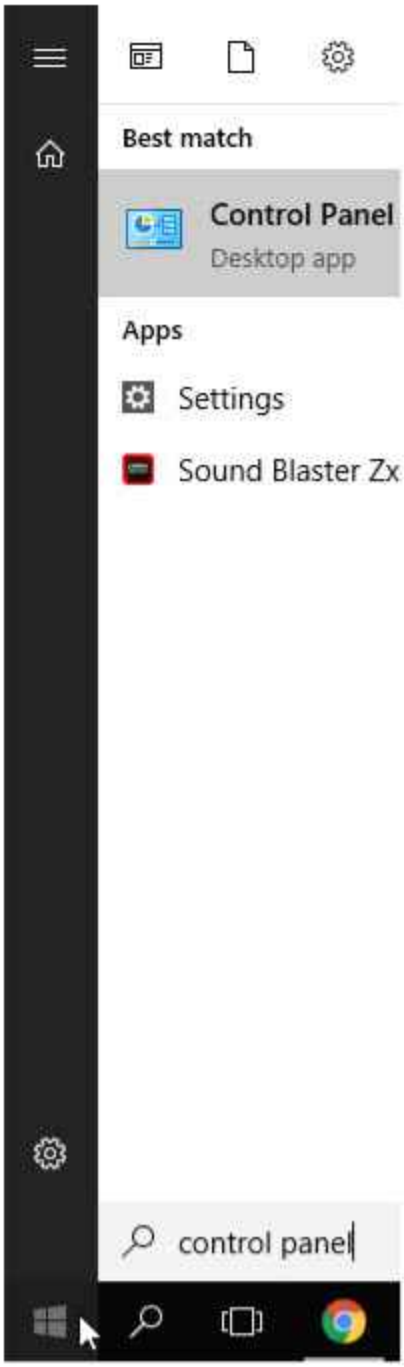
Windows patches are one of the most important things you can do as part of good IT hygiene to keep a system running securely and efficiently. Windows 10 provides rich controls over patches and the ability to install them out of working hours. There are also bandwidth saving capabilities and peer-to-peer protocol distribution of updates where systems nearby can share updates to avoid flooding bandwidth of the upstream connection to retrieve the same files from Microsoft.

All of these capabilities can also be controlled centrally by the administrator.

Networking Windows

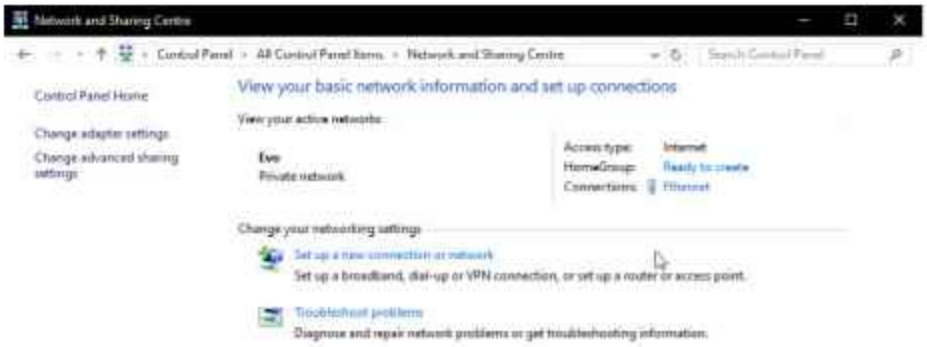
Like most operating systems, Windows can usually get you networked automatically as soon as you connect, no matter if you are plugging in an ethernet cable or connecting to the WiFi. Connection setup is handled by a network protocol called DHCP (Dynamic Host Configuration Protocol), which will be covered elsewhere in this course.

If you are on a network that doesn't support DHCP, or you'd prefer to set your own settings for whatever reason, you can change your network settings to set them manually. The first stage is to access the Control Panel, so hit the Windows icon in the bottom left and type 'Control Panel' and click on the 'Control Panel' entry that comes up.

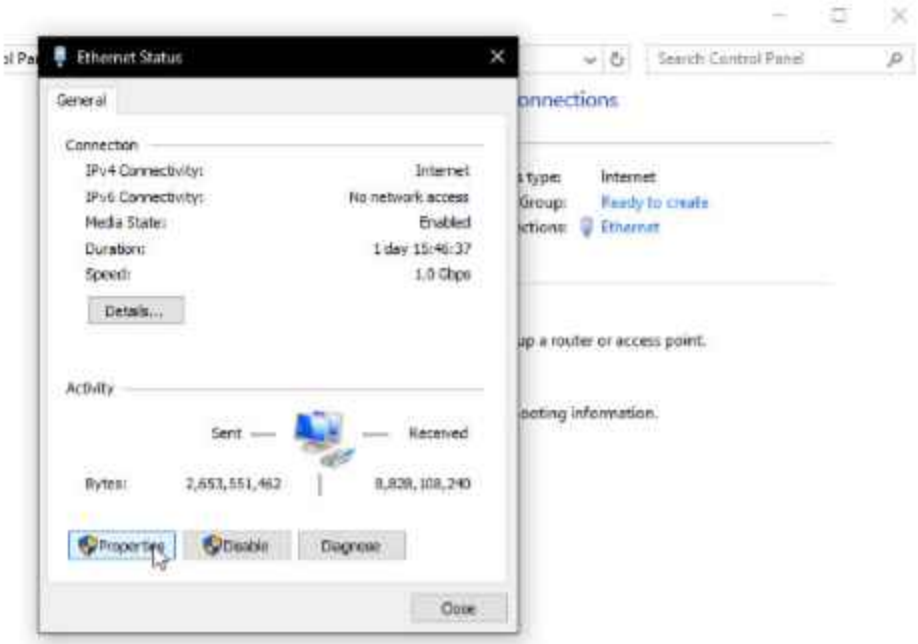


Once in the Control Panel, use the search in the top right and search for 'Network'. The first result should be 'Network and Sharing Centre', select it.

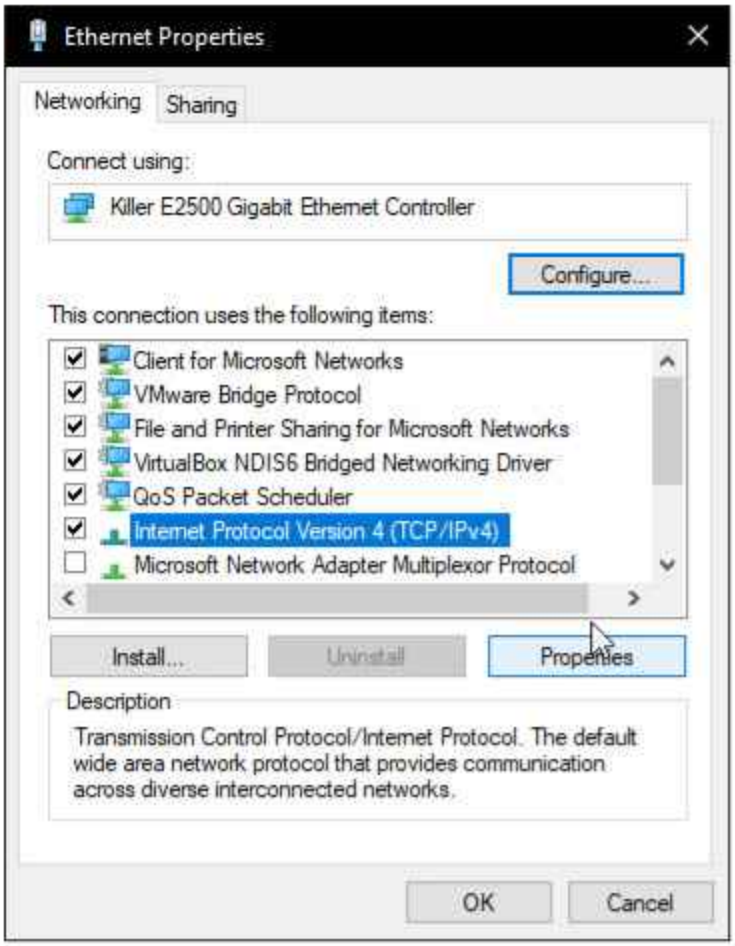
You should see 'Connections', and there will be a blue text link that says either 'Ethernet' if you are connected through a cable, or 'WiFi' if you are connected over Wireless. Click on the blue text link to bring up a status window, then click on 'Properties'.



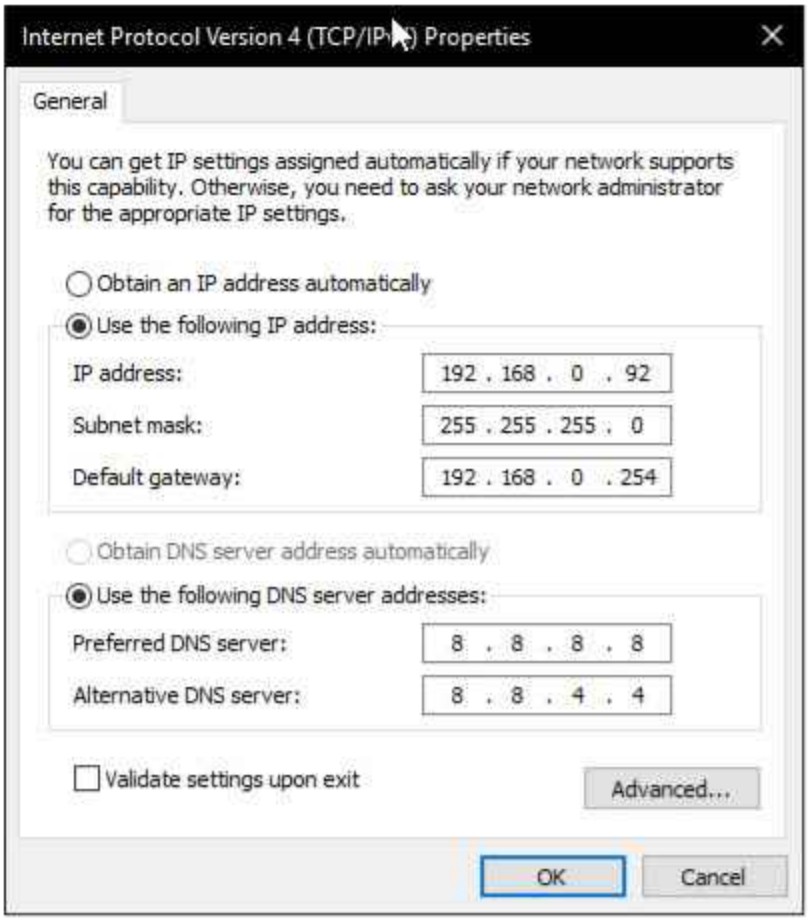
In the 'Properties' window, you will have a list of items.



Select Internet Protocol Version 4 (IPv4) and then click on 'Properties' again.



From this window, you can swap from 'Obtain an IP Address automatically' to 'Use the following IP address' to configure custom network settings. Similarly, for the DNS servers, you can set them statically by ticking 'Use the following DNS server addresses'.



If you do make a change here, make sure to select 'OK' to save the settings, and don't forget that if your network is using IPv6, then you should also look at the settings in the Internet Protocol Version 6 (IPv6) item.

Windows Defender

Windows Defender is the built in anti-virus solution that comes with Windows 10. It is enabled by default, and it will also update itself with the latest virus definitions periodically. Microsoft advises that you stick with it, instead of installing third party antivirus solutions, however other solutions might provide more enterprise management capabilities, or features that you need for your particular security setup. What is important is that it provides robust malicious code prevention capabilities out of the box. Microsoft are improving it all the time too, for example in response to ransomware which has been prolific in the past few years.

Windows Defender can be disabled. However, Windows will automatically turn it back on after a period of time. If you install a third party anti-virus product, Windows Defender will be disabled automatically, and it won't turn back on until your third party anti-virus has been uninstalled. Windows does this because running multiple antivirus products at once can cause unexpected behaviour in the computer. You can chain together multiple AV products if you want to in later versions of Windows, but the gains of doing this against performance are minimal. This makes it a rare use case.

To access the settings for Windows Defender, go to the 'Windows' icon in the taskbar and click on it, then type 'defender'. You should see 'Windows Defender Security Centre' in the search results. Windows Defender can be configured from that location.

Windows Firewall

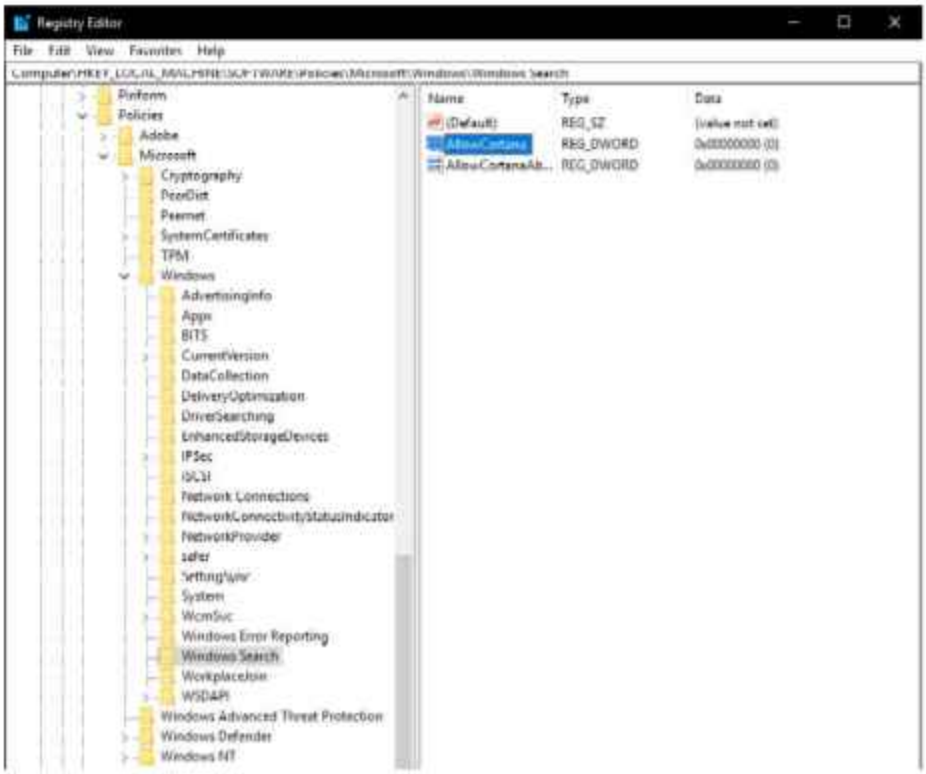
In addition to Windows Defender, Windows 10 has a built in software firewall which is also enabled by default. The firewall will try to prevent unknown connections from coming into your computer from the internet, and for each program that runs, and tries to connect out to the internet, it will also ask you permission to allow it through the firewall. This only happens the first time the program runs, and your choice is saved after that.

Registry

The Windows Registry is a database of settings for both the operating system and for any applications which support storing data in the registry. The settings here are low-level and are not meant to be changed or even seen by the typical end user. However, the registry is incredibly powerful. There are settings stored in the registry that cannot be modified through any other method.

You should not edit the registry without being confident of what you are doing. It is possible to destroy your Windows installation with a single error. You can also export the registry as a backup, however if you mess up too badly there is no guarantee you could restore the registry using the exported file. At that point, your only option would be a full re-install of the operating system.

To view the registry, we need to use a tool called 'regedit', which is installed on Windows by default. To access it, click the 'Windows' icon in the taskbar and type 'regedit'.



As an example of what can be accomplished with the Windows Registry, we will show you how to disable Cortana, the Windows 10 virtual assistant which pops up when you access the Windows start menu. There is no setting to disable this in the Windows Control Panel; however, we can accomplish it by editing the Registry directly.

In the image above, you can see we have gone into

```
HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows\Windows
Search
```

and created a new entry in the registry called 'Allow Cortana'.

