Week 1 - Python Part 1

Programs, programming and languages

A computer *program* is a set of instructions for a computer to follow. Computer *programming* 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, C, Ruby, JavaScript, PHP, and many, many more. Each language has its own advantages and disadvantages, and there is no "best" language that will be the most appropriate choice in all situations. While, for expediency, this course will focus on a single language (Python), many of the fundamental principles you will learn, such as program flow, data manipulation and code modularity, carry over to almost every other language.

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. Python has very simple syntax and is considered one of the best designed programming languages in this respect.

Python

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

- it is a general-purpose language that can be used in a wide variety of situations;
- it is 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 making its learning curve easily managed.

It is used by large technology companies such as Google and Amazon, and also by researchers analysing data in climate models and astrophysics. It has become the *lingua franca* for data scientists - its data processing and visualisation libraries such as **NumPy** and **Pandas** make handling data a breeze.

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 incompatible with each other.

In this course, code will usually be presented in a web interpreter as seen below, the language the code is written in can be seen in the top right corner of the web interpreter, most code will be editable and runnable. Try running the code below to see the exact version of Python 3 we are using (in ed):

```python
import sys
print(sys.version)
```

Python nuts and bolts

Python is an *interpreted* language which means that code is passed directly to an interpreter (a program running on the host machine) for execution rather than being *compiled* into machine-readable instructions to be executed directly by the computer. This means that executing Python code can be as simple as starting an interpreter and typing commands to run. If you installed a version of Python on your computer then there will be an interpreter as part of that installation. Alternatively, you can access an interpreter through the *workspaces* and *code snippets* here on **ed** platform. Increasing usability by simplifying the process from program writing to execution makes Python a very attractive language for the novice programmer.

Programming basics

Here is a very simple Python program:

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

Execution

You can execute the program (i.e. run it) by clicking 'Run'. When you do, 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).

Statements

Each line the program is called a *statement*. Your programs will typically contain many statements. Here is one with two statements:

```python
print('Hello, world!')
print('Goodbye, world!')
```

The print() function

`print()` is a Python *function,* one that you will find yourself using often. `'Hello world!'` is an *argument* of the function - some input that we give to the function. The `print()` function performs the task of printing the argument.

Try modifying the program below to make it print your name (you need to use quotation marks):

```python
print()
```

If you supply `print()` with multiple arguments it will print them all, separated by spaces. You will find this very useful:

```python
print('The value is', 10*2)
```

Getting user input

Another Python function that you will use often is `input()`, which you can use to get input from the user. For example:

```python
print(input('What would you like to print? '))
```

This program asks the use for some input and then prints it.

Syntax errors

Before executing a program the interpreter first checks that it 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) - try running it:

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

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, which is impossible. It generates a runtime error:

```python
print(10/0)
```

Comments

In Python, anything on the same line after a `#` will be ignored by the Python interpreter. This allows you to add comments. 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 example from above, with an explanatory comment added:

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

Comments can start anywhere on a line (but use comments on the same line as the code sparingly):

```python
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 intentions of the 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 that.

Disabling 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
print('Hello, world!')
# print('Goodbye, world!')
```

Whitespace

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

```python

print  ( 'Hello'   )

print(   10    *      2)
```

```python
print('Hello')
print(10*2)
```

Spacing at the start of a line (i.e. *indentation*) is not ignored by Python - it indicates a *compound statement* and we will see this further when we cover the `if` and `while` commands. If you incorrectly add indentation, Python will throw a (syntax) error.

```python
    print('Hello')
```

Spacing inside quotes is also not ignored by Python:

```python
print('Hello     world')
```

There is no standard convention over whether `print('Hello')` should be favoured over `print( 'Hello' )` or `print ( 'Hello' )` however it is good coding practice to use *consistent* spacing rules throughout your program.

Objects

When you program with Python (and many other languages) you will work a lot with *objects.* Objects encapsulate a piece of data (which may be simple, such as a single number; or more complex, such as a collection of smaller objects) and are the building blocks of Python. For example, in the statement `print('The value is', 10*2)`, `'The value is'`, `10` and `2` are all objects. Indeed, even the `print()` function is an object.

