3.1 Propositional Logic

As we get ready to write larger and more complex programs, we're going to take a pause on programming to study formal mathematical logic. You might wonder what logic has to do with software development. As we'll see over the course of this chapter, a firm understanding of logic allows us to precisely identify, define, and write *boolean expressions* and use them in our programs.

It might seem counter-intuitive to spend a whole chapter on logic, as `bool` is the simplest data type in Python. But writing boolean expressions that correctly capture definitions and conditions in a given problem domain can be tricky as these definitions and conditions grow in complexity. It will turn out to be very useful to have a formal mathematical language—logic—to express these complex boolean expressions before turning them into code.

Propositions

We will start our study in this chapter with *propositional logic*, an elementary system of logic that is a crucial building block underlying other, more expressive systems of logic that we will need in this course.

Definition. A **proposition** is a statement that is either True or False. Examples of propositions are:

- $2 + 4 = 6$
- $3 - 5 > 0$
- Every even integer greater than 2 is the sum of two prime numbers.
- Python's implementation of `list.sort` is correct on every input list.

We use **propositional variables** to represent propositions; by convention, propositional variable names are lowercase letters starting at p.[1]

[1] The concept of a propositional variable is different from other forms of variables you have seen before, and even ones that we will see later in this chapter. Here's a rule of thumb: if you read an expression involving a propositional variable p, you should be able to replace p with the statement "CSC110 is cool" and still have the expression make sense.

A **propositional operator** (or **logical operator**) is an operator whose arguments must all be either True or False. Finally, a **propositional formula** is an expression that is built up from propositional variables in combination with propositional operators.

In the following sections, we describe the various operators we will use in this course. It is important to keep in mind when reading that these operators inform both the *syntax* of formulas (what they look like) as well as the *semantics* or *truth value* of these formulas (what they mean: whether the formula is True or False based on the truth values of the individual propositional variables).

The basic operators ***NOT, AND, OR***

We have seen these operators earlier when discussing different types of data. The fact that Python has specific keywords dedicated to these operators should at least hint that they are frequently used. Here, we spend some time introducing the operators more formally and developing our first truth tables.

The unary operator **NOT** (also called "negation") is denoted by the symbol $\neg$. It negates the truth value of its input. So if p is True, then $\neg p$ is False, and vice versa. This is shown in the *truth table* at the side. In Python, we use the `not` keyword to represent this operation.

p	$\neg p$
False	True
True	False

The binary operator **AND** (also called "conjunction") is denoted by the symbol $\wedge$. It returns True when both its arguments are True. In Python, we use the `and` keyword to represent this operation.

p	q	$p \wedge q$
False	False	False
False	True	False
True	False	False
True	True	True

The binary operator **OR** (also called "disjunction") is denoted by the symbol $\vee$, and returns True if one or both of its arguments are True. In Python, we use the `or` keyword to represent this operation.

p	q	$p \vee q$
False	False	False
False	True	True
True	False	True
True	True	True

The truth tables for **AND** and **NOT** agree with the popular English usage of the terms; however, the operator **OR** may seem somewhat different from your intuition, because the word "or" has two different meanings to most English speakers. Consider the English statement "You can have cake or ice cream." From a nutritionist, this might be an *exclusive or*: you can have cake or you can have ice cream, but not both. But from a kindly relative at a family reunion, this might be an *inclusive or*: you can have both cake and ice cream if you want! The study of mathematical logic is meant to eliminate the ambiguity by picking one meaning of **OR** and sticking with it. In our case, we will always use **OR** to mean the *inclusive or*, as illustrated in the last row of its truth table.[2] This is also the behaviour of the `or` operator in Python, which evaluates to `True` when both of its operands are `True`.

[2] The symbol $\oplus$ is often used to represent the *exclusive or* operator, but we will not use it in this course.

AND and **OR** are similar in that they are both *binary* operators on propositional variables. However, the distinction between **AND** and **OR** is very important. Consider for example a rental agreement that reads "first and last months' rent *and* a $1000 deposit" versus a rental agreement that reads "first and last months' rent *or* a $1000 deposit." The second contract is fulfilled with much less money down than the first contract.

The implication operator

One of the most subtle and powerful relationships between two propositions is *implication*, which is represented by the symbol $\Rightarrow$. The implication $p \Rightarrow q$ asserts that whenever p is True, q must also be True. An example of logical implication in English is the statement: "If you push that button, then the fire alarm will go off."[3] Implications are so important that the parts have been given names. The statement p is called the *hypothesis* of the implication and the statement q is called the *conclusion* of the implication.

