Introduction to Python: Transcripts

Unit 1.1 About Python

Title: What is Coding? - Animation

Narration script
Welcome to “What is Coding”. In this animation, we’ll discuss what coding is, why you would want to learn about it and a little general theory about how it works in practice.
Writing and using custom code is an integral part of part of modern research. This derives from the fact that computers are very well suited to performing calculations exceedingly quickly. By instructing a computer to perform specific calculations, we may automate laborious calculations that would otherwise be prohibitively time consuming to conduct. By carefully selecting which calculations to perform, we can produce simulations which predict the behaviour of a physical system which would otherwise be costly, difficult or impossible to observe. For instance, before I joined the Graduate School, my research was in the area of nuclear engineering. A large portion of my work was the production of simulations of different accident scenarios which would be expensive and dangerous to produce in real life as an experiment. But, by using computers to simulate the accident, we could learn more about it and how to prevent or mitigate its effects. Codes can also be used to analyse large amounts of data rapidly. You could use a code to automate the analysis of hundreds or thousands of data points you’ve created from an experiment in a lab, or massive collections of data available from other sources such as space telescope images or medical or social databases. Due to the speed of computers they can also be used to analyse and respond to data in real time. This means production of code is at the heart of many technologies including self-driving cars, high-frequency financial trading and “internet of things” devices.

Now we have some examples of what coding can be used for, we should gain some understanding of how the production and execution of code works on a conceptual level. The act of coding is the production of “source code”. This is a human-readable set of instructions, detailing what you, the coder, wants the computer to do. This source code will be written according to the rules of a particular coding language. There are a large number of languages available, including Python, C, C++, Fortran, Julia and Javascript. Each will have its own features, drawbacks and quirks and the best language to use in a given situation will depend on a number of factors, including the preferences of the coder and their team.

This example piece of source code would execute in a number of languages including Python, Fortran and C. It tells the computer to calculate the sum of the variables named X, Y and Z and save this value into a variable named D. This source code is fairly abstract in the sense that it doesn’t give specific commands to the hardware of the computer.

Instead, when a piece of code is executed, it will commonly be translated into a set of assembly language instructions such as these. This is much less readable than the source code we saw before, but is more specific in terms of what the hardware should be doing. Here we see the value of the variable X is loaded into a memory location A in the processor and the values of variables Y and Z are added to it, before the value stored in A is saved into the variable D.

This may be further processed into a series of machine language commands which are represented in binary and are all but unreadable to most programmers. It’s these commands which will actually be interpreted by the processor in your computer. Code designed to perform the same calculations may appear very different in the source code form, dependent on the language the source code is written in, but the machine language representation may actually be quite similar as the same operations are required of the physical processor.

The good news is you won’t actually have to interact with anything other than source code unless you end up specialising in specific areas of computer science. As a result, it will be sufficient to learn the rules of the language you’re using in terms of source code. Your source code will then cause the desired operations to be carried out by the processor.

It can be useful to remember that the rules of the programming language you use exist to provide an interface between you and the hardware of your computer to tell it what to do. As a coder, it’s your job to work out what arrangement of source code will cause the computer to carry out the calculations you want it to.

As a result of this description of coding, we can say that there are four skills required to be able to code in a given language.

Firstly, you need to know what operations a given language can be instructed to do and what their effects are in terms of calculations in the computer.

Next, you need to be able to work out what how to combine these operations to achieve the overall goal you’re trying to achieve with your code.

Once you have a sequence of operations, you need to know the syntax of the language so that you can create the source code containing the instructions relating to these operations. Finally, once you have your source code written, you need to know how to instruct your computer to run the code you've written.
In this video we've discussed some basics of why we code, how code works and the relationship between coder, source code and the hardware of the machine. This will give you some useful context whilst learning the specifics of Python.

Title: What is Python?

Narration script
In this video, we'll discuss the properties of Python and the consequences of these properties in terms of how Python fits into the family of coding languages.
If you were to find a short description of Python, you would find it described as an "interpreted, high-level, general purpose programming language". That's a long and wordy description, so let's break it down
There are two broad families of coding languages - compiled languages and interpreted languages. Before a code written in a compiled language can be run, the source code will be examined by a program called a compiler. This will produce an executable written in a low-level language that can be directly run on a computer. This executable may then be run repeatedly without further reference to the source code. The source code of a program written in an interpreted language will be examined by the interpreter each time it is run. More specifically, the interpreter will look at only the line that is currently being executed at the time and interpret the statement and execute its commands in real-time.

Generally, compiled languages are faster to run. This is partially because compilers are often able to optimise a piece of code and chose the optimum way to execute the set of commands presented in the source code. An interpreted code will also have to convert the higher-level source code to a lower level language in order to execute it on your computer on the fly, which increases running time.

The main advantage of an interpreted code, such as Python, is that it is much more portable. The source code can be moved to any operating system and, providing an interpreter is installed, the code can be run with relatively little difficulty. By comparison, an executable produced by a compiler will generally be tailored to the architecture it was compiled for and therefore will not be as portable.

Regardless of the advantages and disadvantages of each, the line between "interpreted" and "compiled" languages is not absolute and is becoming increasingly blurred. Any compiled language can be interpreted, and any interpreted language can be compiled.

It is becoming increasingly common for modern languages, including Python, to sit somewhere between these two extremes and Python files are often compiled to speed up execution.

However, Python is still classed as an "interpreted" language as this is historically how it was used and is the most common way it is used today.

A high-level language abstracts much of the detail of how a code is implemented in terms of the computer hardware. Most languages that are commonly used for programming large projects will be high-level languages. However, there is some variation and languages such as Fortran and C are slightly lower-level languages than Python, although they are still much higher-level than Assembly Language or Machine Language.

High-level languages are much easier and faster to program in. The details of implementation are hidden from the user by the language and are handled by the interpreter or compiler, meaning more can be achieved with each line of code.

However, low-level languages have the benefit of giving the user much more direct and precise control over what the computer is doing. With extra effort and skill on the part of the coder, lower level languages can produce more efficient code, although this will come at the cost of extra development time.
Some languages are designed with a specific application in mind. For instance, R is a language intended primarily to undertake statistical analysis and contains many features designed to help this goal. However, Python is a general-purpose language, meaning it is designed to be equally applicable to any situation. This means, by default, it doesn't contain specialised functionality but will be able to handle any scenario.
Finally, we come to the noun of the phrase. A programming language is a set of rules which translates characters written in a source file into operations performed on your computer.

There are some further observations we can make about Python.

Firstly, it is relatively easy to learn. By many measures, it's quite intuitive and easy to understand. The high-level nature of the code means it's also relatively quick to write code in once you're proficient with it, as you can get a lot done with each line.

Python is one of the most popular coding languages in the world, normally ranked in the top three, alongside Java and C. It's popular among a wide range of audiences across a range of industries.

This large user base leads to several advantages. It means there is a very large Python community. This means a lot of time has been spent learning how best to use the language and there is a lot of good practice to draw from. It also means there are many people who you can ask questions of, including online and many common questions will have been asked and answered in online forums such as stackexchange already.

In addition, the popularity of the language means it is well supported by third party programs such as text editors and Integrated Development environments and that many useful libraries have been created and released to the public. The availability of these libraries means the usefulness of the language has been substantially extended.

One drawback of pure Python is that it's quite slow. Compared to languages such as Fortran and C, it may be dozens of times slower. However, libraries such as NumPy or SciPy exist which give access to pre-compiled code written in faster languages. By using such libraries for the computationally intensive parts of the code, some of this performance deficit can be recovered.

In this video we've described some of the features of Python, discussed its advantages and disadvantages and talked about how it fits into the family of coding languages.

1.2 Using Jupyter Notebooks

Title: Before your start

Narration and Script

Python is a programming language which can executed in a number of different ways using a number of different tools. In this course, we will be mainly be using the Jupyter notebooks application. There are many ways to run Jupyter notebooks on your computer. This video will show you how to install and run Anaconda on your machine, which is one of the most common ways to run a Jupyter notebook.
To install Anaconda, open your web browser and go to “Anaconda.com”. Go to the products tab and select “Individual Edition” and click “Download”. Select the appropriate version for your operating system and click it. This will start a download which may take a few minutes to complete, depending on your connection speed.
When the download has completed, open the installer. Follow the prompts in the installer. Choose a location in your computer to install Anaconda if you don’t want to use the default location. On the “advanced installation options” screen, leave the boxes unticked and click “install”. The installation may take a few minutes to complete.
We can then finish following the prompts in the installation process. On the last screen, you have the option to open some webpages to learn more about Anaconda.
We can then open Anaconda. How you do this will vary from operating system to operating system. On Windows, I’ll search for “Anaconda” and open the program. This may take a few seconds to launch and may flash some black screens before Anaconda opens.
On the Anaconda Navigator homepage we can see a number of programs including Jupyter Notebook. Later on, we’ll use Spyder but, for now, we’ll use Jupyter Notebook. Click “launch” to launch it.
After a few seconds the Jupyter notebook interface will open in your browser. Although Jupyter runs in your browser, it’s actually running locally on your machine rather than on the internet. You’ll see a depiction of the directories on your computer. You can create a new Jupyter Notebook by clicking the “new” button followed by “Python 3”. Later in the course when you have the opportunity to download a Jupyter notebook, put it in a location of your choice inside your computer. You can then navigate to it like this and click on it to open it.
For now, I’m going to create a new file. This is the Jupyer notebook interface. If you’ve got to this point, you’ve done all you need to for now. We’ll examine the interface in detail in another video.
Additional resources

Title: Introduction

[Jupyter Notebooks:

- “Using Jupyter Notebooks”

Narration script
The goal of this module is to introduce you to the interface of Jupyter Notebooks provided by Anaconda and to help you get comfortable with it. It also aims to introduce you to some of the features offered by a Jupyter Notebook.
A Jupyter notebook is an open-source file format that typically has the file extension “.ipynb”. This stands for “I-Python Notebook” which is an old name for a Jupyter Notebook. Here we see a number of Notebook files in the directory view of the Jupyter Notebooks application.
A Jupyter Notebook may be opened by a number of apps, including the Jupyter Notebook component of Anaconda. You should have installed this program during the “Before you start” Section of this course. Anaconda isn’t the only place you can run Jupyter Notebooks, but it is the most popular.
In a notebook we can create a combination of text, written in a html-like language called Markdown, and Python code which can be executed in place. This allows us to produce beautiful and instructive documents describing and annotating the code that it contains. This could be used to create intuitive and interactive front-ends to codes and is also well suited to producing notes. During this course, you will collect a series of notebooks with a collection of notes, examples and exercises which will provide you with a lasting record of what you learn.

Title: Cells and Executing code

Narration script
A Jupyter Notebook is made up of a number of cells. We can select a cell by clicking on it. Each cell will have a type: either a Markdown cell like the first cell here, or a code cell like the second one. Let’s have a look at some interesting properties of code cells before we move on to Markdown cells.
To the left of each code cell is the word “In” followed by some square brackets. Initially theses square brackets will be empty, like they are now. We have two options to run the code contained in a code cell. The first option is to click on it to select it and then press shift and enter. This will run the code cell. In this case, the “print” statement within the code executes and the phrase within the parentheses following the word “print” are printed to the space below the code – this is the place where this type of output will be displayed.

We can also run the code by selecting the cell and clicking the “run” button. Note that, this time there was already a cell beneath the cell we ran, so no new cell was created, but the next cell was selected.
We can insert a new cell by clicking “Insert” and then inserting a cell either above or below the currently selected cell. The inserted cell will be a code cell.
If we click a code cell outside of the editable areas, we select the cell in “command mode”, indicated by a blue border. If we click the editable area, we enter “edit mode”, indicated by a green border. This allows us to edit the contents of the cell.
Let’s write some code in this cell. This code creates a variable named “a” and gives it the value 1. We’ll talk more about exactly what’s the going on here later. Now let's run this cell. As we haven’t asked Python to give us any output, such as by using the “print” statement, there is no output produced below the cell.
In the next code cell, we can write print(a). This time the name of the variable we just created is inside the parentheses. When we run the cell, this will cause Python to print the value of “1” which we gave to "a". This shows us one of the important properties of code cells in Jupyter Notebooks – information we’ve saved as a variable persists across different cells.
In the next cell we create, we can choose to give “a” a different value. Let’s give it the value “2”. Now we can run the cell and the value of “a” will be changed.
Now if we go back and run the previous cell again, we see the value printed is now “2”. This demonstrates another very important property of code cells – the order in which they’re run is important but the order in which they’re placed on the page isn’t. In fact, Jupyter helpfully keeps track of what order cells have been run in through the numbers which appear in the square brackets to the left of the code cells.
We can clear all saved variables by opening the “Kernel” menu and clicking restart. We’ll be asked to confirm that we want to delete all variables.
Once we’ve deleted the variables, if we execute the “print” statement we’ll get an error displayed below the code cell. Python is trying to print the value of the variable “a”, but we just deleted the variable “a” so it can’t.
Now, we might decide the output in our notebook looks untidy and we want to get rid of it. To do that, we can click “Cell” and then “All output” and then “Clear”.
If we want to run a set of cells, there are several useful options in the “Cell” menu, such as “Run All” which runs all the cells in the notebook one after another from top to bottom.
The “edit” menu give several useful options, including deleting the currently selected cell using “delete cell”.

There are several other useful commands in the “edit", “insert”, “cell” and “kernel” menus. Take a few minutes to explore some of these options and try using them in a notebook of your own. Feel free to copy some of the code from the examples in this video if you like.

Title: Text Cells in Jupyter

Narration script
Now, we let’s have a look at Markdown cells. We mentioned previously that, when we insert a cell, it’s a code cell, as shown by the square brackets to the left of it.
If we click on this new cell to select it, we can change its type by opening the “Cell” menu and hovering over “Cell Type”. Here, we have the option of three cell types - “Code”, “Markdown” and “Raw NB Convert”. You’re already familiar with Code cells. “Raw NB Convert” cells are a more advanced and specialised type of cell that we won’t be looking at this course. We’re going to turn this cell into a Markdown cell.
If we click in the cell, its border changes to green and we enter edit mode. We can now type Markdown code in this cell. This is a type of code which will be interpreted into rendered text. We’re going to type a couple of words here as an example.
We can run this cell in the same ways that we ran the code cells earlier. This interprets the markdown code and turns it into rendered text. This replaces where the Markdown code was written in the Notebook. In this case, the Markdown we wrote was plain text and so the rendered text is also plain text.
However, we can do more than create plain text using Markdown. Let’s edit the markdown cell we created to see some examples. To do that, we first double click it to re-enter command mode, then we click it once more to enter edit mode.
The first thing we’re going to look at is how to produce titles. The way to code for a title in Markdown is to begin a line with one or more hash symbols. We can create a large heading using a single hash like this. Note that the Markdown text is coloured blue as Jupyter Notebooks knows this line will be rendered as a heading and is showing us this.

When we run the cell, the rendered text appears again. This time we can see the heading has been rendered in a larger font to appear as a heading.
We can also create smaller sub-headings and sub-sub headings by using more hash symbols at the start of the line. For instance:
When we run the cell, we see these are rendered into smaller headings. These headings can be useful for breaking up your document.
We can also insert tables, using a format similar to tables in html documents. First, there should be a blank line before a table starts. In each row of the table, we denote columns using this vertical line character. The placement of this character on your keyboard tends to vary, so you might have to look for it. We're going to create a table with two columns, so we need three vertical strokes with the headings of the table in between.
The second line of our table must have the same number of vertical lines with one or more minus characters in each column.
We can then add rows to our table by writing the same number of vertical lines with the contents of each cell of the table between each.
We can include as many rows of the table as we wish. When we click "run" the Markdown code is rendered into a table.
We can also write Markdown code which will render into a mathematical equation. To do this, we surround a piece of text with two percentage signs, like so.
The syntax used is the same as mathmode in LaTeX. If you're not familiar with LaTeX, then don't worry about it – we're not going to use this in the rest of this course. It is, however, useful to know that, if you need to, you can include complex mathematics which you could use to annotate your notebook, perhaps to explain what your code is doing.
Finally, you can write text in Markdown which will be highlighted as though it is code. This text can't be executed but will be highlighted to mark it as code and to make it easier to read as code.
For an example, let's write a = 2. To do this, we precede the text to be rendered as code with three backticks and then the name of the language whose rules are to be used to highlight it. In this case, we'll write "python". Next, we write the text to be rendered as code. Finally, we place three backticks after it to show where the code-like section ends.