Types

Every object is of a certain *type*. The type of an object determines how Python interacts with it, for example it makes sense to add two numbers, but it does not make sense to add functions. 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*'). Used to store text.
- **Boolean (bool)**. A truth value, either true or false.
- **List (list)**. An ordered container of objects.
- **Tuple (tuple)**. An immutable list (i.e. one that cannot be changed).
- **Set (set)**. An unordered container of unique objects.
- **Dictionary (dict)**. A set of key-object pairs.
- **Function**. A piece of code that can be run by calling it.
- **Class**. A user-defined type of object.

You will be learning more about objects of each type.

None

Python also has a special object, `None`, which represents the absence of an object. Somewhat paradoxical!

Checking the type of an object

You can find out the type of an object by using Python's `type()` function:

```python
print(type(1))
print(type(3.14))
print(type('Hello'))
print(type(None))
print(type(print))
```

You can also use Python's `isinstance()` function, to check whether an object is of a certain type (so-called because an object is an *instance* of its type):

```python
print(isinstance(1, int)) # True
print(isinstance(1, float)) # False
print(isinstance(3.14, float)) # True
print(isinstance(3.14, int)) # False
print(isinstance('Hello', str)) # True
print(isinstance('Hello', float)) # False
```

Attributes

Objects have *attributes*. Attributes are properties that are specific to the object. For example, the string object 'Hello', has an attribute `upper()`, which is a function that produces an upper-case version of the string. You can access this attribute of the string by using *dot notation*:

```python
print('Hello'.upper())
```

It can be useful to think of '.' as representing " 's " (i.e. apostrophe-s) - the statement `'Hello'.upper()` is instructing Python to execute the `upper()` function of the `'Hello'` object.

When an attribute is a function, like this one is, it is also called a *method*. Attributes which are not methods are also known as *fields*.

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

You will be learning more about the attributes that each type of object has.

Expressions

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

Literals

One way of referring to an object is to use a *literal*. Literals show explicitly which object they refer to. Here are some examples:

- Integer literals: `1`, `26`, `-14`
- Floating-point literals: `3.14`, `0.06`, `-9.7`
- String literals: `'Hello'`, `"Goodbye"` (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 refer to an object using a *variable*. You have to specify which object the variable refers to, using the *assignment operator* `=`. Consider the program below:

Run PYTHON

```python
message = 'Hello there'
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:

Run PYTHON

```python
message = 'Hello there'
print(message)
print(message)
print(message)
print(message)
```

You can change the value of a variable in a program as often as you like:

```python
message = 'Hello there'
print(message)
message = 3.14
print(message)
message = True
print(message)
```

Notice that the variable `message` in the program refers to different types of object as the program proceeds - first a string, then a floating-point number, then a boolean. Because variables can do this we say that Python has *flexible typing*.

You can use variables to define other variables:

```python
message = 'Hello there'
new_message = message
print(new_message)
```

Note that assignment associates objects to variables, not variables to variables. If you change the object a variable is assigned to, it does not affect other variables that were assigned to the original object:

```python
message = 'Hello there'
new_message = message
print(new_message)
message = 'Goodbye'
print(new_message)
```

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

```python
print(message)
```

If you want to introduce a variable without giving it a value then you can assign it the `None` object:

```python
var = None
print(var)
```

You can also assign values to multiple variables at the same time:

▸ Run PYTHON

```python
hello, goodbye = 'Hello', 'Goodbye'
print(hello)
print(goodbye)
```

Naming variables

In Python, a variable name can be of any length and contain a sequence of uppercase letters (`A-Z`), lower case letters (`a-z`), digits (`0-9`) and the underscore (`_`). The first character however, cannot be a digit. As long as you follow these rules you can name your variables whatever you like.

Note that variables names are *case sensitive*. 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):

▸ Run PYTHON

```python
message = 'Hello there'
print(Message)
```

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:

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

```python
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`.

