Week 1 content

Programs, languages and Python

Programs

A **computer program** (or **code**) is a set of instructions for a computer to follow. **Computer programming** (or **coding**) is the activity of writing computer programs.

Languages

When you write a program you must use a language that the computer can understand. There are many such languages - Java, Python, R, C, Ruby, JavaScript, PHP, and many, many more. Each language has its advantages and disadvantages - there is no "best" language that will be the most appropriate choice in all situations.

Python

In this course you will be learning Python. Python is one of the most popular languages, for several reasons:

- It's a general-purpose language that can be used in a wide variety of situations.
- It's mature enough that many comprehensive libraries for many disciplines have been built and extensively tested.
- It was developed recently enough to be built upon many of the fundamental programming principles that have been developed and fine-tuned over the years.
- It has a human-friendly syntax that is easy to learn.

Python has become one of the most popular languages for working with data. It has data processing and visualisation libraries such as pandas and Matplotlib that make handling data very easy (you will learn them in this course).

Even though you will be learning just one language, many of the fundamental principles you will learn, such as controlling program flow, manipulating data, and modularising code, carry over to almost every other language.

Version

Like natural languages, programming languages tend to change over time. You'll be using the latest version of Python throughout this course - Python 3. You may encounter some Python 2 code from time to time. The language differences between these two are minor, however they can be

ncompatible with each other.	

Programming basics

Here is a very simple Python program:

```
PYTHON ☐

1 print('Hello, world!')
```

Execution

You can **execute** the program (or **run** it) by clicking 'Run'. You should see "Hello, world!" printed just below the code. You should also see "Program exited with code 0" - this just tells you that the program executed without a problem.

When you run a program like this, the Python **interpreter** executes each line of the program, one at a time, from beginning to end (in this case there is only one line).

Whenever you see a runnable piece of code like this you can also modify it yourself. This is a great way for you to experiment with Python (don't worry, you won't break anything, and everything gets restored when you refresh the page). Try modifying the code above to get it to print your name rather than "world".

Statements

Each line the program is called a **statement**. Your programs will typically contain many statements. Here is a program with two statements (click the "Run" button):

```
PYTHON II

1 print('Hello, world!')
2 print('Goodbye, world!')
```

You normally write each statement on its own line, as above. But you can write two or more statements on the one line, as long as you separate them by semicolons:

```
PYTHON II

1 print('Hello, world!'); print('Goodbye, world!')
```

However, this can make a program more difficult to read, and it's recommended that you don't do this.

Syntax

Like natural languages, such as English and French, programming languages have a specific grammar; this is known as the language's **syntax**. Unlike natural languages, programming languages require

you to be precise and unambiguous. A big part of learning a language is learning its syntax.

Whitespace

Python will ignore blank lines between statements and spaces between arguments in a statement. So the following code snippets are equivalent:

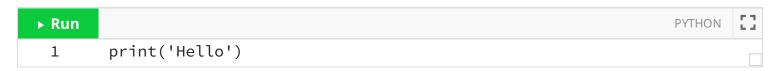
```
      ▶ Run
      PYTHON
      II

      1
      2 print ('Hello')
      3

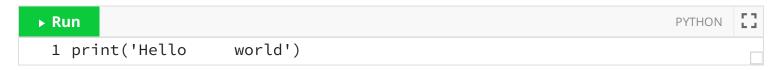
      4 print(
      10 * 2)
      PYTHON
      II

      1 print('Hello')
      2 print(10*2)
      □
```

Spacing at the start of a line is <u>not</u> ignored by Python. It is called **indentation**, and is a very important aspect of Python syntax. You will learn more about indentation when we cover <code>if</code> and <code>while</code> statements. For now you just need to know that if you incorrectly add indentation, Python will generate an error.



Spacing inside quotes is also <u>not</u> ignored by Python:



Although much of your whitespace is ignored by Python it has become conventional to use it in a certain way, described in a set of guidelines called "The PEP 8 Style Guide", or just "PEP 8". According to PEP 8, for example, you should use print('Hello') rather than print('Hello') or print ('Hello').

There is a link to PEP 8 in the Orientation section of this course, for your information. The guidelines are comprehensive, covering much more than just the use of whitespace. You're not expected to know all of them. If you just follow the formatting used in the examples throughout this course then your code should be fine.

Case sensitivity

Python is a **case-sensitive** language. This means that "print" and "Print" are considered to be different words. See what happens when you try to use "Print" instead of "print":

PYTHON

1 Print('Hello world!')

Output and input

Output

As you've seen, you can use print in a program to produce user output:

```
PYTHON ☐

1 print('Hello, again!')
```

A very handy technique when using print is to use an **f-string**. If you append an 'f' to the start of a quoted string (before the first quote) you can get Python to do things inside the string before it prints, such as perform calculations. You do this by using curly brackets. Here are some examples:

```
PYTHON II

1 print(f'The result is {10*2}.')
2 print(f'The results are {10*2}, {10*3}, and {10*4}.')
```

Input

You can also get user input, using input:

```
PYTHON II

1 input('What is your name?')
```

Python pauses until the user hits enter, and then continues executing the program. The example above is not very interesting because the program does nothing with the input it gets. Here's a more interesting example, in which the input gets printed:

```
PYTHON ☐

1 print(input('What is your name? '))
```

It's more interesting if we do something sensible with the input. The following uses an f-string to greet the user:

```
PYTHON ☐

1 print(f"Hello, {input('What is your name? ')}!")
```

Calling functions

You have seen how to use print and input. These are both **functions**. Functions are a very important part of programming. You will learn many different Python functions, and you'll even learn how to create your own.

Calling a function

You **call** a function, or **invoke** it, but putting round brackets after it. Notice the round brackets in the following:

```
PYTHON []

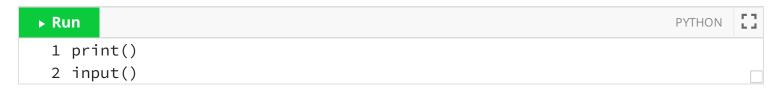
1 print('Hello world!')
2 input('What is your name?')
```

Arguments

Inside the round brackets you can provide the function with **arguments**. In the examples above you have provided the <code>print</code> function with one argument - the string 'Hello world!'. And you have provided the <code>input</code> function with one argument also - the string 'What is your name?'. Each function does something with your argument.

Some functions don't expect any arguments, and will just ignore any you might provide. Some functions demand one or more arguments, and will raise an error if you don't provide them - these are called **required arguments**. Some functions don't expect any arguments but allow you to optionally provide them if you like, and will use them if you do - these are called **optional arguments**. Some functions have a mixture of required arguments and optional arguments. Part of learning a function is learning what arguments it requires and allows.