When we run the markdown cell to render it, we see this text appears in a different font and different parts of the text are in different colours, just as they would be in a code cell. However, we can see that it's still in a Markdown cell as its background is white, not grey.
As a reminder, there is no way to run this text as code, but sometimes it's useful to be able to make something look like code if your text is referencing code that is written in a code cell.
Please take some time to experiment in your own notebook. Create some markdown cells and try to create your own headings, table, equation and code snippet.

Unit 2.1 My First Program

Title: The Print Statement

Narration script
In this video, we're going to look at one of the most common and most useful functions in Python – the "print" function. This function allows us to display values from our code, which can be useful for outputting results or other messages to the user.
By writing "print", followed by a pair of parentheses, we can cause Python to print to the screen a representation of the value of whatever is inside the parentheses. For instance, if we want to print the value 3 to the screen, we would write the following:
When we run a code cell containing a print statement in a Jupyter Notebook, values inside the parentheses are displayed below the code cell.
If, instead of printing an integer, you want to print text, it must be inside single or double quotation marks like this.
Note that when we run a cell with multiple print statements they will execute one after another from top to bottom and the output of each will appear below the code cell.
Outside of a notebook, the output of a print statements in Python will commonly be printed to the terminal used to run the script.

Print statements can be very useful for checking what values variables within the code take or for displaying results at the end of a calculation. We'll be making heavy use of the print statement in this course.

Student exercise:

- In a fresh code cell, write three print statements. They should print:
  - the phrase "Hello World"
  - an integer of your choice
  - a negative non-integer number of your choice

Solution to the student exercise "The Print Statement"

In this exercise we were tasked to write three print statements. Let's begin with the first statement. We write the word "print" to tell Python we want to print a value. We follow this with a set of parentheses. Into these parentheses we put the thing we want printed. In this case, we want to print the phrase "hello world", so we write that in here. I used double quotation marks to tell Python that it's a phrase to be printed. Single quotation marks would also have worked here.

Next, we're asked to print an integer. I'm going to print the value 7. So, again, we write "print" and add a pair of parentheses. Inside the parentheses I write the number 7. I haven't put quotation marks here as I want Python to print a representation of the value 7. If I used quotation marks, Python would print the character "7". This would look the same, but isn't what we were asked to do.

Finally, we're asked to print a negative non-integer. We construct the print statement like normal and give it a negative non-integer value to print by using a minus sign to show it's negative and a full stop for the decimal point.

When we run the cell, we see the three values we were asked to print, printed below it.

Title: Setting Variables

In Python, we create a variable by assigning a value to a variable name with the equals sign like this. Here, we've created a piece of data with the value "1" in our computer and a variable with a variable name of "a" which references this stored value.
By placing the variable name inside the parentheses of a print statement, we ask Python to print the value a variable name references. Here, we ask it to print the value of a variable "a" and, when we run the cell, we see that "a" has the value "1".
We put a single value on the right-hand side of the equals sign, but we can actually put any expression there. An expression is a piece of code which can be evaluated to give a value.
So, we could put another variable name of the right-hand side. This would give the variable on the left the same value as the variable on the right. For instance, we can write "b = a" to create a variable named "b" and give it the same value as "a". We can then check the values of these variables with some print statements.
Before we have assigned to a variable name, that variable name won't exist. For instance, if we try to print the value of the variable "c" we'll get an error. Specifically, Python will tell us this is a "NameError" and that the variable named "c" doesn't exist. For this reason, we must always assign to a variable before we use its value.
Now, complete the exercise to get some practice at creating your own variables.
First of all we're going to create a variable named "word" and give it the value "software".
We can verify the value of "word" by placing the variable name inside the parentheses of a print statement.
When we run the cell, we see "word" does, indeed have the value "software"

Next, we want to give a variable the value of a negative integer. I'm going to call it "neg" and give it the value "-3".
Next, we create a variable named "pos" and give it the value "5.2".
Finally, we're asked to change the value of the variable named "word" to that of the positive non-integer, which I named "pos". Before we do that, we're going to check the value of "word", to ensure it's still "software".
Now we're going to change its value, so it has the same value as "pos" by writing "word = pos".
Now, we'll print the values of both variables.
We see that the value of "word" has been changed to 5.2 and the value of pos is unchanged.

Title: Mathematical Operators

In this video, we're going to look at some mathematical operators in Python and discuss what they do, and how they behave together. We've already looked at the assignment operator, represented by an equals sign, which sets the value of the variable to the left of it equal to the value of the expression on the right.
We're going to use the assignment operator here to create a couple of variables, "a" and "b" to use in the upcoming examples.
The first mathametical operator we're going to look at is the addition operator, represented by the plus sign. An addition operator adds the values to the left and right of it together, like this. This produces the value 13 because 10 plus 3 is 13.
Next, we'll look at the subtraction operator, represented by the hyphen character. A subtraction operator subtracts the value on the right from the value on the left, like this. The value printed is 7 because 10 minus 3 is 7.
The multiplication operator is represented by the star symbol, which can be found on shift+8 on most keyboards. This multiplies the values to the left and right of it together. In this case, 10 times 3 is 30.
The division operator is represented by the forward slash character. This divides the value on the left by the value on the right. In this case, 10 divided 3 is 3 point 3 recurring.

Another interesting thing here is that the final digit of the value printed here is 5 and not 3. This is an interesting by-product of the way that computers store numbers. When Python, or any other coding language, stores a value, it uses a certain number of bits – ones and zeroes – to store the value. As there are a finite number of bits, there are a finite number of combinations of ones and zeroes available, meaning there are a finite number of numbers which can be represented precisely. When Python performs an operation and the value doesn't correspond exactly with one of the numbers it can store precisely, it picks the value it can represent with the closest value to the true value of the expression. This is called "floating-point error" and is common to most coding languages. The value returned is still very close to the correct value. Here it is correct to within 16 significant figures and this is more than accurate enough for most purposes and can be ignored. As you get more advanced, you may have to think about floating point error more but, for now, you can accept it as a quirk of how computers behave.
Next, we're going to look at the exponent operator. This is represented by two stars next to each other, like this. This will return a value equal to the value on the left raised to the power of the value on the right. Here, that is 10 raised to the power of 3, or 10 cubed. This gives a value of 1000
Note that using a hat operator by pressing shift+6 on your keyboard will not give you the exponent operator as it would do in some other programs, such as Excel. It is a different operator that we're not going to discuss today, but it's important to make sure not to confuse the two.
The next operator may be new to you. This is the modulo operator and is represented by a percentage sign. This returns the remainder when the value on the left is divided by the value on the right. For instance, here, 3 times 3 is 9. This is the largest multiple of 3 which fits under the value 10. 10 is 1 bigger than 9, so 10 modulo 3 is 1.
Finally, we're going to look at the integer division operator and is represented by a double forward slash. This divides the value on the left by the value on right and returns the nearest integer below that number. Here, 10 divided by 3 is 3 and a third, so this expression returns 3 as this is the nearest integer below 3 and a third.
We can also form more complex expressions by combining multiple operations. In this case, operations will be carried out according to BODMAS or PEDMAS rules. This means, values inside parentheses will be calculated first, with inner parentheses happening before outer parentheses.

Next, exponents will be calculated. If multiple exponents are on the same line, they will be calculated in order from right to left.
Next, operations relating to multiplication and division will be executed. If multiple operators from this tier are in the same expression, they will be executed from left to right.
Finally, addition and subtraction will be carried out. If multiple operators from this tier are present, they will be executed from left to right.
Let's have a look at an example. In this case, the values within parentheses will be calculated first. The value of the expression in the left set of parentheses is 1 plus 1, which is 2. In the parentheses on the right, the multiplication operator is a higher tier than the addition or subtraction operator, so this will be carried out first. 1 times 3 is 3. Then the addition and subtraction operators are the same tier, so will be carried out from left to right. 1 minus 3 (the result of the multiplication) is minus 2. Then –2 plus 4 is 2. So the expression in the right-hand parentheses evaluates to 2. Now, we can perform the exponent operator, which raises 2 to the power of 2. This is 2 squared, or four. When we run the code, we see this is the value printed.
Now, have a go at the exercise which will let you put your newly found skills with printing, variables and operators to the test in your first code

Title: Student Exercise: Mathematical Operators

Written exercise:

Now you have the opportunity to put your new skills to the test and to write a code of your own. This code relates to a scenario where there are five identical cubes, each with a side length of 3m. In a new code cell, use the operators we discussed to calculate the following properties of these cubes and print them:

- Length of all edges of one cube
- Area of one face of one cube
- Area of all faces of one cube
- Volume of one cube
- Volume of all cubes
- Surface area of all cubes
- Surface area to volume ratios of the cubes

If you want to, you can save intermediate variable to use again later to avoid having to calculate the same values repeatedly. When you're happy with this part of the scenario, now imagine that the cubes are being filled with water, one after another. When one cube is filled, the next one begins to be filled. 100 cubic metres of water is added in total. You may assume the final cube is partially filled. Calculate:

- The number of the cubes needed to hold the water (this should be an integer)
- The amount of space left in the partially full cube after the water has been poured into the cubes

To perform these tasks most efficiently, try to think how you might use integer division and the modulo operator.

Student exercise answer: Mathematical Operators

Narration script

First of all, we're going to create variables to hold the values of the length of side of a cube and the number of cubes.
Like all the other variables we're going to create, I gave these two variables descriptive names. This is generally good practice as it makes the code easier to read.
Next, we're going to print the length of all the edges of a single cube. A cube has 12 edges, so we're going to multiply the length of a side by 12.
Next, we're going to calculate the area of one face of one cube. This is a value we're going to use again, so this time we're going to save it to a variable before printing it. We calculate its value using the exponent operator to sqaure the length of a side of the cube.

Next, we're asked to calculate the surface area of all the faces of a single cube. Again, we're going to save this value so we can use it again. There are six faces on one cube so we can multiply the surface area of a single face, which we already calculated, by 6.	
Next, we're going to calculate and save the volume of a single cube. We can do this by cubing the length of the side of a cube using the exponent operator.	
To find the volume of all cubes, we can multiply the volume of a single cube by the number of cubes. We won't need this value again, so we'll just print it.	
To calculate the surface area of all cubes, we can take the surface area of a single cube and multiply it by the number of cubes.	
To calculate the surface area to volume ratio of the cubes, we can divide the surface area of one cube by the volume of one cube.	
We could also have divided the surface area of all cubes by the volume of all cubes and got the same answer.	
In the next part of the exercise, we're asked to consider the water being poured into the cubes. First, let's save the volume of water to a variable.	
To calculate the total number of cubes containing water, we're going to use the integer division operator. We divide the volume of water by the volume of a single cube.	
However, this will round down the value to the nearest integer, which will relate to the number of completely filled cubes. What we want is the number of cubes containing any water. Because we're allowed to assume there is exactly one partially filled cube, we can add 1 to that value to get the total number of cubes which contain any water.	
To calculate the volume of the partially filled cube which does not contain water, we can use the modulo operator to calculate the volume of water in the partially filled cube. This works because the modulo returns the remainder when the volume of water is divided by the volume of one cube.	
However, we were asked to calculate the volume remaining unfilled in the partially-filled cube. To do this, we can subtract the value we just calculated from the volume of a single cube as follows. This works without parentheses because the order of operations means the modulo operator will be executed before the subtraction operator.	

Title: Comments

Before we move on, we're going to quickly discuss comments. Comments are common to many coding languages and provide a means for us to write in a source file in a way that will be ignored by the interpreter.
In Python, a simple comment can be inserted using a hash character. When a hash character is used, the remainder of a line is ignored by the interpreter. For instance, we can add a comment here...

...or here. When we run the cell, we see the code executes as if the comments weren't there. Here, we've used comments to annotate our code. We've provided a description of what we're doing in the code. This can be very useful to someone looking at your code as it means they can understand what you were doing and why and so understand how the code works. This can be invaluable if you give your code to someone else, work on a collaborative project, or even re-visit a section of your own code months or years after writing it. Writing this type of comment in the most useful way is a skill it takes time to acquire and we're not going to go into too much detail here, but it's important to at least be aware of the syntax.
Another use for comments is to temporarily disable a section of code. For example, putting a comment here will disable the change of value of x to 4. This is sometimes referring as "commenting out" code and can be useful when debugging.
By removing this hash symbol, we "un-comment" the line and cause it to execute again.
Another piece of syntax commonly used to comment codes is to create a multi-line string, which is created by placing a triple set of quotation marks at the beginning and end of the comment. This can be useful for creating a longer comment as it doesn't require a new hash on each line.

Unit 2.2 Variables and Assignments

Title: Variables and Assignment

Narration script
Welcome to the video "Variables and Assignment". After watching this video, you will be able to: • Describe what a variable is and how it relates to data stored in your computer • Describe the rules for variable names in Python • Use the assignment operator to give values to variables

In all coding languages, a variable is a label used in the source code to reference a piece of information stored in your computer's memory. By using the variable name in the source code, it's possible to retrieve the referenced value to be used in calculations. A variable name is associated with a value through "assignment"

In Python there are important rules to be aware of regarding the names of variables. These rules include:

- Variable names must start with a letter or the underscore character
- Variable names can only contain alphanumeric characters and the underscore character
- Variable names are case sensitive For instance "x1" with a lower case "x" is a different name to "X1" with an upper case "X" Both "x1" and "X1" may exist independently at the same time.
- Variable names may not use reserved keywords which typically have a special function in the language

This is the full list of reserved keywords in Python 3.

Fortunately, most places you might write Python code, including Jupyter notebooks, will highlight the word in a different colour whenever you use a restricted keyword. This means you don't have to remember the whole list.

In this case here we tried to use "and" as a variable name and this is a restricted keyword, so we got an error.

Before a variable can be used in a calculation or expression it must be created by being assigned to, like this. This code cell creates a variable named "a" and gives it the value 3.

Here, the "equals" sign is referred to as the assignment operator.

The variable name to be assigned to is placed on the left of the equals sign. In this case, this variable is named "a"

On the right-hand side of the assignment operator we write an expression. This may be a single value, the name of another variable or a calculation. Here we've written a simple calculation.

We can check the variable has been created and has had a value assigned to it by using the print statement. We see here that "a" does indeed have the value "3" as we would expect.

If we assign a value to a variable which already exists, it will be given a new value and the old value will be lost.

If we try to use a variable that hasn't been created through assignment yet, we'll see we get an error. For instance, here we try to print the value of "b" before assigning to it.

When we run this code, the statements execute from top to bottom so, when Python tries to access the value of “b” to print it, no variable with that name exists. This causes Python to raise a NameError telling us it doesn’t recognise that variable name.
When examining the assignment operator, it’s important to know that the value of the expression to the right of it is calculated and **then** the value is assigned to the variable on the left of the operator.
This means the following code is completely valid. The line “x = x + x” will double the value of x. This is one way in which the assignment operator behaves differently to the equals sign in algebra, where this line would only make sense for x = 0.
Another way in which the assignment operator is different to the equals sign in algebra is that the variable name must be written to the left of the operator. For example, this code will not work because Python cannot assign to the value 3. Python tells us this by raising a SyntaxError.
This video should have helped solidify some of the concepts introduced in the “My First Code” unit and given you a better idea of how variables are created by assignment and the rules relating to both variables and assignment.

Unit 2.3 Common Variable Types

Title: Types

Narration script
In Python, every value, including variables and results of expressions, is an “object”. An object is an example or “instance” of a particular “type” or “class” of data. You may think of a type as a blueprint for an instance of that type. This describes how Python should store the data associated with an instance of that type and what features and behaviours the instance will have. In Python, the words “type” and “class” are largely interchangeable.
You can find out the type of an expression by placing it inside the parentheses following the word “type”.
Here we’ve made an expression which is a simple calculation and asked Python for its type.

If we put this inside the parentheses following the word “print”, we will print the type of the expression.
When we run the code we see it has the class of “int”, which is short for “integer”. Recall, that the word “class” means the same as “type” in modern Python.
We can also test the type of some other values. We can create a varaible call “a” and give it the value 1.1. When we ask for the type of “a”, we see it is a float.
When we ask for the type of the value “words”, we see it is a "str”, which is short for string.
In this unit, we’ll go on to discuss the properties of all of these types and more.