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:

Run PYTHON

```python
var = 1 # var refers to the integer 1
print(var, type(var))

var = float(1) # var refers to the floating point number 1.0
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()`:

Run PYTHON

```python
# number will be a string:
number = input('Enter a number: ')
print(number, type(number))

# number will be an integer:
number = int(input('Enter a number: '))
print(number, type(number))

# number will be a float:
number = float(input('Enter a number: '))
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:

Run PYTHON

```python
var = int('hello')
print(type(var))
```

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

Run PYTHON

```python
var = 2 # var refer to the integer 2, by default
print(var, type(var))

var = float(var) # var now refers to the floating-point number 2.0
print(var, type(var))

var = str(var) # var now refers to the string '2.0'
print(var, type(var))
```

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

Run PYTHON

```python
print(int(1.2))
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:

Run PYTHON

```python
HOURS_PER_DAY = 24
MINUTES_PER_HOUR = 60
num_days = int(input('How many days? '))
num_minutes = num_days * HOURS_PER_DAY * MINUTES_PER_HOUR
print(num_days, 'days is', num_minutes, 'minutes')
```

Note that just with any other variable, it is possible for the program 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.

Working with numbers

Numbers in Python

Python has 3 builtin types for numbers:

- `int` - representing whole numbers (positive and negative integers) with unlimited precision (i.e. there is no *a priori* maximum or minimum value an `int` object can have)
- `float` - representing floating point numbers (non-whole numbers) with a limited precision (i.e. a `float` object is limited to a certain number of significant figures)
- `complex` - representing complex numbers [not covered in this course]

While numeric types are largely interchangeable (for example you can add an `int` to a `float`), a common source of runtime errors is when objects of the wrong type are being used (for example using a `float` when an `int` was expected or vice versa).

Operating on numbers

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

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

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

```python
print(10**2) # 10 to the power 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
print(10//3) # 10 divided by 3 and rounded down
```

Note the effect of using `//` on negative numbers:

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

Order of operations

Python has an inbuilt precedence with its arithmetic operations: `**` will be evaluated before any of `*`, `/`, `//`, `%` and these will be evaluated before any of `+`, `-`. With the exception of `**` operations with the same precedence are evaluated left-to-right. `**` is evaluated right-to-left. Parentheses can be used to change the inbuilt evaluation order.

```python
print(1+2*3) # Same as 1 + (2*3)
print(2**3*4) # Same as (2**3) * 4
print(24/6*2) # Same as (24/6) * 2
print(24/6/2) # Same as (24/6) / 2
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 the variable. Python provides *augmented assignment operators* to allow for more concise code: `+=`, `-=`, , `*=`, `/=`, `//=`, and `%=` .

- `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 /= 2` is equivalent to `x = x / 2`
- `x //= 2` is equivalent to `x = x // 2`
- `x %= 2` is equivalent to `x = x % 2`

```python
x = 12
x += 2
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`

Run PYTHON

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

Floating-point numbers and `==`

Because floating point numbers have limited precision you will experience strange results when attempting to compare them with `==`. For example:

Run PYTHON

```python
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 PYTHON

```python
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.

Run PYTHON

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

Note: the `round()` function implements round half to even:

Run PYTHON

```python
print(round(4.5,0))
print(round(5.5,0))
print(round(-4.5,0))
print(round(-5.5,0))
```

Numeric functions

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

Run PYTHON

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

Working with strings

Strings in Python

Strings are ordered sequences of characters, and a character is essentially anything you can type on the keyboard in one keystroke (technically a character is a unicode character, so it does include many other symbols including foreign letters, mathematical symbols and non-printable text).

As we have seen, you define a string literal with text between either single quotes `'` `'`, or double quotes `"` `"`.

Run PYTHON

```python
literal_one = 'A string'
literal_two = "A string" # These are the same string literals
print(literal_one)
print(literal_two)
```

One of the most useful strings is the empty string - a string with 0 characters. This object is different to the `None` object seen earlier.

Run PYTHON

```python
empty = ''
none = None
print(empty)
print(none)
```

Sometimes you will need to use a string literal that contains quote marks or other special characters that Python will interpret as syntactical elements. You can do this by *escaping* those special characters, by prefixing them with a backslash `\`.

Run PYTHON

```python
# Backslash used to escape a quote mark
print('Penny\'s dog')

