# Python Fundamentals 1

Logical Operations and Conditional Statements PF1

## Learning Objectives

1 What is a logical operation and how to use them
2 What is a conditional statement and how to use them

## Introduction

Before we delve into both logical operations and conditional statements, it is important to define what both of these have in common: they allow the flow of a program to be modulated in response to an interaction with data. Conceptually, the directive 'if a is true, do b' is a simple arrangement of words that implies an action only happens, in response to a condition being met.

While both logical operators and conditional statements differ in Python, both have at least this much in common. Witl this in mind, let's dive into each concept, in more detail.

## Logical Operations in Python

Intrinsic to Python is a method of making decisions by comparing values with one another. For example, we can ask questions such as: "is `value_a` equal to `value_b`?" Or "is `value_c` greater than `value_d`?" The answer that is returned in the form of the data type **boolean** (i.e. either `True` or `False`). From a Pythonic perspective, we term these questions **logical operations**; and without them, your block of code would just run from top to bottom without interacting with your data, at all.

Logical operations are useful whenever it is required of a program, to make choices. For example, they allow you to check whether inputted data meets certain criteria, to compare results from calculations, or even to combine several checks into a single decision. Logical operations are the building blocks that conditional statements rely on: once a condition evaluates to `True` or `False`, Python knows whether to execute a particular block of code. In this way, logical operations provide the "questions" that drive the "answers" of decision-making in your programs.

Python has specific syntax (often common punctuation marks) that constitue what we call **logical operators**. Below is list of built-in logical operators in Python:

- `==` – **equality** operator; checks if two values are the same.
- `!=` – **inequality** operator; checks if two values are different.
- `<` – **less-than** operator; checks if the value on the left is smaller than the one on the right.
- `>` – **greater-than** operator; checks if the value on the left is larger than the one on the right.
- `<=` – **less-than-or-equal-to** operator; checks if the value on the left is smaller than or equal to the one on the righ
- `>=` – **greater-than-or-equal-to** operator; checks if the value on the left is larger than or equal to the one on the right.
- `and` – logical **conjunction**; returns `True` only if both conditions are true.
- `or` – logical **disjunction**; returns `True` if at least one condition is true.
- `not` – logical *negation*; inverts the truth value of a condition (True becomes False, and vice versa).
- `~` – **bitwise negation** (complement); flips all the bits of an integer (not the same as logical `not`). We won't be using this in this course, but have included it here for completion. You can read more about it here at Python.org

Let's define the values of two variables (as integers), in the code cell below:

```
# Define variables:
```

We can then use the comparative operator == (also referred to as the **equality operator**) to check whether one value is equal to another.

```
# Equality operator (==)
```

```
# Inequality operator (!=)
```

```
# Less-than operator (<)
```

```
# Greater-than operator (>)
```

```
# Less-than-or-equal operator (<=)
```

```
# Greater-than-or-equal operator (>=)
```

```
# Logical conjunction (and)
```

```
# Logical disjunction (or)
```

```
# Logical negation (not)
```

```
# Have a play around with the logical
```

Run Code

FACT

While we are lumping all of these operators under the heading "logical operators", there is actually a subtle difference between some of them. In Python we technically have both **comparison operators** *and* **logical operators**. In the list above, we have added a line break between the three further 'sub-categories' of operator.

The *first six* operators in the list, constitute **comparison operators** (like ==, <, > and `!=`) which are used to compare tw values and return a *Boolean* result (`True` or `False`).

The *second group of three* operators in the list are termed **logical operators** (`and`, `or`, `not`) and these are used to *combine* or *invert* those Boolean results.

And lastly - on its own at the *bottom of the list*, is the **bitwise complement operator** ~, which looks similar because it also flips values, but it works at the binary level of integers, inverting every bit rather than simply switching `True` and `False`. It is therefore *not* a logical operator in the strictest sense.

For emphasis: while we are superficially categorising all of the above under the heading "logical operators", technicall speaking, they belong to different categories. Common and fundamental to all, these operators are **asking questions** ar working with truth values. It is therefore common and useful to group them together under the umbrella of logical operations when first learning Python.

## What is a conditional statement?

One of the most useful aspects of programming, is the ability to execute an action in response to the presence or absence of a condition. For example, if the time is 2:00, and the sun is in the sky, append 'PM' to give 2:00 PM. If it's dark and the sun is down, append 'PM'. This is a lexical way of expressing a condition, and an action that depends on whether a condition is satisified, or not.

Aside from some of the logical operators we covered earlier, Python provides three syntaxes that allow us to express these conditions, and execute code in response to whether or not they are satisfied. These are `if`, `elif` and `else`. In summary:

- `if` – checks a condition, and runs a block of code only **if*** the condition returns the boolean `True`.
- `elif` – shorthand for 'else if', this lets you test another condition *if* the previous condition in your code returns th boolean `False`.
- `else` – usually the last statement in a block of conditionals, `else` provides a block of code to run when **none** of the previous conditions return the boolean `True`.