Title: Numeric Types

There are three numeric types in Python, which are used to hold numerical values: integers, floats and complex numbers. We’re going to look at integers and floats and, for those of you that are interested, there will be an extension section on complex numbers.
Integers (or “int”s for short) can be used to represent whole numbers, such as –4 or 1000. We can create an integer by writing a number without any decimal point. When we print its value, it will be displayed with no decimal point.
Floats are a type of value which can represent non-whole numbers. We can create a float by writing a number with a decimal point.
Floats can also represent whole numbers, so long as what follows the decimal point is zero. Note that, even though this is a whole number, it will still be printed with a decimal point to indicate it’s a float.
When printing a float, Python will print as many values after the decimal point as it takes to precisely represent the number stored in memory. I the last cell, this could be achieved with a single zero.
In this case, Python will print a large number of digits to represent the number as precisely as possible.
In this example, we also note that the 4 and the 3 are integers, but the 1.3 recurring is a float. This demonstrates that Python’s mathematical functions may operate on integers but return a float.
In fact, the division operator will always return a float, if it operates on a mixture of ints and floats. However, for some operators, the type of value returned will depend on the types of the values operated on. The addition operator is an example of such an operator.
If we operate on two ints, an int will be returned.
If we operate on two floats, a float will be returned.
And, if we operate on a float and an int, a float will be returned.
This shows us Python will convert between floats and integers dependent on the operation being performed and the values being used in the calculations. Sometimes, however, we will want to make sure we specifically have an integer or a float. To do this we can use the following functions.

If we write “float” and a pair of parentheses containing a number, the result will be the value expressed as a float. Note that the value has a “.0” after it to indicate it’s a float.
If we write “int” and then a pair of parentheses containing a number, the result will be the number rounded toward zero to the nearest integer, expressed as an integer.
Note that, here, we chose to put a variable in the parentheses. In this case, the value associated with the variable was used in the calculation of the new value but was not altered and it still 2.6.
If, instead of rounding down, we want to round to the nearest whole number, we can use the "round” method instead. In this case, we’ve placed a calculation in the parentheses. Python will calculate the value of d + 1 and the round function will return this value rounded to the nearest integer, which is 4.
In this video you’ve learned about what types of data the int and float types can store, how we specify values of these types, how we convert between these types, how mathematical operations may return ints or floats and how these types are displayed when printed.

[Jupyter Notebooks: Numerical Types Sample Solution]

Written exercise:

At each stage of the following, use print statements to check you get the result you expect. Perform all of these calculations in a single cell:

- Create a variable named `my_number` and set its value to 10.8
- Create another variable named `round_down` which is the value of `my_number`, rounded toward zero
- Create another variable named `round_nearest` which is the value of `my_number`, rounded to the nearest integer
- Create another variable named `new_float` which is the value of `round_nearest` expressed as a float

Run your code and check it works as you expect. Then edit your code so you give `my_number the value of` -3.6 instead of 10.8 at the start of the cell and re-run the cell.

Student exercise answer

Narration script

First, we're going to create the variable "my_number" and use the assignment operator to give it the value 10.8. We'll put in a print statement as well to check its value.
Now, we create "round_down" and use the "int" function to give it the value of my_number rounded toward zero to the nearest integer. This gives it a value of 10.
To instead round to the nearest integer, we use the "round" function to produce the "round_nearest" variable. We see this gives a value of 11 instead of 10.
We can create a variable called "new_float", which is the value of "round_nearest" expressed as a float, by using the "float" function. We see this gives new_float the value of 11.0. The ".0" indicates that it is indeed a float.
Finally, we're asked to change the value of "my_number" to –3.6 and re-run the cell. When we do this, we see that "round_down" has rounded toward zero to give a value of –3 and "round_nearest" has rounded to –4.

Title: Strings

In this video we'll discuss strings. A "string" is type which contains a collection of characters. We've already seen how to create a string, using either single or double set of quotation marks. This can contain characters, number or punctuation such as spaces or full-stops. When we run this code, we see Python tells us "a" is a "str" - this is the name of the type, which is short for "string".
A string is what is known as a "collection" in Python. This means it may contain many different related values. In the case of a string, all the contained values must be characters.
More specifically, a string is referred to as a "sequence". This is a special type of collection, where the order of the contained values is important. It also means that we may access an individual character of a string by writing the name a string variable, followed by a set of square brackets containing an integer. This integer is referred to as the "index". Here, we are asking Python to print the character in the string "a" with the index 1.

The value Python will print here is “h”. You might have been expecting a capital “P” to be printed as this is the first character of the variable “a”. The reason this doesn’t happen is that Python, like many other coding languages, gives the first value in a sequence the index 0 and the second value an index of 1 and so on.

The reason for this is largely historical. If it seems odd to you, you’re not alone. If you’re working with a sequence such as a string, and are getting the wrong value or an error, it can be worth checking “am I remembering the first index is zero not one?”. You’ll probably make this mistake more than once if you haven’t encountered this convention before. The good news is that it becomes natural after a little practice.

In our example, this means the capital “P” has an index of 0 and the “h” has an index of 1, so the print statement here prints a “h”.

Another way we can access characters within a string is to use a negative index. In this case, an index of –1 will access the last character, an index of –2 will access the penultimate character and so on. Here, using an index of –2 accesses character “1” as it’s the second-to-last character of the string “a”.

When we access a character from a string like this, it is itself a string, containing a single character.

We can also extract a string which contains multiple characters of the original string. We do this by sampling characters from a range of characters in the original string. We do this by putting three values, separated by colons, in the square brackets following the string, like this.

The first two numbers define the range of indices from which we sample characters. The first of these is the index of the first character which belongs to this range. The second is the index of the first character after the range we sample from. That means, in this case, the characters we are sampling from have the indices 1-4.

The third value is known as the “step”. This tells Python how often we want to sample the characters in the range we defined with the first two values. In this case, a step of 2 means we are asking Python to access every other character in the string. A step of 1 would mean every character in the range, a step of 3 would mean every third character in the range and so on.

So, in this case, we’ve selected the characters with indices 1-4 and asked for every other character. The first character returned is the character at the start of the range – the “h” which has an index of 1. The next character is the “a” which has an index of 3. The character with index 5 is not returned as this is after the range of characters we’re sampling from. Thus, we get a string with the characters “ha”.

When we place two colons inside a set of square brackets following a string, we don’t need to specify all three values. If we omit the first value, Python assumes we want to sample from the beginning of the string.

If we omit the second value, Python assumes we want to sample until the end of the string.

If we omit the third value, Python assumes we want to take every character from the sampled range.

Python can also tell us the number of characters in a string. By writing “len” and then a set of parentheses with a string inside, Python will evaluate the length of that string. In this case, the variable “a” is a string with 9 characters.

The plus sign has a special meaning for strings in Python. If the plus sign is placed between two strings, Python will interpret it as the concatenation operator. This means Python will create a new string that is the combination of the two strings either side of the concatenation operator. For example, we can create a new string that contains “Phrase 1.” from the string “a” and a new string “Another phrase” we create here. Note that this doesn’t alter the contents of the variable “a” but creates an entirely new string.
In this video, you’ve learned what strings are, what type of information they store, how to access individual characters or sets of characters from a string, how to find the length of a string and how to join strings together.

[Jupyter notebook: string_exercise]

Written exercise:
 	
Open the Notebook “strings_exercise.ipynb”. Look at the code and write down what you predict what will be printed. Then run the code and make sure what is printed matches your prediction.
 	
In a new code cell, write code which does the following:
• Define a string of your choice with at least ten characters
• Print the first character of the string
• Make two new strings
  ○ One from the first three characters
  ○ One from the last three characters
• Join these strings together to form a new string
• Make another string containing every other character of your original string beginning with the second character

Student exercise answer Narration script	
First, we’ll analyse the code provided as part of the exercise.	
We know the variable “first_string” is a single character from “string1” as there is only one value inside the square brackets. Because this value is a minus 3, we know it’s the third character from the end of the string, in this case “h”.	
When Python calculates the value of “second_string”, we have three values separated by colons in the square brackets, so we know Python is going to sample from a range of characters in the string. The first value is 1 and the third value is 3. Remembering the first character has index 0, this means the second	

character is the first one returned and then every third character is returned. So the characters “sya” are returned. The character “u” is not returned as it has the index 10 and the second value in the square brackets is 10. This is the index of the first character not in the range to be sampled, so the "u" is not returned.	
When calculating the value of “third_string”, there are two strings, joined by a concatenation operator. In the first string, there are two colons, so we’re still asking Python to sample in a specified range of characters. However, as the first and third values are missing, the sampled range begins at the start of the string and the step is 1, so every character in the range is sampled. The second value in the square brackets is 3, meaning the sampled range ends just before the character with index 3. This means the first string is “is “, with a space after the “s”. The second string to be joined is the entirety of “string1” so the final string is “is Python”.	
Finally, we ask Python to calculate the length of “third_string” using the “len” function. If we count the character of the string when it’s printed above, we see it has 9 character including the space, so this is the number that is printed.	
Now we’re going to move on the second part of the exercise. I’ll create a new code cell and create a string called “my_string”. I’m going to give it the value “Python is fun”.	
To print the first character of the string, I need to write the name of the string variable, then a set of square brackets containing the index of the first character, which is zero	
Next, I’m going to make the two short strings. I'm going to name the string containing the first three letters “start_string” . We want to use the first three characters, so I’m going to leave the first value in the square brackets blank to show we want to sample from the start of the string. Because we want the characters with indices 0, 1 and 2, we want the second value to be 3 as that’s the first value not in the sampled range. Finally, we want to sample every character in the range we’ve selected so we can leave out the step so it defaults to 1.	
I’m going to store the last three characters of the string in a variable named “end_string”. The first character we want is the character three characters from the end of the string. We could count from the start of the string to find the index of this character, but it’s easier to write “-3” for the first character to show we want our new string to begin 3 characters from the end of our original string. We	

want the new string to run until the end of the original string so we leave the second value blank. We want every character in this range so we leave the step blank too.	
Next, we want to create a new string which is the combination of these two new strings. To do that, we use the concatenation operator and save the result into "joined_string" like this:	
Finally, we want to create a string which contains every other character from our original string. I'm going to call this variable "alternating_string". When filling in the square brackets, we want the step to be 2 so we take every other character. The question asked for the new string to begin with the second character of the original string, so the first value should be 1, as this is the index of the second character. As we want the new string to run until the end of the old string, we can leave the second value in the square brackets blank.	
This concludes the exercise.	

Title: Lists

[Jupyter Notebooks: Lists Sample Solution]

Written exercise:

Create a new code cell. In that cell write commands which:

- Create a list named `cuddly_animals` with the names of at least two cuddly animals as items
- Append the name of another cuddly animal to the end of the list
- Insert the name of yet another cuddly animal so it is the first entry in this list
- Change the second entry in the list to a final cuddly animal

What happens if you try to print the value of an item of a list using an index greater than the number of items in the list?
What happens if you try to set the value of an item of a list using an index greater than the number of items in the list?

Student exercise answer Narration script	On screen
To create the list of cuddly animals, I'm going to create a variable named cuddly_animals. On the right-hand side of the assignment operator I use a pair of square brackets to create a list. Inside the brackets we place the words "rabbit" and "hamster" inside quotation marks and separated by a comma.	Live code and run: cuddly_animals = ["rabbit", "hamster"]
When we print cuddly_animals, we see it's a list containing those two animals.	Live code and run: print(cuddly_animals)
To add another cuddly animal to the end of the list, we can use the "append" method like this. This adds "rat" to the end of the list.	Live code and run: cuddly_animals.append("rat") print(cuddly_animals)
To insert an item at the start of the list, we use the insert method, like this. We want our new animal to go at the start of the list and so the first argument we provide is zero, to show we want to insert the value so it has an index zero. Next, after a comma, we provide the second argument, the name of the animal we want to add. Let's use "chinchilla".	Live code and run: cuddly_animals.insert(0, "chinchilla") print(cuddly_animals)
Next, we're asked to change the second value to another cuddly animal. To do this, we'll use item assignment. We write the name of the list we want to edit, "cuddly_animals", and then a set of square brackets with the number "1". This tells Python we want to change the item in the list with index "1". Then, we use the assignment operator to change this value to "ferret"	Live code and run: cuddly_animals[1] = "ferret" print(cuddly_animals)
We're asked when happens when we try to print an item from the list with an index greater than the number of items in the list. So let's try printing the entry from the list with index 99. We can do this by using the print statement. Inside the parentheses we write the name of the list and then the index 99 inside a pair of square brackets. When we run this, we get an IndexError. Reading a little closer, the error says "List index out of range". This is telling us that Python can't print the value from	New code cell. Live code and run: print(cuddly_animals[99])

cuddly_animals with an index of 99 because there aren't enough items in the list for one of them to have an index of 99.	
The final thing we're asked to do is assign to the value of cuddly_animals with an index of 99. We attempt to do item assignment as normal. However, when we run this code we get another IndexError. This time, Python tells us "List assignment index out of range". This means that Python cannot assign to the item in the list with an index of 99, because no item in the list has an index that high.	New code cell. Live code and run: cuddly_animals[99] = "gerbil"
This concludes this exercise on lists.	

Title: Extension: Complex Numbers

Narration script
This video, "Complex numbers" is an extension section relating to how complex numbers are handled in Python. It is entirely optional and is not required for the completion of this course. By the end of this video you will know how to create complex numbers in Python and perform operations with them.
There are two ways you can create a complex number in Python. The first is to write the word complex and then a pair of parentheses. Inside the parentheses put the real and imaginary parts, separated by a comma. In this case, we've created a complex number with a value of 1 + 2j.
We can check its type like this
and print its value like this.
Another syntax we can use to create a complex number is by writing a number, followed by the letter "j". This creates an imaginary number. This is a complex number by itself, but it is only the imaginary part.
We can add a real part simply by using the addition operator and a real number, like this.

Addition and other operators also work when applied to two complex numbers. For instance, we can add or multiply the two complex numbers we just created.
In this video we've seen two ways to create complex numbers and seen how we can use them in mathematical operations in the same way as real numbers.

Title: Extension: Dictionaries

Narration script
This is an extension video relating to the dictionary type in Python. This video is entirely optional and is not required for the completion of this course. By the end of this section you will know how to create dictionaries and how to use their basic functions.
Dictionaries are similar to lists, except the values stored in a dictionary have no order and they use a "key" instead of an "index" to reference their stored values. This key does not need to be an integer, as with a list. Instead, it may be any "hashable" value. We won't go into the details of what this means here, but this includes strings, integers and reals. Like a list, any type of value may be stored as a value in a dicitonary.
We're going to create a dictionary which represents the atomic numbers of the first few elements in the Periodic table. We'll begin by creating an empty dictionary. To do this, we use a pair of curly brackets. When we print it, we see that Python shows us it's a dictionary by printing a set of curly brackets.
We can also specify some initial values of items in the dictionary and the keys we're using for them. To do this we specify key-value pairs inside the curly brackets. In each pair, the first value is the key and the second is the value. These are separated by a colon. For example, here, we've given the dictionary an item with a value of "1" and a key of "H", to represent Hydrogen, which has an atomic number of 1.
If want to specify multiple items when we create a dictionary, we can do that by specifying multiple key-value pairs, separating them with commas. Let's add an entry for Helium.
We can also assign a value after creating a dictionary, in a very similar way to a list. By writing the name of the list and a set of square brackets containing the key of the value to be added, then the assignment operator, followed by the value, we add the value to the dictionary with the specified key. Note that, unlike a list, we were able to assign a value to a key that didn't previously exist using the item assignment syntax.
We can also use item assignment to change an existing value, like this:
We can access a single value of the list by placing the key inside a set of square brackets immediately after the name of the dictionary.

Dictionaries are useful in Python when the natural way to organise a collection of data isn't by each entry having an integer in a continuous sequence beginning from zero. In this example, it made sense to organise the data of the atomic numbers of different elements by their symbol, while there would not have been a sensible way to organise this data in a list.

In this video, you've learned what a dictionary is, what keys and values are in the context of a dictionary item, how to create a dictionary, how to add and change values, and how to access a stored value.

[Jupyter Notebook: Dictionaries Sample Notebook]

Written extension exercise:

Create a dictionary of your own call "`contacts`":

- It should have two entries, each having the key of a person's name
- The entries themselves should also be dictionaries and have keys `phone_number`, `address` and `email` with appropriate values
- Create at one of the values in "contacts" when you create the dictionary and at least one using item assignment
- Print "contacts" to check it looks how it should.