When you create a new entry in the registry (right click where you want to add it and go to New), it will ask you for the type of entry to be added. For this use, it needs to be DWORD(32 Bit). You will then have to give the entry a name; in this case, it must be 'Allow Cortana'. Finally, you have to set a value, in our case 0 (to turn Cortana off).

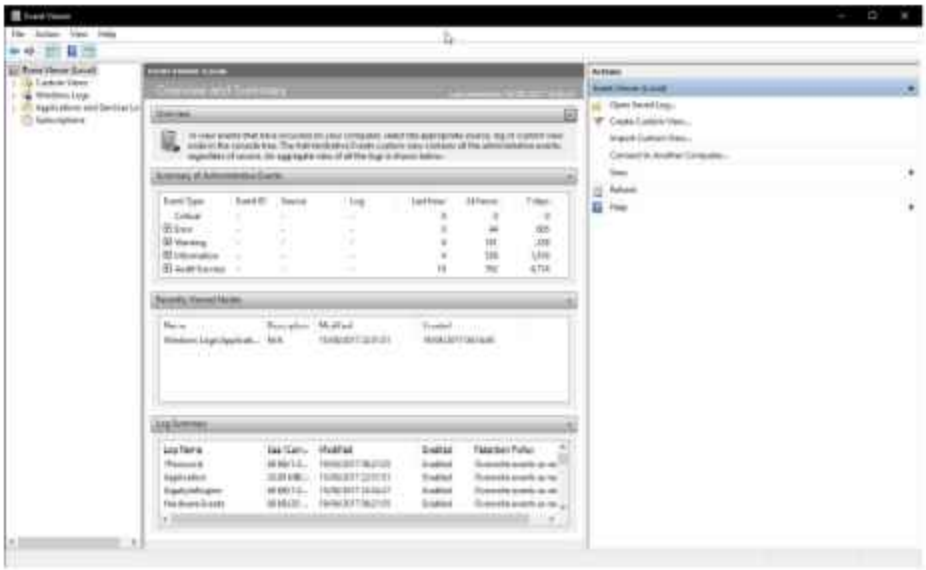
After the changes, you must close regedit and restart the computer; then you should be able to access the 'Start' menu from the task bar without Cortana appearing.

Note: This works at the time of writing, but Microsoft can and often does change the way things work during updates. If this does not work for you, try searching online for an updated method to disable Cortana. It will almost certainly involve using regedit. There are other methods, such as using 'group policy'; however, the group policy editor is only available in the Pro version of Windows 10.

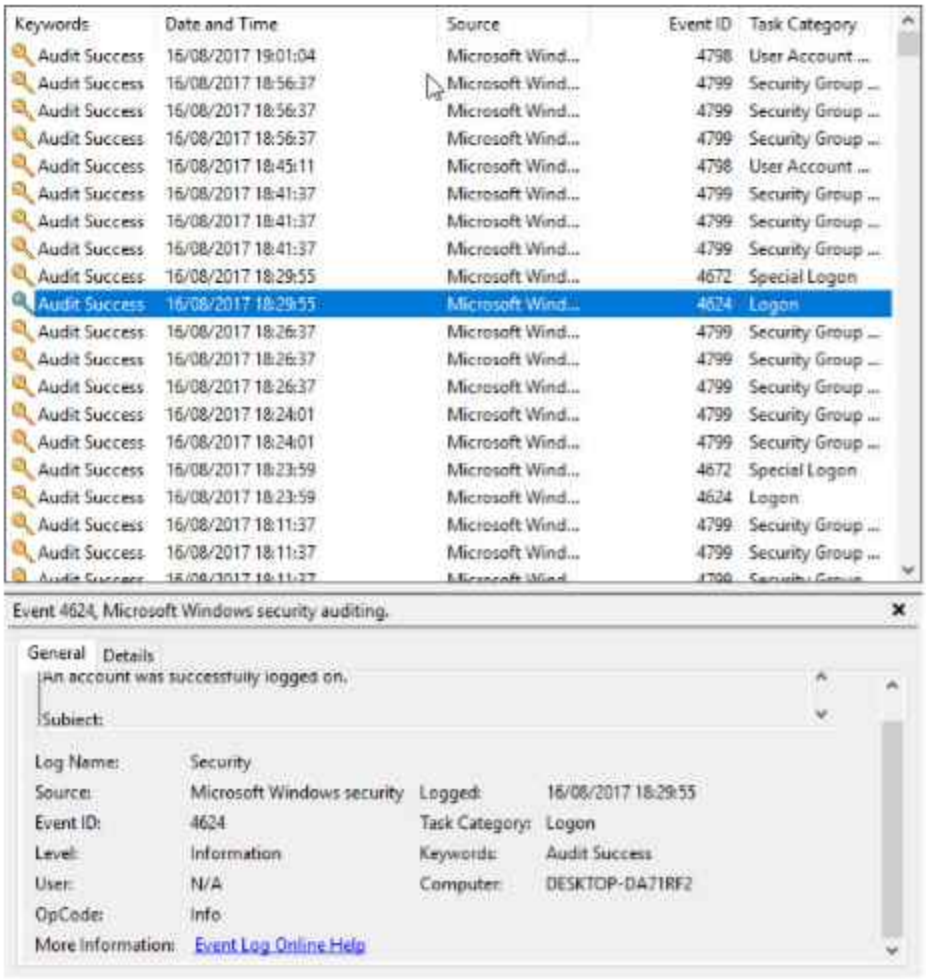
Log Files

Like most operating systems, Windows keeps a 'log' of events that occur on the computer. Log files are useful in many circumstances, from determining what was responsible for a crash occurring, to trying to trace if an attacker has compromised the system.

To view the log files, we can use the Event Viewer. The Event Viewer can be accessed by going to the 'Windows' icon in the taskbar, and then typing 'event viewer'. One of the search results should be 'Event Viewer'. Click on it to open, and you should be presented with a window that looks like this:



From here, you can expand the 'Windows Logs' folder and pick a log file to examine:



Keywords	Date and Time	Source	Event ID	Task Category
Audit Success	16/08/2017 19:01:04	Microsoft Wind...	4798	User Account ...
Audit Success	16/08/2017 18:56:37	Microsoft Wind...	4799	Security Group ...
Audit Success	16/08/2017 18:56:37	Microsoft Wind...	4799	Security Group ...
Audit Success	16/08/2017 18:56:37	Microsoft Wind...	4799	Security Group ...
Audit Success	16/08/2017 18:45:11	Microsoft Wind...	4798	User Account ...
Audit Success	16/08/2017 18:41:37	Microsoft Wind...	4799	Security Group ...
Audit Success	16/08/2017 18:41:37	Microsoft Wind...	4799	Security Group ...
Audit Success	16/08/2017 18:41:37	Microsoft Wind...	4799	Security Group ...
Audit Success	16/08/2017 18:29:55	Microsoft Wind...	4672	Special Logon
Audit Success	16/08/2017 18:29:55	Microsoft Wind...	4624	Logon
Audit Success	16/08/2017 18:26:37	Microsoft Wind...	4799	Security Group ...
Audit Success	16/08/2017 18:26:37	Microsoft Wind...	4799	Security Group ...
Audit Success	16/08/2017 18:26:37	Microsoft Wind...	4799	Security Group ...
Audit Success	16/08/2017 18:24:01	Microsoft Wind...	4799	Security Group ...
Audit Success	16/08/2017 18:24:01	Microsoft Wind...	4799	Security Group ...
Audit Success	16/08/2017 18:23:59	Microsoft Wind...	4672	Special Logon
Audit Success	16/08/2017 18:23:59	Microsoft Wind...	4624	Logon
Audit Success	16/08/2017 18:11:37	Microsoft Wind...	4799	Security Group ...
Audit Success	16/08/2017 18:11:37	Microsoft Wind...	4799	Security Group ...

Event 4624, Microsoft Windows security auditing.

General Details

An account was successfully logged on.

Subject:

Log Name:	Security		
Source:	Microsoft Windows security	Logged:	16/08/2017 18:29:55
Event ID:	4624	Task Category:	Logon
Level:	Information	Keywords:	Audit Success
User:	N/A	Computer:	DESKTOP-DA71RF2
OpCode:	Info		
More Information:	Event Log Online Help		

Here is an example of the 'Security' log, where we can see a 'logon' event.

These logs can also be useful if you get a Blue Screen of Death and need to track down the cause, or maybe you just want to find out if anyone logged into your computer while you were away.

Windows Permissions

Contents

In this module, we will be covering:

- User Accounts
- Groups in Windows
- User Account Control
- File Permissions
- Hidden Files

User Accounts

A user account is what you use to sign in to your computer. Seems simple, right? Several accounts are created automatically during installation so we'll have a look at those, before looking at how to create and manage users on your machine.

Default Accounts

Along with any user you specify during installation, the following users are always created when you install Windows:

- **Administrator**: This account has 'full' control over the machine. There are some things it can't do (generally something that would break the operating system), but it can be used to manage other users and install applications.
- **Guest**: The Guest account is used by people who do not have an actual account on the computer. It is limited in what it can do, and care should be taken to ensure that this account is not able to do anything that could cause harm.
- **DefaultAccount**: This is used as the template for all new accounts created on the machine. Any changes made to this account will be carried over.

All three of the accounts will be disabled by default, which means you can't log in using them. But they're there and can be enabled if you need.

Account Types

Windows 10 has three account types (it's actually closer to two as you'll see shortly): Administrator, Standard and Child.

The Administrator account type is able to make changes that affect all users, including modifying other accounts, installing applications and changing security settings on the machine.

A user with the Standard account type can do most things that an Administrator can, however they are prevented from doing anything that would affect other users; they can't delete certain files or change things that affect everyone.

The third type, Child, is effectively the Standard account with parental controls enabled automatically so things like usage time can be monitored or limited. It can only be created as part of a family, it's not possible to create a Child type by any other method.

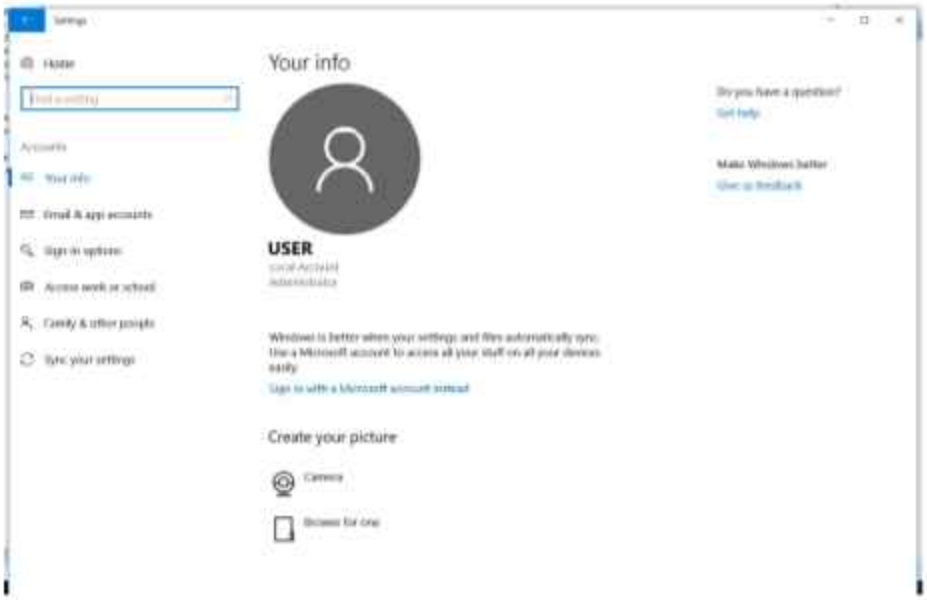
Creating a New User

Assuming the machine will be shared, you'll more than likely need to create additional accounts for each user. There are many different ways to create new user accounts in Windows. We'll have a look at two of the simplest: using the Control Panel and using the Local Users management console.

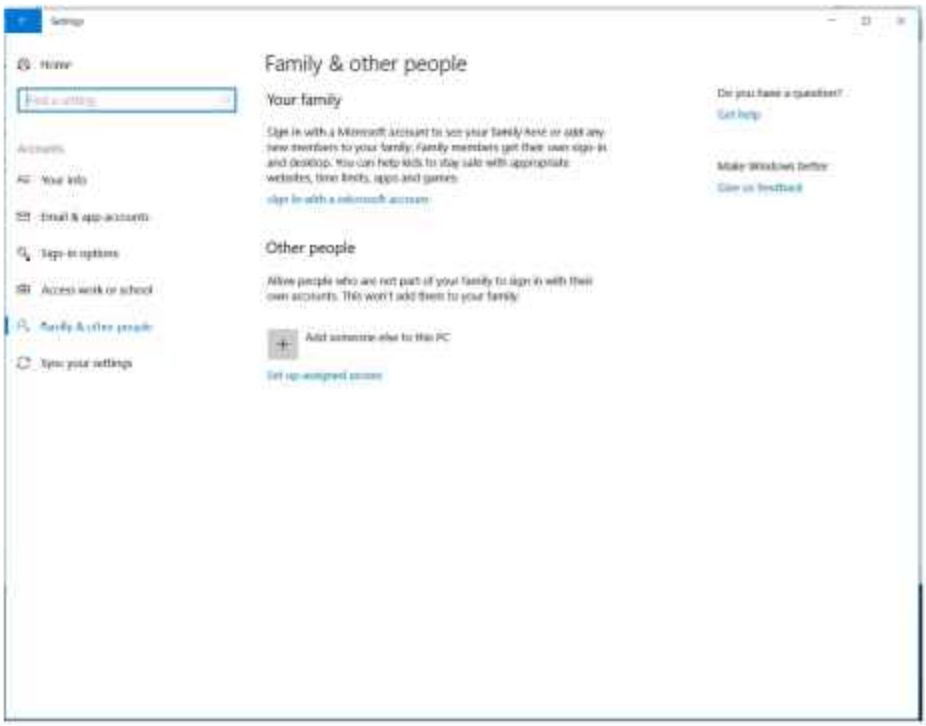
Creating a User With The Control Panel

Using the Control Panel to create a new user is by far the easiest way to tackle this task. If you hit 'Start' and then select 'Settings' (the cog icon on the left) you'll be presented with a screen much like the following:

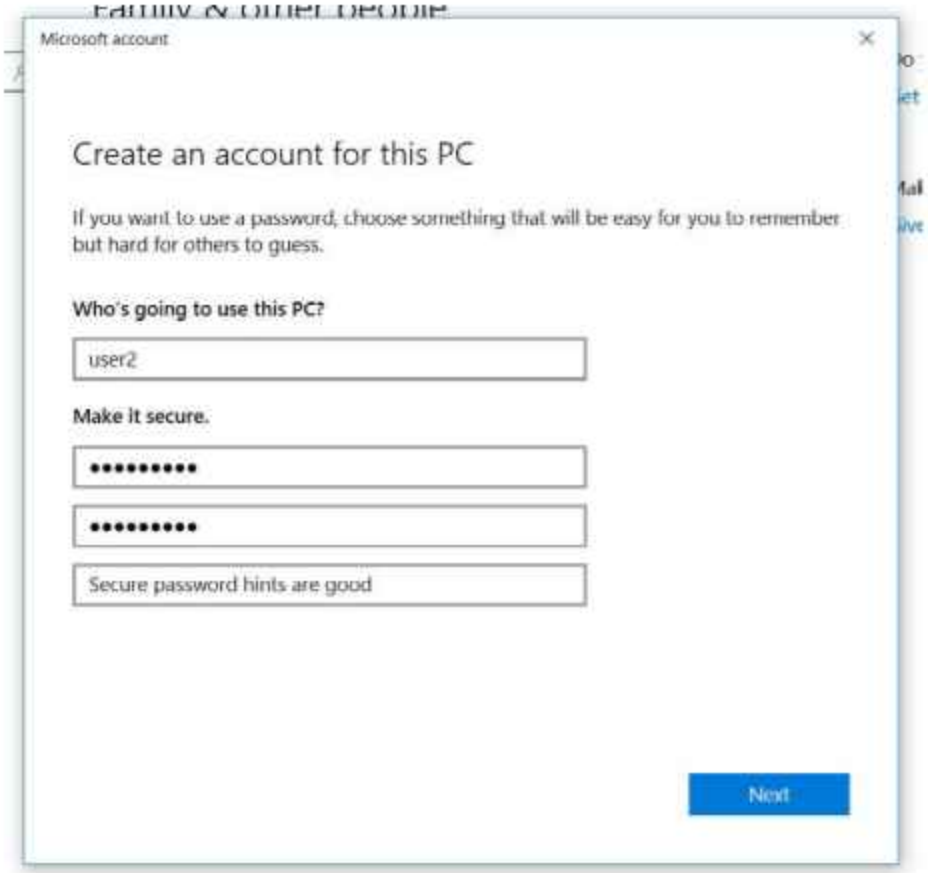
Here we want to select 'Accounts', which will take us to a section where you can manage not only your own account but also create and manage others. The screen will look like this:



This page allows you to change various aspects of your account, such as profile picture and lock screen settings. The section we're interested in here is 'Family & other people'. Selecting this will take us to a page where we can manage other users of this machine, so let's do that now. We end up on this page:

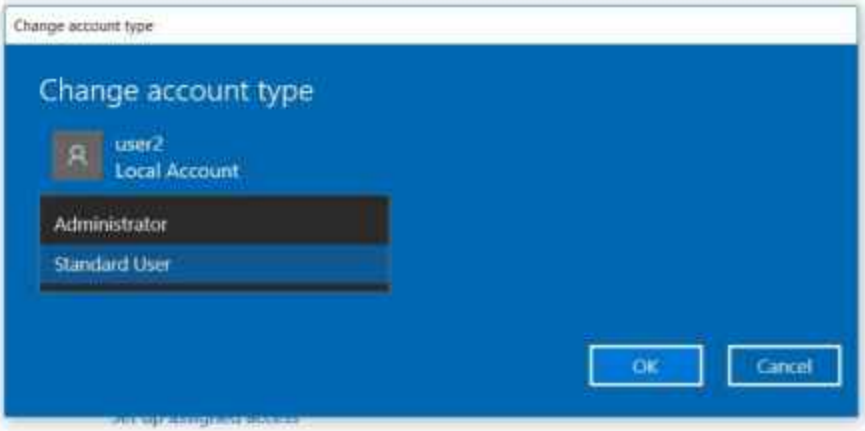


You can now begin to add a new account by selecting 'Add someone else to this PC'. This will display a prompt where you set the username and password for the new account. When creating an account this way that's all you can set; you can't choose what type of account it will be while creating it. The screen will look like this. In this case, it has been pre-filled to create a 'user2' account that will be modified in the next step.



You will see the new user appear where there previously was none.

This new account will, by default, be a standard user account. This means it is limited in its abilities and can not make changes that will affect all users on the machine. To change this select 'Change account type' and you'll be presented with a screen to choose either 'Standard User' or 'Administrator'.



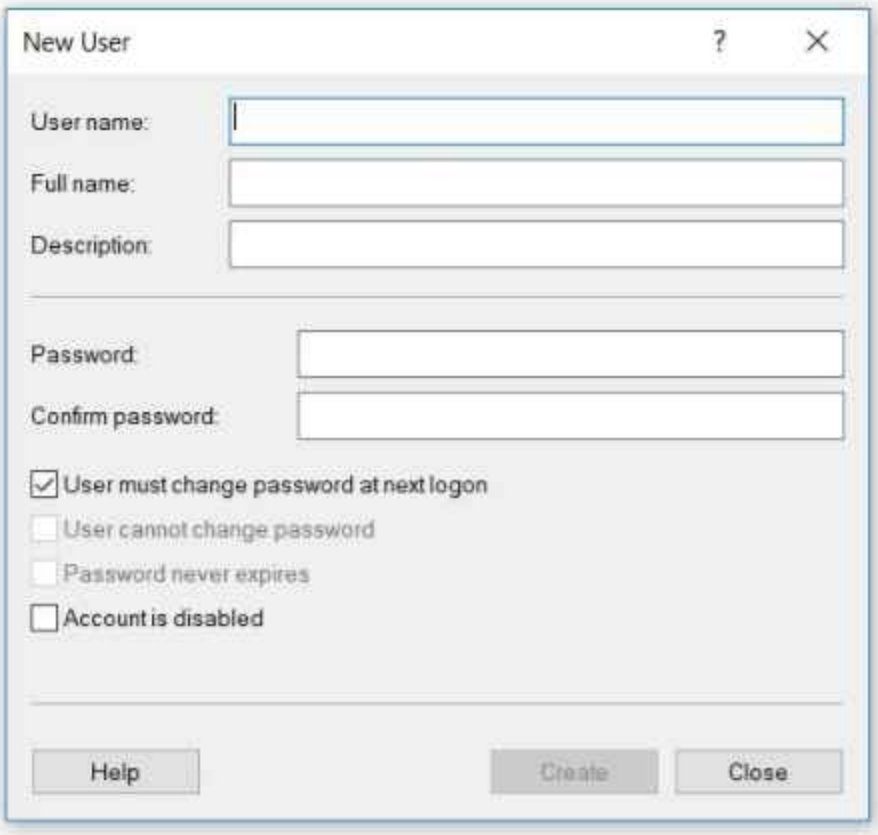
This screen also allows us to delete user accounts, this is done by selecting the 'Remove' option. Removing an account will also delete all data that was created on the machine by that user, so be careful to remove the correct account when doing this as recreating the user will not bring back any files.

Creating a User with the Management Console

Now let's take a look at a less simple, but more powerful way to create users: the Local Users Management Console. If you hit 'Start' and type 'lusrmgr.msc' you'll see the console suggested. It'll look like this:

After the console has loaded, if you select 'Users', you'll see all the users who currently have an account on the machine. Notice you can also see the default accounts mentioned earlier. This is where we can enable or disable these accounts.

Creating a user from this screen is as simple as right-clicking and selecting 'New User'. This will display the following screen:



You'll notice that this screen gives us a lot more choice when creating the account. We can force the user to change their password when they log in for the first time, make it so they can never change their password, or create the account as disabled. If you do not select the 'Password never expires' box, users created using this method will be forced to change their password every 30 days (this does not apply to users created using the control panel method above).

Groups

When managing users in Windows it can become rather unwieldy attempting to keep track of permissions and which level of access each account should have. Groups are the solution to this: a user can be added to one or more groups and will inherit their permission from the group itself, no need to set the permissions for each user.

You can create your own groups if needed, however Windows comes with several built in that are suitable for most needs initially, these are:

- **Administrators**: Any members of this group have full access to the computer. They can access all files and make changes that affect all users, such as installing new software or changing system settings. The Administrator account is a default member of this group.
- **Users**: This is the default group new standard users will be assigned to. Members of this group can perform most common tasks like running applications and using printers, although they are not able to install any new programs or make similar changes to the system.
- **Remote Desktop Users**: If you wish to be able to connect to the computer remotely, you will need to be a member of this group. Remote desktop sessions cannot be created unless you're a member of the group. Note that this doesn't mean you can't use remote desktop on this machine, just that you can't connect to it.
- **Guests**: Any members of the Guests group do not have permanent profiles on the machine. Each time they log in a new profile is created for them and at log off the profile is deleted. The Guest account is a default member of this group.

Creating new groups

To create a new group we need to revisit the management console. You can get there by heading to the start menu, typing 'lusrmgr.msc' and clicking the link when it appears.

Creating a new group is as easy as right clicking on the Groups folder and selecting 'New Group'. You'll be prompted to enter a name and description for the group, and to add any users, although only the name is mandatory at this stage.

Once the group is created you can then begin to add users and use it to manage permissions, which will be covered shortly.

User Account Control

User Account Control (UAC) was introduced by Microsoft in an attempt to make the Windows Operating System more secure. UAC works by assigning 'tokens' to the user when they log in. All users, whether they have a standard or administrative account, are given a standard token which is used to run applications with a limited level of access.

When an administrator wants to make changes to the system the screen is dimmed and a prompt is shown asking them to verify the request. If this is accepted a special administrative token is released and used to run the application with all the powers that come with being an administrator. The tokens, and the permissions that are associated with them, are only valid as long as the application is running and a new request is required next time the access is needed.

UAC Prompts

As previously mentioned, if the user who tries to carry out a task is an administrator they are presented with a prompt to approve the action, it looks much like this:

The prompt shows which application is requesting the elevated privileges and asks the user to confirm the action. But what if the user is not a member of the Administrators group? In that case a prompt as below will be displayed:

Before the application can run with administrative access, an administrator is required to enter their username and password. This is because a standard user does not have the administrative token to pass to the application.

UAC Levels

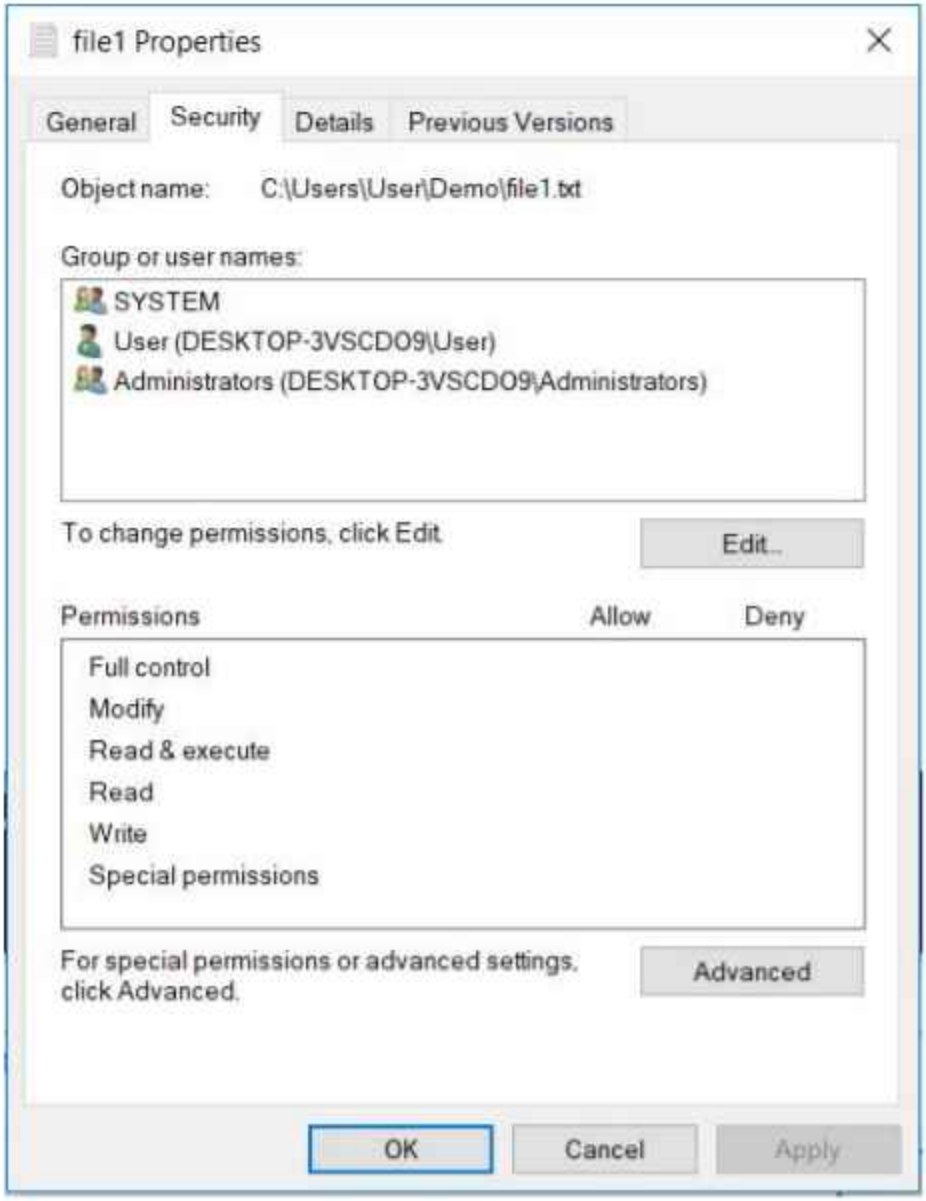
You can configure UAC to be more, or less, permissive. If you search 'UAC' on the start menu and select 'Change User Account Control Settings' you will be presented with four options.

The options, and what they mean for UAC are:

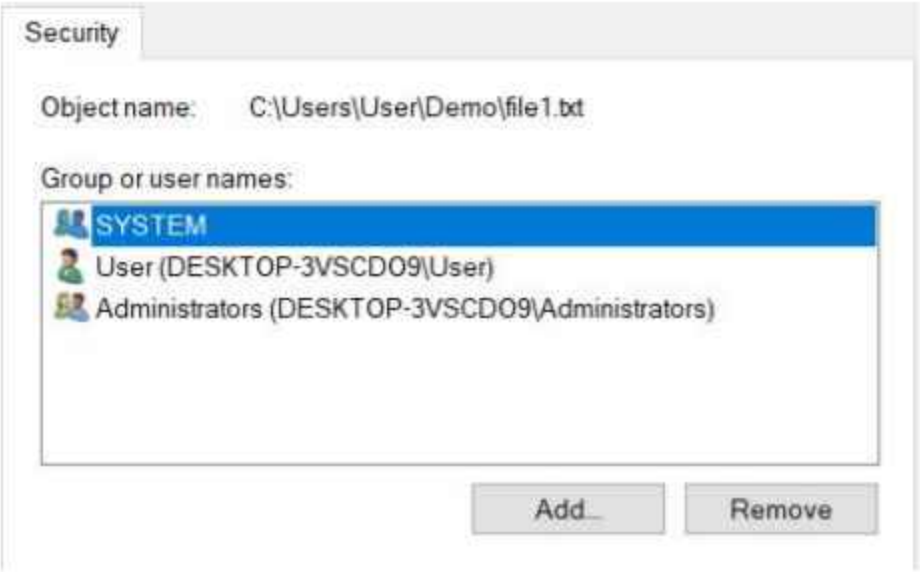
- **Always notify**: The UAC prompt is shown when apps try to install software or make changes to your computer and when you try to change Windows settings. The desktop is dimmed when a UAC prompt is shown.
- **Notify me only when apps try to make changes to my computer**: This is the default setting for UAC. UAC prompts aren't shown when you try to make changes to Windows settings, but are shown when attempting to install software or run as administrator. The desktop is dimmed when a UAC prompt is shown.
- **Notify me only when apps try to make changes to my computer (do not dim my desktop)**: This is the same as above but the desktop isn't dimmed when a UAC prompt is shown.
- **Never notify**: This is the equivalent of turning off UAC. Although UAC will still be active, all requests from administrators will be automatically approved while requests from standard users will be denied without showing a prompt in either case.

File Permissions

Windows has a very granular permissions system for files and folders. To manage this each file or folder has an Access Control List (ACL) which stores who is allowed which level of access. To access this list you need to right click the file and select 'Properties', then once the new window opens head to the Security tab, which will look like this:



At the top of the window we can see a list of users or groups that have been added to the ACL for this file. Selecting one of the items from this list will populate the bottom section with their permissions. You can also edit the users or groups in the ACL by hitting the edit button and then when the screen below is displayed, adding or removing to the list.



All permissions can either be allowed or denied. It's worth mentioning that a deny will always override an allow. For example if a user is a member of a group that is allowed to access a file, and also a member of another which is not, they will always be denied access. It's worth noting that the default action when a user does not have permissions set is to deny all.

The permissions available for files and folders are broadly the same. For files you are able to set the following permissions:

- **Full Control:** gives you all available permissions for the file.
- **Modify:** allows you to read, write, modify and execute the file.
- **Read & Execute:** allows you to display the file's contents, and run the file if it's a program.
- **Read:** allows you to open the file, and view its contents.
- **Write:** allows you to write data to the file.