What about print and input - do they require any arguments? You can try and see:



No, no arguments are required (we didn't get an error).

Multiple arguments

To provide a function with multiple arguments you just separate them by commas. The print function allows you to optionally provide multiple arguments. Let's see what happens if you do:

```
PYTHON ☐

1 print('The', 'cat', 'sat', 'on', 'the', 'mat')
```

It prints each of the arguments, separated by spaces.

Many functions expect you to provide arguments in a very particular order. Again, part of learning the function is learning this order.

Providing arguments by keyword

Remembering the correct order of arguments can be hard. Fortunately, many functions allow you to provide arguments **by keyword**, which means that you can tell the function which argument is which by naming them, rather than putting them in a particular order.

The print function, for example, has some optional arguments that you can provide by keyword. You can use the keyword sep to specify what string to use to separate the items that get printed:

And you can use the end keyword to specify a string to add at the end:

```
PYTHON II

1 print('The', 'cat', 'sat', 'on', 'the', 'mat', end='.')
```

You can use them together, and because you are using keywords you can put the two in any order:

Return values

When you call a function, it always **returns** a value. At the very least it returns the special value None (you'll learn what this is), but typically it returns something more interesting.

The input function returns the string that the user inputs. You can do whatever you want with this string, including ignoring it. In the program below, the first line ignores it, the second line prints it:

```
PYTHON II

1 input('What is your name? ') # Return value ignored
2 print(input('What is your name? ')) # Return value printed
```

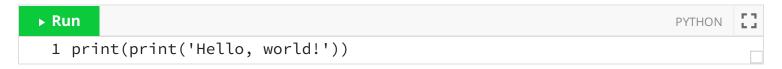
Notice what's happening in that second line: The string 'What is your name? ' is provided as an argument to a call of the input function, and the string that is returned is then provided as an

argument to a call of the print function.



One thing to keep in mind when using the input function is that it always returns a string, even if the user inputs numbers. You will learn more about this, and the importance of it, in later slides.

What does the print function return? It's easy to find out, just print it:



As you can see, it returns None.

Notice what's happening in this example: The string 'Hello, world!' is provided as an argument to a call of the print function; that function prints the string and returns None; None is then provided as argument to another call of the print function, which prints it. That's why two things get printed first Hello, world!, then None.

Encountering errors

We need to talk about errors early on, because errors are a fact of programming life.

There are three main types of error that you might encounter when writing and running programs:

- Syntax errors
- Runtime errors
- Logical errors

Syntax errors

Before executing a program the Python interpreter first checks whether the program has correct syntax. If it finds any syntax errors it will tell you, and not run the program. The following program contains a syntax error (it's missing a quotation mark in line 2) - try running it:

```
PYTHON II

1 print('The program is running')
2 print('Hello, world!)
```

Notice that the first line is not executed - that's because the program never gets to run, because of the syntax error.

Runtime errors

Sometimes the syntax of a program is fine, but Python raises an error during its execution. These are called "runtime errors". When this happens, execution stops, and Python returns some information about the error.

For example, the following program asks Python to divide a number by zero in line 2, which is impossible. It generates a runtime error:

```
PYTHON ☐

1 print('The program is running')
2 print(10/0)
```

Notice that the first line is executed this time - that's because the programs runs, at least until it gets to line 2.

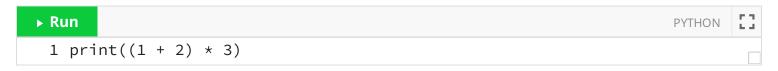
Logical errors

Sometimes the syntax of a program is fine, and it executes without any runtime errors, but it doesn't do what you intended it to do. This is called a "logical error".

For example, suppose you write the following program to add 1 and 2 and then multiply the result by 3:



The result you're expecting is 9, because 1 + 2 is 3, and 3 times 3 is 9. But the program above prints 7. Error! This is not a syntax error, nor a runtime error - it's a logical error. The problem is that Python assumes that you want the multiplication to happen first, then the addition (you will learn more about this). So it calculates $2 \times 3 = 6$, then 1 + 6 = 7. You should write you program like this instead:



This removes the logical error.

Adding comments

In Python, anything after a hash symbol # on a line will be ignored by the Python interpreter. This allows you to do some useful things.

Explaining your code

It is important that the programs you write are easily understandable by someone who reads them. To help with this, it can be a good idea to add comments throughout.

Here is an earlier example, with an explanatory comment added:

```
PYTHON II

1 # Ask the user for input and then print it:
2 print(input('What would you like to print? '))
```

Comments can start anywhere on a line, but comments that come after code can be difficult to read use them sparingly:

```
▶ Run

1 print(input('What would you like to print? ')) # Ask the user for input
```

You should not use comments to state the obvious. Comments should explain the intention of your code, instead of describing the exact procedure it is performing. What the code is *doing* is often obvious, but it may not be obvious *why* it is doing it.

Here in Ed we'll add quite a few comments to code that wouldn't ordinarily be added, for instructional purposes.

Disabling your code

You can also use # to disable one or more lines of code, either because they are not yet finished and will cause an error, or because your program is not working and you are trying to find the cause of the error, or because you are not using them but want to save them just in case you decide to use them again. For example:

```
PYTHON II

1 print('Hello, world!')

2 # print('Goodbye, world!')
```

Keep in mind that disabled code can be distracting, so get rid of it as soon as you can, either by enabling it again, or deleting it altogether.

Multiline comments?

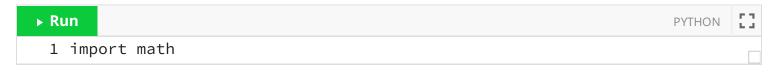
Alas, the only way you can have a multiline comment in Python is to put a # at the start of each line. (Some languages have special techniques.)

As a workaround you could put triple quotes before the first line of code and after the last line, but this does not make the lines into a multiline comment that Python ignores - it makes them into a multiline string which Python evaluates and then does nothing with. It's not good practice.

Importing modules

You won't get far using Python before you need to **import a module**. A module is just a file that contains Python code, with a ".py" extension. Although the core Python language contains a lot of functionality, some of the functionality you'll want is included in modules, rather than in the core. You can access that functionality by importing those modules.