How do you print the phone number of a particular person in "contacts"?

Student exercise answer

Narration script

In this exercise, we're building a dictionary to act as an address book. First, we'll create a dictionary and assign it to a new variable, contacts. Currently, this dictionary is empty. We want to give it a value when we create it. Let's create an entry for a person named Alice. We want the key of this new value to be "Alice", so let's write that in. We want each value in contacts to be a dictionary, so let's start this value off as an empty dictionary.

Now we can start to give the dictionary relating to Alice some values. Let's give her a phone number, an address and an email.

We want to give contacts a value relating to another person. Let's call them Bob. We want to use item assignment for this one, so we use the square brackets following "contacts" to set this value. Again, we want this to be a dictionary.

Let's again give it values for their phone number, address and email address.

Now, we can print our dictionary of contacts. We see that it contains two dictionaries, each describing the contact details of a person. The name of the person is the key for the appropriate dictionary, so we can be happy we've constructed this correctly.

If we want to print just the phone number of a single person, we can first of all print the dictionary relating to that person. Let's do this for Alice. We see that the dictionary of Alice's contact details is printed.

To access Alice's phone number, we can put another pair of square brackets after the pair containing "Alice". This will access a value stored in the dictionary relating to Alice's details which we printed before. If we write "phone number" in this second set of square brackets, we see we get Alice's phone number printed.

Unit 2.4 Conditionals

Title: Boolean operators

Narration script
The Boolean type is important in Python for controlling the flow of a program. An instance of the Boolean type can take two values – True and False.
We can create a Boolean value by writing the words “True” or “False”, using a capital “T” or “F” respectively. We can save a Boolean value to a variable just like any other data type.
We can check the type of this variable we’ve just created using the “type” function. We see that it is indeed a “bool”.
Before we look at how we can use Boolean variables to control the flow of a program, we’re going to look at how we can use Boolean operators to generate Boolean values dependent on the state of the variables in our program.
The first operator is the “less than” operator. This will give a value of “True” is the value of the expression on the left of it is less than the value of the expression on the right, and “False” if the value of the expression on the left of it is greater than or equal to the one on the right.
The next operator is the “greater than” operator. This gives a value of “True” if the value on the left is greater than the value on the right.
The equality operator is written with two “equals” signs and tests if the values on the left and right of it are equal. If they are, “True” is returned. If not, “False” is returned.
Not that, for numeric values, it is the numeric value that is compared. This means the values of the integer 1 and the float 1 point 0 are the same, even though they are different types.
The inequality operator is made up of an exclamation mark and an equals sign and will return “True” if the values on either side of it have the different values and “False” if they have the same value.
The “in” operator is written with the word “in”. It checks if the value on the left of it appears in the collection to the right of it. This could be a list or a string. Here, we’ll check to see if the value 3 appears in this list with the values 4, 3, 5. Because the list does contain the value “3” the value “True” is returned.
If the value on the left isn’t in the collection on the right, “False” will be returned instead.
For a string, the “in” operator will check to see if the string on the left is contained in the string on the right. In this example, “True” is returned because the string “ytho” is found in the string “Python”
We can also combine Boolean variables using logical operators which result in a Boolean value. The first of these is the “not” operator. This operator is placed before a single Boolean expression and will return the opposite of that Boolean expression. For example, “not True” will be False.

The “or” operator requires a Boolean expression either side of it and will return True if either of those expressions evaluate to True. In this example, the expression on the left is False and the one on the right is True. This means the overall expression evaluates to True.
Like the “or” operator, the “and” operator requires a Boolean expression on both the left and right of it. If both are True then the overall expression will be True. If either are False, the overall expression will be False. So, when we change the previous example to an “and” we get False as the expression on the left is False.
This concludes this video on Boolean operators. You now know how to set a value in Python to be True or False, including as a response to the values of expressions in your code.

[Jupyter notebook: booleans_exercise]

Written exercise: Open the Notebook “booleans_exercise.ipynb”. Look at the code and write down what you predict what will be printed. Then run the code and make sure what is printed matches your prediction.

Student exercise answer

Narration script

You were asked to examine this code and predict what would be printed. Let's run the code and see what is printed before we try to explain why.

In Case A the "not" operator operates on the Boolean value True. The not operator inverts this value, so the result is "False".

In Case B we have two Boolean expressions, one either side of an "or" operator. The one on the left is False, because 3 is not greater than 4. However, the one on the right is True, because the number 1 does appear in the list [1,2]. This means at least one of the Boolean expressions adjacent to the "or" operator is True, so the overall value returned is True.

In Case C, we have an "and" operator, so we need both Boolean expressions to be True for the overall expression to be True. The first uses the modulo operator to return the remainder when 10 is divided by 2. 10 divides by 2 exactly so the remainder is 0, so that expression is True. As an aside, checking if the modulo of a Value A divided by a Value B is zero is a good way to check if Value B is a factor of Value A. The second Boolean expression here is True because 5 is not the same value as 6. This means both Boolean expressions are True and so the overall expression evaluates to True.

In Case D, we create a variable "list1" which is a list containing the values 1 and 2. Again, we're using an "and" statement, so we need both Boolean expressions in the parentheses to be True for the printed value to be True. The first value is True as 5 multiplied by 2 is 10, which is greater than 8. However, the second expression is False because "list[1]" refers to the second item in list1 which is 2. This is not equal to one, so the Boolean expression on the right is False. This means there is at least one False statement adjacent to the "and", so the overall value is False.

In Case E we need to look at the equality operator before considering the "not". "list2" is another list containing the values 1 and 2. This means it has the same value as "list1" so this expression is True. However, this expression follows a "not" operator, so the overall value inside the parentheses is actually False.

Title: If Blocks

Narration script
Now you know how to create True/False values dependent on the values of expressions in your code, we're going to see how we can use this to control the flow of our code. Specifically, we're going to look at how we can use an if block to decide which, if any, of a selection of pieces of code are executed.
To begin an "if" block, we use the word "if". This tells Python we're about to specify an "if" block. This means we're going to write a piece of code which will be executed if a particular statement has the value of True.
Following this, we should write an expression which gives a Boolean value. Let's make a simple one to start with. This is the expression which must evaluate to True if the next section of code is to be executed. We follow this with a colon to tell Python we've finished specifying our Boolean expression.
We then start a new line of code. This line of code must be indented. Fortunately, Anaconda knows this line needs to be indented and does this automatically for us.
The section of indented code we're writing now is the section of code that will be executed if the Boolean expression above it is True. This can be multiple lines, but each line should be indented the same amount.
Once we've finished specifying the code to be conditionally executed, we can specify more code, this time without the indent. In Anaconda we can remove the automatic indent by pressing backspace. This code will be executed after the "if" block has completed, whether the conditionally executed code was executed or not. As such, the "if" block is ended when the indentation finishes.
When we run this section of code, Python evaluates the value of 2 < 1, which is "False". Because this is not "True" the indented section of code is not run and only the final print statement is executed.
However, if we change the "less than" sign to a "greater than" sign, the Boolean expression is now True. This means the indented code is now executed and assignment to "a" and the printing of the value of "a" both happen before the final print statement.
We can include multiple "if" statements inside each other if we want. We do this by indenting the word "if" and then indenting the content of this inner "if" block two steps. Again, Anaconda helpfully does the indentation automatically for us.
If we then include code which is one step indented, this will be part of the conditionally executed code of the outer "if" statement, but not the inner one. This ability to have constructs such as "if" statements nested within each other and their relationships determined with the degree of indentation is common in Python and applies to other constructs we'll look at later in this course, including "for" loops.
For now, let's simplify the code before moving on.

We can add more pieces of code which may be executed to an “if” block. The first way we can do this is by using the “elif” statement, which is short for “else if”. This is indented the same amount as the “if” statement. In this case, that means it’s not indented at all.

Following the “elif” we again write another expression which will return a Boolean. The value of this expression will only be evaluated if the Boolean expression associated with the “if” statement above it was False. Following the Boolean expression, we again use a colon to indicate to Python the expression has finished.

Then, we can begin writing the lines of code which will be executed if the Boolean statement was true. We do this on a new indented line. Just like after the “if” statement, we can include many lines of indented code if we wish.

When we run this code, Python checks the value of the Boolean expression following the “if” statement. Because this is False, Python doesn’t execute the first piece of indented code and moves on to check the Boolen expression following the “elif” statement. As this is True, the following print statement is executed.

We can include multiple “elif” statements if we chose by repeating the syntax. However, note that once any Boolean expression following an “if” or “elif” statement has been evaluated to be True and the associated code has been executed, Python will skip to after the end of the “if” block. It will not check any Boolean expressions associated with any following “elif” statements, nor will it execute any associated code.

In this case, even though the Boolean expression of the second “elif” statement would be True, Python will not try to check this and the associated print statement will not trigger.

Finally, if we chose, we could use an “else“ statement. This must follow any “elif” statements and is inserted by writing the word “else” on a new line. This is followed by a colon and, on the next line, indented code. This code will be executed if and only if the code associated with the preceding “if” statement and any “elif” statements is not executed as none of the Boolean expressions were True.

In this case, because the Boolean expression associated with the first “elif” statement was True, the else statement isn’t executed.

However, if we modify the Boolean expressions associated with the two “elif” statement so that they’re both False, the code following the “else” statement will be executed.

As a summary, you do not need to include any “elif” statements in an “if” block. But, if you do, you can include as many as you like.

You don’t need to include an “else” statement at all. But, if you do, you can only have one. A maximum of one block of indented code will be executed. If an “else” statement is included, exactly one block will be executed. Without an “else” block, it’s quite possible for no indented code to be executed if all the associated Boolean expressions are False.

The fact that only one set of code inside an “if” block will be executed and that the Boolean statements are evaluated from top to bottom means that care must be taken in constructing the logic of an “if” block, to ensure the block performs the correct way in all cases.
“If” blocks are useful when you want your code to do different things in different cases, dependent on the value of pieces of data in your code. For instance, you might use an “if” block to examine values obtained from an experiment. If they’re above a certain threshold, you might classify them as an anomaly and discard them. Otherwise, you process them normally.
You should now understand the syntax of an “if” block. The exercise which follows will allow you to practice the logic required to construct an “if” block to perform a specific task.

[Jupyter notebook: booleans_exercise]

Written exercise: In a new code cell, write a piece of code which defines a as an integer. Follow this with an if block which will print "Fizz" if “a” is a multiple of 3 and "Buzz" if it's a multiple of 5. If it's a multiple of 3 and 5, print "FizzBuzz" instead. If it's not a multiple of 3 or 5, print the value of “a” instead. Test your code for a few values of `a to ensure it works as you expect.` The modulo operator may be useful to you.

Student exercise answer

Narration script

There are a few ways we can structure this piece of code. The first thing I'm going to do is create the variable "a" and give it a value of 15.
Now, I'm going to write an "if" statement to see if it's a multiple of both 5 and 3. I test this first because, if this is the case, we always want to print "FizzBuzz". By comparison, if we only know it's a multiple of 3, we don't know if we should be printed "Buzz" or "FizzBuzz".

To test if "a" is a multiple of 3, I evaluate the modulo of "a" and 3. If this is zero, "a" is a multiple of 3. So, we can use the equality operator to check if a modulo 3 is zero. We also want to check if it's also a multiple of 5, so we use an "and" operator. This means our first piece of indented code will only be triggered if a is a multiple of both 3 and 5, so we can use it print "FizzBuzz".
Now we can write the first "elif" statement. We'll check to see if a is a multiple of 3. If the code has reached this point, we know it is not multiple of both 3 and 5. So, if it's a multiple of 3, we know it is a multiple of 3, but not a multiple of 5. In this case, we want to print "Fizz".
Next, we test to see if it's a multiple of 5. This will only be tested if "a" is not a multiple of 3 because, if it was, one of the previous bits of code would have been executed. As a result, if the line below is executed, we know "a" is multiple of 5, but not of 3, so we print "Buzz".
If none of the earlier statements have been triggered, we know "a" is not a multiple of 3 or 5, so we can use an "else" statement to print the value of "a" instead.
After writing a piece of code like this, it's always a good idea to test it. We already know we get "FizzBuzz" if "a" is a multiple of both 3 and 5, as desired. So let's change the value of "a" to 9, which is a multiple of 3 but not 5. Sure enough, we get "Fizz" printed.
Next, we'll change "a" to 10, so it's a multiple of 5 but not 3. Just as desired, we get "Buzz" printed.
Finally, we'll change "a" to 8 so it's not a multiple of 3 or 5. In this case, we get the value of "a", which is 8, printed.
Now we've tested all the cases, we can be happy that our code fulfils the specification provided in the problem description.

Unit 2.5 Loops

Title: Syntax

Narration script
A “for” loop allows you to repeatedly execute a piece of code, using a “loop variable” which changes with each execution of the piece of code. This can allow the same process to be applied to many different pieces of data. In the first example we’re going to look at, the loop variable will take its values from a list.
Let’s create the list we’re going to loop over. We’ll fill it full of integers for this example.
Now we’ll write the for loop itself. To do this, we first use the word “for”. This is followed by the name of what’s called the “loop variable”. This is a variable which will take on different values from the list in different iterations of the loop. It can have any valid variable name, but it’s best to give it a name that’s descriptive of what it represents. Here, it’s going to take values from the items of the list, so I’m going to call it “item”.
Now we need to tell Python where our loop variable will get its values from. To do this we write the word “in”, followed the iterable value we want the loop variable to take its values from. An “iterable” value is a value which can provide a series of values one after another in a context such as a for loop. Most collections, including lists and strings are iterable. Here we’re going to use the variable name “my_list” to use the list we created earlier.
Following the value we want to iterate over, we need to use a colon which indicates to Python we’ve finished specifying this part of the for loop. Then we start a new line which is the start of the block of code which will be executed for each iteration of the loop. Let’s print the value of “item”. When we run the code, we see that the print statement has been called four times and, each time, “item” has a different value from the entries in “my_list”.
You may have noticed that, when I pressed enter after the colon, Anaconda automatically indented the line with the print statement on. This is because the block of code to be executed each iteration of the loop must follow the colon and this block of code must be indented. The indentation tells Python that the code is to be executed each iteration of the for loop. If we want to define more code to be executed after all iterations of the loop have finished, we place another non-indented line of code after the indented section. So, now, when we run the code we see that after all values of the list have been printed, the print statement in the final line is executed.
We can also have multiple lines of code inside the indented section. Let’s create a new variable in a calculation based on the value of “item” and then print it. Now, when we run the code, we see that there are two print statements per iteration. The first prints the value of “item”, the second printing the value of “a”, which is one larger than the value of item. So, in the first iteration of the for loop, “item” has the value 1 and “a” has a value of 2. The next iteration, “item” has a value of 2 and “a” has a value of 3, and so on.
We can also place other, more complex pieces of code inside a for loop, such as an if statement. When we do this, we indent the line containing the “if” the same amount as the rest of the indented text inside the for loop. Let’s make this if statement trigger if a is a multiple

of 2. When we press enter after the colon, Anaconda will automatically double indent the next line. This signifies that this code is within both the for loop and the if statement.

This pattern of using indentation to signify which Python constructs nest inside each other can be used for any indented pieces of code, including any combination of if blocks, for loops and functions, which we'll look at in a future video. We can nest any number of constructs inside each other in this way, adding an extra layers of indentation for each. However, after a few levels of indentation, this can make code difficult to read so, often, it's good practice to reorganise your code if you're indented more than two or three times.

In this case, we're going to print the word "Even" if "a" is even. When we run the code, we see this printed at the end of the first and last iterations of the loop when "a" is 2 and 6 respectively.

This concludes this video. You've now seen all you need in order to use a loop to use each item of an iterable variable, such as a list or string, one after another. Now, have a go at the exercise to practice constructing a loop of your own.

Written exercise:

Create a string of your choice with around twenty characters. Use a for loop to loop over all the characters of your string and print all the vowels ("a", "e", "i", "o", "u") in either upper or lower case. Each letter may be printed on a different line.