[3] In some contexts, we think of logical implication as the temporal relationship that q is inevitable if p occurs. But this is *not* always the case! Be careful not to confuse implication with causation.

How should the truth table be defined for $p \Rightarrow q$? First, when both p and q are True, then $p \Rightarrow q$ should be True, since when p occurs, q also occurs. Similarly, it is clear that when p is True and q is False, then $p \Rightarrow q$ is False (since then q is not inevitably True when p is True). But what about the other two cases, when p is False and q is either True or False? This is another case where our intuition from both English language it a little unclear. Perhaps somewhat surprisingly, in both of these remaining cases, we will still define $p \Rightarrow q$ to be True.

p	q	$p \Rightarrow q$
False	False	True
False	True	True
True	False	False
True	True	True

The two cases when p is False but $p \Rightarrow q$ is True are called the **vacuous truth** cases. How do we justify this assignment of truth values? The key intuition is that because the statement doesn't say anything about whether or not q should occur when p is False, it cannot be disproven when p is False. In our example above, if the alarm button is *not* pushed, then the statement is not saying anything about whether or not the fire alarm will go off. It is entirely consistent with this statement that if the button is not pushed, the fire alarm can still go off, or may not go off.

The formula $p \Rightarrow q$ has two logically equivalent[4] formulas which are often useful. To make this concrete, we'll use the example "If you are a Pittsburgh Pens fan, then you are not a Flyers fan".

[4] Here, "logically equivalent" means that the two formulas have the same truth values; for any setting of their propositional variables to True and False, the formulas will either both be True or both be False.

The following two formulas are equivalent to $p \Rightarrow q$:

- $\neg p \vee q$. On our example: "You are not a Pittsburgh Pens fan, or you are not a Flyers fan." This makes use of the vacuous truth cases of implication, in that if p is False then $p \Rightarrow q$ is True, and if p is True then q must be True as well.

- $\neg q \Rightarrow \neg p$. On our example: "If you *are* a Flyers fan, then you are *not* a Pittsburgh Pens fan." Intuitively, this says that if q doesn't occur, then p cannot have occurred either.

  This equivalent formula is in fact so common that we give it a special name: the **contrapositive** of the implication $p \Rightarrow q$.

There is one more related formula that we will discuss before moving on. If we take $p \Rightarrow q$ and switch the hypothesis and conclusion, we obtain the implication $q \Rightarrow p$, which is called the **converse** of the original implication.

Unlike the two formulas in the list above, the converse of an implication is *not* logically equivalent to the original implication. Consider the statement "If you can solve any problem in this course, then you will get an A." Its converse is "If you will get an A, then you can solve any problem in this course." These two statements certainly don't mean the same thing!

In Python, there is no operator or keyword that represents implication directly. When we want to express an implication as a Python expression, we can use the first equivalent form from above, writing $p \Rightarrow q$ as $\neg p \vee q$, or in Python syntax, `not p or q`.

Biconditional ("if and only if")

The final logical operator that we will consider is the *biconditional*, denoted by $p \Leftrightarrow q$. This operator returns True when the implication $p \Rightarrow q$ and its converse $q \Rightarrow p$ are both True.

In other words, $p \Leftrightarrow q$ is an abbreviation for $(p \Rightarrow q) \wedge (q \Rightarrow p)$. A nice way of thinking about the biconditional is that it asserts that its two arguments have the *same* truth value.

p	q	$p \Leftrightarrow q$
False	False	True
False	True	False
True	False	False
True	True	True

While we could use the natural translation of $\Rightarrow$ and $\wedge$ into English to also translate $\Leftrightarrow$, the result is a little clunky: $p \Leftrightarrow q$ becomes "if p then q, and if q then p." Instead, we often shorten this using a quite nice turn of phrase: "p if and only if q," which is abbreviated to "p iff q."

In Python, we don't need a separate operator to represent $\Leftrightarrow$, since we can simply use `==` to determine whether two boolean values are the same!

Summary

We have now seen all five propositional operators that we will use in this course. Now is an excellent time to review these and make sure you understand the notation, meaning, and English words used to indicate each one.

operator	notation	English	Python operation
NOT	$\neg p$	p is not true	`not p`
AND	$p \wedge q$	p and q	`p and q`
OR	$p \vee q$	p or q (or both!)	`p or q`
implication	$p \Rightarrow q$	if p, then q	`not p or q`
biconditional	$p \Leftrightarrow q$	p if and only if q	`p == q`