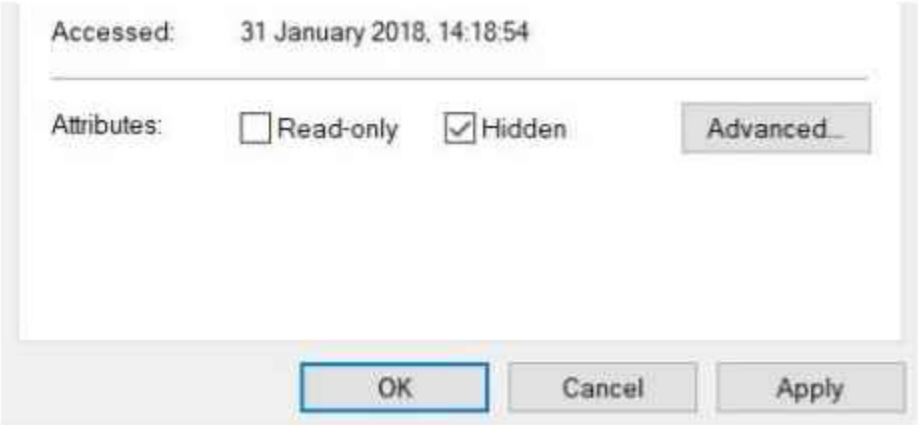
Similarly for folders, these are the options you have at your disposal:

- **Full Control:** gives you all available permissions for the folder and its contents.
- **Modify:** allows you to read, write, modify, and execute files in the folder.
- **Read & Execute:** allows you to display the folder's contents, the contents of files inside, and run any programs in the folder.
- **List Folder Contents:** allows you to display the folder's contents, and the contents of files inside.
- **Read:** allows you to open the folder, and view any files or subfolders.
- **Write:** allows you to add new files or subfolders.

You might notice there's some overlap there. That's because you need both Read & Execute and List Folder Contents to execute files within a folder. This is because files within a folder inherit their permissions from the folder, and files can't have the List Folder Contents permission, so must inherit Read & Execute.

Hidden Files

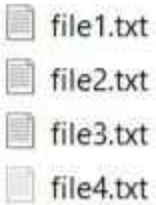
In Windows when we hide a file, the 'hidden' attribute gets added, these files then no longer show up when viewing folders. You can set a file or folder as hidden by right clicking on it and selecting 'Properties'. Once the properties window has popped up, check the Hidden box and hit OK. The properties window will look like this:



Once you have done this, unless you have set Explorer to show hidden files, you'll no longer see this file. It will still exist, and can be edited or used as normal, you just can't see it.

To show hidden files you need to head into folder options, this can be found on the ribbon at the top of the explorer window under 'View'. Once this is open, select the 'View' tab and change the option to show hidden files and folders like so:

Any hidden files will then show up, but with a translucent icon to show that they are hidden. In the image below "file4.txt" is hidden, and as you can see has a different appearance to the other files.



The Windows CLI

Contents

In this module, we will be covering:

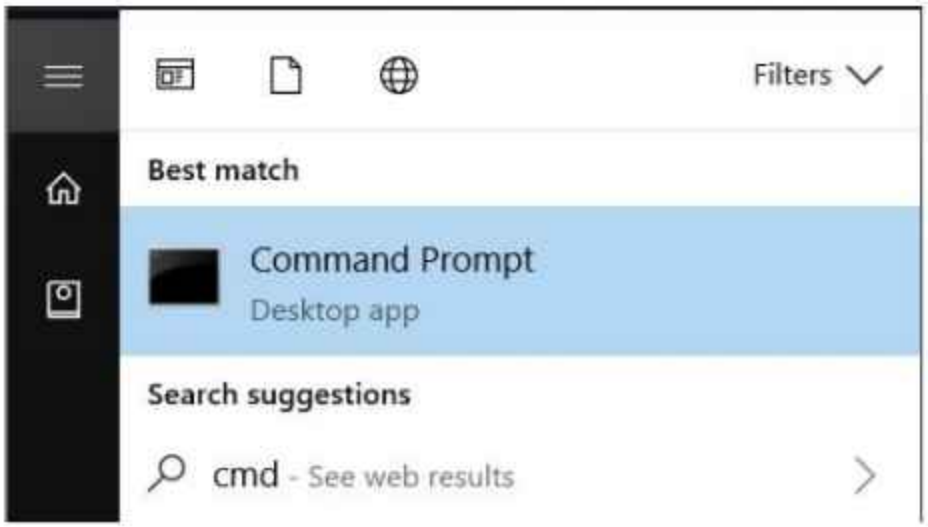
- The Windows Command Line
- Command Line Navigation
- Command Line Commands
- Command Line Networking
- Command Line User Management

What is the Command Prompt

The Command Prompt is a command line interpreter (CLI) application available on most Windows operating systems. We use the Command Prompt to, wait for it... execute entered commands. Most of the time it's used to automate tasks by running scripts or batch files, carry out administrative tasks, and troubleshoot and solve issues. The Command Prompt is officially called the "Windows Command Processor" but is generally referred to as the command shell or prompt, and sometimes by its filename cmd.exe. You might occasionally hear it called "the DOS prompt" or as MS-DOS itself. Command Prompt is a Windows program that emulates much of what was available in MS-DOS but it is not actually MS-DOS.

Accessing the Command Prompt

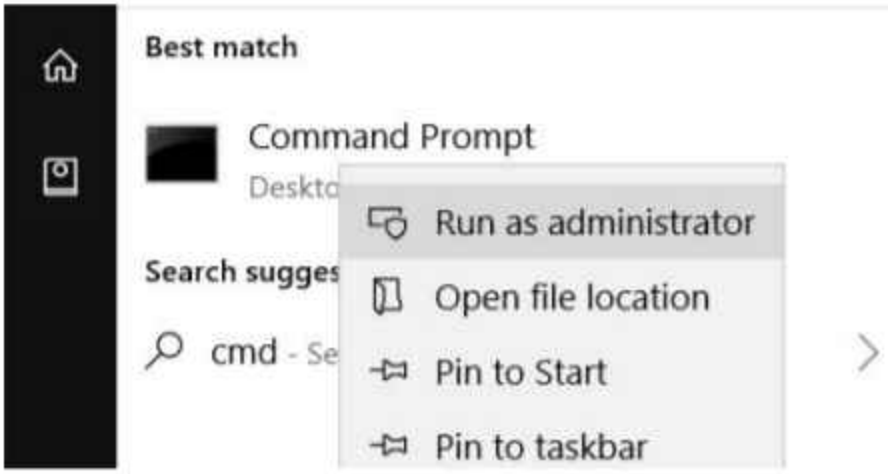
There are many ways to open the prompt; the easiest being to hit start and type cmd. This will search for and suggest the Command Prompt. As below, clicking the suggestion will open the Prompt.



Notice that when we open the prompt this way, even if our user account is an administrator, the prompt is opened as a standard user.

```
Microsoft Windows [Version 10.0.16299.192]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\User>
```

If you need an administrative prompt you need to explicitly request it. You can do so by right clicking on the suggested link:



This will (depending on settings) trigger a User Account Control prompt, to confirm you want administrative level access, before opening a new prompt as an administrator. Notice that the prompt does not default to the user's home directory, but instead to `C:\Windows\System32` and the title bar of the prompt clearly labels it as being run with administrator privileges.

```
Microsoft Windows [Version 10.0.16299.192]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Windows\system32>
```

As always, you should only drop into an administrative prompt if absolutely necessary, and once you've completed your task be sure to return to being a standard user.

Changing directory

The first command you're going to need is `cd`, this enables you to change directory (move to a different folder). For example `cd \` will take you to the root of the file system, which in this case is `C:\`.

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.16299.192]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\User\Documents>cd \
C:\>
```

You can use tab completion to make typing directory names easier. Just start typing the name you want and hit tab on your keyboard. A suggestion will appear and complete your directory. You can continue to hit tab to cycle through suggestions.

It's worth noting that Command Prompt is not case sensitive, it doesn't matter whether you use `CD`, `cd` or `Cd`: all are perfectly valid. The same goes for file names.

Changing drives

Your machine is likely to have more than one drive available, either a second hard drive or possibly a connected USB device. To access another drive, we just type that drive's letter, followed by ":". For instance, if you wanted to change the drive from "C:" to "D:", you should type "d:" and then hit enter on your keyboard. You'll notice the drive letter on the prompt change to the new drive, as below:

```
C:\Users\User\Documents>D:

D:\>
```

If you know the directory you need to go to on the new drive you can navigate straight to it by using `cd` with the `/D` switch. This allows you to go to a specific folder on the new drive, so for example if we wanted to get from the D: drive to C:\Windows we could use `cd /D C:\Windows` and we'll go straight there:

```
D:\>cd /D C:\Windows

C:\Windows>
```

Viewing directory contents

It's all well and good being able to navigate the file system, but you probably want to be able to see what's inside the directories. We can do this using the `dir` command.

The `dir` command is used to show files and directories in the current directory. The command also shows the last modification date and time, as well as the file size. When you run `dir` the results will be like the following:

```
C:\Users\User\Demo>dir
Volume in drive C has no label.
Volume Serial Number is DEF5-A66E

Directory of C:\Users\User\Demo

30/01/2018 13:12    <DIR>          .
30/01/2018 13:12    <DIR>          ..
30/01/2018 12:42    <DIR>          dest_dir
30/01/2018 13:12                14 file1.txt
30/01/2018 12:42    <DIR>          source_dir
               1 File(s)             14 bytes
               4 Dir(s)  36,092,047,360 bytes free
```

By default `dir` does not show hidden files and folders. To do this we need to add a switch to the command, we'll use the `/A` switch which is used to set file attributes to display. There are filters to narrow down what's shown here but for now we're happy to be shown everything. Running the command again, this time as `dir /A` results in the following output, showing the previously hidden file:

```
C:\Users\User\Demo>dir /A
Volume in drive C has no label.
Volume Serial Number is DEF5-A66E

Directory of C:\Users\User\Demo

30/01/2018 13:12    <DIR>          .
30/01/2018 13:12    <DIR>          ..
30/01/2018 12:42    <DIR>          dest_dir
30/01/2018 13:12                14 file1.txt
30/01/2018 13:12                14 hidden_file.txt
30/01/2018 12:42    <DIR>          source_dir
               1 File(s)             14 bytes
               4 Dir(s)  36,092,047,360 bytes free
```

Common Command Prompt commands

We're going to take a look at some of the more useful commands when working in the Windows Command Prompt. Using the commands covered here you should be able to get by navigating and working with the Command Prompt for most day to day tasks.

mkdir

The 'mkdir' command is used to create a directory. This one is nice and straightforward, you use it like this:

```
C:\Users\User\Demo>mkdir new_dir

C:\Users\User\Demo>dir

 Volume in drive C has no label.
 Volume Serial Number is DEF5-A66E

 Directory of C:\Users\User\Demo

30/01/2018 14:08    <DIR>          .
30/01/2018 14:08    <DIR>          ..
30/01/2018 13:07    <DIR>          dest_dir
30/01/2018 13:12                14 file1.txt
30/01/2018 14:08    <DIR>          new_dir
30/01/2018 12:42    <DIR>          source_dir
               1 File(s)             14 bytes
               5 Dir(s)  36,090,421,248 bytes free
```

You can also use a full path when creating a folder; in this case the command would look like this:

```
mkdir C:\Users\User\mynewfolder
```

'mkdir' will create any missing folders along the way, so don't worry if they don't exist, you don't need to do them one at a time.

copy

Copy allows us to copy one or more files to a different location, while leaving the original file untouched. Using copy is as simple as:

`copy <source> <destination>`

It looks like this:

```
C:\Users\User>copy file1.txt file2.txt
   1 file(s) copied.

C:\Users\User>
```

We can also use it to copy multiple files into a new folder, like so:

```
C:\Users\User>copy *.txt .\copy_target_dir
file1.txt
file2.txt
   2 file(s) copied.

C:\Users\User>
```

One thing we can't do with 'copy' is copy a directory. Strange, right? If you attempt this what will actually happen is the files from within the directory itself are copied, the root folder and any subdirectories are left behind. This is because 'copy' is not capable of handling directories.

```
C:\Users\User>copy copy_target_dir new_dir\
copy_target_dir\file1.txt
copy_target_dir\file2.txt
   2 file(s) copied.

C: \Users\User>dir new_dir
Volume in drive C has no label.
Volume Serial Number is DEF5-A66E

Directory of C: \Users\User\new_dir

30/01/2018 12:01    <DIR>      .
30/01/2018 12:01    <DIR>      ..
30/01/2018 11:51            10 file1.txt
30/01/2018 11:51            10 file2.txt
               2 File(s)  20 bytes
               2 Dir(s) 36,093,161,472 bytes free
```

To do this we need to use 'robocopy'.

robocopy

Robocopy is a command that allows you to copy files, directories, and even entire drives from one location to another. It's more robust than just plain old 'copy', and allows you to do far more.

To copy an entire directory tree, you need to use the '/S' switch. This copies the directory and all subdirectories, excluding any empty directories. If you also want to copy empty directories it's the '/E' switch.

```
C:\Users\User\Demo>robocopy source_dir dest_dir /s

------------------------------------------------------------------------------
   ROBOCOPY     ::  Robust File Copy for Windows
------------------------------------------------------------------------------

Started : 30 January 2018 22:34:43
Source  : C:\Users\User\Demo\source_dir\
Dest    : C:\Users\User\Demo\dest_dir\

Files   : *.*

Options : *.* /S /DCOPY:DA /COPY:DAT /R:1000000 /W:30

------------------------------------------------------------------------------

New Dir 1   C: \Users\User\Demo \source_dir\
100%   New File   9   file4.txt
New Dir 0   C:\Users\User\Demo\source_dir\inner\
New Dir 0   C: \Users\User\Demo\source_dir\inner_dir\

------------------------------------------------------------------------------

        Total   Copied  Skipped Mismatch    FAILED  Extras
Dirs  :     3     3     0     0        0     0
Files :     1     1     0     0        0     0
Bytes :     9     9     0     0        0     0
Times :  0:00:00 0:00:00                0:00:00 0:00:00

Speed   :  600 Bytes/sec.
Speed   :  0.034 MegaBytes/min.
Ended : 30 January 2018 22:34:43
```

You may notice that there's a lot more output about the progress of the command and the eventual outcome. Be sure to check `robocopy /?` for all options.

move

The 'move' command is used to move one or more files from one location to another. Unlike 'copy' when using 'move' we can move directories without any extra commands. We use move with the following syntax:

```
move <source> <destination>
```

That's really all there is to it.

```
C:\Users\User\Demo>move source_dir dest_dir
   1 dir(s) moved.