Student exercise answer

Narration script

We're going to start off by creating a string with the value "A longish string to test"
We want to test each character in turn to see if it's a vowel and, if it is, print it. The first step to this is to write a for loop which will give us each character in turn. A string is iterable so we can use it in a for loop, like this. I've chosen the name "letter" for the loop variable here because it's descriptive of what that variable contains.
We can put in a print statement to check if this variable is, indeed, taking the value of each letter in turn. As we would expect, the variable letter is the character "upper-case A" in the first iteration of the loop, then it's a space in the next iteration and then an "l" in the next and so on.
However, we want to only print the value of letter if it's a vowel. To do this, we need to put the print statement within an if statement that tests this and put that "if" statement within the for loop. Note that this means we need to add a level of indentation to the print statement so it sits inside the "if" statement.
The most compact way to test if the variable "letter" is a vowel is using the "in" operator. Because we know "letter" will always be a single character, we can test if it is "in" a string containing the characters we're looking for.
In this case we want to test if letter is one of "a", "e", "i", "o" or "u", in lower or upper case. When we run the code, we see the vowels are printed as desired.
If we change the value of "my_string" and re-run the code, we get the vowels of this new string printed instead.
To summarise, the for loop caused the variable "letter" to have a value equal to each character of "my_string" in turn. The "if" statement checks if it's one of the characters we're interested in and, if so, it is printed.

Title: The Range Construct

Narration script
We've seen how you can loop over an iterable collection such as a list or string. However, sometimes you'll want to use a series of numbers in a predictable pattern and the "range" construct allows us to do this without, for example, creating a list of all the numbers explicitly.
For example, let's say we want to calculate the value of "n" factorial using a for loop. This means we want the value 1 times 2 time 3 times 4 and so on up to n.
We can generate the numbers we want to multiply together in a loop using the range construct. Let's set a variable "n", which is the value we intended to calculate the factorial of. We start the "for" loop in the usual way. I'm going to call the loop variable "number".
However, we can use the word "range" after the word "in" to cause the loop variable to take its values from the range we're going to create. We specify which values we want the range to give to the loop variable by specifying a value in parentheses after the word "range". This value is one larger than the largest value that "number" will take.
We can see the values of "number" by putting a "print" statement in the body of the for loop. When we run the code, we see that the first time the loop runs, "number" has the value 0, the next time it has the value 1 and so on. The final value "number" takes is 4, which is one less than the value of "n".
However, the values we want to multiply together, start with 1, not 0, so we need to modify the set of numbers that will be produced by the range. To do this, we can provide more numbers in the parentheses following the word "range", just as we did when we looked at indices of strings and lists earlier in the course. In this case, if we add one more number, the first number will represent the first value "number" will take and the second will represent the value one larger than the largest value to be returned. So, if we put a "1" in before the "n", we see "number" now takes the values 1-4.
We actually want the values of 1-5, so we need to increase the second value in the parentheses by 1, like this. Now, the values of "number" run from 1 to 5.
If we want to, we can also include a third value in parentheses. This acts like the "step" when indexing a collection. For example, using a step of 2 means that number will take every other value in the specified range. In this case, that means 1, 3 and 5.
For now though, we want every number in the specified range, so we could change it to 1 or delete it entirely, so the step defaults to 1. Let's go back to not having a specified step at all.
Now we're happy we've got the correct values of "number", we need to multiply them together. We can do this by using a value to track the running total of the product. Let's create a variable named "factorial" and start it off with the value of 1. Then, inside the loop, we can multiply it by the value of "number". This means we will multiply it by 1, then 2, then 3, then 4, then 5.

While we're at it, let's change the print statement inside the for loop so we print the value of factorial as well so we can see it develop with successive iterations of the loop. This syntax is an example of how we can ask a single print statement to print multiple values by separating them with commas.

We'll also add a final print statements at the end, after the end of the loop body to check it has the correct final value of 120. As a reminder, this print statement isn't indented to show that it's not part of the loop body.

Now, we when we run the code, we see how the value of "number" and "factorial" increase with each iteration of the loop. The first iteration, "number" is 1 and factorial is 1 times 1, which is also 1. The next iteration, number is 2 and factorial is 1 times 2, which is 2. The third iteration, number if 3 and so this increases the value of factorial from 2 to 6. This continues until, after the fifth iteration, we see that "factorial" has the desired value of 120.

Now, we can also work out the factorials of different values simply by changing "n" and re-running the code. Here, changing it to ten means we execute more iterations of the loop and get a different final value of factorial.

In this video you've learned how to use a "range" to generate a sequence of numbers. You've also seen an example of how we can use this sequence of numbers to perform an iterative calculation to solve a real problem. The method of iteratively updating a single variable (in our cause, "factorial") is a variation of the "accumulator" pattern, which is a useful design patter to be aware of as it's frequently useful.

Written exercise:

Use the range construct to calculate the value of the following expression:

$$\sum_{x=1}^{100} x^2 = 1^2 + 2^2 + 3^2 + \cdots + 100^2$$

You should get a value of 338,350

Student exercise answer

Narration script

We want to calculate the sum of x-squared for x = 1 up to x = 100. The first thing we want to do is to generate the different values of x. We can do this using a for loop with a range construct, like this. Note that, in the range construct, I provided a start value of 1 and a stop value of 101. This means the smallest value “x” will take is 1 and the largest value will be 100.

Now we need to do something with each value of “x” by filling in the body of the for loop. For each value of “x” we want to use the value of x-squared.

But now we need to do something with this value. We’re going to use a similar pattern to when we calculated the factorial in the example earlier. We begin by creating a variable we want to hold the final value of the sum. Let’s call it “evaluation”. We want to repeatedly add the different values of x-squared to this value, so we want it start off with a value of 0, so we’ll initially assign a value of zero to it.

Now, we can modify the value of evaluation inside the for loop, by adding the value of x-squared to it. Finally, we put a print statement after the for-loop to print the final value of “evaluation”.

The first time the for-loop executes, “evaluation” initially has the value 0 and “x” has a value of 1. “evaluation” is updated to have the value of one squared, which is one. Then the next iteration of the loop executes, “x” has the value of 2 and “evaluation” is updated to have the value of 1 plus 2-squared, which is 5. This process continues until we’ve added the square of every number between 1 and 100 to “evaluation”, as required. This gives the expected value of 338,350 for “evaluation”.

Title: Extension: nested loops

Narration script
We've already seen how an if statement can be embedded within a for loop. At the time, I mentioned how a for loop could be embedded within another for loop. In this optional video I'll show you an example, before giving you an exercise to practice the logic and syntax of nested for loops.
For this example, I'm going to try and print to the screen the number of factors each number up to 100 has. The first thing we're going to do is to use a for-loop with a range to loop over the values between 1 and 100. "x" is the name of the variable which represents the value we're currently examining to see how many factors it has.
Now, for each number equal to or less than "x" we want to check if that number is a factor of x. We can generate all these potential factors by nesting another "for" loop inside the outer for loop. By placing "x+ 1" as the upper limit of the range, we ensure that "potential_factor" will take every value between 1 and the current value of "x".
Before we go any further, let's test these loops by putting a print statement inside the inner loop. Let's print both the value of "x" and the value of "potential_factor". We'll also temporarily reduce the maximum value of x from 100 to 5 so we get a more manageable amount of output to read. The value that "x" takes in the first iteration of the outer for loop is 1. This means the range in the inner for loop produces numbers which are 1 or greater and less than 2. This means that the inner loop will only have one iteration with "potential_factor" taking the value 1. In the next iteration of the outer loop, "x" takes the value 2. This means that "potential_factor" will take the values 1 and then 2 in the inner loop. The next iteration of the outer loop, "x" has the value 3 and "potential_factor" takes the values 1, 2 and 3. This continues until the fifth iteration of the outer loop where "x" has the value 5 and "potential_factor" takes the values 1-5 in the inner loop.
Now we're convinced the loops are working, let's start counting up the factors of each value of "x". Let's add an "if" statement that will only trigger if "potential_factor" is a factor of "x". To check if "potential_factor" is a factor of "x", we ask for the remainder when "x" is divided by "potential_factor" using the modulo operator. If this is zero, then "potential_factor" is a factor of "x".
To check this is working, let's put the print statement within the if statement so we'll only print the values of "x" and "potential_factor" if "potential_factor" is a factor of "x". Note that, now we've added the if statement, we'll need to indent the print statement another level to do this. When we run the code, we see the print statement only occurs when "potential_factor" is a factor of "x", for instance when "x" and "potential_factor" are both 1 or when "x" is 4 and "potential_factor" is 2.

Now, we can identify if “potential_factor” is a factor of “x” we need to count up the number of factors for each value of “x”. Inside the outer for loop, we can assign a value of 0 to a value named “n_factor”. This will be set to 0 each time a new value of “x” is considered.
We then loop over all the potential factors of “x” and increase the value of “n_factor” if it is indeed a factor of “x”. We can get rid of the print statement as we don’t need it anymore.
Once this loop is complete, we can print the number of factors a value has. Note that this print statement is indented the same amount as the first two lines inside the outer for loop to indicate it’s not part of the if statement or the inner for loop, but is part of the outer for loop. Let’s give the print statement a nice message.
Finally, let’s change the upper value of x back to 100 and run the code. We see that the code correctly identifies that 1 has 1 factor, 2 and 3 have 2 factors and that 4 has 3 factors and so on.
This example demonstrates how we might use nested for loops in practice. While you’re getting used to using nested loops, remember to think carefully about which loop you want a statement to be within and the level of indentation required. Now have a go at writing your own nested for loop in the exercise.

Written exercise: Write a piece of code which constructs a list containing every prime number between 2 and 1000. Then print the list of primes.

Student exercise answer

Narration script

We want to construct a list of prime numbers, so let's initialise a list to hold the primes.
Now, let's loop over all the candidate numbers that might be prime. The smallest prime is 2 and we're checking values up to 1000. We'll check to see if they're prime and, if so, add them.
We'll use a variable "is_prime" to store if a number is prime or not and start off by assuming it is prime.
Next, we'll add an inner for loop, looping over each value which might be a factor of "candidate". We can use a trick here, but don't worry if you did something else that works.

We only need to check if "candidate" has any prime factors. If it has no prime factors, it must also be a prime. For example, if it's a multiple of 4, it must also be a multiple of 2. So, checking if it's a multiple of 4 gives us no extra information about if it is a prime compared to checking if it's a multiple of 2. So, we can loop over all the primes we've found so far instead of looping over every number smaller than candidate.
Looping over every number smaller than "candidate" using range would also work, but it would be slower to run.
Now, we check to see if "potential_factor" is a factor of candidate using the modulo operator. If it is a factor, we know "candidate" isn't prime, so we can switch "is_prime" to False.
Ater the inner for loop has completed, "is_prime" should be True if "candidate" is prime and False if it's not. So, we can use an if statement that's inside the outer for loop, but outside the inner for loop and use this to append the value of "candidate" to "primes", if it's prime.
Finally, outside the outer for loop, we'll print the value of "primes". We can check it contains the correct values by running the code. The first values are 2, 3, 5, 7, 11 and so on. As a result, we can conclude that "primes" contains primes and not any non-primes, fulfilling the conditions of the exercise.

Unit 2.6 Functions

Title: Calling a function

Narration script
Functions are a vital and incredibly useful tool in Python. They are a way to store and label a piece of code which can then be called from other places in the program. This allows code to be compartmentalised and reused, improving code readability and speeding up the development process. First, we're going to look at how to call a function and use the value it returns. Many functions "return" a value. This means that the function will calculate a value, often as a response to the values of values provided to it. This value will be returned, and we can use it as part of an expression if we wish. In this video, we're going to use the "max" function as an example, which calculates and returns the maximum value of a collection. This is an "intrinsic" Python function. This means that it is available in Python without having to write it ourselves or import it manually. We'll talk about writing our own functions and importing functions later in the course.
To call a function, we need to write the name of the function, followed by a set of parentheses. Into the parentheses, we place the arguments that we want to provide to the function in this call. The arguments we provide to a function are the values we give to a function for it to use in calculations. In this case, we'll provide a list. When we run the code, Python will execute the code contained within the "max" function, using the data from the list.
We see that value returned by the function ("99") is printed. This is a specific behaviour of Jupyter Notebooks where, if the final line of a code cell is an expression, its value is automatically printed. This behaviour is specific to Jupyter Notebooks in Anaconda and is not a core Python feature. We can suppress this behaviour by making an arbitrary and unimportant assignment to a variable after this line. Now, when we run the code, the value returned by the "max" statement is not printed.

The value returned by the max statement is not used in any way in the code as it exists – it is calculated and discarded. However, we can use this value in the same way as any other value. We can use it as an argument in another function, such as the print function. In this example, we can print the value returned.
We can also save the value to a variable and print it if we want.
If we want, we can use the returned value as part of a more complex expression. Let's increase the value of "result" by 5 by adding 5 to the value returned from the "max" function. We see "result" now has the value of 104.
When we call a function, we can also use a variable name as an argument and the value of that variable will be passed to the function.
We can also write expression inside the parentheses of a function and the value of that expression will be passed as an argument to the function. Here, lets concatenate my_list and a new list. This means a list with the values 3, 6, 4, 10, 2 is passed to max and, when we run the code, the max function returns a value of 10 and so a value of 15 is printed.
In this video, you've seen how we can call functions, provide data to them using arguments and use their returned results as values in an expression. In the next video, we'll look at some useful intrinsic functions provided by default in Python.

Title: Intrinsic Functions

Narration script
Now you know how to call functions and use their results, we're going to look at some "intrinsic" functions in Python. Intrinsic functions are always available and you don't need to do anything special to use them, such as importing them from a module. Thie collection of intrinsic functions discussed in this video is far from exhaustive and there are others you will come across as you get more familiar with Python. We'll also revisit some functions we've used in the past and discuss them in a little more detail, or show you a more interesting or advanced way of using them.
The "abs" function takes a numeric expression as an argument and returns the absolute value of the value of that expression. For instance, the absolute value of 2 is 2 and the absolute value of –3.1 is 3.1. Note that, in both cases, the returned value is of the same type as the value provided as an argument. In the first case, the returned value is an int and, in the second case, the returned value is a float.

The “all” function takes a collection of bools as an argument and returns True is all of the values are True and False if any of the values are False. Here, the first expression was True as all the values in the list were True and the second value was False as there is a False in the list.
The “any” function also takes a collection of bools an argument, but it returns True if any of the bools are True and will only return False if they are all False. So the first expression was True because at least one of the values in the list was True, and the second expression was False because no values in that list were True.
We’ve already seen the “float” function before and seen how it can create a float from an integer, like this. However, it can also create floats from other variable types, such as a string which contains a representation of a numeric value.
The “int” function can also be used to create an integer from a string, but only if it contains a representation of an integer, like this.
If we try to provide it with a string containing a non-integer, a ValueError will be returned as the function cannot convert the string to a number and round it at the same time.
If we want to do this, we could convert it into a float first, using the float function inside the parentheses of the int function. This means the float function will first produce the numeric value of –0.1, as a float, before the int statement rounds this to the integer 0.
The “input” function takes a single string as an argument. This string will be printed to the screen as a prompt to the person running the code. This is intended to be used as an instruction for the user to write a string. This string will be returned by the input function and provides one way for the user to provide a value into a code. Let’s write an example, where we write a piece of code which asks the user their name and then prints them a message saying “hello”. When we run this code, the message appears below the code cell, along with a box to enter our response. If I put my name in, the value will be saved to “a” and the print function will print the greeting in reply.
We’ve already looked at the “len” function. To recap, it returns the number of items in a collection, such as the number of characters in a string or the number of items in a list.
We’ve already seen how the “max” function returns the maximum value of a collection. But, we can also use the max function to find the maximum of a number of of individual values.
Similarly, the “min” function behaves in the same way but returns the minimum value instead.
We’ve looked at the “print” function before and seen that it will print some representation of the value or values provided to it as arguments. When this is an int or a float, it’s fairly obvious what will be printed. But there are many other types which can also have some representation of the data they contain printed.