# Backslash used to escape a new line symbol
print('This is one line.\nThis is a second line')

# Backslash used to escape a tab symbol
print('Here\tThere')
```

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:

▸ Run PYTHON

```python
print("Penny's dog")
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 PYTHON

```python
# Using \n
text = 'This is one line.\nThis is a second line'
print(text)

# Using triple single quotes
text = '''This is one line.
This is a second line.'''
print(text)

# Using triple double quotes
text = """This is one line.
This is a second line."""
print(text)
```

Operating on strings

You can concatenate strings using the `+` operator:

▸ Run PYTHON

```python
first_name = 'Leo'
last_name = 'Tolstoy'
full_name = first_name + ' ' + last_name
print(full_name)
```

You can also use the `+=` augmented assignment operator:

▸ Run PYTHON

```python
name = 'Leo'
name += ' '
name += 'Tolstoy'
print(name)
```

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

▸ Run PYTHON

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

Comparing strings

Just as with numbers, Python has a number of builtin comparison operators that allow you 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) the value of `str2`, otherwise it is `False`
- Similarly we have `str1 <= str2`, `str1 > str2`, and `str1 >= str2`

```python
print('A' == 'A')
print('A' == "A") # Whether you define literals with ' or " does not mat
```

```python
print('fish' in 'selfishness')
print('fine' in 'selfishness') # Substrings have to be contiguous
```

```python
print('A' < 'B')
print('AA' < 'AB')
print('A' < 'AA')
```

String functions

We have seen the `str()` function to convert a non-string object into a string:

```python
num = 3
message = str(num*num)
print(message)
```

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

```python
print(len('abcde'))
```

String methods

Many more useful methods for modifying strings can be found as attributes of the string object. For example, you have the following string methods available to use:

- `str.capitalize()` - Returns `str` with the first character uppercase and the rest lowercase
- `str.count()` - Returns the number of times a specified value occurs in `str`
- `str.endswith()` - Returns true if `str` ends with the specified value
- `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.format()` - Returns `str` formatted as specified
- `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.islower()` - Returns true if all characters in `str` are lower case
- `str.isspace()` - Returns true if all characters in `str`are whitespace characters (i.e. space, tab, or new line)
- `str.isupper()` - Returns true if all characters in `str` are upper case
- `str.lower()` - Returns `str` converted to lower case
- `str.lstrip()` - Returns `str` with whitespace stripped from the left end
- `str.replace(old, new)` - Returns `str` with `old` replaced by `new`
- `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.rstrip()` - Returns `str` with whitespace stripped from the right end
- `str.split()` - Splits `str` using a specified delimiter and returns the result as a list
- `str.startsswith()` - Returns true if `str` starts with the specified value
- `str.strip()` - Returns `str` with whitespace stripped from both ends
- `str.title()` - Returns `str` with the first character of each word in upper case
- `str.upper()` - Returns `str` with every character in upper case

Working with booleans

Booleans in Python

Boolean objects are the simplest datatype in Python, but often play the most significant role in determining the execution path a program takes (the *program flow*). They can be one of two values: `True` or `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

```python
print(3<5)
print(1+1==3)
```

Things are a little more interesting with variables:

Run PYTHON

```python
guess = int(input('Enter a number: '))
print('Your number is less than 5?')
print(guess<5)
```

Operating on booleans

In many cases a boolean object represents a propositional statement - a sentence that is either true or false (e.g. "3 is less than 5" or "1+1 equals 3"). Just as you can build complex propositions from simpler ones using logical connectives such as "not", "and" or "or", you can build more complex boolean objects by combining smaller boolean objects with the logical operators `not`, `and`, and `or`:

- `not x` is `True` if `x` is `False`, otherwise it is `False`
- `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`

Run PYTHON

```python
print(not True)
print(not False)
print(not 3<5) # It is not the case that 3 is less than 5
```

Run PYTHON

```python
print(True and True)
print(True and False)
print(False and True)
print(False and False)
print((3<5) and (1+1==3)) # Are both these statements True?
```

Run PYTHON

```python
print(True or True)
print(True or False)
print(False or True)
print(False or False)
print((3<5) or (1+1==3)) # Is at least one of these statements True?
```

Run PYTHON

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

If statements

In a simple program Python executes the statements of the program one-by-one, from start to finish. However, you will often want Python to execute a certain statement only if a certain condition is satisfied. For this you can use an `if` statement.