**Conditional statement type: `if`**

For the first time in the course so far, we will also see the importance of **indentations** in Python. Let's start with the Python keyword `if` as the first illustrative example. Its syntax specifies that `if` is followed by a condition, and then a colon `:`. If you hit return, you will notice that your cursor will automatically be tabbed inwards by a one-tab or four-space indentation. This indentation defines the **scope** of the conditional statement's executable code. For example, in th example below - if the patient's temperature is above 37.5ºC, the print statement will run. If the condition is not satisfied, the code will not execute:

```
# The 'float' class
is used to always
```

Run Code

If the lines that follow the conditional statement are *not* indented by a single indentation (or four spaces), the code will fall *out* of the scope of the conditional, and will be executed, irrespective of the condition being met.

In the code cell below, the temperature is fixed at 35, which does not meet the condition of our `if` statement; but still, our print statement at the bottom of the code block will run, regardless:

```
temperature = 35.0
```

**Conditional statement type: `elif`**

Following on with this logic in mind, we can look at the `elif` conditional (note this is named as shorthand for 'else if'). These always follow an `if` conditional, and explicitly state that if the condition of the `if` isn't met, then proceed to the next `elif` statement. `elif` follows the same syntactical rules as `if`, in that the `elif` keyword is followed by both a condition and a colon `:`, with newline characters automatically indenting a single tab-space in, to specify the code within its scope. This code is then executed if the condition of the `elif` statement is met.

Let's develop the above example to illustrate this. For ease, we will predefine the temperature in the code, and not use the `input()` function.

```
temperature = 35.5
```

IMPORTANT BOX

Note, when you use return to move the cursor to a new line, when you're inside a conditional statement, it will always return an indented new line (i.e. within the scope of the conditional statement you're already in). In order to specify an `elif` statement, you need to hit the backspace key, to make sure your `elif` conditional statement begins on the *same indentation level* as the if conditional, above it. This way, the Python interpreter knows that the `elif` is not inside the scope of the top `if` statement, and thus will run only if the condition is *not* met, rather than if it *is* met.

The condition of the above `elif` statement means that the print statement will print, only if the temperature is greater than or equal to 35.5 *and* less than 36.0.

We can continue, and add another two `elif` statements to define normal body temperature and severe hypothermia, as follows. The `input` function has been reintroduced, so that you can easily change the temperature, to see which condition you satisfy, and thus, which statement is printed:

```
# Change the
temperature to test
```

Run Code

**Conditional statement type: `else`**

The final type of conditional statement is `else`. This specifies that if all conditions specified in your code are *not* met then - irrespective of the condition - this block of code will always execute. Think of it as a final 'catch-all' statement. So long as there is an `else` statement in your code, something will always execute.

We can build on our example, and add in an `else` statement: this specifies that all values not within the ranges specifie by *any* of the conditional statements in our code, will tell our Python interpreter to execute the `else` statement's code.

```
# Change the
temperature to test
```

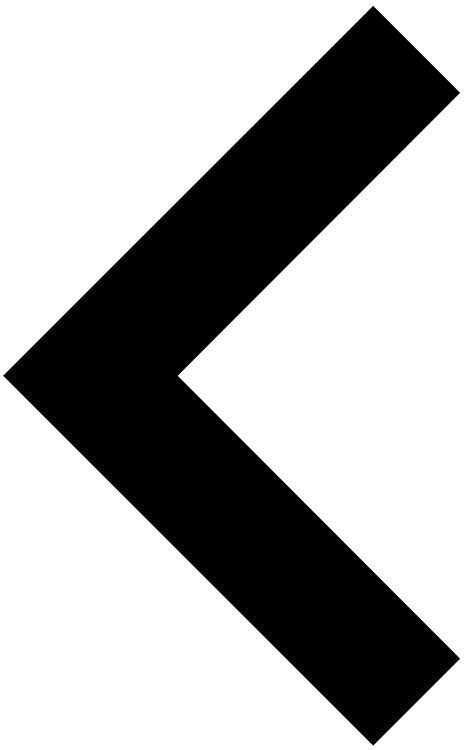
Run Code

```
# Have a go creating
your own if, elif
```

Run Code

## Summary

- **Logical operations** allow comparisons that return `True` or `False`
- **Comparison operators** include ==, `!=`, <, >, <=, >=
- **Logical operators** include `and`, `or`, `not` for combining conditions
- The **bitwise complement** ~ flips all bits of an integer (not the same as `not`)
- **Conditional statements** control program flow based on conditions
- `if` runs a block of code only if a condition is `True`
- `elif` allows checking further conditions if the first is `False`
- `else` runs code when none of the previous conditions are `True`
- **Indentation** and **colons** are essential parts of Python’s conditional syntax
- Conditional logic is useful for modelling real-world decisions (e.g. patient temperatures)



Previous Next