For example, there is a lot of non-core mathematics functionality in the "math.py" module. You can import that module as follows (note that you don't include ".py" - Python will figure it out):



This makes all of the code that has been saved in the math.py file available for use in your program. For example, in math.py there is a function floor, which rounds a number down to the nearest integer. Having imported math.py you can now call this function:

```
PYTHON II

1 import math
2 print(math.floor(3.14))
```

Note that you call the function using math.floor, rather than using just floor. If you'd like to be able to use this function without having to add the prefix, you can do so using the from keyword when you import:

```
PYTHON II

1 from math import floor
2 print(floor(3.14))
```

If you'd like to import more functions then you can just separate their names by commas:

```
PYTHON II

1 from math import floor, ceil
2 print(floor(3.14))
3 print(ceil(3.14))
```

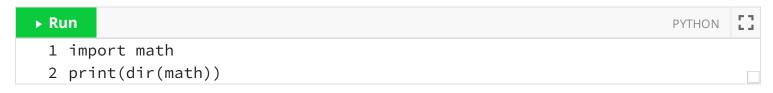
You can import everything by using an asterisk:

```
PYTHON II

1 from math import *
2 print(floor(3.14))
3 print(ceil(3.14))
```

But you should avoid doing this because you might end up with the same function name being used twice.

You can use the dir function to see what is available to you in a module that you have imported:



When you import a module you can give it an alias using the as keyword, to save yourself some typing. One module that we will be using a lot in the second half of this course is the pandas module. It is standard to give it the alias "pd" when importing it:



When Python gets installed on a computer many modules are installed with it. These are called **built-in modules**. These are automatically available for you to import. If you want to import other modules they will first need to be installed on the computer. You can also write your own modules, and import them - you will learn how to do that in Week 3.

Objects

When you program with Python (and many other languages) you work a lot with **objects**. In Python, just about everything is an object. The number 10 is an object, the string "Hello, world!" is an object, even the print function is an object.

Types

Every object is of a certain **type**. In Python, the main types are:

- Integer (int). A whole number, positive or negative, including zero (i.e. ..., -2, -1, 0, 1, 2, ...).
- **Floating-point number (float)**. A positive or negative number, not necessarily whole, including zero (e.g. 3.14, -0.12, 89.56473).
- String (str). A sequence of characters (e.g. 'Hello', 'we34t&2*').
- Boolean (bool). A truth value, either true or false.
- **List (list)**. An ordered collection of objects.
- **Tuple (tuple)**. An immutable list (i.e. one that cannot be changed).
- Set (set). An unordered collection of unique objects.
- **Dictionary (dict)**. A set of key-value pairs.
- Function. A piece of code that can be run by calling it.
- **Class**. A user-defined type of object.

Python also has a special object, None, which represents the absence of an object. It is of type **NoneType**. It is the only object of this type.

You will be learning more about the objects of each type. This week you will learn about integers, floats, strings, and booleans. Next week you will learn about lists, tuples, sets, and dictionaries. In Week 3 you will learn about functions and classes.

Checking the type of an object

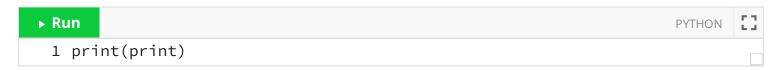
You can check the type of an object by using Python's type function:

```
PYTHON II

1 print(type(1))
2 print(type(3.14))
3 print(type('Hello'))
4 print(type(None))
5 print(type(print))
```

Notice in the last example above that we have provided the print function as an argument to the

type function. You can do that - the print function is an object, just like numbers and strings are, and you can provide it as an argument to the type function to find out what type of object it is. Most of the time you'll be calling print, and providing arguments to it, but occasionally you might provide it as an argument to some other function, as we have done above. In fact, you can even provide print as an argument to itself:



When an object is of a certain type we say that it is an **instance** of that type. You can also use Python's isinstance function to check whether an object is of a certain type:

```
PYTHON II

1 print(isinstance(1, int)) # True
2 print(isinstance(1, float)) # False
3 print(isinstance(3.14, float)) # True
4 print(isinstance(3.14, int)) # False
5 print(isinstance('Hello', str)) # True
6 print(isinstance('Hello', float)) # False
```

Attributes

Objects have attributes.

Each attribute is either a function or a non-function. If it's a function, it's called a **method** of the object. You can think of this as something that you can do with the object. If it's a non-function, it's called a **field** of the object. You can think of this as some fact about the object. So attributes come in two varieties:

- Methods things you can do with the object
- Fields facts about the object

Each attribute has a name, and you can refer to an object's attribute using **dot notation**. For example, a string object has an attribute upper. You can refer to this attribute of the string "Hello" like this: "Hello".upper. It also has an attribute lower, to which you can refer like this: "Hello".lower.

Both of these attributes are functions, so both are methods. Since they are functions you can invoke them using round brackets (), like this: "Hello".upper(), "Hello".lower(). The first is telling Python to invoke the upper method of the "Hello" object; the second is telling Python to invoke the lower method.

What do these two methods do? The first returns the string in upper case, the second returns the string in lower case:

```
PYTHON I

1 print("Hello".upper())
2 print("Hello".lower())
```

Note that every string object has these two methods. But when you invoke these methods they might return different values for different strings - it depends on which string object they are invoked on:

```
PYTHON I

1 print("Hello".upper())
2 print("Goodbye".upper())
```

Which attributes an object has depends upon what type of object it is. String objects have the upper attribute, but function objects do not. If you try to access this attribute of a function you will get an error:

```
PYTHON :

1 print("Hello".upper())
2 print(input.upper())
```

Part of learning Python, and other languages, is learning what attributes the different types of objects have.

Expressions

To work with an object you need to refer to it, and to refer to it you use an **expression**.

Expressions come in two varieties: **simple expressions** and **complex expressions**. We'll consider simple expressions first, and then complex expressions.

There are three types of simple expressions:

- Literals
- Variables
- Constants

Literals

A **literal** shows explicitly which object it refers to. Here are some examples:

- Integer literals: 1, 26, 0
- Floating-point literals: 3.14, 0.06, 0.0, 2.4e12, 5.12E-2
- String literals: 'Hello', "Principles of Programming" (you can use single or double quotes, but they must match)
- Boolean literals: True, False (there are only two)
- None literal: None (there is only one)

Variables

You can also introduce your own names for objects, and these are called **variables**.

When you introduce a variable you have to specify which object it refers to. This is called **assigning** the variable, and you do it using the **assignment operator**. This is sometimes also called **binding** the variable - you bind it to an object.

Here's an example:

```
PYTHON 1 message = 'Hello there' # Introduce a variable and assign it an object 2 print(message)
```

The first line introduces a variable message and assigns it the string object 'Hello there'. The second line uses this variable to print the object.

Once you introduce a variable you can use it as many times as you like throughout your program:

```
PYTHON CONTRACTOR OF THE PYTHON CONTRACTOR OF
```

The object that the variable refers to is often called the **value** of the variable. You can change the value of a variable (i.e. change which object the variable refers to) as often as you like:

```
PYTHON II

1 message = 'Hello there'
2 print(message)
3 message = 3.14 # Assign a new value
4 print(message)
5 message = True # Assign a new value
6 print(message)
```

Notice that the variable message in this program refers to different types of object as the program proceeds - first it refers to a string, then to a floating-point number, and then to a boolean. Because variables in Python can do this we say that Python has **flexible typing**. (Some languages don't allow variables to change their reference to a different type of object.)