For example, we can print a representation of a “range” object, where it’s a bit less clear what will be printed. When we do this, Python tells us this is a “range” object and tells us the relevant parameters, the start value, the stop value and the step value. In this way Python has communicated the important properties of this object. Many other complex types show similar behaviour when printed. The takeaway here is that when you “print” an object, Python is not simply printing the “value” of the object. In fact, Python will have a different process used for each type that it uses to describe the important information about an object of that type.
The “str” method returns a representation of the passed value as a string. So str(-1.3) will return a string containing the characters “hyphen, one, full stop, three”.
However, the “str” function can also produce a representation of any data type. In fact, this function returns the same string as is produced and printed by the print statement. For example, we can again print the string representation of the range by saving it to a variable, then printing that variable.
The “sum” function is a useful function which returns the sum of a collection of numbers, such as this list of numbers.
Finally, the “type” function is one we’ve already seen, which returns the type of the object passed to it as an argument. For example, we can get the type of an integer and save it to a variable.
We can print this variable and see it’s the type of an integer.
We can also print the type of our variable “integer”. When we do this, we see that it is itself a type (recall that “type” and “class” are all but interchangeable in Python). This is an example of how every data type, including types themselves, are actually examples of types in Python. This is getting into the underlying object-oriented nature of Python in some depth at this point. This is fascinating and useful but going into it in too much depth is beyond the scope of this course. So, don’t worry about it too much for now.
This video has introduced to some of the common and useful intrinsic functions which exist in Python and given some examples of how they can be used.

Title: Defining your own functions

Narration script
You should now be fairly comfortable with how to call functions. In this video, we’re going to look at how you can define functions of your own. This is a really useful thing to be able to do as it allows you to reuse your code, speeding up the development process significantly, as well as making your code easier to read.
To begin defining a function, we use the word “def”. This is followed by the name of the function you want to define. I would recommend picking a name which is descriptive of the task performed by the function, let’s call ours “greater_than”.

When naming function, you should pay attention to avoid the names of intrinsic functions. Although defining a new function called, for instance, “print” would be allowed, it would hide the intrinsic “print” function and prevent you accessing it. This would normally be a bad idea.

Next, we add a pair of parentheses. These will contain the names of the variables we want to exist in the function and take the values of the arguments passed to the function when it is called. These variables exist only within the function and cannot be accessed from outside the function. They can also have the same names as variables which exist outside the function. Let’s call our variables “x” and “y”.

Most functions take arguments, but a function does not need to have any arguments. In that case, we would still need the parentheses, but they would be empty.

We finish this line using a colon.

The colon tells Python we’re ready to start defining the body of the function – that is, the code which will be executed when the function is called. Just like an if statement or a for loop, the body of the function must be indented and the end of the indentation will signal to Python that we’ve finished defining our function.

We want our function to return the value True if “x” is greater than “y”. One way we can do this is by using an if block. We can write this if block with the condition x > y.

Just like when we put an if statement inside a for loop, the body of an if statement inside a function needs to be indented twice. Fortunately, Anaconda knows this and will automatically indent for us. In order to return the value True we must use the “return” construct. This will return the value of the expression within the parentheses and the execution of the function will cease. This means, if we put a print statement after the “return” statement here, for example, it will not be executed as the function will have already returned at this point. Let’s do this an example.

We see that, running the code, there is no output at all. We have only defined the code and have not run it yet.

Now we’ve written a valid function, let’s call it and see what happens. To call the function, the call to it should be after the function is defined. If we tried to call a function above where we define it, Python would give an error.

We call our function in the same way we call an intrinsic function – by writing it’s name and then expressions whose values are to be used for the arguments inside parentheses. We’ll put this function call within the parentheses of a print function so we print the value returned by our function.

When we give our functions the values 3 and 2 as arguments, the variable “x” inside the function will have the value 3 and “y” will have the value 2. This means, in the “if” statement, the condition will be True and so the value True will be returned. Note that the print statement printing the “A” is not reached because the function returns before it can be reached.

Let’s delete it as it’s superfluous.

Now, let's try running it with a value for "y" that's bigger than "x". Let's give it the value of 10. Now, the condition of the if statement is False so the return statement isn't reached. No return statement is executed in this case and the function finishes executing when the indented section finishes.

But we might wander what value will be printed? Let's run the code and find out. The value that has been printed is "None". None is a special value which is used in Python used to describe the case when an expression has no value. It is created in a couple of cases, including when a function returns no value.

The "print" function is an example of an intrinsic function that returns a value of None. We can check this by assigning to a new variable the value returned by the print statement and then printing it.

We can see here that the print statement correctly prints the value "1" and "a" is given the value "None" as the print statement doesn't return a value.

It's not necessarily wrong when a function doesn't return a value. In the example of the print function it makes sense – the function of the print statement is to output a value. It doesn't calculate anything, so it makes sense that it has nothing to return. Nonetheless, when you write your own functions, you should think about whether you always want it to return a value and, if so, make sure every path through the function ends with a return statement.

We want our function to return a value, so let's get rid of these two lines and complete our function.

We can do this by adding an else section to the if statement. This section of code will be executed if x is not greater than y, so we want to return False in this case. Now when we run this code with a set of arguments where y is bigger than x, the value False is returned and printed instead.

We can print the results of another call to this function. This time though, we're going to use a variable to pass one of the arguments. I'm going to call this variable "x" and give it a value of 5. We'll call the function and compare it to a value of 8.

Note that I've provided "x" as the second argument so its value 5 is passed to the variable "y" inside the function. The variable "x" inside the function is separate to the variable "x" outside the function and the two do not interfere with each other.

Finally, as a demonstration, let's see what happens when we print "y" outside the function. When we do this, Python gives us a NameError – it doesn't know this variable "y". This is because the variable "y" we use inside the function doesn't exist outside the function. So, when we try to print it, it doesn't exist.

Note that we must define a function before we can call it from the unindented main body of the code. If we try to call greater_than before we define it, we'll get an error. Let's restart the kernel to get rid of the saved version of the function. Python tells us that greater_than hasn't been defined when it's called. This is because the interpreter reads the call to the function before the definition of the function.

With that, we conclude this video. In it you should have learned the syntax of how to define and call your own functions in Python and the key rules associated with doing so.

Written exercise:

A dot product of two vectors **A** and **B** (which each have "n" entries) is defined as follows:

$\mathbf{A}.\mathbf{B} = \mathbf{A}_1\mathbf{B}_1 + \mathbf{A}_2\mathbf{B}_2 + \cdots + \mathbf{A}_n\mathbf{B}_n$

where $\mathbf{A}_1$ is the first value of **A** and so on.

In Python, you may choose to represent a vector as a list of numbers, with each item representing a different value of the vector.

Write a function which takes two lists of numbers as arguments and returns the dot product of the vectors they represent. You may assume the lists contain only numbers and that they contain the same number of values. Test your function with a few pairs of vectors.

Student exercise answer

Narration script

First, of all, let's define our function. I'm going to call this function "dot_product" as that's a descriptive name for the purpose of this function. We want it to take two arguments, one for each vector. Let's call them "vector1" and "vector2" as those are descriptive names for the data they'll contain.

Next, let's think about how we're going to achieve the goal. We want to take a sum of many values, so using the accumulator pattern we've used before where we create a variable with a value of zero and add to it inside a for loop would be good choice. Let's create the variable we want to eventually contain the result and call it "result".

Now we'll create the for loop. Because we want to loop over two lists at the same time, we can't use a simple "for item in list" construct. There are a couple of ways that we could cause this for loop to loop over the two lists, but I'm going to choose one which uses a combination of features you've already used.

I'm going to use the range construct to generate the indices of the items of the two lists. The loop variable will be the index of the lists, so let's call it "index". We want it to take values from a range and we want the range to give it values from 0 up to one less than the number of items in the list. To do this, we don't need a start value or a step, but we do need to specify the stop value. If we use the length of one of the lists, we guarantee the appropriate indices will be returned.

For instance, if the lists have three entries, len(vector1) will be 3. This means the loop variable will take on the values 0, 1 and 2, corresponding to the indices of the three values of the lists.

Now, inside, the loop, we want to increase the value of "result". We want to increase it by the product of the items of the lists with the current index, which we can write like this.

Finally, outside the for loop, we return the value of "result".

Let's check this function with an example. We'll choose some random numbers for our vectors.

We expect our answer to be 1 times –1, which is –1, plus 4 times 3, which is 12, plus 2 times 1. This gives us an expected result of 13. When we run the code, we see it produces an answer of 13, indicating the function is working as expected.

Title: Functions calling Functions

Narration script
We've seen how we can define and call a single function. However, functions really start to improve your Python experience when you begin to call functions from within other functions. This allows a web of related functions to begin to work together, making writing larger codes easier.
Let's say we want to write a function that returns the sum of the squares of all the values in a list. There are two steps here, finding the square of a number and summing the squares of multiple numbers. Let's make a function for each of those steps. The first one we'll create is the function to square a number. We'll call this function "square". It should take one value and return the square of it. This example is pretty trivial, but the principle here translates to other more complicated examples as well.
We can test this function out with a couple of examples to check it works. Both of these results look as we would expect, so we can move on to the next step.
We'll define the next function, which calculates the sum of the squares of the values in the list, so we'll call it "sum_of_squares". We want this function to use the "square" function, but we don't actually need to define square before we define sum_of_squares. What matters is that both functions are defined before they are called. "sum_of_squares" takes the list of values as an argument.
Again, we're going to use the accumulator method where we repeatedly adding to a value to keep track of a sum. So let's initialise the variable we're going to accumulate our final value in. We'll call it result again. Then, we'll loop over the list of values. For each of these values, we'll call the square function with "value" passed as the argument and add the returned value to "result". Finally, we'll return the value of "result".
Let's test this function with a sample list. We expect a returned value equal to 1 plus 9 plus 16, which is 26. When we run the code, we do indeed get the value 26 printed, showing us that this code has worked.

In this video we began by writing a function designed to do a single task, and then used this in another function designed to perform a more complex task using the first function. This pattern of development is a great way to develop code, as it ensures complexity is added, and tested, incrementally and means we have multiple useful functions we may well be able to reuse. When you develop complex code projects yourself, try to bear this pattern in mind and see if you can develop you code in this way.

Title: Extension: Recursive Functions

Narration script
We've seen how a function can call other functions. However, it's also possible for a function to call itself in Python. This is called recursion and a function which does this is called a recursive function. This is something which isn't always useful but occasionally is. In this optional video, we'll look at the syntax for such a function and construct one, to see how the logic of a function like this works.
We're going to write a function which returns the factorial of an integer. The factorial of a value "n" is written as "n exclamation mark" and is equal to "1 times 2 times 3 and so on, up to times n". Additionally, 0-factorial is defined to be equal to 1.
We could calculate this value using a for loop, but we're going to use the property that n-factorial is equal to n-minus-one-factorial multiplied by n to write this as a recursive function instead.
Let's begin to write our function. We'll call it "factorial" and have it take a single argument "n"
Now, let's work on the logic of our function. If n is greater than 0 we want to calculate the value of n-factorial by using the fact that this is equal to n-minus-1 factorial multiplied by n. Let's write that down. To call the factorial function recursively, we simply call it like a normal function.
We then use the fact that, if n is equal to zero, we want to return the value 1 as 0-factorial is defined to be equal to 1. We can look at this logic and conclude that it will work as we expect it to, as both the conditions lead to appropriate return conditions. This is a valid way to approach designing an recursive function, but it relies on considering all the different values the arguments may be given and what should be returned in each case.
We can try a few cases to check the code works as we expect. A value of 2 for n leads to a value of 2 returned, which is correct as 2 x 1 = 2.
In this case, the first time the factorial function is called, n is 2, which is greater than zero, so the factorial function is called again with a value of 1 as an argument. This value will be multiplied by 2 when the function returns.
In the function call with a value of 1 as an argument, "n" is once again greater than 0, so factorial is called again with a value of 0 and its result will be multiplied by 1 before being returned.
When the factorial function is called with a value of zero the elif condition is triggered and a value of 1 is returned. This time, the function will exit and return the value 1 to the call of the function where "n" had a value of 1. The value of 1 will be multiplied by 1 and returned to

the call to factorial where n had a value of 2. This will be multiplied by 2 and the value returned. The final value returned will be 2, which is correct.
A similar process will happen when we provide a value of 5. The values 5, 4, 3, 2, 1 and another 1 will ultimately be multiplied together to give a value of 120, which is the value of 5!.
There are two cases where this code will produce results which aren't necessarily what we want. The first is when a non-integer value is passed to the function. Let's try this with the value 0.5. The first condition is triggered and factorial is called again with a value of –0.5. This time though, neither the if nor elif conditions will be triggered and the function will complete without executing a return statement so "None" will be returned. In the call to the function with a value of 0.5 for "n", this None will be multiplied by 0.5. This doesn't make sense, so Python gives us an error.
We can improve our function by adding an if statement to check if "n" is not an integer. In this case, we give a warning and allow the function to finish with no return statement being triggered, causing None to be returned. This will also cause the function to behave sensibly if someone passes a value such as a string as a value for "n"
The other condition is when "n" is an integer less than 0. In this case, none of the conditions will trigger and None will be returned. Returning None might a good outcome, but we can add another condition to catch this case and give the user a warning, like this. The else statement will be triggered if n is an int and is less than 0 and give a useful warning before None is returned.
In this video, we discussed what a recursive function is and written one. We've talked through exactly how it works and how it's important to considered all the possible input values.

Written exercise:

The Fibonacci sequence is a sequence found in many places in nature. The nth Fibonacci number F(n) is defined as follows:

$F(0) = 0$
$F(1) = 1$
$F(n) = F(n - 1) + F(n – 2)$ for $n > 2$

Write a recursive function which will return the value of the nth Fibonacci number. Test you function with a few values. For reference, The first few Fibonacci values are as follows:

n	F(n)

0	0
1	1
2	1
3	2
4	3
5	4
6	8
7	13
8	21

Student exercise answer

Narration script

Let's begin by defining our function so it has the name "fibonacci" and takes a single value "n" for the value we want to calculate the Fibonacci value of.
Now, let's give a warning if "n" isn't an integer.
Next, we'll add aw warning if "n" is less than zero.
Now, let's deal with the special cases of the Fibonacci values of 0 and 1. We note that, in these cases the Fibonacci value of n is equal to n, so we can write this to give the right answers in those cases.
Finally, lets deal with the remaining cases, where n is an integer equal to or greater than 2. In this case, we want to return the value of the Fibonacci value of n-1 plus the Fibonacci value of n-2. Our function is now complete. Let's test it.
When n is not an integer, we get an appropriate warning and None is returned.
When n is negative, we get an appropriate warning and None is returned.
When n is greater than or equal to 0 and less than 2, we get the appropriate value returned.
When n is greater than or equal to 2, we get the appropriate value returned. We conclude that each of the four conditions is triggering at the correct time and is returning the correct value.

Unit 2.7 Importing modules

Title: Importing a module

Narration script

A Python module is a collection of Python source code that provides a group of related functionality. Using modules is an important part of using Python effectively. Python by default contains many modules with extra functionality you may import into your program. Other developers may write modules containing useful, specialised content you may want to use in your code. Finally, you will want to organise large

projects into a number of source files and combining these into a single program involves treating them as modules. In this video, we're going to look at how to import a module and gain access to its functionality.

The module we're going to use an example is the "math" module. This is an important and useful module, as it contains definitions for a number of common mathematical functions.

It is an example of an "intrinsic" Python module. This means it is included in a standard installation of Python and doesn't need to be installed separately. There are many useful intrinsic modules available in Python. A link provided along with this video links to part of the Python documentation, which lists all the intrinsic modules provided with Python. For example, "audiooop" provides tools for dealing with audio files, "cmath" provides tools for dealing with complex numbers and "io" provides tools for reading from and writing to external files.

But let's get back to importing the "math" module. The first and simplest way to import a module is to write the word "import", followed by the name of the module. This makes the module and its contents available within in the code following the import statement.

We can inspect the contents of a module we've imported by writing "help(math)". When we run the code, Python gives us the name and a description of the module. Then it tells us the names and descriptions of all the functions that are part of the module. For instance, it describes the "acos" function, which returns the arcosine, measured in radians, of the single argument passed to it.

Scrolling down further, we see definitions of the gamma function, various different types of logarithms and all manner of trigonometric functions. At the bottom, we see there are also some values which have been defined, including "e", pi and tau.

To actually use one of these functions or values, we need to write the name of the module, followed by a full stop, followed by the name of the value or function. For instance, to print the value of pi from the math module, we can write "print(math.pi)". This will print the value of the variable.

Just like any other variable, we can use this variable in more complex expressions if we like. For instance, we can write "a = math.pi / 6" to divide this value by 6 and save the result to the variable "a".

We can use this value as an argument in another function from the math module, the sin function.