The following program uses an `if` statement:

```python
1 number = int(input('What is your favourite number? '))
2 if number == 42:
3     print('That is my favourite number too!')
4 print('Good bye')
```

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.

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

```python
if True:
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
if True:
	print('Hello') # Indentation using a tab
    print('Hello') # Indentation using 4 spaces
```

```python
if True:
    print('Hello') # Indentation using 4 spaces
     print('Hello') # Indentation using 5 spaces
```

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

Conditions

Any Boolean expression can be used between the `if` and the `:` :

```python
if True:
    print('The condition is true')
if 2 > 1:
    print('The condition is true')
if 2 > 1 and 2 < 3:
    print('The condition is true')
if 1 > 2 or 2 > 1:
    print('The condition is true')
if 'cat' == 'cat':
    print('The condition is true')
```

Being 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
if False:
    x = 1
print(x)
```

Since line 2 is not executed the variable `x` does not get defined, 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:

▸ Run PYTHON

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

Elif clauses

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

▸ Run PYTHON

```python
number = int(input('What is your favourite number? '))
if number == 42:
    print('That is my favourite number too!')
elif number == 21:
    print('That is my second favourite number')
else:
    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:

▸ Run PYTHON

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

Nesting

Inside the block of an if statement we can have other if statements:

Run PYTHON

```python
number = int(input('What is your favourite number? '))
if number > 10:
    print('That is a big number')
    if number > 100:
        print('It is bigger than 100')
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:

PYTHON

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

PYTHON

```python
if is_admin and not admin_expired:
    activate_launch()
elif is_person and has_override:
    activate_launch()
elif is_person and special_override:
    activate_launch()
else:
    call_security()
```

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

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:

Run PYTHON

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

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:

Run PYTHON

```python
n = 1
while n <= 10:
    if n % 2 == 0:
        print(n)
    n = n + 1
print('Finished')
```

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

```python
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")
```

Continuing

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

```python
1 i = 0
2 while i < 10:
3     i += 1
4     if i == 5:
5         continue
6     print(i)
7 print('Finished')
```

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.

Breaking

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

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

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.

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
name = input('What is your name? ')
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
while True:
    name = input('What is your name? ')
    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 'x'. Remember to let the user know that they can do this:

```python
while True:
    name = input('What is your name? (Enter x to stop) ')
    if name == 'x':
        break
    print('Hello', name)
```

Stopping execution with control-c

As well as clicking 'Stop', you can also enter control-c on the keyboard:

```python
while True:
    pass
```

Working with files

So far we have seen programs interact with the user via the `print` function (for outputs) and the `input` function (for inputs). A third way for a program to interact with the external environment is through reading from and writing to files.

To read from or write to a file, we must first open the file.

The Python `open` function returns a file object that represents the file that we have opened. It is this object that allows us to perform operations on the underlying file itself.

Run PYTHON

```python
# Open a file for writing
f = open('myfile', 'w')

# Write to the file
f.write('some text')

# Close the file after we are done
f.close()

# Open the file again for reading
f = open('myfile', 'r')

# read returns all the data in the file
data = f.read()
```

Managing open files

We are expected to close files after opening them. 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.

Python provides the `with` block to make opening and closing a limited resource less error prone. Rewriting the previous example with `with` blocks:

```python
# Open a file for writing
with open('myfile', 'w') as f:
    f.write('some text')