If you want to check the type of the object that a variable refers to, you can use the type function:

```
PYTHON []

1 x = 'Hello there'
2 print(type(x)) # str
3 x = 3.14
4 print(type(x)) # float
5 x = True
6 print(type(x)) # bool
7 x = print
8 print(type(x)) # function
```

Any type of object can be assigned to a variable, not just numbers, strings or booleans. Look at the last example above - we have assigned the print function to the variable x. Having done that, x is then another name for the print function, and we can use it accordingly:

```
PYTHON []

1 x = print
2 x('Hello, world!')
```

You can use variables to assign other variables:

```
Print(new_message)
PYTHON II

1 message = 'Hello there'
2 new_message = message
3 print(new_message)
```

Be careful to understand what's going on here. In line 2 you are getting new_message to refer to whatever it is that message refers to at the time. If a different object is later assigned to message, it will not automatically be assigned to new_message as well - there will be no change in what new_message refers to. Let's check:

You must assign a value to a variable before you use it, otherwise Python will generate an error:

```
PYTHON II

1 print(message) # Error - message has no value
```

If you want to introduce a variable but don't yet have any significant value to assign it then you can assign it the value None:

```
PYTHON II

1 message = None
2 print(message) # No error - message has a value
```

You can assign multiple variables the same value:

```
PYTHON II

1 x = y = z = 0 # All get assigned 0

2 print(x, y, z)
```

Or different values, using a technique called **unpacking**:

```
▶ Run

1 x, y, z = 1, 2, 3 # x gets 1, y gets 2, z gets 3
2 print(x, y, z)
```

This technique gives you a way to swap the values of two variables:

```
PYTHON []

1 x, y = 1, 2
2 print(x, y)
3 x, y = y, x # Swap the values of x and y
4 print(x, y)
```

Naming rules

You have a fair bit of freedom in what names you can introduce, but there are some rules and restrictions.

First, you cannot use any of Python's keywords, such as import, from, True, False, None, etc. There are 33 keywords in version 3 - you will be learning most of them.

Second, a name can only contain uppercase letters (A-Z), lower case letters (a-z), digits (0-9) and the underscore (_). Moreover, the first character cannot be a digit.

Keep in mind that Python is case sensitive, and this applies to your names as well. The program below generates an error because the variable Message is not given a value before it is used (the variable message is, but that's a different variable):

```
PYTHON II

1 message = 'Hello there'
2 print(Message) # Error - Message does not have a value
```

It is best to choose names for your variables that make the intention of your program as clear as possible. Consider the following two pieces of code:

```
var1 = 10
var2 = 120
print(var1 * var2)

days = 10
fish_per_day = 120
print(days * fish_per_day)
```

Both pieces of code do the same thing, but the second makes it much clearer what is going on. In effect, by choosing variable names carefully we can use them to help explain the code.

It is fairly standard to make variable names lowercase, with words separated by underscores, e.g, fish_per_day. This is often called **snake case**.

Unbinding variables

You can unbind a variable from an object by using del.

```
PYTHON II

1 x = 10
2 print(x)
3 del x # Unbind x from its value
4 print(x) # Now we get an error
```

Note that the word "del" is a bit misleading. You are not deleting the variable or the object it is bound to - you are just severing the connection between them.

Although, when you do this the variable and/or the object might get removed from memory, via a process called **garbage collection**. An object is removed from memory by Python when nothing is referring to it.

Also note that unbinding a variable is not the same as **rebinding** it to None:

```
PYTHON []

1 x = 10
2 print(x)
3 x = None
4 print(x) # No error - x has a value
```

Choosing types

You can get Python to return an object of a certain type by using the functions int, float, str, bool, list, tuple, set, and dict. For example, if you would like var to refer to the floating-point number 1 rather than the integer 1 you can use the float function:

```
PYTHON II

1 var = 1 # var refers to the integer 1
2 print(var, type(var))
3
4 var = float(1) # var refers to the floating point number 1.0
5 print(var, type(var))
```

You might need to do this when you are getting user input. Python treats user input as a string, so if you are asking the user to enter a number then you will need to convert the input, using int or float:

```
1 # number will be a string:
2 number = input('Enter a number: ')
3 print(number, type(number))
4
5 # number will be an integer:
6 number = int(input('Enter a number: '))
7 print(number, type(number))
8
9 # number will be a float:
10 number = float(input('Enter a number: '))
11 print(number, type(number))
```

Sometimes Python can't return an object of the type you are asking for, and it will raise an error. For example, Python cannot make every string into an integer:

```
PYTHON 1

1 var = int('hello') # Error - cannot make 'hello' into an integer
2 print(type(var))
```

You can use the same functions to change the type of object a variable refers to after it has been set:

```
1 var = 2 # var refer to the integer 2, by default
2 print(var, type(var))
3
4 var = float(var) # var now refers to the floating-point number 2.0
5 print(var, type(var))
6
7 var = str(var) # var now refers to the string '2.0'
8 print(var, type(var))
```

Note that when it operates on a floating-point number the int function truncates all decimal places:

```
PYTHON II

1 print(int(1.2))
2 print(int(-1.2))
```

Constants

If you intend the value of a variable not to change, then you are using it as a **constant**. It is conventional to indicate this by naming it using all capital letters, with underscores separating the words, e.g, MAX_INT. One of the main reasons for using constants is to give an indication as to why a particular value is being used (e.g. using the constant HOURS_PER_DAY instead of the literal 24), so again, it's best to use names that help to explain what your program does:

Note that just with any other variable, it is *possible* to change the value of a constant - it is up to the programmer to ensure that variables that are intended to be constants do not change after they are defined.

Complex expressions

A **complex expression** is an expression that contains subexpressions, combined using arithmetical and other kinds of operators (you will be learning about various kinds of operators).

For example, $10 \star 2$. This is an expression, because it refers to an object - the number 20. And it is a complex expression, because it contains a subexpression - the literal 10. Actually, it contains another subexpression too - the literal 2. But it only needs to contain one subexpression to count as complex. These two subexpressions are combined using the \star operator.

You will see many examples of complex expressions as we proceed.

Working with numbers

Python distinguishes two types of numbers:

- Integers
- Floating point numbers

There are also complex numbers, but we won't be considering them in this course.

