The following are taken from Chapter 5 of *How to Think Like a Computer Scientist: Learning with Python 3* by Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers. From the Open Book Project.

5. Conditionals

Programs get really interesting when we can test conditions and change the program behaviour depending on the outcome of the tests. That's what this chapter is about.

5.1. Boolean values and expressions

A *Boolean* value is either true or false. It is named after the British mathematician, George Boole, who first formulated *Boolean algebra* — some rules for reasoning about and combining these values. This is the basis of all modern computer logic.

In Python, the two Boolean values are `True` and `False` (the capitalization must be exactly as shown), and the Python type is **bool**.

```
>>> type(True)
<class 'bool'>
>>> type(true)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
NameError: name 'true' is not defined
```

A **Boolean expression** is an expression that evaluates to produce a result which is a Boolean value. For example, the operator `==` tests if two values are equal. It produces (or *yields*) a Boolean value:

```
>>> 5 == (3 + 2)   # Is five equal 5 to the result of 3 + 2?
True
>>> 5 == 6
False
>>> j = "hel"
>>> j + "lo" == "hello"
True
```

In the first statement, the two operands evaluate to equal values, so the expression evaluates to `True`; in the second statement, 5 is not equal to 6, so we get `False`.

The `==` operator is one of six common **comparison operators** which all produce a `bool` result; here are all six:

```
x == y               # Produce True if ... x is equal to y
x != y               # ... x is not equal to y
x > y                # ... x is greater than y
x < y                # ... x is less than y
x >= y               # ... x is greater than or equal to y
x <= y               # ... x is less than or equal to y
```

Although these operations are probably familiar, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (`=`) instead of a double equal sign (`==`). Remember that `=` is an assignment operator and `==` is a comparison operator. Also, there is no such thing as `=<` or `=>`.

Like any other types we've seen so far, Boolean values can be assigned to variables, printed, etc.

```
>>> age = 18
>>> old_enough_to_get_driving_licence = age >= 17
>>> print(old_enough_to_get_driving_licence)
True
>>> type(old_enough_to_get_driving_licence)
<class 'bool'>
```

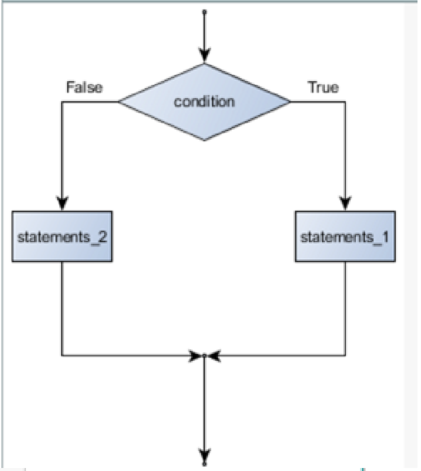
5.5. Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the **if** statement:

```
if x % 2 == 0:
    print(x, " is even.")
    print("Did you know that 2 is the only even number that is prime?")
else:
    print(x, " is odd.")
    print("Did you know that multiplying two odd numbers " +
                                         "always gives an odd result?")
```

The Boolean expression after the `if` statement is called the **condition**. If it is true, then all the indented statements get executed. If not, then all the statements indented under the `else` clause get executed.

Flowchart of an if statement with an else cla



The syntax for an `if` statement looks like this:

```
if BOOLEAN EXPRESSION:
    STATEMENTS_1        # Executed if condition evaluates to True
else:
    STATEMENTS_2        # Executed if condition evaluates to False
```

As with the function definition from the last chapter and other compound statements like `for`, the `if` statement consists of a header line and a body. The header line begins with the keyword `if` followed by a *Boolean expression* and ends with a colon (:).

The indented statements that follow are called a **block**. The first unindented statement marks the end of the block.

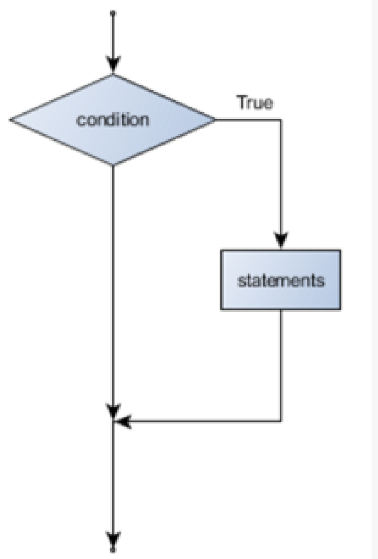
Each of the statements inside the first block of statements are executed in order if the Boolean expression evaluates to `True`. The entire first block of statements is skipped if the Boolean expression evaluates to `False`, and instead all the statements indented under the `else` clause are executed.