C: \Users\User\Demo>dir dest dir\source_dir
Volume in drive C has no label.
Volume Serial Number is DEF5-A66E

Directory of C:\Users\User\Demo\dest_dir\source_dir

30/01/2018 12:42    <DIR>      .
30/01/2018 12:42    <DIR>      ..
30/01/2018 12:41              10  file1
30/01/2018 12:42              10  file2

          2 File(s)         20 bytes
          2 Dir(s) 36,092,719,104 bytes free
```

del

When we want to delete a file, we use 'del'. Be careful with this one though as it does not prompt for confirmation before removing files, and once they're gone, they're gone. Using the del command is another nice easy one, to delete a file you just do the following:

`del <filename>.`

You can also use wildcards when deleting, for example:

`del *.txt`

This will delete all files with the .txt extension in the current folder, but again be careful you don't delete too much by accident. It's often a good idea to use '/P' which will prompt before deleting files unless you're absolutely sure you don't need them. The prompt for deletion looks like this:

```
C:\Users\User\Demo>del /P *.txt
C:\Users\User\Demo\file.txt, Delete (Y/N)? n
```

Much like the 'copy' command, we can't delete directories with this. It will delete all files within the directory itself without removing the folder once done. To do this we need 'rmdir'.

rmdir

To remove directories from the command line we need to use the 'rmdir' command. This command is essentially the opposite of 'mkdir' we saw earlier, we use it like this:

```
rmdir <directory>
```

You don't get any output from this command, the only way to know that it worked is to check the folder has gone. It's pretty safe to assume that if no errors were produced that it worked, but let's check:

```
Volume in drive C has no label.
Volume Serial Number is DEF5-A66E

Directory of C:\Users\User\Demo

30/01/2018 14:15    <DIR>          .
30/01/2018 14:15    <DIR>          ..
30/01/2018 14:10    <DIR>          a
30/01/2018 13:07    <DIR>          dest_dir
30/01/2018 14:08    <DIR>          new_dir
30/01/2018 12:42    <DIR>          source_dir
               0 File(s)              0 bytes
               6 Dir(s) 36,090,220,544 bytes free

C: \Users\User\Demo>rmdir new_dir

C: \Users\User\Demo>dir /A:D
Volume in drive C has no label.
Volume Serial Number is DEF5-A66E

Directory of C:\Users\User\Demo

30/01/2018 14:15    <DIR>          .
30/01/2018 14:15    <DIR>          ..
30/01/2018 14:10    <DIR>          a
30/01/2018 13:07    <DIR>          dest_dir
30/01/2018 12:42    <DIR>          source_dir
               0 File(s)              0 bytes
               5 Dir(s) 36,090,220,544 bytes free
```

Something to note is that if the target is not empty by default the command will fail. You can override this by passing the '/S' switch, which will delete all files and sub-directories, deleting an entire directory tree. Obviously, as with any deletion, this is something that should be done with extreme caution.

```
C:\Users\User>rmdir Demo
The directory is not empty.

C:\Users\User>rmdir /S Demo
Demo, Are you sure (Y/N)? y

C:\Users\User>
```

more

The 'more' command is used to display the contents of a file one page at a time. It works much the same as the more command from Linux. A good use of 'more' is where a command has a large output and you wish to page through it.

Here we have used the 'dir' command to list a directory with a large number of things inside: by piping the output to 'more' we can page through it by hitting enter. Notice at the bottom of the output there is "-- More --" shown: this lets us know there is more text to come.

```
C:\Users\User\Demo>dir "c: \Windows" | more
Volume in drive C has no label.
Volume Serial Number is DEF5-A66E

Directory of c:\Windows

24/01/2018 10:38    <DIR>          .
24/01/2018 10:38    <DIR>          ..
29/09/2017 13:46    <DIR>          addins
04/01/2018 04:11    <DIR>          appcompat
24/01/2018 02:09    <DIR>          apppatch
30/01/2018 10:35    <DIR>          AppReadiness
...
30/09/2017 14:35    <DIR>          Help
29/09/2017 13:41           976,896 HelpPane.exe
-- More --
```

find

'find' searches inside files for a specified string of text. After searching the files, 'find' displays any lines of text that contain the search string. This is useful for quickly searching the contents of files when looking for something specific. Like most commands we've seen you can also use wildcards in your search.

Here you can see the result of:

```
find C:\Users\User\* "hello" 2>nul
```

```
C:\Users\User\Demo>find c:\Users\User\* "hello" 2> nul

---------- C:\USERS\USER\#

---------- C:\USERS\USER\FILE1.TXT
"hello"

---------- C:\USERS\USER\FILE2.TXT
"hello"

---------- C:\USERS\USER\NEWDIR
"hello"
"hello"

---------- C:\USERS\USER\NEWDIR.TXT
"hello"
"hello"
```

This command is looking at all files in the "User" directory and checking them for the string "hello". You might notice the `2>nul` at the end, this has been used to hide error messages, as without it any access denied messages will also be displayed. This was not desirable in this case.

where

The 'where' command is roughly equivalent to the 'which' command in Linux, we can use it to locate files on a computer. The command searches the current directory and any directories listed in the PATH variable by default so can be handy when looking for the location of an executable.

Here you can see a search for text files where the name starts with "file":

```
C:\Users\User\Demo>where "file*.txt"
C:\Users\User\Demo\file1.txt
C:\Users\User\Demo\file2.txt
C:\Users\User\Demo\file3.txt
```

The default behaviour can be changed by using the '/R' switch and specifying a directory; this will cause 'where' to search this directory and all sub directories for the file. Notice that adding the '/R' switch to the command revealed a new file missed before because it is not in the current directory or PATH.

```
C:\Users\User\Demo>where /R . "file*.txt"
C:\Users\User\Demo\file1.txt
C:\Users\User\Demo\file2.txt
C:\Users\User\Demo\file3.txt
C:\Users\User\Demo\source_dir\file4.txt
```

Command line networking

One of the great things about the Windows Command Prompt is that it makes some tasks much quicker. A great example of this is checking the network settings for the local machine. By opening up Command Prompt and typing `ipconfig` you can access the current settings. It'll look something like this:

```
C:\Users\User\Demo>ipconfig

Windows IP Configuration

Ethernet adapter Ethernet0:

   Connection-specific DNS Suffix  : localdomain
   Link-local IPv6 Address         : fe80::218b:93c8:5647:3d61%5
   IPv4 Address.                   : 172.16.16.129
   Subnet Mask                     : 255.255.255.0
   Default Gateway                 :

Ethernet adapter Ethernet1:

   Connection-specific DNS Suffix  : localdomain
   Link-local IPv6 Address         : fe80::4d2f:4374:5c:3027%3
   IPv4 Address.                   : 192.168.196.135
   Subnet Mask                     : 255.255.255.0
   Default Gateway                 : 192.168.196.2

Ethernet adapter Bluetooth Network Connection:

   Media State                     : Media disconnected
   Connection-specific DNS Suffix  :
```

'ipconfig' is a command line utility that's available on all versions of Windows. This utility allows you to retrieve your network configuration and also allows some control over active TCP/IP connections.

Setting IP

You may encounter a situation where you are not given an IP address on a network, this could be because the network does not automatically assign them, or possibly there's a fault. In any case, you're going to need an IP to communicate on the network.

We can achieve this from the command line by using the 'netsh' utility. The 'netsh' command is used to start Network Shell, which can then be used to manage the network settings of either the local, or a remote, computer.

To set the IP of a machine, you would run the following command in an administrative Command Prompt (the change affects all users, so that's an admin task):

```
netsh interface ip set address <connection name> static <IP>
<subnet> <gateway>
```

This looks quite daunting at first, but you'll get the hang of it. Here's what the command will look like:

```
netsh interface ip set address "Ethernet0" static 172.16.16.150 255.255.255.0
172.16.16.1
```

Lets just verify that worked by checking 'ipconfig' again. We should now have the IP 172.16.16.150.

```
C:\Windows>ipconfig
Windows IP Configuration

Ethernet adapter Ethernet0:

   Connection-specific DNS Suffix  :
   Link-local IPv6 Address . . . . . : fe80::218b:93c8:5647:3d61%5
   IPv4 Address. . . . . : 172.16.16.150
   Subnet Mask . . . . . : 255.255.255.0
   Default Gateway . . . . . : 172.16.16.1
```

Accessing network resources

Now that we've seen how to manage the interface on your machine, you may want to access files stored on the network. We can mount shared folders by using the 'net' command. This command is used to manage almost all aspects of a network and its settings including shares, print jobs and users. Run `net /?` for a list of all the commands possible.

To mount a network drive you need to use 'net use', and you'll need to use this any time you want to view mounted shares, add a new share or manage existing ones. We're going to execute the following command:

```
net use x: \\DESKTOP-3VSCD09\Share
```

This will mount the share on the named machine (the bit after the '\'), to the 'X:' drive on the local machine. We can then navigate to the 'X:' drive.

```
C:\Users\User\Demo>net use x: \\DESKTOP-3VSCD09\Share
The command completed successfully.

C:\Users\User\Demo>X:
X:\>
```

To remove the mounted drive we use the '/delete' switch like so:

```
net use x: /delete
```

This will remove the shared folder from your machine and release the X: drive to be used for something else.

```
C:\Users\User\Demo>net use x: /delete
x: was deleted successfully.

C:\Users\User\Demo>X:
The system cannot find the drive specified.
```

Command line user management

You've already seen that using the command line can make tasks quicker. Another great time to head to the terminal is when you need to manage users on a machine. Using the 'net' command seen in the previous section, it is also possible to manage just about everything about a user account. You can view all user accounts currently on a machine by runningnet user, which will display a table of current users.

```
C:\Windows\system32>net user

User accounts for \\DESKTOP-3VSCDO9

-------------------------------------------
Administrator   DefaultAccount  Guest
User            user2           WDAGUtilityAccount
The command completed successfully.
```

You might notice the following commands fail if you're running a standard user Command Prompt. This is another case of system wide changes and means you'll need an administrative session to manage users.

Adding a new user

Adding a new user is as simple as running:

```
net user /add <username> <password>
```

This command will create a new user and set the password to what you specified. You can also use * in place of a password and you will be prompted to enter the password on a new line (where it will not be displayed, which is good). Adding a user looks like this:

```
C:\Windows\system32>net user /add user3 *
Type a password for the user:
Retype the password to confirm:
The command completed successfully.
```

You can now verify that the user was created by runningnet user again. The new user account will be visible in the table.

```
C:\Windows\system32>net user

User accounts for \\DESKTOP-3VSCDO9

-------------------------------------------
Administrator      DefaultAccount  Guest
User               user2           user3
```

```
WDAGUtilityAccount
The command completed successfully.
```

Your new user account is now available to be logged in to for the first time using the password you set during creation. You can do so much more when creating users this way, such as configuring account expiry and setting times where the account can log on. Check out `net user /?` for a full list of options.

Removing a user

Removing a user is just as simple as adding one, and once again you're going to use the trusty 'net' command to do it. Once you have identified the account to be removed, doing so is as simple as running:

```
net user /delete <username>
```

As with all deletions, be careful here as you'll get no warnings. You're running as an administrator so the system trusts you know what you're doing.

```
C:\Windows\system32>net user /delete user3
The command completed successfully.
```

Just like adding a user, you can verify the account has been deleted by checking the list of users in `net user` to ensure the account has been removed.

Changing user groups

The 'net' utility is also where you go to manage the groups a user is a member of. To do this you will need to use `net localgroup`. When you create a new user, by default they are just a standard user, so to gain higher privileges they need to be added to the Administrators group.

To check current group membership run:

```
net user <username>
```

This will print details about that account like below.

```
C:\Windows\system32>net user user3
User name                    user3
Full Name
Comment
User's comment
Country/region code          000 (System Default)
Account active               Yes
```

```
Password last set                    30/01/2018 19:52:06
Password expires          Never
Password changeable                  30/01/2018 19:52:06
Password required         Yes
User may change password             Yes

Workstations allowed                 All
Logon script
User profile
Home directory
Last logon                           Never

Logon hours allowed                  All

Local Group Memberships              *Users
Global Group memberships    *None
The command completed successfully.
```

Looking towards the bottom of the output you can see the user is currently only a member of the Users group. To add them to the Administrators group you need to run:

```
net localgroup Administrators /add user3
```

```
C:\Windows\system32>net localgroup Adminstrators /add user3
The command completed successfully.
```

Running the`net user user3` command again shows us the user is now a member of the Administrators group.

```
C:\Windows\system32>net user user3
User name                 user3
Full Name
Comment
User's comment
Country/region code       000 (System Default)
Account active            Yes
Account expires           Never

Password last set         30/01/2018 19:52:06
Password expires          Never
Password changeable       30/01/2018 19:52:06
Password required         Yes
User may change password  Yes

Workstations allowed      All
Logon script
User profile
Home directory
Last logon                Never

Logon hours allowed       All

Local Group Memberships   *Administrators *Users
Global Group memberships  *None
The command completed successfully.
```

As with all previous commands, running`get localgroup /?` will get you more details on possible options when managing group membership this way.

Contents

PowerShell is extremely powerful and Microsoft's path forwards for a command line shell, scripting and administration.

In this module, we will be covering:

- PowerShell
- PowerShell Commands
- PowerShell Objects

PowerShell

As we have seen, the Windows Command Prompt is fairly powerful and we can do almost everything we need in there, but PowerShell is a tool that's much more powerful than the Command Prompt. One reason for this is that it's based on the .NET framework and includes a command line shell and a scripting language all in one package.

You can access PowerShell by hitting start and typing 'powershell'. If you do this you'll be presented with the command line interface for PowerShell, which looks a lot like Command Prompt, and in many ways it is. The PowerShell prompt looks like this:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.
PS C:\Users\User>
```

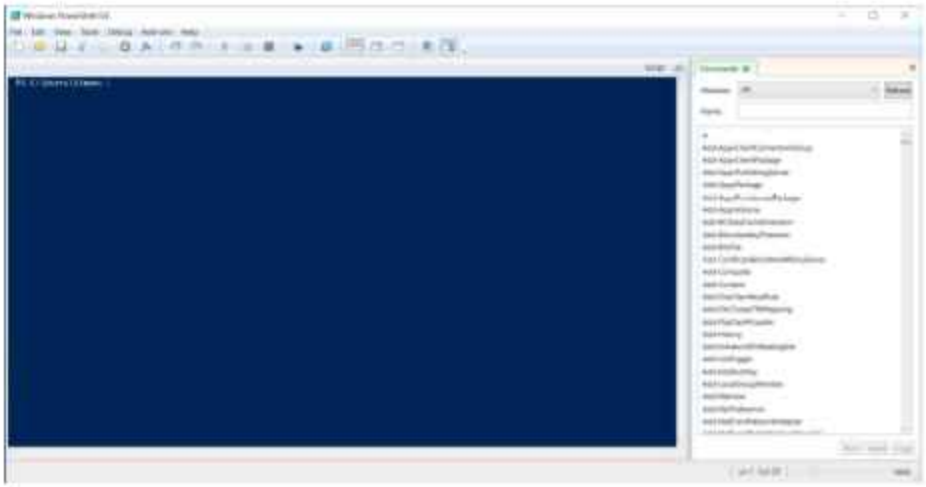
PowerShell Cmdlets

A cmdlet is a lightweight command that is used within PowerShell. A cmdlet generally does one very small thing and then usually returns the result to you as a .NET object. You can then continue to process the result either with more cmdlets, or by writing to a file or the screen if it's the final step. It's worth noting that cmdlets are not standalone executables, so you can't run them outside of PowerShell. They are actually instances of .NET code that is run by PowerShell itself.

PowerShell ISE

The PowerShell Integrated Scripting Environment (ISE) allows you to write, test and run PowerShell scripts. It includes tab completion for commands and a search function to allow you to find commands if you're unsure of their name. To open the PowerShell ISE go to start and type 'powershell'. One of the suggestions will be PowerShell ISE, the icon will look like this:

Once you click that the ISE will open and you'll be presented with a window like the one below:



This is the ISE. You can begin to create your script in the blue section of the window. The command search is shown on the right, and once you have written some commands you can test them by hitting 'run' on the command ribbon above.

Don't worry if you don't know any commands just yet, we're going to begin covering them in the next section.

PowerShell Commands

PowerShell commands use a 'verb-noun' naming system, so each cmdlet name is made up of two parts: a verb (Get, Start, Stop) and a noun (Service, Process, Date). This naming convention was chosen to help cut down on difficulty when attempting to remember commands, for example it's very clear what `Stop-Service` is going to do, and because of the common verb being used, it's fairly easy to work out what `Stop-Computer` will do (hint: it shuts down the computer).

In this section we're going to look at some common commands. We aren't going to come close to covering all the available cmdlets (a standard Windows installation comes with over 250 cmdlets installed), though by the end you should be familiar enough with them to begin exploring PowerShell on your own. To find out more about any PowerShell commands use the '-?' argument to show the help file.