Operating on numbers

You can add, subtract, multiply, and divide numbers by using the **operators** +, -, *, /, respectively, and combinations of them:

```
Print(1 + 2)
1 print(1 + 2)
2 print(10 - 5)
3 print(3 * 4)
4 print(20/4)
5 print((1 + 2)*(3 + 4))
6 print(10/(3-1))
```

You can raise one number to the power of another number by using the **exponentiation operator**, **:

```
PYTHON []

1 print(10 ** 2) # 10 to the power 2
2 print(2 ** 3) # 2 to the power 3
```

You can divide one number by another number and round down to the nearest whole number by using the **integer division operator**, //:

```
PYTHON ☐

1 print(10 // 3) # 10 divided by 3 and rounded down
```

Note the effect of using // on negative numbers:

```
PYTHON II

1 print(-10 // 3) # -10 divided by 3 and rounded down
```

You can divide one number by another number and get the remainder by using the **modulus operator**, %:

```
PYTHON II

1 print(10 % 3) # The remainder when 10 is divided by 3
```

Order of operations

Unless you specify the order in which operations are to be performed, by using brackets, Python performs them in a very particular order:

- First, ** is performed, from right to left
- Next, *, /, //, and % are performed, from left to right.
- Next, + and are performed, from left to right.

```
PYTHON 1

1 print(1 + 2 * 3) # Same as 1 + (2*3)
2 print(2 ** 3 * 4) # Same as (2**3) * 4
3 print(24 / 6 * 2) # Same as (24/6) * 2
4 print(24 / 6 / 2) # Same as (24/6) / 2
5 print(2 ** 2 ** 3) # Same as 2**(2**3)
```

Augmented assignment operators

You will often find yourself wanting to operate on a variable and then re-assign the result to that same variable. Python provides **augmented assignment operators** to allow for more concise code: +=, -=, *=,

- x += 2 is equivalent to x = x + 2
- x = 2 is equivalent to x = x 2
- x *= 2 is equivalent to x = x * 2
- $x \neq 2$ is equivalent to $x = x \neq 2$
- x //= 2 is equivalent to x = x // 2
- x % = 2 is equivalent to x = x % 2

```
PYTHON II

1 x = 12
2 x += 2 # Equivalent to x = x + 2 (assign to x the result of x + 2)
3 print(x)
```

Comparing numbers

Python has a number of **comparison operators** which you can use to compare numbers:

- num1 == num2 is True if the value of num1 equals the value of num2, otherwise it is False
- num1 is num2 is True if the value of num1 **equals** the value of num2, and num1 and num2

have the same type, otherwise it is False

- num1 != num2 is True if the value of num1 does not equal the value of num2, otherwise it is
 False
- num1 is not num2 is True if the value of num1 **does not equal** the value of num2, or num1 and num2 **have different types**, otherwise it is False
- num1 < num2 is True if the value of num1 **is less than** the value of num2, otherwise it is False
- num1 <= num2 is True if the value of num1 is less than or equal to the value of num2,
 otherwise it is False
- num1 > num2 is True if the value of num1 **is greater than** the value of num2, otherwise it is False
- num1 >= num2 is True if the value of num1 is greater than or equal to the value of num2,
 otherwise it is False

```
PYTHON []

1 print(1 == 1.0)
2 print(1 is 1.0)
```

Floating-point numbers and ==

Because floating point numbers are stored with limited precision you might experience strange results when attempting to compare them with == . For example:

```
▶ Run

1 print(0.1 + 0.2 == 0.3)
```

Why does this comparison return False? Because 0.1 + 0.2 is not what you would expect:

```
▶ Run

1 print(0.1 + 0.2)
```

When you compare floating point numbers using == or != you should always round them float to a specified precision, using the round function: round(x, n) rounds the value of x to n decimal places.

```
PYTHON []

1 print(round(0.1 + 0.2, 1))
2 print(round(0.3, 1))
3 print(round(0.1 + 0.2, 1) == round(0.3, 1))
```

Note: the round function implements round half to even:

```
PYTHON 13

1 print(round(4.5,0))
2 print(round(5.5,0))
3 print(round(-4.5,0))
4 print(round(-5.5,0))
```

Applying functions

Python has the functions int, float, abs, pow, round that you can use to manipulate numbers:

```
PYTHON ☐

1 print(int('3') + float('2.0')) # Convert strings to numbers and add them
2 print(abs(-5)) # Absolute value of -5
3 print(pow(2, 4)) # 2 to the power of 4. This is the same as 2**4
4 print(round(3.567, 2)) # Round 3.567 to 2 decimal places
```

Generating random numbers

It can be very useful to generate random numbers. You can do this by importing the functions random and randint from the random module.

```
PYTHON II

1 from random import random, randint
2 print(random()) # Random float between 0 and 1
3 print(randint(10, 20)) # Random integer between 10 and 20 (inclusive)
```

Other mathematical functions

A lot of other mathematical functions that you might need are in in math module:

```
1 import math
2 print(math.sin(0.5)) # Degrees are in radians
3 print(math.cos(0.5))
4 print(math.tan(0.5))
5 print(math.sqrt(225)) # Square root
6 print(math.log(10)) # Natural logarithm, base e
7 print(math.ceil(2.05)) # Round up to nearest integer
8 print(math.floor(2.95)) # Round down to nearest integer
```

Working with strings

A string is a sequence of characters. The characters might be ones you can type on your keyboard, or any of the hundreds of thousands of other **unicode characters**, including symbols and foreign language letters.

String literals

As you have seen, string literals have the characters between quotes, either single quotes or double quotes.

```
PYTHON I

1 literal_one = 'A string' # You can use single quotes
2 literal_two = "A string" # Or double quotes
3 print(literal_one)
4 print(literal_two)
```

The empty string

One of the most useful strings is the **empty string** - a string with no characters. The literal for an empty string is ''', or "".

Note that the empty string is not the same thing as None - they are different objects:

```
PYTHON II

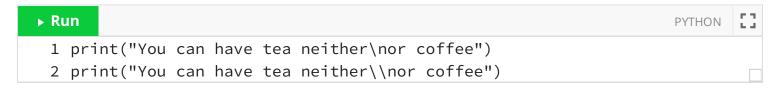
1 print('') # Empty string
2 print(None) # None
3 print('' is None) # They are not the same
```

Escaping special characters

Sometimes you will need to use a string literal that contains quote marks or other characters that have special meaning in Python. You can do this by **escaping** those special characters, which means prefixing them with a backslash \setminus .

```
1 # Backslash used to escape a quote mark
2 print('Penny\'s dog')
3 print("Penny's dog is called \"Mac\".")
4
5 # Backslash used to escape a new line symbol
6 print('This is one line.\nThis is a second line')
7
8 # Backslash used to escape a tab symbol
9 print('Here\tThere')
```

This means that the backslash itself has a special meaning in strings, so to include it you must escape it:



If you have a lot of backslashes and no special characters then you can tell Python to ignore the special meaning of the backslash by preceding the string with r (for "raw"):

```
▶ Run

1 print(r"You can have tea neither\nor coffee")
```

You can avoid having to escape single quote marks by enclosing the whole string in double quotes. Similarly, you can avoid having to escape double quote marks by enclosing the whole string in single quotes:

```
PYTHON ☐

1 print("Penny's dog")
2 print('Penny said, "This is my dog".')
```

If you like to include line breaks in a string then you can either use \n , as in the example above, or you can use triple quotes around the string:

```
# Run

1 # Using \n
2 text = 'This is one line.\nThis is a second line'
3 print(text)
4

5 # Using triple single quotes
6 text = '''This is one line.
7 This is a second line.'''
8 print(text)
9

10 # Using triple double quotes
11 text = """This is one line.
12 This is a second line."""
13 print(text)
14
```

You can also use triple quotes around a string that contains both single and double quotes:

```
PYTHON []

1 text = '''She said, "I don't know how you do it!"'''
2 print(text)
3 text = """She said, 'I don't know what "" means'"""
4 print(text)
```

You'll get an error if you end up with four quotes in a row.

Concatenating strings

You can **concatenate** strings (i.e. join them) using the + operator:

You can also use the += augmented assignment operator:

```
Print(name)
PYTHON

1 name = 'Leo'
2 name += ' '
3 name += 'Tolstoy'
4 print(name)
```

You can duplicate a string a given number of times by using the ★ operator:

```
PYTHON II

1 print('a' * 10)
```

Comparing strings

Just as with numbers, you can use Python's comparison operators to compare strings:

- str1 == str2 is True if str1 and str2 are the same sequence of characters, otherwise it is False
- str1 != str2 is True if str1 and str2 are not the same sequence of characters, otherwise it is False
- str1 in str2 is True if str1 appears as a substring in str2, otherwise it is False
- str1 not in str2 is True if str1 does not appear as a substring in str2, otherwise it is False
- str1 < str2 is True if str1 **is lexicographically less than** (i.e. would appear earlier in the dictionary than) str2, otherwise it is False
- Similarly we have str1 <= str2, str1 > str2, and str1 >= str2

Applying functions to strings

You can use the len function to find the length of a string:

```
PYTHON ☐

1 print(len('abcde'))
```

Calling string methods

String objects have many useful methods. For example, they have a method upper which returns a string with the same characters but all in upper case.

```
PYTHON []

1 s = 'hello'
2 print(s.upper())
```

None of a string's methods **modify the string in place** - they all **return a new string**. So, if you want to modify a string by using one of the methods you have to assign the result of the method back to the string. To illustrate:

```
PYTHON ::

1    s = 'hello'
2    print(s)
3
4    s.upper() # Returns a new string - does not change s
5    print(s)
6
7    s = s.upper() # Assign the new string back to s - changes s
8    print(s)
```

Some of the most useful string methods are listed below.

- str.isupper Returns true if all characters in str are upper case
- str.islower Returns true if all characters in str are lower case
- str.isalpha Returns true if all characters in str are from the alphabet
- str.isdigit Returns true if all characters in str are digits
- str.isnumeric Returns true if all characters in str are numeric
- str.isspace Returns true if all characters in str are whitespace characters (i.e. space, tab, or new line)
- str.startswith Returns true if str starts with the specified value
- str.endswith Returns true if str ends with the specified value
- str.upper Returns str with every character in upper case
- str.lower Returns str with every character in lower case
- str.title Returns str with the first character of each word in upper case
- str.capitalize Returns str with the first character uppercase and the rest lowercase
- str.format Returns str formatted as specified
- str.find Searches str for a specified value and returns the index at which it is first found (or -1 if it was not found)
- str.index like str.find but raises an error if the value is not found
- str.rfind Searches str for a specified value and returns the index at which it is last found (or -1 if it was not found)
- str.count Returns the number of times a specified value occurs in str
- str.strip Returns str with whitespace stripped from both ends

- str.lstrip Returns str with whitespace stripped from the left end
- str.rstrip Returns str with whitespace stripped from the right end
- str.replace Returns str with a specified value replaced by a specified value

Working with booleans

Boolean objects are the simplest type of objects in Python, but often they play the most significant role in determining the path that a program takes (as you will see).

There are two boolean objects: True and False. They most commonly appear as the result of the comparison operations we have seen for the other types, for example 3 < 5 will evaluate to True; and 1 + 1 == 3 will evaluate to False:

```
      ▶ Run
      PYTHON
      I

      1 print(3 < 5) # True</td>
      2 print(1 + 1 == 3) # False
```

Operating on booleans

You can build complex boolean expressions by combining simple boolean expressions with the logical operators not, and, and or:

• not x is true if x is false, otherwise it is false.

1 print(True or True) # True
2 print(True or False) # True
3 print(False or True) # True
4 print(False or False) # False

5 print((3 < 5) or (1 + 1 == 3)) # True

- x and y is true if x is true and y is true, otherwise it is false.
- x or y is true if x is true or y is true or both, otherwise it is false

```
PYTHON 

1 print(not True) # False
2 print(not False) # True
3 print(not 3 < 5) # False

PYTHON 

1 print(True and True) # True
2 print(True and False) # False
3 print(False and True) # False
4 print(False and False) # False
5 print((3 < 5) and (1 + 1 == 3)) # False
```

```
PYTHON II

1 guess = int(input('Enter a number: '))
2 print('Your number is between 2 and 5?')
3 print(guess > 2 and guess < 5)
```

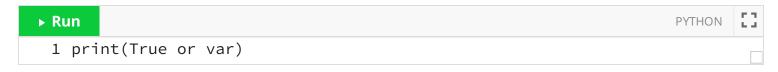
Chaining comparisons

It is possible to combine comparisons into a chain:

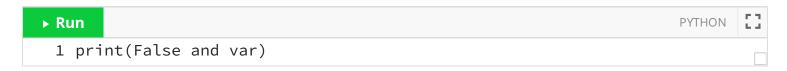
Short circuiting

The operators or and and are said to be **short circuiting** operators, because they only evaluate their second expression if they need to.

To illustrate, the program below does not generate an error, even though the variable var has not been defined. That's because Python doesn't even look at that variable. It doesn't need to, because the first True is enough to make the whole expression true.



Similarly, the program below does not generate an error, even though the variable var has not been defined. That's because Python doesn't need to look at it - the first False is enough to make the whole expression false.