# f is automatically closed when control exits the with block

# Open the file again for reading
with open('myfile', 'r') as f:
    # read returns all the data in the file
    data = f.read()

print(data)
```

`with` blocks always ensure the resource is closed, even if an exception occurs inside it. As soon as the control exits the block, the resource will be closed.

Arguments for the `open` function

The Python documentation for `open` shows that the function has quite a few parameters, here we'll just examine the two most commonly used.

Filename

The first, and only compulsory argument for `open` is the name of the file to be opened. 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.

Mode

The second argument for open is the mode - indicating whether the file is being opened for reading (i.e. input) or writing (i.e. output) and whether the file is text-based, or binary. If we don't specify the mode, Python assumes that the file is text, and we want to open the file for reading.

The available options are:

- `'r'` - Open the file for reading. `open` will throw an exception (covered in Week 3) 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.
- `'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.

These options assume by default that the file being accessed is a text file. This can also be made more explicit with the arguments `'rt'`, `'wt'` and `'at'`. If the file to be accessed is a binary file, the arguments for reading, writing and appending are `'rb'`, `'wb'` and `'ab'` respectively.

Text files vs Binary files

The main difference between a text file and a binary file is in how Python reads from and writes to the file. A text file is a file which the Python interpreter expects to be made up of (mostly) human-readable **characters**, and Python will read/write the information from such files in this manner. A binary file is considered by Python to be a sequence of 0's and 1's, and Python will not assume that the information can be broken up into human-readable characters. Binary files are predominantly used to store data/information that should not be altered arbitrarily - for example, executable files, or data files that should only be changed by a specific program. While it is technically possible to read/write text into a binary file (and vice-versa), it is not good practice to do so.

Storing data in human-readable characters introduces a lot of overhead and redundancy - text files are not an efficient way of storing information. Storing the same amount of information in a binary file can make a noticeable difference, as indicated by the difference in file sizes of an ordinary text file and a zipped version of the same file.

However, from a portability issue, text files are the most practical form of *delivering* data. By having human-readable content, no assumptions need to be made about the client's software. As long as there is agreement on how the information should be parsed (usually through a standard), data presented as a text file can offer the most flexibility. Indeed, **comma separated value files** (csv files) are a very common form of data presentation.

Reading from text files

The `read()` method of a file object will return a `string` (in the case of text file) or a sequence of `bytes` (in the case of a binary file) that contains the entire contents of the file. If the file being accessed is a text file, the `readline()` method allows the program to access content up to (and including) the next newline ('\n') character.

▸ Run PYTHON

```python
# Set up a file for reading
with open('myfile', 'w') as file:
    file.write('Line 1: Some text\n') # Note: you have to include the ne
    file.write('Line 2: Some more text')

# Demonstrate the readline() method
with open('myfile', 'r') as file:
    line_one = file.readline()
    line_two = file.readline()

print(line_two) # Just print the second line of the file
print(line_one.endswith('\n')) # The last character of line_one is the n
print(line_two.endswith('\n')) # line_two does not end in a newline char

```

The `read()` and `readline()` methods are cumulative - until the file is closed the read methods will skip over any content that has been already read using an earlier call to either `read()` or `readline()`. So, for example, if `read()` accesses the entire file, any subsequent `read()` or `readline()` call will simply return the empty string.

▸ Run PYTHON

```python
# Set up a file for reading
with open('myfile', 'w') as file:
    file.write('Line 1: Some text\n')
    file.write('Line 2: Some more text')

# Demonstrate using readline() and read() after a previous read() call
with open('myfile', 'r') as file:
    content = file.read() # this reads the entire contents of the file,
    next_line = file.readline() # this call to readline() will not retur
    more_content = file.read() # and this call to read() will not return

print(next_line == "")
print(more_content == "")
```

There are some more useful approaches for reading (and writing) multiple lines from files that we will see once we have covered lists (Week 2).

Further reading

You might find the following helpful:

- The Python Tutorial at w3schools.com

Greetings

Write a program that asks the user for their name, and then prints "Hello, <name>!"

Example

```
Enter your name: Ed
Hello, Ed!
```

Hint: You'll want to use the `input()` and `print()` functions for this challenge.

▸ Run PYTHON

```python
message = input('Message: ')
print(message)
```

BMI calculator

Write a program that calculates the user's BMI (body mass index). The formula is:

- BMI = weight in kilograms / (height in meters * height in meters)

Example

```
What is your weight in kg? 70
What is your height in m? 1.82
Your BMI is 21.1
```

Odd/even checker

Write a program that asks for an **integer** number and then displays whether the number is odd or even.

Example 1:

```
Enter a number: 1
The number 1 is odd.
```

Example 2:

```
Enter a number: 2
The number 2 is even.
```