Get-Command

The `Get-Command` cmdlet is a great place to start. Using this command we can find all the cmdlets that are, for example, using the 'Computer' noun. We do this by running:

```
Get-Command -Noun Computer
```

This returns all cmdlets where the noun is 'Computer', the output will be like below:

```
PS C:\Users\User> Get-Command -Noun Computer

CommandType Name                                Version Source
----------- ----                                ------- ------
Cmdlet      Add-Computer                        3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Checkpoint-Computer                 3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Remove-Computer                     3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Rename-Computer                     3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Restart-Computer                    3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Restore-Computer                    3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Stop-Computer                       3.1.0.0 Microsoft.PowerShell.Management
```

You can also search on the verb of the cmdlet, so if you wanted to know all the commands that allow you to stop something, you would run:

```
Get-Command -Verb Stop
```

And you would receive the output as a list like this:

```
PS C:\Users\User> Get-Command -Verb Stop
CommandType Name                    Version    Source
----------- ----                    -------    ------
Function    Stop-DscConfiguration   1.1        PSDesiredStateConfiguration
```

```
Function        Stop-Dtc                                    1.0.0.0         MsDtc
Function   Stop-DtcTransactionsTraceSession 1.0.0.0    MsDtc
Function        Stop-EtwTraceSession                        1.0.0.0         EventTracingManagement
Function   Stop-NetEventSession          1.0.0.0    NetEventPacketCapture
...
```

Get-ChildItem

The `Get-ChildItem` cmdlet is used to list the contents of a folder, and the output is very similar to running 'dir' in Command Prompt. What is very different is how easy it is to filter the output:

```
PS C:\Users\User\Demo> Get-ChildItem

    Directory: C:\Users\User\Demo
Mode       LastWriteTime  Length  Name
----       -------------  ------  ----
d-----  30/01/2018 22:28          dest_dir
d-----        30/01/2018 22:28                    source_dir
-a---  30/01/2018 15:07       8 file1.txt
-a---        30/01/2018 15:07                    8 file2.txt
-a---  30/01/2018 15:07      16 file3.txt
-a---        30/01/2018 16:14                   10 new
```

Get-Content

The `Get-Content` cmdlet is used to display the contents of a file. It works much the same as 'cat' in Linux and, in fact it has 'cat' as an alias.

```
PS C:\Users\User\Demo> Get-Content \file1.txt
"one"
```

Get-Process

`Get-Process` is used to gather information about running processes on the machine.

```
PS C:\Users\User\Demo> Get-Process
Handles NPM(K)  PM(K)   WS(K)  CPU(s) Id     SI ProcessName
------- ------  -----   -----  ------ --     -- -----------
299      18  8936   24372   0.13   3540   1 ApplicationFrameHost
256          13   4572      18416       2.47      3280       1 conhost
513      19  1600   4864         452   0 csrss
346          16   1636      4848                    556       1 csrss
346      15  2816   12824   0.17   5280   1 ctfmon
...
```

Start-Process

We can use `Start-Process` to start running, you guessed it, a new process. If you want to start an application running on a machine at a certain point in a script, this is the cmdlet you need. If you wanted, for example, to open notepad the command to run would be:

```
Start-Process -FilePath "notepad"
```

This is a fairly simple example and the options available are vast. You can configure the window style, which user to run the application as and even request to run as admin. Check out all the options by using `Start-Process -?` to display the help file.

Stop-Process

`Stop-Process` is what you would use to end a process. You can either stop a process using its ID or its name. To stop a process by its ID, first you need to find it. To stop the notepad process from the previous example, we find the ID by running:

```
Get-Process -Name notepad
```

This gives us the following output:

```
PS C:\Users\User\Demo> Get-Process -Name Notepad
Handles NPM(K)  PM(K)   WS(K)  CPU(s)  Id    SI  ProcessName
------- ------  -----   -----  ------  ----  --  -----------
235      14  3104   28424  0.16   6056  1   notepad
```

Now that we have the ID (6056) we can stop the process. To do this we run:

```
Stop-Process -ID 6056
```

The process will be immediately terminated. The alternative, if we have a lot of processes to stop, would be to close by name. In this case we can even use wildcards, so:

```
Stop-Process -Name note*
```

This will terminate any process that starts with "note**Be careful with this one.** If stopping by name, it's often a good idea to use the '-Confirm' argument, which will ask for confirmation before terminating each process that matches.

Aliases

While a lot of the cmdlets above seem to have rather long names compared to their Command Prompt or Bash alternatives, this doesn't have to be the case. PowerShell comes with several aliases set up for common cmdlets. The most useful are probably:

- **cat** is `Get-Content` used to display file contents
- **cd** is `Set-Location`used to change directory
- **dir** is `Get-ChildItem` used to list directory contents
- **ls** is `Get-ChildItem` used to list directory contents
- **rm** is `Remove-Item` used to delete files or folders

If you want to check out what other commands have an alias set, or if you want to set your own, you can do so by using the `Get-Alias` and `Set-Alias` cmdlets.

Running `Get-Alias` will print a list of all currently set aliases, or if you want to see which command is behind the alias you can use the '-Name' argument like this:

```
PS C:\Users\User\Demo> Get-Alias -Name cat
CommandType Name                                       Version Source
----------- ----                                       ------- ------
Alias       cat -> Get-Content
```

Setting an alias is done by using the following arguments:

```
Set-Alias -Name <alias_name> -Value <cmdlet_to_run>
```

So if you wanted to add "list" as an alias of `Get-ChildItem` the command would look like:

```
Set-Alias -Name list -Value Get-ChildItem
```

PowerShell Objects

Unlike traditional command-line interface commands, Windows PowerShell cmdlets are designed to deal with objects. These are much more than the text output you're more likely to be used to, as they contain a bunch of information about the result, and some functions you can call to display or manipulate that data. You'll quickly find that command output in PowerShell always carries along extra information that you can use if you need it.

We can check what information is stored inside the result of a command by using the `Get-Member` cmdlet. You can see an example of this being run below:

```
PS C:\Users\User\Demo> Get-ChildItem | Get-Member
   TypeName: System.Io.DirectoryInfo
Name                 MemberType          Definition
----                 ----------          ----------
LinkType             CodeProperty        System.String LinkType{get=Ge...
Mode          CodeProperty   System.String Mode{get=Mode;}
Target               CodeProperty        System.Collections.Generic.IE...
Create       Method       void Create(), void Create(S...
CreateObjRef         Method              System.Runtime.Remoting.ObjRe...
...
Delete               Method              void Delete(), void Delete(b...
```

Notice that there are Properties and Methods. Properties contain information about the object, in this case, as it was used against `Get-ChildItem`, we get things like time last accessed, and the parent and root directories. The Methods allow us to manipulate the object (and the data it represents). You can see a Delete Method in the example above, using that Method will delete the directory that this object represents.

Storing Objects

Just as in other commandline interfaces, we can store the result of a command in a variable. To do that in PowerShell we declare the variables using a $ symbol. Staying with the example of `Get-ChildItem` from above you can store the result in a variable by running the following command:

```
$child_items = Get-ChildItem
```

We can then access this result using the variable, as you can see below:

```
PS C:\Users\User\Demo> $child_items = Get-ChildItem
PS C:\Users\User\Demo> $child_items

    Directory: C:\Users\User\Demo
Mode       LastWriteTime  Length  Name
----       -------------  ------  ---
d-----  30/01/2018 22:28          dest_dir
d-----  30/01/2018 22:28          source_dir
-a---   30/01/2018 15:07       8  file1.txt
-a---   30/01/2018 15:07       8  file2.txt
-a---   30/01/2018 15:07      16  file3.txt
-a---   30/01/2018 16:14      10  new
```

We can also use the Methods and Properties from the object through the variable, This is done by using a command like this:

```
$child_items.Name
```

This will produce a list containing the name of each file in the folder this object represents. The command and its output look like this:

```
PS C:\Users\User\Demo> $child_items.Name
dest_dir
source_dir
file1.txt
file2.txt
file3.txt
new
```

Accessing the Methods inside a variable is done in the same way; the command syntax is:

```
$child_items.<method_name>()
```

The output of the method will be displayed on the screen.

CPU & Memory

Contents

In this module, we will be looking at how the CPU and RAM work in-depth:

- CPU: Components
- CPU: Memory Registers
- CPU: Fetch - Decode - Execute
- Memory: Stack and Heap
- Memory: Stack Frame
- Memory: Instructions vs Data
- Memory: Return Pointer

Note: This module may look intimidating, bu**DON'T PANIC**. We will be demonstrating all of this in detail in other modules using practical examples.

CPU Components

Control Unit (CU)

The Control Unit of a CPU is the part of the CPU responsible for directing electrical signals to the computer, in order to execute program instructions. Think of it like the director of a movie on a film set, or the conductor of an orchestra. It doesn't perform any execution of the program instructions itself, it only directs other parts of the computer system to do so.

Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit is the part of the CPU responsible for performing arithmetic and logical operations. The arithmetic part of the ALU can carry out four kinds of arithmetic:

- Addition
- Subtraction
- Multiplication
- Division

The logic part of the ALU carries out logical operations, typically comparisons. The logical operations it can perform are:

- Equal-To: compares two values to see if they are equal.
- Less-Than: compares two values to see if one is less than the other.
- Greater-Than: compares two values to see if one is greater than the other.

You can combine the operations above and include others. For example, NOT can negate the comparison. So you could test for NOT Equal To, to tell if two values don't match. You could also use OR to test for multiple things, such as Less Than OR Equal To, which would be true if the two values matched or if one was less than the other.

There will be more on logic in the data modules, so don't worry about it too much for now.

Registers

The CPU has its own memory, called memory registers. These are actually even faster to use than RAM because they are physically inside the CPU and therefore the electrical signals have a shorter distance to travel. The downside is that there are a limited number and they can hold an extremely limited amount of data. We will look into this more in the section on CPU registers.

CPU Memory Registers

As we mentioned in the previous section, the CPU contains several memory registers which are even faster to access than RAM. However, they can hold an extremely limited amount of data. The exact amount varies depending on the architecture of the CPU.

In a CPU with a 32-bit architecture, each memory register can hold 32 bits of data. That's 4 bytes of data, or enough for four characters if you use ASCII encoding. In a CPU with a 64-bit architecture, each memory register can hold 64 bits of data or 8 bytes. That's eight characters if you use ASCII encoding.

General Purpose Registers

On a 32-bit Intel CPU (x86), there are four memory registers which can be split up into parts. These are called:

- EAX
- EBX
- ECX
- EDX

You can split each register up into parts. If you store something in the 'AX' register, that refers to the lower 16 bits of the full 32-bit EAX register.

If you store something in the 'AL' register, that refers to the lower 8 bits of the 'AX' register. You can remember it based on the middle letter in EAX (A), and 'L' stands for 'lower'. There is also the 'AH' register, which is the higher 8 bits of the 'AX' register.

8 bits	***8 bits***	***8 bits***	***8 bits***
EAX	EAX	EAX	EAX
		AX	AX
		AH	AL

It's important to note that it's all still one register, you're just accessing different sections of the register. The same method applies to the other general purpose registers. For example, EBX can be addressed with: EBX, BX, BH, BL.

There are also registers that cannot be split up quite so far into parts. These are:

- ESP
- EBP
- ESI
- EDI

Although these are also called general purpose registers, they are used for very specific purposes and using them to store data can, in some cases, cause unintended consequences. They can be addressed as SP, BP, SI, and DI to access the lower 16 bits, but you **cannot** split them further.

ESP is the Stack Pointer register. It contains a memory address, which points to the top of the current stack frame in RAM (more on this shortly).

EBP is the Base Pointer. It contains a memory address, which points to the bottom of the current stack frame in RAM (again, more on this shortly).

ESI (Source Index) is typically used to hold a memory address of data when that data is being used as a source in an operation. For example, if you are copying data from one location to another, ESI will contain the memory address of the data you are copying.

EDI (Destination Index) is typically used to hold a destination memory address. If you were to copy data from one location to another, EDI holds the memory address that the data is going to be copied into.

Special Purpose Registers

There are a few registers that cannot be used freely by programs. These are known as special purpose registers. The best example of a special purpose register is EIP (Again on an Intel 32-bit x86 processor). EIP, also known as the 'instruction pointer' holds the memory address of the next instruction the CPU is going to execute.

The other special purpose registers won't be covered here, for the sake of simplicity and your sanity.

The Fetch - Decode - Execute Cycle

The CPU's job is to execute program instructions. The way it does this is as follows:

- The control unit 'fetches' the next instruction from RAM.
- The control unit 'decodes' the instruction (translates it into a form it can understand) and retrieves the necessary data from memory and places it into the arithmetic logic unit (ALU) of the CPU.
- The arithmetic logic unit (ALU) 'executes' the instruction, and operates on the data provided in the previous step.
- The arithmetic logic unit stores the result of the execution either in a memory register or RAM.

Once these four steps are complete, the cycle begins again, and the next instruction is retrieved.

RAM

Enough about the CPU for now, let's look at the RAM. RAM as a piece of hardware is one contiguous piece of data storage, but it is separated into sections by software. There are two sections of RAM, the stack and the heap.

The Stack

The stack is a very structured and orderly section of memory. When you launch a program, the instructions for that function are loaded onto the stack, and then each function is assigned an area of memory called a 'stack frame'. The stack frame contains the local variables that are used by the function.

Take, for example, a function that asks the user to type in some text and prints it to the screen. On the stack will be the instructions for asking the user for input and printing that text to the screen. Within the stack frame will be an area reserved for the data that will be printed to the screen. Since at this point the function has not run yet, there will be no data there yet, but when the function runs and the user types in the text they want to print, that text will get saved into the reserved memory location.

Finally, right at the bottom of the stack frame will be the return pointer. When the CPU enters a function, it will save the previous value of EIP to the bottom of the stack frame. We call this the return pointer. When the CPU gets to a 'ret' (return) instruction, it will load the return pointer from the bottom of the stack frame into EIP (remember, the instruction pointer that points to the next instruction that the CPU will execute) and therefore leave the function and the stack frame.

The Heap

The heap is much simpler. It's an unstructured area of memory that can be used to store data. The heap is somewhat slower to access than the stack, but the benefit is you can store whatever you like on there without knowing beforehand what the size of the data will be. By contrast, the stack is very structured, and you need to know how much data to reserve on the stack when you write the program.

Instructions vs Data

The topic is very important in cyber security. What is the difference between instructions and data in memory? The answer is there is no difference: to the CPU it all looks like binary data. However, when the CPU is pointed to a memory location via the instruction pointer, it treats whatever is at that location as an instruction, whether it makes sense as an instruction or not.

When the CPU is instructed to copy data from a memory address, it goes to the memory address in question and the value in that memory address is treated as data. This is because the instruction to copy data from a memory address expects data to be there.

If for some reason you manage to make the instruction pointer point to the bottom of a stack frame (the data section), the CPU would try to execute that data as if it were instructions (and likely crash as a result). Likewise, if you try to perform an operation on a memory address that is high up in the stack frame (instructions), the CPU would treat it as data rather than an instruction, because it expects to get data.

We will look at how we can take advantage of this particular fact later on in the course.

Monitoring Execution

Tracking Execution

In this module we will introduce GDB and how to track execution in greater depth. Things will look a little intimidating from here forwards, but these skills will be invaluable in helping you understand lower level flaws that attackers use, and basic architecture.

GDB Setup

In the next few sections, we will be covering several common types of flaws in binary applications. When it comes to exploiting these programs, it's essential for us to be able to see what is happening as the program executes. To do this, we can use a debugger. In our examples, we will be using GDB, the GNU Debugger. It's a debugger that is installed on nearly every Linux system. GDB on its own is not particularly user-friendly, unfortunately, so we'll be using an extension for GDB which adds some extra functionality. The extension we'll be using is called 'pwndbg'; it's fairly easy to install.

You'll first need to install 'git':`$ sudo apt install git`

Then to install 'pwndbg':

```
git clone https://github.com/pwndbg/pwndbg
cd pwndbg
./setup.sh
```

This will install 'pwndbg' into GDB. Now if you type `$ gdb` you should get a `pwndbg>` prompt (quit with `q`).