If you swap the order in either case, you get an error:

```
▶ Run

1 print(var and False)
```

Controlling program flow

In a simple program, Python executes the statements of the program one-by-one, from start to finish. However, you will often want your program to deviate from this:

- **Conditional execution**. You might want to execute a certain statement only if a certain condition is true.
- Loops. You might want to loop through a block of code multiple times.
- **Handling exceptions**. You might want to run a block of code in a "cautious" way, so that if an error arises you can deal with it without your whole program stopping.

We will look at these one-by-one in the next few slides.

If statements

You will often want Python to execute a certain statement only if a certain condition is true. For this you can use an if statement.

The following program uses an if statement:

If the user enters the number 42 then Python executes line 3, otherwise it skips line 3 and goes straight to line 4.

Syntax

The syntax for an if statement is:

```
if <expression>:
    <statement(s)>
```

There are two parts to this statement - the part between if and : is called the **header** of the statement; the rest is called the **body** of the statement. The body of the statement is a **block** of statements.

Notice that the body contains one or more statements (as many as you like), so an if statement contains other statements as part of it. Because of this we call it a **compound statement**.

Also notice that the body is **indented**. This is required - if you don't use indentation then Python will issue an error:

```
PYTHON II

1 if True:
2 print('Hello')
```

You can use either the tab character or the space character to create the indentation, but you must use the same character for each line, and the same number of those characters, otherwise Python will issue an error. Each of the following will cause an error:

```
PYTHON II

1 if True:
2  print('Hello') # Indentation using a tab
3  print('Hello') # Indentation using 4 spaces - Error

PYTHON II

1 if True:
2  print('Hello') # Indentation using 4 spaces
3  print('Hello') # Indentation using 5 spaces - Error
```

It is standard to use 4 spaces for indentation. You can set this to be the default behaviour in the code editor for ed by selecting "Soft tabs" under the settings menu (icon: ♥) in the top right of an editor window.

The statement block after if must contain at least one statement. It is common use the pass statement as a placeholder for unfinished code - it is a statement that does nothing:

```
PYTHON I

1 if True:
2 pass # There must be at least one statement in the body
```

Conditions

Any boolean expression can be used between the if and the : :

```
PYTHON []

1 if True:
2   print('The condition is true')
3 if 2 > 1:
4   print('The condition is true')
5 if 2 > 1 and 2 < 3:
6   print('The condition is true')
7 if 1 > 2 or 2 > 1:
8   print('The condition is true')
9 if 'cat' == 'cat':
10   print('The condition is true')
```

Be careful with variables

If you introduce a variable inside the block of an if statement then that variable will only be defined if the block is executed. This might cause you some unexpected errors. For example:

```
PYTHON II

1 if False:
2  x = 1
3 print(x) # Error - x does not have a value
```

Since line 2 is not executed the variable \times does not get assigned a value, so when it is used in line 3 Python issues an error.

Else clauses

You can add an else **clause** to an if statement, to tell Python what to do if the condition of the if statement is false:

```
1 number = int(input('What is your favourite number? '))
2 if number == 42:
3    print('That is my favourite number too!')
4 else:
5    print('That is not my favourite number.')
```

Elif clauses

You can add elif clauses (short for 'else if') to chain together multiple conditions:

```
1 number = int(input('What is your favourite number? '))
2 if number == 42:
3    print('That is my favourite number too!')
4 elif number == 21:
5    print('That is my second favourite number')
6 else:
7    print('That is neither of my favourite numbers.')
```

Abbreviations

If you only have one statement in an if body then you can put it on the same line as the header. The same applies to elif and else. Note that you still need the colon:

```
PYTHON []

1 number = int(input('What is your favourite number? '))
2 if number == 42: print('That is my favourite number too!')
3 elif number == 21: print('That is my second favourite number')
4 else: print('That is neither of my favourite numbers.')
```

Nesting

Inside the block of an if statement you can have other if statements. These other if statements are said to be **nested**:

```
1 number = int(input('What is your favourite number? '))
2 if number > 10:
3    print('That is a big number')
4    if number > 100: # This if statement is nested
5    print('It is bigger than 100')
6 print('Good bye')
```

Breaking up complex expressions

Do not try to do too much in one go by building overly complex expressions; code should not be concise at the expense of readability. Consider the following two pieces of code:

```
1 if (is_admin and not admin_expired) or (is_person and (has_override or s call_security()  
3 else:
4    activate_launch()

1 if is_admin and not admin_expired:
2    call_security()  
3 elif is_person and has_override:
4    call_security()  
5 elif is_person and special_override:
6    call_security()  
7 else:
8    activate_launch()
```

The second piece of code is vastly more clear on the conditions required for a launch.

Ternary expression

Python has an expression whose value depends upon a condition.

```
<expression> if <expression> else <expression>
```

It is called a **ternary** expression, because it combines three expressions into one.

Here is an example of it being used to assign a value to a variable:

```
PYTHON II

1 x = 23
2 parity = "even" if x % 2 == 0 else "odd"
3 print(parity)
```

This is equivalent to using the following if statement:

Notice that the ternary expression is an *expression*, not a statement. This means that it returns a value, which you can assign to variables or print:

An if statement is not an expression, so it doesn't return a value that you can assign to variables or print.:

Be careful when using this ternary operator that your code does not become difficult to read. It is usually best to put parentheses around it.

While statements

Sometimes you might want to repeat a set of statements for as long as a certain condition is true. For this you can use a while statement.

Here's a program that uses a while statement to print the first 10 positive integers:

```
PRUN

1 n = 1
2 while n <= 10:
3    print(n)
4    n = n + 1
5 print('Finished')</pre>
```

When Python gets to line 2 it evaluates the condition after while. If the condition is true then it executes the statement block below, in lines 3-4, and then returns to line 2 again. If the condition is false then it skips the block and goes straight to line 5.

You could achieve the same effect by using 10 different print statements, but using a while statement is more elegant and less repetitive. And if you don't know in advance how many integers to print, for example if you want to ask the user, then it might be impossible to use just print statements.

The while block

The while block can contain any statement(s) you like, including if statements and other while statements. For example, here's a program that prints the even numbers between 1 and 10. It uses an if statement inside the while loop:

Here is an example of a while loop used to iterate through a string, counting the number of times 'e' occurs in it.

```
PYTHON II

1 string = 'The quick brown fox jumped over the lazy dog'
2 occurrences = 0
3 i = 0
4 while i < len(string):
5    if string[i] == 'e':
6     occurrences += 1
7    i += 1
8 print("The letter 'e' occurs", occurrences, "times")</pre>
```

Continuing

You can use a continue statement to skip to the next iteration of a loop:

When the value of i gets to 5 the continue statement is executed, and Python jumps directly back to line 2 and continues. The number 5 does not get printed, but 6 - 10 do.

Why might you use continue? It can help to keep your code from getting to many levels of indentation. We will see examples of this.