There is no limit on the number of statements that can appear under the two clauses of an `if` statement, but there has to be at least one statement in each block. Occasionally, it is useful to have a section with no statements (usually as a place keeper, or scaffolding, for code we haven't written yet). In that case, we can use the `pass` statement, which does nothing except act as a placeholder.

```
if True:          # This is always True,
    pass          #   so this is always executed, but it does nothing
else:
    pass
```

5.6. Omitting the `else` clause

Flowchart of an if statement with no else clause



Another form of the `if` statement is one in which the `else` clause is omitted entirely. In this case, when the condition evaluates to `True`, the statements are executed, otherwise the flow of execution continues to the statement after the `if`.

```python
if x < 0:
    print("The negative number ",  x, " is not valid here.")
    x = 42
    print("I've decided to use the number 42 instead.")

print("The square root of ", x, "is", math.sqrt(x))
```

In this case, the print function that outputs the square root is the one after the `if` — not because we left a blank line, but because of the way the code is indented. Note too that the function call `math.sqrt(x)` will give an error unless we have an `import math` statement, usually placed near the top of our script.

Python terminology

Python documentation sometimes uses the term **suite** of statements to mean what we have called a *block* here. They mean the same thing, and since most other languages and computer scientists use the word *block*, we'll stick with that.

Notice too that `else` is not a statement. The `if` statement has two *clauses*, one of which is the (optional) `else` clause.

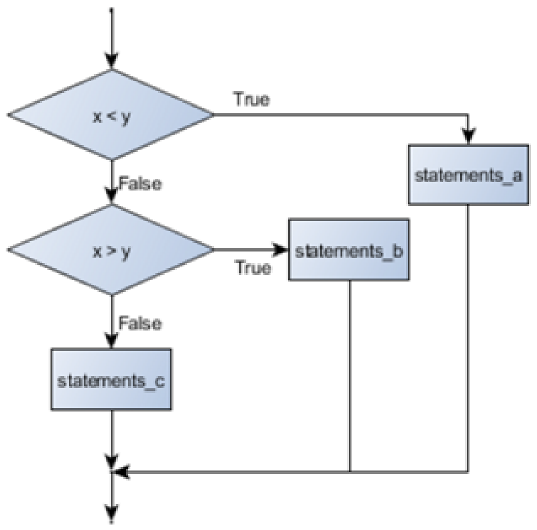
Continues on the next page …

5.7. Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

```python
if x < y:
    STATEMENTS_A
elif x > y:
    STATEMENTS_B
else:
    STATEMENTS_C
```

Flowchart of this chained conditional



`elif` is an abbreviation of `else if`. Again, exactly one branch will be executed. There is no limit of the number of `elif` statements but only a single (and optional) final `else` statement is allowed and it must be the last branch in the statement:

```python
if choice == "a":
    function_one()
elif choice == "b":
    function_two()
elif choice == "c":
    function_three()
else:
    print("Invalid choice.")
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

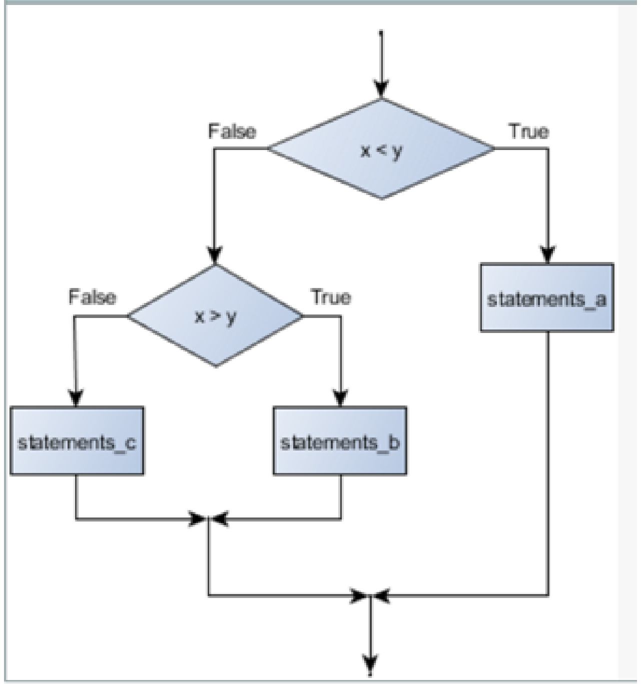
Continues on the next page …

5.8. Nested conditionals

One conditional can also be **nested** within another. (It is the same theme of composibility, again!) We could have written the previous example as follows:

```
if x < y:
    STATEMENTS_A
else:
    if x > y:
        STATEMENTS_B
    else:
        STATEMENTS_C
```

Flowchart of this nested conditional



The outer conditional contains two branches. The second branch contains another `if` statement, which has two branches of its own. Those two branches could contain conditional statements as well.

Although the indentation of the statements makes the structure apparent, nested conditionals very quickly become difficult to read. In general, it is a good idea to avoid them when we can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:            # Assume x is an int here
    if x < 10:
        print("x is a positive single digit.")
```

The `print` function is called only if we make it past both the conditionals, so instead of the above which uses two `if` statements each with a simple condition, we could make a more complex condition using the `and` operator. Now we only need a single `if` statement:

```
if 0 < x and x < 10:
    print("x is a positive single digit.")
```