We're going to start off by looking at a very simple program so we can familiarise you with GDB. The program source code is:

password.c

```
// Compile with gcc -m32 -o password password.c

#include <stdio.h>
#include <string.h>

int main() {
        char *password = "ThisIsMyPassword\n";
    char input[151];

    puts("Please enter the password:");
        fgets(input, 149, stdin);

        if (strcmp(input, password) == 0) {
        puts("Success!");
        } else {
        puts("Fail!");
        }

        return(0);
}
```

Once you have the program compiled, we can look at it in GDB using:

```
$ gdb ./password
```

Running the Program

At this stage, we can run the program from within GDB by typing:

```
pwndbg> run
```

or

```
pwndbg> r
```

If we do, the program will ask us to type the password. We can pretend we don't have the source code, so we don't know what the password is. So we'll just type something, and the program will tell us we failed and finish executing:

```
root@kali:~/Binary# gdb ./password
...
Type "apropos word" to search for commands related to "word"...
Loaded 113 commands. Type pwndbg [filter] for a list.
Reading symbols from /password (no debugging symbols found) done.

pwndbg> r
Starting program: /root/Binary/password
Please enter the password:
blah
Fail!
[Inferior 1 (process 4397) exited normally]
pwndbg>
```

So far so good; now let's actually do something useful with GDB. Let's pause the program's execution as soon as it starts. To do this, we'll need to set something called a 'breakpoint' at the place where the program starts. A breakpoint is a marker that will tell GDB to pause the program when it reaches that point. Once the program is paused, we can start looking at things such as which instruction is currently being executed, what the next instruction is and what the state of the CPU is.

First, let's find out what functions there are in this program. We can do this with:

```
pwndbg> info functions
```

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x000003d8 _init
0x00000410 strcmp@plt
0x00000420 fgets@plt
0x00000430 puts@plt
0x00000440 __libc_start_main@plt
```

```
0x00000460 _start
0x000004a0 __x86.get_pc_thunk.bx
0x000004b0 deregister_tm_clones
0x000004f0 register_tm_clones
0x00000540 do_global_dtors_aux
0x00000590 frame_dummy
0x00000599 __x86.get_pc_thunk.dx
0x0000059d main
0x00000650 __libc_csu_init
0x000006b0 __libc_csu_fini
0x000006b4 _fini
pwndbg>
```

Now we have a list of functions that are in the program and the memory addresses they start at (the memory addresses are the ones that start 0x...). In a program written in C, usually, the start of the program is the main function. To set a breakpoint there, we can do:

```
pwndbg> break *main
```

or

```
pwndbg> break *0x0000059d
```

It doesn't matter if we use the symbol (the name) or the memory address, either is fine. It's also possible to shorten 'break' to just 'b':

```
pwndbg> b *main
```

or

```
pwndbg> b *0x0000059d
```

Once we set it, GDB will confirm it has been set successfully:

```
pwndbg> break *main
Breakpoint 1 at 0x59d
pwndbg>
```

Now if we run the program:

1

```
Breakpoint 1, 0x5655559d in main ()
LEGEND: STACK | HEAP | CODE | DATA |RWX | RODATA
```

2

```
[---------- REGISTERS ----------]
*EAX 0xf7f9edbc (environ) -> 0xffffd42c -> 0xffffd5cf <- 0x435f534c ('LS_C')

EBX 0x0
*ECX 0x541be3a3
*EDX 0xffffd3b4 <- 0x0
*EDI 0xf7f9d000 <- 0x1b9db0
*ESI 0x1
EBP 0x0
*ESP 0xffffd38c -> 0xf7dfb456 (__libc_start_main+246) <- add esp, 0x10

*EIP 0x5655559d (main) <- lea ecx, [esp + 4]
```

3

```
[---------- DISASM ----------]
>0x5655559d <main>          lea     ecx, [esp + 4]
 0x565555a1 <main+4>   and    esp, 0xfffffff0
  0x565555a4 <main+7>        push    dword ptr [ecx - 4]
 0x565555a7 <main+10>  push   ebp
  0x565555a8 <main+11>       mov     ebp, esp
 0x565555aa <main+13>  push   ebx
  0x565555ab <main+14>       push    ecx
 0x565555ac <main+15>  sub    esp, 0xa0
  0x565555b2 <main+21>       call    __x86.get_pc_thunk.bx          <0x565554a0>

  0x565555b7 <main+26>       add     ebx, 0x1a49
 0x565555bd <main+32>  lea    eax, [ebx - 0x1930]
 ```

4
```

[---------- STACK ----------] 00:0000| esp 0xffffd38c -> 0xf7dfb456 (__libc start main+246) <- add esp, 0x10

01:0004| 0xffffd390 <- 0x1 02:0008| 0xffffd394 -> 0xffffd424 <- 0xffffd5b9 <- 0x6f6f722f ('/roo')

03:000c| 0xffffd398 -> 0xffffd42c -> 0xffffd5cf <- 0x435f534c ('LS_C')

| **04:0010** | **0xffffd39c <- 0x0** |
|---|---|
| 07:001c | 0xffffd3a8 -> 0xf7f9d000 <- 0x1b9db0 |

```
### 5
```

[---------- BACKTRACE ----------]
```

f 0 5655559d main f 1 f7dfb456 libc start main+246 Breakpoint *main pwndbg>

There's quite a lot going on here, and all of it is important:

1. This just tells us which breakpoint caused the program execution to pause.

2. This area shows us the CPU registers. If you recall, the CPU has multiple registers, which can store small amounts of data. It's quicker for the CPU to access these than it is for the CPU to access RAM or storage. Data is usually stored in the registers while the CPU acts on it.

3. At this position, we can see the code that is being executed. This is assembly code and NOT C because the C compiler converts our C code to assembly, which the CPU can understand. Don't panic too much about this, for now, assembly is a huge topic, and while it is a programming language worth learning, for the next few demonstrations, we will explain any of the assembly you need.

4. At this position we have the stack, this is the data which is stored in RAM by the program.

5. At this position is the backtrace, in other words, which functions were called to get us to this point in the code. `__libc_start_main` is the standard function which is called to launch a program, and that ran the main function.

As a quick reminder, EIP is the instruction pointer, so it points to the next instruction. You can see it points to 0x5655559d in the screenshot, the code area of GDB agreed that this is where we are. Note the instruction that is being shown at position 3 **has not** been executed yet; this is the instruction that is about to be executed.

We can either continue the program at this stage using:

`pwndbg> continue`

or

`pwndbg> c`

If you get to the end of the program, you can run it again, and the breakpoint will still be there.

Moving through the program

If you instead want to step through the program instruction by instruction, you can do either:

`pwndbg> step`

or

`pwndbg> next`

After doing that once, if you hit enter again without typing anything it will repeat the last instruction you typed so that will save you some time.

Using these two commands, we can run the next instruction before pausing the program execution again.

Note there is one major difference between step and next:

- If you step through the program, if you hit any instructions that are 'call blah' then you will step into that function. This can be useful, or it can be really annoying. If you step into standard library functions like printf for example, this can be terrible. On the other hand, if the function is a custom function then that might be what you want.
- If you use next to move through the program, if you hit any function calls, then you will step over that instruction and move on to the next in your current function. The code in the function you stepped over will be executed, so it won't be skipped, but you won't have to go through it instruction by instruction.

Let's use 'next' in our example:

`pwndbg> next`

pwndbg> next 0x565555a1 in main () LEGEND: STACK | HEAP | CODE | DATA |RWX | RODATA [----------------- REGISTERS -----------------] EAX 0xf7f9edbc (environ) -> 0xffffd42c -> 0xffffd5cf <- 0x435f534c ('LS_C')

EBX 0x0 *ECX 0xffffd390 <- 0x1 EDX 0xffffd3b4 <- 0x0 EDI 0xf7f9d000 <- 0x1b9db0 ESI 0x1 EBP 0x0 ESP 0xffffd38c -> 0xf7dfb456 (__libc_start_main+246) <- add esp, 0x10

*EIP 0x565555a1 (main+4) -> 0xfff0e483 <- 0xfff0e483 [------------------ DISASM ----------------] 0x5655559d

lea ecx, [esp + 4]

0x565555a1

and esp, 0xfffffff0 0x565555a4
push dword ptr [ecx - 4] 0x565555a7
push ebp 0x565555a8
mov ebp, esp 0x565555aa
push ebx 0x565555ab
push ecx 0x565555ac
sub esp, 0xa0 0x565555b2
call __x86.get_pc_thunk.bx <0x565554a0>

0x565555b7

add ebx, 0x1a49 0x565555bd
lea eax, [ebx - 0x1930] [------------------ STACK -----------------] 00:0000| esp 0xffffd38c -> 0xf7dfb456 (__libc_start_main+246) <- add esp, 0x10

01:0004| ecx 0xffffd390 <- 0x1 02:0008| 0xffffd394 -> 0xffffd424 -> 0xffffd5b9 <- 0x6f6f722f ('/roo')

03:000c| 0xffffd398 -> 0xffffd42c -> 0xffffd5cf <- 0x435f534c ('LS_C')

04:0010	0xffffd39c <- 0x0
07:001c	0xffffd3a8 -> 0xf7f9d000 <- 0x1b9db0
[---------- BACKTRACE ----------]	
> f 0 565555a1 main+4	
f 1 f7dfb456 __libc_start_main+246	
pwndbg>	

So at this point, we've executed the 'lea' instruction, and we're paused before executing the 'and' instruction. If we want to run the same command again, we can just press return without typing anything.

Inspecting memory

You can inspect memory at any valid memory address using GDB.

For example:

`pwndbg> x/i 0x565555a1`

The first x here is short for examine. After the '/' we say what format we want the information to be displayed in. In this case 'i' means instruction, so we get:

pwndbg> x/i 0x565555a1 => 0x565555a1

: and esp,0xfffffff0

We can also view it as hexadecimal with:

pwndbg> x/x 0x565555a1 0x565555a1

: 0xfff0e483

The second 'x' here is for hexadecimal. It's important to understand that assembly code is just numbers to the CPU. We labelled the numerical instructions that the CPU understands to make things easier and we call that assembly language, but the truth is it's just numbers.

We can view information at a memory address as a string with:

pwndbg> x/s 0x565555a1 0x565555a1

: "\203\344\360\377q\374U\211\345SQ\201",

In this case, the 's' stands for string, but the data at that memory address isn't a string, which is why we get a messed up result here. This is important to note about GDB: it doesn't know and doesn't care what format the data at a certain location is in, it only cares about the format you've asked it to display it in.

You can also use the memory registers as a reference for example:

[---------- REGISTERS ----------] EAX 0xf7f9edbc (environ) -> 0xffffd42c -> 0xffffd5cf <- 0x435f534c ('LS_C')

EBX 0x0 *ECX 0xffffd390 <- 0x1 EDX 0xffffd3b4 <- 0x0 EDI 0xf7f9d000 <- 0x1b9db0 ESI 0x1 EBP 0x0 ESP 0xffffd38c -> 0xf7dfb456 (__libc_start_main+246) <- add esp, 0x10

*EIP 0x565555a1 (main+4) -> 0xfff0e483 <- 0xfff0e483

...

[---------- STACK ----------] 00:0000| esp 0xffffd38c -> 0xf7dfb456 (__libc_start_main+246) <- add esp, 0x10

01:0004| ecx 0xffffd390 <- 0x1 02:0008| 0xffffd394 -> 0xffffd424 -> 0xffffd5b9 <- 0x6f6f722f ('/roo')

03:000c| 0xffffd398 -> 0xffffd42c -> 0xffffd5cf <- 0x435f534c ('LS_C')

04:0010	**0xffffd39c <- 0x0**
07:001c	0xffffd3a8 -> 0xf7f9d000 <- 0x1b9db0

...

pwndbg> x/x $esp 0xffffd38c: 0xf7dfb456

Notice it has calculated the address in the 'esp' register for us.

And if you need to show more than just that one entry, you can put a number after the '/'. For example, to show 20 addresses with the address of $esp as the starting point:

pwndbg> x/20x $esp 0xffffd38c: 0xf7dfb456 0x00000001 0xffffd424 0xffffd42c 0xffffd39c: 0x00000000 0x00000000 0x00000000 0xf7f9d000 0xffffd3ac: 0xf7ffdc04 0xf7ffd000 0x00000000 0x00000001 0xffffd3bc: 0xf7f9d000 0x00000000 0x9cf78c8b 0xdc38809b 0xffffd3cc: 0x00000000 0x00000000 0x00000000 0x00000001 fififi

Quitting

If you've been thoroughly traumatised by all that above and you just want to leave, type:

```
pwndbg> quit
```

or

```
pwndbg> q
```

to exit.

Debugging 'password'

Now that you are familiar with GDB, let's look at how we find the password for the 'password' program we prepared in the last section. We'll load the password program into GDB:

```
$ gdb ./password
```

Now we'll check for the main function:

```
pwndbg> info functions
```

And set the breakpoint:

```
pwndbg> break *main
```

and finally, run the program:

```
pwndbg> run
```

Now we're here:

```
Breakpoint 1, 0x5655559d in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[-------------------- REGISTERS --------------------]
*EAX 0xf7f9edbc (environ) -> 0xffffd42c -> 0xffffd5cf <- 0x435f534c ('LS_C')
EBX 0x0
*ECX 0xe3fa2f7b
*EDX 0xffffd3b4 <- 0x0
*EDI 0xf7f9d000 <- 0x1b9db0
*ESI 0x1
EBP 0x0
*ESP 0xffffd38c -> 0xf7dfb456 (__libc_start_main+246) <- add esp, 0x10
*EIP 0x5655559d (main) <- lea ecx, [esp + 4]
[-------------------- DISASM --------------------]
> 0x5655559d   <main>     lea    ecx, [esp + 4]
0x565555a1          <main+4>       and         esp, 0xfffffff0
0x565555a4     <main+7>    push    dword ptr [ecx - 4]
0x565555a7          <main+10>      push        ebp
0x565555a8     <main+11>   mov     ebp, esp
0x565555aa          <main+13>      push        ebx
0x565555ab     <main+14>   push    ecx
0x565555ac          <main+15>      sub         esp, 0xa0
0x565555b2     <main+21>   call    __x86.get_pc_thunk.bx      <0x565554a0>
```

```
0x565555b7          <main+26>     add         ebx, 0x1a49
0x565555bd     <main+32>   lea     eax, [ebx - 0x1930]
[---------------- STACK ----------------]
00:0000| esp   0xffffd38c  -> 0xf7dfb456 (__libc_start_main+246) <- add esp, 0x10
01:0004|              0xffffd390       <- 0x1
02:0008|       0xffffd394  -> 0xffffd424 -> 0xffffd5b9 <- 0x6f6f722f ('/roo')
03:000c|              0xffffd398       -> 0xffffd42c -> 0xffffd5cf <- 0x435f534c ('LS_C')
04:0010|       0xffffd39c  <- 0x0
       |
07:001c|       0xffffd3a8  -> 0xf7f9d000 <- 0x1b9db0
[---------------- BACKTRACE ----------------]
> f 0 5655559d main
   f 1 f7dfb456 __libc_start_main+246
Breakpoint *main
pwndbg>
```

Let's step through the program with 'next' until we see something interesting in the code:

```
EBX 0x56557000 (_GLOBAL_OFFSET_TABLE_ ) <- 0x1ef8
ECX 0xfbad0084
EDX 0xf7f9e870 <- 0x0
EDI 0xf7f9d000 <- 0x1b9db0
ESI 0x1
EBP 0xffffd378 <- 0x0
ESP 0xffffd2c0 -> 0xffffd2d5 <- 0x10000000
EIP 0x565555f0 (main+83) -> 0xfffe2be8 <- 0x0
[---------------- DISASM ----------------]
 0x565555e0 <main+67>   sub    esp, 4
 0x565555e3 <main+70>   push   eax
 0x565555e4 <main+71>   push   0x95
 0x565555e9 <main+76>   lea    eax, [ebp - 0xa3]
 0x565555ef <main+82>   push   eax
> 0x565555f0 <main+83>   call   fgets@plt          <0x56555420>
   s: 0xffffd2d5 0x10000000
   n: 0x95
   stream: 0xf7f9d5a0 (_IO_2_1_stdin_) - 0xfbad2088
 0x565555f5 <main+88>   add    esp, 0x10
 0x565555f8 <main+91>   sub    esp, 8
 0x565555fb <main+94>   push   dword ptr [ebp - 0xc]
 0x565555fe <main+97>   lea    eax, [ebp - 0xa3]
 0x56555604 <main+103>  push   eax
[---------------- STACK ----------------]
00:0000| esp   0xffffd2c0 -> 0xffffd2d5 <- 0x10000000
01:0004|       0xffffd2c4 <- 0x95
02:0008|       0xffffd2c8 -> 0xf7f9d5a0 (_IO_2_1_stdin_) <- 0xfbad2088
03:000c|       0xffffd2cc -> 0x565555b7 (main+26) <- add ebx, 0x1a49
04:0010|       0xffffd2d0 -> 0xf7ffda74 -> 0xf7fd30e0 -> 0xf7ffd918 <- ...
05:0014| eax-1  0xffffd2d4 <- 0x1
06:0018|       0xffffd2d8 -> 0xf7fd3110 -> 0x565552db <- inc edi /* 'GLIBC 2.0' */
07:001c|       0xffffd2dc <- 0x1
[---------------- BACKTRACE ----------------]
> f 0 565555f0 main+83
  f 1 f7dfb456 __libc_start_main+246
pwndbg>
something
```

Okay so here we are, we've hit the 'call fgets' instruction. If you look up 'fgets' you'll see it's function that is part of a standard C library, for receiving user input from the command line. In other words, this is the bit where it asks us to type the password. We don't know the password yet, so I've just entered 'something'.

Now we're going to continue stepping through with 'next'.

The next place I want to pause at is here:

```
EAX 0xffffd2d5 <- 'something\n'
EBX 0x56557000 (_GLOBAL_OFFSET_TABLE_) < Ox1ef8
ECX 0xf7f9e87c <- 0x0
EDX 0xffffd2d5 <- 'something\n'
EDI 0xf7f9d000 <- 0x1b9db0
ESI 0x1
EBP 0xffffd378 <- 0x0
ESP 0xffffd2c0 -> 0xffffd2d5 <- 'something\n'
EIP 0x56555605 (main+104) -> 0xfffe06e8 <- 0x0
[---------------- DISASM ----------------]
 0x565555f5 <main+88>  add    esp, 0x10
  0x565555f8 <main+91>       sub          esp, 8
 0x565555fb <main+94>  push   dword ptr [ebp - 0xc]
  0x565555fe <main+97>       lea          eax, [ebp - 0xa3]
 0x56555604 <main+103> push   eax
>0x56555605 <main+104>     call         strcmp@plt                        <0x56555410>
    s1: 0xffffd2d5 <- 'something\n'
        s2: 0x565556d0 <- 0x73696854 ( 'This')
 0x5655560a <main+109> add    esp, 0x10
  0x5655560d <main+112>     test         eax, eax
 0x5655560f <main+114> jne    main+136            <0x56555625>
  0x56555611 <main+116>     sub          esp, 0xc
 0x56555614 <main+119> lea    eax, [ebx - 0x1903]
[---------------- STACK ----------------]
00:0000| esp          0xffffd2c0 -> 0xffffd2d5 <- 'something\n'
01:0004|                         0xffffd2c4 -> 0x565556d0 <- push esp
02:0008|              0xffffd2c8 -> 0xf7f9d5a0 (_IO_2_1_stdin_) <- 0xfbad2288
03:000c|                         0xffffd2cc -> 0x565555b7 (main+26) < add ebx, 0x1a49
04:0010|              0xffffd2d0 -> 0xf7ffda74 -> 0xf7fd30e0 -> 0xf7ffd918 <- ...
05:0014| eax-1 edx-1   0xffffd2d4 <- 0x6d6f7301
06:0018|              0xffffd2d8 <- 'ething\n'
07:001c|                         0xffffd2dc <- 0xa676e /* ng\n' */
[---------------- BACKTRACE ----------------]
> f 0 56555605 main+ 104
  f 1 f7dfb456 __libc_start_main+246
pwndbg>
```

So, here the program is calling the 'strcmp' function. If you look that up, you'll see it's a C library function for comparing two strings. In other words, this is the bit that checks if the password we typed matches the password it expects.

Because we are using 'pwndbg', it's showing us the two parameters that are being passed to this function. One of these is the password we typed at the prompt, 'something'. The other must be the password it expects. Remember, we can view a string at a memory address with:

```
pwndbg> x/s memoryaddress
```

So we'll do:

```
pwndbg> x/s 0x565556d0
```

And we get:

```
pwndbg> x/s 0x565556d0
0x565556d0: "ThisIsMyPasswor"...
pwndbg>
```

Close, but that ellipsis (...) means the string is incomplete, it just doesn't all fit into one memory address. So let's look at two memory addresses instead:

```
pwndbg> x/2s 0x565556d0
0x565556d0: "ThisIsMyPasswor"...
0x565556df: "d\n"
pwndbg>
```

So the password is:

```
ThisIsMyPassword\n
```

That last \n is a newline character which is naturally added when you press enter to submit the password, so we can ignore it. Now that we know the password, we can test it out. Quit gdb with:

```
pwndbg> quit
```

And then run the program and enter the password we found:

```
$ ./password
```

Debugging challenge

Step 1

Navigate to the lab directory as instructed and run the binary, and try to enter a password. Our goal is to get the password and bypass authentication using debugging. Open the binary with gdb.

Step 2

Once GDB is open, the first command you should run is `info func`. As explained previously, this will list all of the functions used by, or contained in, the binary itself. Make sure you run `info func` before running the program, otherwise it will print all of the functions that get loaded dynamically as well, which is a lot...

In the output of `info func`, you will see references to several interesting functions:

- `main`
- `fgets`
- `string_compare`
- `string_reverse`

The `main` function is the entry point for the program, and is where execution begins in a binary written in C or C++.

Step 3

Let's take a look at the `main` function using the `disas main` command to disassemble the code.

In the output you will see many lines of x86 Assembly code. You are not expected to understand what every line of the code is doing, however you can pick out key bits of functionality by looking for function calls that look like:

```
call 0x4a0 [fgets@plt](fgets@plt)
```

This indicates that a C function is being called by the binary.

As we know from before, the `fgets` function allows a program to retrieve input from the user. We can infer from the usage of this function that this is where the user's input is taken and stored for later use by the program.

Step 4

Moving further down the output, we can also see a call to the `string_reverse` function. The name of this function clearly shows its purpose is to reverse the string that is given to it. If we look carefully at the way both the `fgets` and `string_reverse` functions are called, we can identify that they are both operating on the input from the user.

Step 5

Let's take a closer look.

fgets call	*string_reverse call*
`lea eax,[ebp-0x70]`	`lea eax,[ebp-0x70]`
`push eax`	`push eax`
`call 0x4a0 fgets@plt`	`call 0x62d`

If we compare line by line, we can see that the first line of each call is identical. This implies that they are both operating on the same variable. The first line loads the address of [ebp-0x70] into the "eax" register. This is required because `fgets` requires the address of a place in memory that it can store the input from the user, and the `string_reverse` function requires the address of the string that needs to be reversed.

The second line simply pushes the contents of the "eax" register onto the stack. This is because when a function is called in C or C++, it retrieves its parameters from the stack.

The third line is the call to each function respectively. As you can see, the setup for these calls is remarkably similar, and this is because each function is operating on the buffer that stores the user's input.

This implies to us that the input being written to [ebp-0x70] by `fgets`, is also being reversed by the call to `reverse_string`, and is manipulating our input. Knowing this, the easy way to solve this lab would be to use the "strings" program to output contiguous ASCII characters and identify the reversed password.

Step 6

Now all you need to do is reverse the string and input it into the binary, and we have won!

You can reverse the string manually, or by using a small Python one-liner:

```
python -c "print ('<password_here>'[::-1])"
```

This gives us the final password.

Step 7

Alternatively, you could follow the process from the previous demonstration and place a breakpoint on `string_compare`, then find the password already reversed in the call to that function.

Advanced Storage

Contents

In this module, we will be looking at advanced storage mechanisms:

- Explaining RAID
- Cloud Storage Mechanisms

You will understand the different types of RAID arrays and the differences between file, block and object storage.

Explaining RAID

RAID is an acronym which stands for Redundant Array of Independent Disks. When you have several disk drives you can group them into what is known as a RAID array. A RAID array appears to the Operating System as a single logical drive, although it is made up of several physical disks.

There are several different types of RAID setups which have different benefits and drawbacks. Below we will cover some of the common types.

RAID 0

RAID 0 is known as 'striping'. This mode is optimised for performance and maximises read/write speed. The drawback of RAID 0 is the lack of fault tolerance. If a single drive in the RAID array fails all the data across all drives becomes unrecoverable.

RAID 1

RAID 1 is known as 'mirroring'. This mode essentially duplicates data across both drives. The advantage here is fault tolerance. If a drive fails the data will continue to exist. The disadvantage of this mode is your storage capacity is halved. If you have a RAID 1 array of two 4 GB drives then your actual storage capacity is 4 GB.

RAID 5

RAID 5 is known as 'striping with parity'. This mode requires at least three drives to function. This mode is optimised for maximum read speed but has a somewhat slower write speed. Additionally this mode can survive the failure of a single drive in the array, although the array will function with reduced speed until the drive is replaced.

RAID 6

RAID 6 is known as 'striping with double parity'. This mode is similar to RAID 5 but it requires at least four drives to function. As with RAID 5 read speeds are fast, however write speeds are slower. This mode can survive the failure of two drives simultaneously although once again with reduced speeds until the drives are replaced.

RAID 10

RAID 10 is a combination of RAID 1 and RAID 0 and combines the benefits of both striping and mirroring. Read and write speeds are maximised but the data is also mirrored which enables this array to survive a disk failure. This array does maintain the same drawback as RAID 1 in that only half of disk space is available to use for storage.

Cloud Storage Mechanisms

When it comes to storing data on the cloud there are three mechanisms that we must discuss. These are:

- File
- Block
- Object

File Storage

File based storage is the traditional approach to storing data. Data is stored on a filesystem with a name and some metadata and then it is organised into hierarchical folders. Data is retrieved by knowing the path to the file. This type of storage is the kind we are all familiar with, and it is also used commonly in Network Attached Storage (NAS) and shared file servers. This type of storage is less common on the cloud.

Block Storage

Block storage is one of the more common forms of storage you will see on the cloud. It is also utilised in a Storage Area Network (SAN).

Data is split up into blocks of equal size and each block is assigned a unique identifier. The storage system controls where each block of data is stored according to what the most efficient location is. When data is accessed on a block storage system the blocks are reassembled by the storage system based on their identifiers and then the complete data is presented to the user.

Block storage has the added benefit of separating data from systems. This effectively enables access to data from any system that can request it from the storage system without the data being tied to any one system.

A good example of block storage in the cloud is Amazon's 'Elastic Block Storage' offering.

Object Storage

Object storage is also a common form of storage in the cloud. There is no concept of hierarchy in object storage. Data is stored in a fiat structure as objects. Each object contains not only the data but also metadata and a unique identifier that references that object.

Object storage has the ability to scale infinitely by combining storage devices into storage pools. The primary use case for object storage is for holding relatively static, unstructured data. Object storage is most commonly accessed through the use of API calls. For example a GET request would retrieve an object and a POST or PUT request would upload an object.

A good example of object storage in the cloud is Amazon's 'S3' offering.

Containers

Containers

Containers use virtualization, but in a much more lightweight way designed to enable low-cost packaging of applications, libraries and configuration. They do not replicate the whole operating system, which makes them lighter, but also limits you to architecturally more comparable runtimes.

Containers have become crucial to modern technology practice. Developers and cloud application deployment use them pretty much as a default!

Containers vs virtualization

We've already covered virtualization at some length, but containers are a really powerful concept that leverages virtualization capabilities in a different way to produce wonderful efficiencies and fiexibility.

Containers are: * Overall less isolated * Efficient and take up less space * Implicitly version-controlled and portable * Great for running 'n' instances of applications/configurations with reasonable isolation on top of one host.

Virtual Machines are: * Much more isolated * Able to run diverse setups and operating systems - Windows, Mac OS X and Linux all side by side * Much heavier as you need to copy the 'entire OS and data' * Less portable, though still much more than a traditional server

Containers can run on top of virtual machines if you want to, say, run a Linux docker on top of a Windows system, and indeed Docker has released capabilities like the LinuxKit to facilitate this. That is, however, at its core, stacking virtualization capabilities with containers, so the model and architecture holds true.

Docker Introduction

Docker containers are a hugely powerful capability that is being used in a wealth of different use cases across IT and engineering. Containers are a lightweight form of virtualization that use the OS-level virtualization capabilities as we have previously discussed, but do not encapsulate the whole OS of the guest. Think of it as the host operating system providing services to the container, which make it feel like a different system, and isolate it - but not as thoroughly. We can, however, run a Ubuntu Linux container on top of a Mac, bundling up the applications, library and data required to deliver a specific app - and do it with lean resources!

- Images - executable code built in layers. A read only template or recipe for a container. The packaged requirements for app/server environment deliver.
- Containers - isolated from each other, bundling configuration, software and libraries. These are the environments we run, but they can be started/stopped too.
- Docker Daemon - does all the real work behind containers, from building to running and delivery.
- Docker Client - The component you use to issue instructions, such as the Docker CLI.
- Docker Hub - A registry of Docker images, plus you can roll your own! This is a powerful concept in providing portability.

Docker has lots of use cases and you can take a quick tour of the internet, finding people using it in creative ways, but to list a few features and use cases that are common:

- Developers - replicating development, build and production environments. You can share the recipe with other developers or systems.
- Portable deployment of applications, packaging configuration.
- Server consolidation! Containers are often a more efficient use of resources - why replicate a whole operating system if you don't need to?
- Empowering code pipelines from development to build, staging, and through to production -- all with version control at the container layer, not just application/code.
- Fun, running different tool versions and research! I can, for example, run a different version of Python, and a Linux package for it that is not as portable on a Mac build of Python!

We will explore Docker practically, and at first it can be a little mind-bending. It looks like a virtual machine, but less 'complete' and graphical. That being said, once you are a container power user it will change the way to use computers fundamentally.

Docker CLI Basics

In this walkthrough we take you through the basic use of the Docker CLI and how to grab images, build containers and interact with them. Docker has a very rich set of functionality and many arguments to be explored -- we could spend a week of course content on this alone! The crucial concepts to make sure you understand are:

- Containers use virtualization capabilities, but are not virtual machines that encapsulate a whole OS.
- Docker images are recipes to build containers and can be hosted in the repository. Look how easily we pulled these images when we needed them, just by name!
- Docker containers are the running instantiation of an image, they can be started and stopped -- and eventually removed to save space if we no longer need them,
- `docker pull` - Grabs an image from the repository for you, perhaps a specified version like ubuntu:18.04 or ubuntu:latest.
- `docker run` - Enables you to run a container and execute something inside it, either by default, explicitly by being passed as an argument -- and both interactively like a shell, or in the background.
- `docker images` - Lists images stored locally and provides arguments for handling them, such as`docker images rm`

There are a huge number of examples online, which are worth exploring and getting familiar with on your own system. This is a powerful technology that comes up in a huge number of places in modern IT!

Building Containers

Here we take a quick tour through building a container using Docker and a Dockerfile. There are huge automation opportunities here, and more powerful capabilities like Docker compose, but this is a good introduction to the topic and illustrates the building in layers covered in earlier modules.