Breaking

You can use a break statement to break out of a loop entirely:

```
PRun

1 i = 0
2 while i < 10:
3    i += 1
4    if i == 5:
5     break
6    print(i)
7 print('Finished')</pre>
```

When the value of i gets to 5 the break statement is executed, and Python jumps directly to line 7. The number 5 does not get printed, and nor do 6 - 10.

Note that if the loop is nested inside another loop, the break statement terminates only the inner

loop.

Keeping a program running

When you run the program below is stops after it gets and prints a name. To run it again you have to click 'Run' again:

```
PYTHON II

1 name = input('What is your name? ')
2 print('Hello', name)
```

It can be convenient to have the program keep running - getting it to start again automatically after it does its thing. You can get it to do this by adding a while loop, with a condition that always evaluates to true:

```
PYTHON []

1 while True:
2    name = input('What is your name? ')
3    print('Hello', name)
```

Now the program will keep running, until you click 'Stop'.

If you want to get a bit fancier, you could get your program to stop when the user enters a certain value, such as 'q'. Remember to let the user know that they can do this:

```
PYTHON []

1 while True:
2    name = input('What is your name? (Enter q to quit) ')
3    if name == 'q':
4        break
5    print('Hello', name)
```

Handling exceptions

If your code interacts with the outside world, you may encounter unexpected circumstances. Perhaps a file that you are trying to open doesn't exist, or when you try to save the user's data you find that the disk is full, or perhaps the user enters a non integer value when you are expecting an integer, or maybe you divided by zero.

Exceptions are a mechanism for dealing with the unexpected.

The approach in Python is known as **structured exception handling**. This means that if Python encounters an exception in a block of code, it will search for an exception handler enclosing that block. If none exists, it will look for an exception handler enclosing that outer block, and so on.

It is important to understand that exception handling should be used to handle exceptions that are recoverable. If your program cannot handle the exception or recover from it, it should not even try. Instead it should let the exception terminate your program so the user can deal with it instead. For many cases of exceptions, there is no remedy.

Try and except

You can handle exceptions by using a try statement.

Consider the following program, which asks the user to enter a number and then returns the square of that number:

```
PYTHON II

1 num = input("Please enter a number: ")
2 result = float(num) ** 2
3 print("The square of the number is", result)
```

If the user enters a string of letters instead of a number then an error occurs and the program very ungracefully ends.

Python complained with a ValueError. To handle this, we'd write a try ...except ... block around the code that could potentially raise an exception.

```
PYTHON II

1 num = input("Please enter a number: ")
2 try:
3    result = float(num) ** 2
4    print("The square of the number is", result)
5 except:
6    print("You did not enter a number.")
```

In the above code, when control reaches line 2, an exception is raised. Python finds the nearest exception handler and moves control to line 3, which proceeds into the block at line 4. If there was no exception on line 2, then the code in the except block would not be executed.

If you don't want to do anything with the exception then you can use the pass statement:

```
PYTHON II

1 num = input("Please enter a number: ")
2 try:
3    result = float(num) ** 2
4    print("The square of the number is", result)
5 except:
6    pass
```

When an exception occurs, there is information about the exception contained in a special Exception object. You can assign that object to a variable and then display that information to the user:

```
PYTHON []

1 try:
2 int('string')
3 except Exception as err:
4 print('Exeption: ', err)
```

Else

If you have some code that you want to execute only if the try block succeeds then you can add an else clause:

```
PYTHON []

1 x = 'Hello'
2 try:
3     x = int(x)
4 except:
5     print('The conversion was not successful.')
6 else:
7     print('The conversion was successful.')
```

Working with files

So far we have seen programs that interact with the external environment via the input and print functions. Another way they can interact is by reading from and writing to files. This is very simple in Python, which is why Python is a popular tool for working with files.

This is what you will typically want to do:

- Open the file
- Read from the file, or write to the file
- Close the file

Opening a file

To work with a file you must first open it. You do so using the open function:

```
PYTHON II

1 f = open('MyData', 'w')
```

The function expects a **file** as the first argument. This can be given as an absolute or relative filename. If given as a relative filename, it is relative to the directory that Python was executed from - in Ed this is always the same directory as the program.

You can also supply a **mode** as the second argument. This indicates whether the file is to be opened for reading (i.e. input) or writing (i.e. output). If you don't specify the mode, Python assumes that you want to open the file for reading.

The available options for the mode are:

- 'r' Open the file for reading. open will throw an exception if the file does not exist.
- 'w' Open the file for writing. If the file exists, the contents are completely overwritten. If the file does not exist, it will be created.
- 'x' Open the file for writing. If the file already exists this will throw an exception. If the file does not exist, it will be created.
- 'a' Open the file for appending. The file is opened at the end and any writes to the file will append to the end. If the file does not exist it is created. This option is useful for adding information to a file for example a log file.

The open function returns a file object (also called a **file handle**) that represents the file that you have opened. It is this object that allows you to perform operations on the underlying file itself.

Reading from and writing to a file

The read method of a file object returns a string that contains the entire contents of the file. The write method takes a string and adds it to the file.

```
1 # Open a file for writing
2 file = open('myfile', 'w')
3
4 # Write to the file
5 # If you want a newline anywhere you have to add it, using \n
6 file.write('Line 1: Some text.\n')
7
8 # Write some more to the file
9 file.write('Line 2: Some more text.')
10
11 # Now open the file for reading
12 file = open('myfile', 'r')
13
14 # Print the contents
```

Closing a file

You can close a file by using its close method. It is important to close a file after you are done working with it, to free up resources back to the system. A program can only have a limited number of files open while it is running. If a running program reaches this limit, it will receive a "Too many open files" error when it attempts to open more files.

So the program above should look like this instead:

```
PYTHON []

1 file = open('myfile', 'w')
2 file.write('Line 1: Some text.\n')
3 file.write('Line 2: Some more text.')
4 file.close()
5
6 file = open('myfile', 'r')
7 print(file.read())
8 file.close()
```

Using a with block

It's a good idea to work with a file inside a with statement. This ensures that the file is always closed after use, even if an error occurs inside your program. As soon as control exits the with statement the file will be automatically closed.

Here is the previous example written with with statements:

Run

1 # Open a file for writing
2 # The file is automatically closed after the with statement
3 with open('myfile', 'w') as file:
4 file.write('Line 1: Some text.\n')
5 file.write('Line 2: Some more text.')
6
7 # Open the file again for reading
8 # The file is automatically closed after the with statement
9 with open('myfile', 'r') as file:

print(file.read())

10

Further reading

You might find the following helpful:

- Chapters 2, 5, 7 and 10 of Python Crash Course
- The Python Tutorial at w3schools.com