So, if we write "print(math.sin(a))" we'll calculate and print the value of the sine of pi divided by 6, which is 0.5 give or take some floating point error.

There are other ways of importing a module. We can choose to give the module an "alias". This is an alternative name for the module that we provide and use to reference it in our code.

For instance, we can write "import math as mt". This will still give us access to the math module, but means we can access its functionality by writing "mt dot" rather than "math dot". This might be useful if the module name is particularly long and you don't want to have write a long name repeatedly in your code.

As an example pf how to use a feature from a module imported this way, we can write “mt.sqrt” to access the square root function of the math module.

So far, we’ve imported an entire module and then accessed features of it. However, we can also import features of a module selectively using the syntax “from module_name import feature_name”, like this.

We can also give a feature an alias by adding “as alias_name”. So, here, wer’re importing the “asin” function of the “math” module and locally calling it “arcsin”.

When we use these two features, we now don’t need to use the prefix “math.” as we imported the features directly rather than the module as a whole. So, we can write this expression to calculate four times the arcisne of one over root 2.

There’s a final way we can import the contents of a module and that’s by writing “from module_name import *”. This will import every feature of a module using the name it had in the module.

This means, for instance, we can print the logarithm of 1 by writing “print(log(1))”.

This might seem to be a tempting option – we've imported everything from the module in a short concise way and made it so we can use those features very easily, simply by writing their name, without having to write “module_name.” before it.

However, there are a couple of reasons why this is generally a bad way to import the contents of a module. First of all, when someone looking at the code sees the function “log”, there’s no way to trace back where it came from, particularly if multiple modules have been imported in this way.

Secondly, if two modules have features with the same name and are both imported in this way, it can be easy to lose track of which module the feature in question has been imported from.

For instance, we might import in this way another module designed to record experimental data with a function named “log”. In this case it would be confusing to both the person writing the code and someone else reading it at a later date as to which of the two “log” functions would be executed when the word “log” is written.

For these reason, it’s generally a bad idea to use statements like “from math import *”. But it is important to be aware of what this command does in case you see it in someone else’s code.

To summarise, in this video, you’ve seen the different ways to import modules. The first of these is “import module_name”, which allows you to import features by writing “module_name.feature_name”.

Alternatively, you can give a module an alias by writing “import module_name as alias_name”. You can then use a feature by writing “alias_name.feature_name”.

You can also selectively import individual features from a module by writing “from module_name import feature_name”. This allows you to use the feature just by using the feature name.

You can also give an indidviual feature an alias using the syntax “from module_name import feature_name as alias_name” and then use the alias name to access the feature.

It’s also possible to import the contents of an entire module by writing “from module_name import *”. This allows you to use any feature from the module by simply using its name. However, for reasons we discussed earlier, this is often not a good idea.

Otherwise, which of these options you use is normally a case of personal preference combined with which method makes your code clearest.

Written exercise:

The equation for calculating the length of the hypotenuse of a triangle is:

$a^2 + b^2 = c^2$

Where “c” is the length of the hypotenuse and “a” and “b” are the lengths of the other two sides.

Import the math module using a method of your choice and using the “sqrt” function to write a function which calculates the length of the hypotenuse of a triangle given the lengths of the other two sides as arguments. Check your function works with a couple of pairs of values.

Student exercise answer

Narration script

To begin with, we're going to import the math module. I'm going to choose to do this by simply importing the whole module by writing "import math".
Now, we can define our function. Let's call it hypotenuse, as that's a descriptive name for it. It needs to take two arguments – the lengths of the non-hypotenuse sides. Let's call them "length1" and "length2" as those are descriptive names.
Now, the value we want to calculate is the square root of the sum of the squares of these two values. We can use the square root function from the math module by writing "math.sqrt".

This calculates the square root of the argument passed to it. So, within the parentheses, we can write a "length1" to the power of 2 plus "length2" to the power of 2.

This means we've written an expression which calculates the desired value.
Now, all we need to do is make the function return this value using the return statement.
Next, we should test the function works correctly. First, let's test it with side lengths 3 and 4.

3 squared is 9 and 4 squared is 16. 9 plus 16 is 25 and the square root of this is 5, which is the value we expect the hypotenuse to have.

When we run the code, we get the right answer printed.
Let's try another set of values. Side lengths of 10 and 12 don't produce a neat answer, but we can calculate with a calculator that we expect a result of about 15.6. When we run the code again, we get this answer printed, indicating the function has worked. So, we can be reasonably confident that our function is working as desired.
In this video we've seen how you can import an intrinsic module into your code and then use it in a function you define. This is an important method used to incorporate features of modules into your own custom code.

Title: Importing your own module

Narration script
We've seen how we can import intrinsic Python modules into our code. However, it's also very useful to split our code up into multiple source files and use these as modules. In this video we'll see how we can create our modules in Python and import them into our code. To do this, we'll need to use a different piece of software from Anaconda to create ".py" files to use as modules. A ".py" file is a file intended to contain Python code, like the contents of a code cell in Jupyter Notebooks.
First, we need to open Spyder. We can do this from the main Anaconda screen. If we click on Spyder, the standalone Spyder program will open. Spyder has a lot of functionality, but we're just going to look at a couple of key things. The panel on the left allows us to edit a ".py" file.
We see that, when we opened Spyder the file we're editing is called "temp.py". It contains a couple of sample lines of code that we can delete.
We can choose to save this to a particular location by selecting file, then "save as". We can choose a location and a name. Let's save this one in the same directory as where I'm keeping my Python notebooks. We'll call this file "module1.py". We can write some code into the window on the left. Let's keep it simple for now and simply write "print(hello)". We can run the code by clicking the green triangle in the toolbar. When we do this, the output of the code appears in the window in the bottom right. This is similar to running a code cell in a Jupyter notebook.
Now, let's write a simple function in this file instead. This function is called "function1" and prints the phrase "Function1". Let's save the file.
We can then create another file named "module2" by selecting "file", "new file" and then saving the file. We saved it in the same directory as the first file. From here, we can import from the first file using the file name of that file as a module name. Let's import "function1" from there and then call it. We see that the function we wrote in the other file has been executed. We've successfully imported and used the feature from that module.
Let's modify this module so we have a function to import later. When this function is called, the phrase "Function2" will be printed, before function1 is called and the phrase "Function1" is printed. Let's save the file.

Now, let's create a new Jupyter Notebook in the same directory. In this file, we can import from module2. Let's import function2 then call it.

We see we get "Function2" printed, followed by "Function1". This means we've called function2 from module2.py, which has called function1 from module1.py. This is a simple example of how it's possible to have a web of functions spread across multiple source files, which call each other.

This can be a very powerful way of developing code. It allows us to break down a large problem into a series of smaller problems, then work on different grouped pieces of functionality in a single module. This can be tested and then that functionality can be imported from other modules. This allows us to build up functionality from initially simple functionality to complex programs. It also has the advantage that individual modules can be reused in other projects that require that functionality, for instance, by simply copying the relevant source files into the directory of the new project.

In this video, we've seen how we can use Spyder to create and run .py files, and how we can import .py files as a module into other .py files or into a Jupyter Notebook.

Written exercise:

It's desired to write some code which gives information about various regular polygons.

Write three ".py" source files with the specified contents:

- square.py:
  - square_area: takes a side length as an argument and returns the area of a square (side_length^2)
  - sqaure_perimeter: takes a side length as an argument and returns the perimeter of the square (4 * side_length)
- traingle.py:
  - triangle_area: takes a side length as an argument and returns the area of a square (($3^{0.5}$/4)side_length^2
  - triangle_perimeter: takes a side length as an argument and returns the perimeter of the square (3 * side_length)
- shapes.py:
  - area: takes a string (either "square" or "traiangle") and a side length as arguments. Uses square_area or triangle_area as indicated by the string argument to calculate the area and returns the result.

- perimeter: takes a string (either “square” or “traiangle”) and a side length as arguments. Uses square_perimeter or triangle_perimeter as indicated by the string argument to calculate the area and returns the result.

Import the functions from “shapes” into a Jupyter notebook and call “area” and “perimeter” to find the area and perimeter of both a square and a triangle with a side length of 3.

Student exercise answer

Narration script

Let's first create the "square.py" module in Spyder. We'll create a new directory for this exercise named "shapes_exercise" and save this file to that directory. Now, let's write the two functions in this file.

First, "square_area". We'll call the variable to hold the length of one side "side length", and return the value of this squared.

Now, we'll add "square_perimeter". Again, we'll call the argument "side_length" and we want to return this value multiplied by 4.
Now, we'll create the "triangle.py" file.

We'll create the "triangle_area" function first. To complete this function, we need to use the square root function of the math module, so let's import the math module. Now we can return root 3 over 4, times side_length squared.

The triangle_perimeter function is fairly simple, we simply need to return the side_length multiplied by 3.
Now, we'll create the "shapes.py" file. We want to use aspects of both square.py and triangle.py so let's import those modules.

Now let's define the "area" function. We want it to take an argument which tells us if the area of a square or triangle is being requested. Let's call that "shape". We also need to know the side length of the shape.

Now, dependent on the value of "shape" we want to call either "square_area" from the "square" module and return that value or "triangle_area" from the "triangle" module and return that value.

Finally, we want to think about what happens if "shape" has a different value. Let's print an error message and not return a value at all.

Now, we can repeat this for the "perimeter" function, following the same pattern.

Now we've got a set of functions where we return the area or perimeter or a couple of regular polygons. By adding more modules and more elif statements, we could extend this to calculate the area and perimeter of even more shapes.
Finally, we're asked to create a Jupyter Notebook which imports "shapes" and tests it. Let's do that now. To import "shapes", we need it to be in the same directory.
First, let's import the shapes module.

Now, let's test the "area" function by calling it for a square, then a triangle.

Now, let's do for the same for the perimeter function.

Now let's run the code and check the output. The area of a square with a side length of 3 is indeed 9. The area of an equilateral triangle with a side length of 3 is about 3.9. The perimeter of a square with side length 3 is indeed 12 and the perimeter of an equilateral triangle with a side length of 3 is 9. So we can be happy that our code is working correctly.
This exercise has walked us through the production of several different source files and a Jupyter Notebook. We developed these beginning with small and simple modules relating to each shape, before creating a module that tied together those functions, then used a Jupyter Notebook as a way of using all this functionality.

Unit 3.1 Error messages

Title: Introduction

Narration script
When running a piece of Python code, errors may occur. Sometimes, this will be because the contents of that part of the source file are invalid. You may have used a function which doesn't exist, or forgotten some indentation.
This type of error can be found by "linting" which is a feature built into many places where you might write Python code, including Sypder. As an example, if we write this simple if statement but don't indent the body of it, Spyder is able to notice that we've used a colon on one line, but have not indented the next. This is always wrong and is simple to spot, so Spyder can pick up on this and tell us about it by displaying this red circular warning sign. When we hover over this symbol we get told Spyder analysed the code and expected an indented block. This

linting behaviour is a useful way of finding errors without running the code, but is a feature of Sypder, not Python itself and is not something that is always present when writing Python.
For instance, the same code in a Jupyter Notebook, doesn't lead to a warning at all.
Most errors which occur in Python are more complex than this. For instance, errors which occur when using variables which haven't been defined yet or when variables have an invalid value for an operation cannot be as easily identified by the editor in use. These errors can only be found when the code is run. When the problem occurs during the running of the code, Python, will "raise an exception".
Most of this module will focus on what this means and how we can use this behaviour to help us debug our code.
We're going to look at what happens when Python encounters an error and how to read an error message. We'll look at some common types of error message and how to "handle" and "raise" an error.

Title: Anatomy of an Exception

An exception is a special data type in Python which records the reason an error occurred and the location in the code where it occurred. If this error occurs within a function then the function will immediately return, unless the exception has been "handled", which we'll discuss later. This "raises" the exception to the piece of code which called the function. If this piece of code is also a function, it will also return. This will continue until the main piece of code running receives the exception. At this point, if the exception isn't handled, the entire code will terminate and the exception will be displayed.
Let's look at an example. First, let's create a function called "divide" which will return the value of its first argument divided by its second argument.
Now, let's give it some values which result in an error. Let's call this function but give the argument "b" the value 0 so Python tries to divide by zero inside the function. We'll also put a couple of print statements in to demonstrate which lines of code are executed and which lines of code aren't. Let's run the code and see what happens.
An exception is raised. We can examine the error message which is generated by this exception to find some more information about what went wrong. The first thing we see is the type of exception that was raised. This is a ZeroDivisionError, which was generated because we tried to divide by zero.
Next, we see the "traceback". This is a record of which lines of code and which functions were involved in reaching the code where the error occurred. This begins with the last piece of code which was terminated – the highest level in the hierarchy of function calls. This tells us the error occurred in line 6 when we called the divide function. Note that, in Jupyter Notebooks, we can turn on line numbering in the "view" menu.
The second part of the traceback tells us the error occurs in the function "divide", specifically in line 2.

The fact that, in the previous line, we were told where the divide function was called from allows us to know which call to the divide function caused the problem.
Finally, the exception gives us a little more information – telling us the exception occurred because we divided by zero.
We can use all this data together to work out what went wrong. We know the problem is that we divided by zero. We know this problem occurred in the divide function in line 2 and we know that this occurred when the function was called from line 6. This is all useful information as we know what when wrong, where the problem occurred and the context in which that code was being executed. In this case, we can see that using a value of "0" as the second argument for the "divide" function in line 6 caused us to divide by zero in line 2. Using these pieces of information is key to interpreting the exception which is raised and understanding why the error occurred and how to fix the problem.
It's also worth noting that neither of the statements, where we printed an upper or lower case "a", were executed. The print statement in the "divide" function wasn't executed because the function returned when the error occurred, so this line was never reached.
The print statement outside the function wasn't executed because, after "divide" returned, the exception wasn't handled, so the main part of the code in the code cell halted before the print statement could be executed.
In this video we've discussed exactly what happens when an error occurs and an exception is raised, what information is presented in an exception and how we use this information to work out what went wrong.

Title: Types of Exception

In the last video, we looked at what happens when an exception is raised. Different types of exception will be raised by different errors in the code. A full list of exceptions and descriptions of when they occur can be found in the Python documentation. A link to the relevant section is provided along with this video.
In this video, we're going to look at some common types of exception that can be raised and the errors in code that lead to them being raised. The goal of this is to make it easier for you to understand what problem has caused an exception to be raised when you see it in your code.
The first type of exception we're going to look at is the AttributeError. This occurs when we try to access an attribute of a value when that value doesn't have that attribute. For example, the append method is an attribute of the list class. One way we can access this by writing the name of a list variable, a full stop and then the name of the attribute. So, we can see this code runs without a problem.
However, the string class does not have an "append" method, so when we try to use the append method of a string in the same way, we get an AttributeError telling us the string type does not have an attribute named "append".

When you see this error, it's likely that you've either misspelled the name of an attribute, you are mistaken that the type in question has the attribute you're trying to use, or the variable is a different type than you think it is.
The next type of exception you might encounter is a ModuleNotfoundError. As the name suggests, this exception occurs when you try to import a module, but there is no module available with that name. Let's try importing a module which doesn't exist. We get a ModuleNotFoundError as there isn't a module with this name.
This may occur because you've misspelled the name of a module, or because the module is not a default Python module and needs to be installed in your Python distribution before it can be imported.
An IndentationError occurs when Python is expecting an indented section of code, but the code isn't indented. This example leads to an IndentationError because the print statement is not indented.
An IndexError can occur when a collection is indexed with an integer, such as a list or a string, and the value used to index the collection or perform item assignment is out of the valid range. For instance, if we try access the character with index 99 in a short string, we get an IndexError.
Or, if we try to assign to an item of a short list with an index of 99, we also get an IndexError. Note that the final part of the error message differs between these two IndexErrors as Python gives us a more detailed explanation. In the first case, it says "string index out of range". In the second, it says "list assignment index out of range". This is an example of how the exact error message can sometimes give extra information, past the type of exception raised.
A NameError is raised when we attempt to use a name of something, for instance a variable or function, but there is no existing variable or function with that name. For example, if we try to print the value of a random variable that was haven't yet defined, we'll get a NameError telling us it doesn't exist.
A SyntaxError occurs when we write a piece of code which simply doesn't make sense. For instance, if we try to put a value on the left side of an assignment operator, Python can't assign a value to it and we get an error message telling us we can't do this.
A TypeError occurs when we try to do something with a value which is the wrong type. This might be if we try to concatenate a string and an integer. The error we get here tells us that we can't concatenate a string and an integer, only a string and a string.
Or, we might get a TypeError if we try to pass a value of the wrong type to a function. For instance, if we import the math module and then try to take the square root of a string we'll get a TypeError. Taking the square root of a string doesn't make sense, so the square root function returns an error telling us that we can only take the square root of a real number, like an int or a float, not a string.
An UnboundLocalError is very similar to a NameError. It occurs when we're inside a construct such as a function and attempt to use a variable that we haven't declared yet.

For example, let's create a function where we use the variable "a" before we've assigned to it.

When we run this code, we see we get an UnboundLocalError raised telling us "a" is referenced before we've assigned to it.

ValueErrors occur when we've used the right type of value, but the value is not valid for the case it is being used for. For instance, if we again import the math module and this time try to take the square root of –1 we get a value error. Taking the square root of an integer is a completely valid thing to do but taking the square root of a negative number is not. As such, it is the value of the argument not the type which causes the problem and so Python raises a ValueError, telling us the value is not in a valid domain for the mathematical operation.

We've already seen a ZeroDivisionError is raised when we divide by zero, but it also occurs when we use a zero as the denominator in integer division or modulo operations. For example, let's take the modulus of 5 and 0. We see a ZeroDivisionError is raised. Python even helpfully tells us that the problem is we've performed an integer division or modulo operation by zero.

This video has run through some of the most common exception types that can be raised in Python, discussed what they mean and when they might be raised. As you get more experienced with Python you will become more knowledgeable of these and other exception types.

For now, you should be familiar with these common exceptions and, when they occur as you code, you should have a fighting chance of understanding what went wrong and fixing it.

Written exercise:

Download the notebook "debugging_exercise.ipynb". This Jupyter Notebook contains a function named "standard_deviation" which is designed to calculate the standard deviation of a list of numbers passed to it as an argument. It also contains two calls to this function. The first should return a value of approximately 7.1 and the second should return a value of approximately 395.

However, the notebook contains several errors. Run the code cell, read the exception raised carefully and use it to find the error in the code. Then fix the error and repeat the process until the code runs correctly.

Student exercise answer

Narration script

Here we have our code which is riddled with bugs. Let's run it and see what the first error is.
The first error is a SyntaxError which occurs on line 3. The error occurs in the line where we defined the function. The hat symbol on the line below gives a further clue about where the error occurred – at the end of the line.
There's no further information in the exception, but we have enough information to notice that the line is missing a colon at the end. Let's add that colon in.
Now, when we run the code, we get a different error. Now we get an IndentationError on line 5 as we haven't indented the first line after the definition of the function. Let's indent that line and re-run the code.
Next, we get a ModuleNoteFoundError. This tells us that, in line 1, Python could not import the "mat" module. That's because it doesn't exist, and we've misspelled the name of the "math" module. So, let's correct that.
Now, the exception raised is a NameError. The name "le" in line 15 isn't known to Python. That's because it's actually a misspelling of the function name "len". Let's correct that.
Now we get another NameError. This time, it's the word "mean". However, this isn't misspelled, so what's gone wrong? Well, this time, the problem is that we misspelled "mean" as "men" in line 14 when we created this variable. This means the variable "mean" wasn't available when it was used in line 15. This is an example of a time when Python will raise an exception on one line, due to a mistake we've made on another line. It's always a good idea to keep the possibility of this kind of error in mind while debugging.
This next exception is a little difficult to work out, so don't worry if you got stuck on it. Now, we get a ValueError in line 15. Specifically, this is a math domain error. Let's try some simple debugging to work out what's going on. First, let's try printing the value of the expression we're taking the square root of. We get a negative value. That's strange. The equation we've used in what is now line 16 is correct and we expect this value to be positive.
Let's delve a little deeper and break down these two terms. We can do that by putting a comma before the subtraction operator so we can see each of the terms of this expression separately. Both terms are negative. This isn't necessarily wrong for the expression which is the negative of the square of the mean but, for the first term, we expect the total of the square of the entries to be positive and the length of the list to be positive. So why is this negative?

Let's focus on this term as it being negative is suspicious.

Putting a comma between the numerator and the denominator of the division.
The length of the list is reported as four, which is correct for the first time the function is called. However, the total of the square of the values should be positive as the square of any value should be positive. This gives us a clue that "total_square" is being calculated incorrectly.

The most important line where “total_square” is calculated is line 11 and here’s where we find the error. We’re subtracting the square of the current value from the running total rather than adding it. That’s why the value of “total_square” is negative and that’s why we get a ValueError when we take the square root. Let’s fix this by changing the subtraction operator to an addition operator instead. Now, when we run the code we see that the code now runs to completion and the values of “total_square” are positive both times the function is called.

What we’ve done here is an example of an important technique when debugging. We found an error in a line, localised it to a particular variable in that line, then found out where the variable got its value and what was wrong in that line. This process can be performed repeatedly in complex problems, until we find a problem we can fix.
Now we think we’ve fixed that error, let’s delete the print statement and re-run the code. Now the code runs we can compare the results of the function to results we expect. We expected values of 7.1, then 395 to be returned and that’s what we’ve got. We can conclude we've fixed the problems in this function.

This video has shown some real-world examples of where we might encounter specific exceptions and how we can go about fixing the errors that cause them.

Title: Error Handling

So far, we’ve looked at what happens when an exception is raised, and what errors in the execution of the code cause common exceptions to be raised. In this video we’re going to look at how we can “handle” an error . This allows our code to sense when an error occurs and to do something different without exiting. This can be used to recover from errors which are expected to happen in some cases, or to provide useful feedback to a user.

Normally when an error occurs, an exception will be raised and the current function will immediately return. We can avoid this happening by “handling” the error using a “try-except” block.

Let’s rewrite the divide function we wrote earlier to give an example of how this works. The place we expect an error to occur is in the second line where we perform the division.

To add a try-except block, we need to write the word “try” before the block where we think an error might occur. This is followed by a colon, which indicates the following indented block is the section where we’re monitoring to see if an error occurs. So, let’s indent the line where we think the error might occur and the return statement

This is normally followed by one or more “except” statements. This means we write the word “except” followed by the name of an exception that we want to “catch” and provide alternative behaviour for. We know we might get a ZeroDivisionError so let’s specify that one here. This is

followed by a colon which indicates the following indented block is to be executed if the specified exception is raised in the section of code following the word “try”. Let’s give the user a more explicit error message printed to the screen.
When we create a “try-except” block like this, we need a single try statement and at least one except statement or a “finally” statement, or both. We can add a “finally” statement by writing the word “finally” and following it with a colon. Let’s add a print statement saying “Division complete”.
The word “finally” specifies that the following indented section of code is to be executed no matter what else has happened after any code in “try” sections or relevant “except” sections have been run. This code will be executed whether no exception was raised, an exception was raised and caught, or if an exception was raised and not caught. In this case, it will even be printed if the function returns inside the “try” block.
Let’s try calling our function with a couple of example sets of values and see what happens. First, let’s call it with a valid numerator and denominator. 10 divides by 1 with no problem, so “result” gets the value 10. This value is returned and the finally block is executed, printing “Division Complete”. Then the value returned from the function is printed to the screen.
Now, let’s set the denominator to zero. This time when we run the code a ZeroDivisionError is raised inside the try block before the return statement can be reached. This exception is caught by the except statement and the error message “The denominator may not be zero” will be printed before the “finally” statement is executed and “Division Complete” is printed. As no return statement has been reached, the function returns the default value of None and this is printed by the final print statement.
Now let’s try introducing an error we haven’t handled. Let’s make the denominator a string. This will raise a TypeError as the divide operator can’t operator with a string as a denominator. This isn’t caught by our “except” statement and so the exception will be raised but, before the function returns, the finally block is executed and “Division Complete” is printed. As the exception is not caught and is still raised, the code stops executing before the value returned by the function can be printed and the exception is displayed to the user instead.
If decide we want to catch this exception, there are a couple of places we could do it. The first place we could do it is in the function itself by adding a new “except” statement that catches a TypeError. Let’s add that now, along with a print statement stating both arguments must be real numbers. When we run the code now, we see this except statement is triggered. If we wanted to, we could keep adding more and more except statements to catch different exceptions that might occur.
Now, let’s delete that except statement and look at another place we can put it. We can put a try-except block around the call to the function, like this. Let’s run the code and check it works. This works because when, the exception is raised within the divide function, the finally statement executes but then the exception is still raised to the main body of code, which allows is to be caught by the except statement after the function call.
This is an example of how we can use a try-except statement around a function call and catch exceptions that were raised in that function, or functions called by that function and so on.

The example we've looked at here has been designed to show the syntax of the try-except block in a simple way. However, we probably wouldn't choose to produce this function in reality. In many cases, stopping the code is exactly the right thing to do as, when an invalid operation is performed, often something has gone wrong and the code should stop. So, it's important to think carefully about whether it's correct to implement a try-except block in your code and what code should be placed in the "try" section.
It's also worth noting that a try-except block can, in many cases, be replaced by an if block. For instance, we could write the following, which is similar to the try-except block that caught the ZeroDivisonError. So, even when sensing a problem is the right thing to do, you might need to decide whether it's better to do this in a try-except block or in an if block. The most important factor to take into account when making this decision is how to maximise the clarity and the reusability of your code.
This concludes this video on catching exceptions. This is a very useful tool for managing errors within your code. Catching exceptions allows you write code which recovers from anticipated problems, essentially acting as another way to control the flow of your program. Alternatively, it allows you to provide a custom error message to your user.

Title: Raising Exceptions

We've discussed when operations in Python may result in an exception being raised and how to handle an exception. The last key skill relating to exceptions is the ability to write code which raises an exception. This can be useful when you can test if there's a problem and want to halt the code and provide an error message to the user if so.
We can manually raise an exception by writing the word "raise" followed by the name of the exception we want to raise, followed by a custom error message provided as a string within parentheses. For example, to raise a ValueError, we could write the following.
This can be useful when we can detect a problem in our code that Python wouldn't raise an exception for. Say we're reading a phone number from a string to store it as an integer. Let's write the skeleton of a function that does this. For simplicity we'll assume no phone numbers begin with a zero. This works on a string of numbers quite nicely.
However, let's say we know that each phone number should be at least 6 digits. This function will work just fine on a short string. But we know the integer returned is invalid. We can use an "if" statement to check the value of the number calculated and raise an exception if it's too low. A ValueError would be an appropriate type of error to raise here as the data passed is the right type – it just has the wrong value. When we run the code with a long enough string, it runs the same as before as the raise statement isn't executed. Let's change the string to a shorter one.

When we run the code now, we receive a ValueError with a custom message. Our function now performs a logical check and produces its own useful error message.

The raising of exceptions give us a tool to raise an exception within Python ourselves, allowing us to provide specific feedback on errors you anticipate or to provide exceptions which can be used by exception handling tools like the try-except block.

Written exercise:

A quadratic equation has the following form:

$ax^2 + bx + c = 0$

The values of "x" are known as the "roots" of the equation. The quadratic formula solves a quadratic equation for "x" given values of "a", "b" and "c". The quadratic formula has the following form:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Write a function which solves the quadratic equation and returns a list containing the root(s) of the equation. It should include logic that does the following:

- If "a" and "b" are both zero, a ValueError should be raised informing the user than the value of "x" is undefined.

- If the discriminant (b^2 - 4ac) is negative, a different ValueError should be raised informing the user the roots are imaginary
- If “a” is zero but “b” is not, return a single value in the list equal to –c/b.
- If the discriminant is zero but “a” is non-zero, return a single value in the list equal to -b/2a
- If the discriminant and “a” are both positive, return a list containing the two roots of the formula (produced the positive and negative roots of the square root)

When writing your function, use at least one try-except statement. There is not a single correct way to complete this exercise, you’re encouraged to take all you’ve learned in this course and use it to create a function of your own. Test your function with the following cases.

	a	b	c	Expected Result
Case 1	2	-5	-12	[-1.5, 4]
Case 2	-2	-4	-2	[-1]
Case 3	0	1	2	[-2]
Case 4	1	1	1	ValueError (imaginary roots)
Case 5	0	0	1	ValueError (undefined x)

Student exercise answer

Narration script

Before starting this video where I'll construct a sample answer, I'll reiterate that there are multiple ways to solve this problem. I'm going to present one solution. If your solution looks different but it reproduces the sample results, then it's probably a valid solution.

We begin by defining our function. Let's call it "quadratic" and make it take the arguments a, b and c to represent the three coefficients of the quadratic equation being solved.
The first case I'm going to check for is when both "a" and "b" are zero. If this case is true, it will make it more difficult to check the other conditions, so it makes sense to raise an exception if this is the case before we write the rest of our logic. This means we won't need to worry about both "a" and "b" being zero in the rest of our function.

We can test if both "a" and "b" are 0 using an if statement. If they are, we want to raise a ValueError telling the user that "a and b were both zero so x was undefined".
Next, we'll calculate the discriminant. We'll do this and save the value as we'll be using this value in a few places and it saves us writing it out repeatedly.
Now, we can test if the discriminant is zero. If it is, we need to return a single root. The value returned in this case should be equal to –b divided by 2 times a.
Now, we're going to write a try-except block. Let's write out the whole quadratic equation and then consider what exceptions might be raised and what should happen if they are. We're going to need the math module, so lets import that now.

For now, let's assume we're solving for both roots of the equation and returning both as items in a list. Let's start by writing the term for the negative root of the equation. Minus b, minus the square root of the discriminant, all over 2 times a. Then, the same for the positive root, with the exception that we're adding the square root of the discriminant this time.
Now, we can start thinking about the exceptions that might occur. The first of these relates to the case where the discriminant is negative. This will cause a ValueError to be raised when we try to take the square root of the negative value. In the problem brief, we were told that, in the case of a negative discriminant, we should return a ValueError stating the roots of the equation are imaginary. So, let's add an except statement to the try-except block to catch a ValueError. If this exception is raised, we want to raise a ValueError with a message telling the user that the equation had imaginary roots.
The other type of exception that might get raised is a ZeroDivisionError, which will be raised if the value of "a" is zero. Let's add the except statement. Now, we already checked if both "a" and "b" were zero in the if statement at the start of the function. If they were both zero the function would have already finished executing when the ValueError was raised there. So, we know that if we get a ZeroDivisionError here, that "a" is zero but "b" is not. In the problem brief we were told that, in this case, we should return a list with a single value equal to minus c over b. So let's do that now.

Now, we've completed our function. One quick check we can do relates to the fact that there were five different cases in the problem specification. We can count the number of "return" and "raise" statements as these are the different ways our function might return. We have a raise statement, a return, another return, a raise and a return. That's five possible ways our function could exit, corresponding with the five different cases in the problem specification. So that's one promising sign that we've written a function that corresponds with the brief.
Now, let's try the test cases provided with the problem brief. First the case with a equal to 2, b equal to –5 and c equal to –12. We get two values returned: -1.5 and 4. These are the values we expected, so the function meets this part of the brief.
Next, we'll try the values, -2, -4 and –2. As expected, this returns a single root of the equation with a value of –1.
Now, we'll try the values 0, 1 and 2. We expect this to return a single root with a value of –2 and, indeed, it does.
Next, we'll try the case where a, b and c all have the value of 1. In this case, we're expecting a ValueError to be raised, telling us that all the roots of the equation are imaginary. When we run the code, we see we get this exception raised as expected.

It's worth spending a little time looking at the error message that's displayed here as it's a bit different from what we've seen before. Firstly, we see the ValueError that occurred when we tried to take the square root of a negative number. This was in the try section of our function. We then caught that error and raised our own ValueError instead. As a result, the message provided with the exception says "During the handling of the above exception, another exception occurred" before giving the exception we raised. This is how Python responds to the raising of an exception within the handling of another exception, essentially giving information on both.
The final case we're going to use to test our function is where "a" and "b" have a value of 0 and "c" has a value of 1. Sure enough, we get an exception raised telling us that x was undefined as "a" and "b" were both zero.
So, we've tested all the different types of result we expect our function to give and found each of them is correct. This is a good indication that we've produced a working function.

If you managed to produce a function similar to this one, then you've done a good job in this course. This exercise required some non-trivial logic and the use of a variety of different pieces of Python syntax to produce its result.