4. Conditionals and loops

4.1. Conditional execution

4.1.1. The `if` statement

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, which has the genaral form:

```
if BOOLEAN EXPRESSION:
    STATEMENTS
```

A few important things to note about `if` statements:

1. The colon (`:`) is significant and required. It separates the **header** of the **compound statement** from the **body**.
2. The line after the colon must be indented. It is standard in Python to use four spaces for indenting.
3. All lines indented the same amount after the colon will be executed whenever the BOOLEAN_EXPRESSION is true.

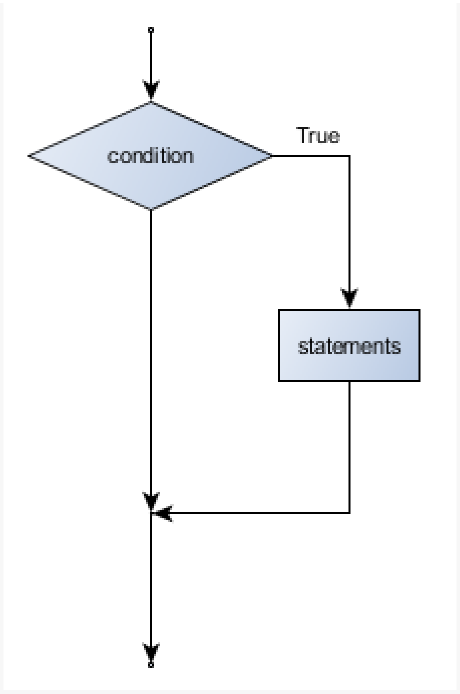
Here is an example:

```
food = 'spam'

if food == 'spam':
    print('Ummmm, my favorite!')
    print('I feel like saying it 100 times...')
    print(100 * (food + '! '))
```

The boolean expression after the `if` statement is called the **condition**. If it is true, then all the indented statements get executed. What happens if the condition is false, and `food` is not equal to `'spam'`? In a simple `if` statement like this, nothing happens, and the program continues on to the next statement.

Flowchart of an if statement



As with the `for` statement from the last chapter, the `if` statement is a **compound statement**. Compound statements consist of a header line and a body. The header line of the `if` statement 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 statement inside the block must have the same indentation.

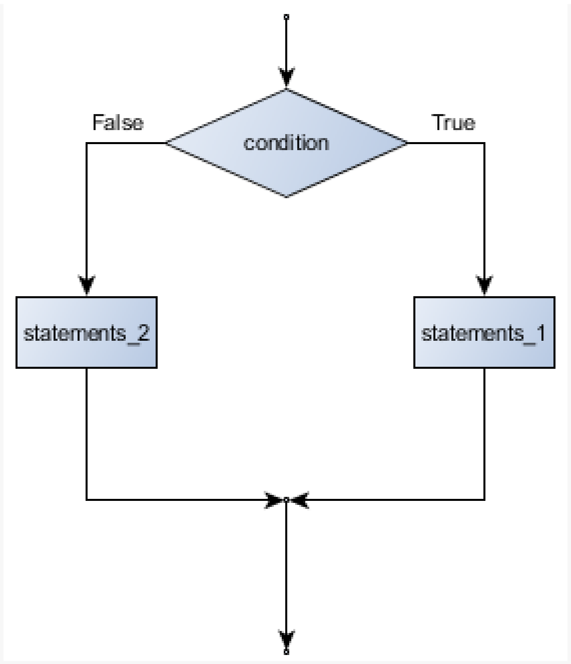
4.1.2. The `if else` statement

It is frequently the case that you want one thing to happen when a condition it true, and **something else** to happen when it is false. For that we have the `if else` statement.

```
if food == 'spam':
    print('Ummmm, my favorite!')
else:
    print("No, I won't have it. I want spam!")
```

Here, the first print statement will execute if `food` is equal to `'spam'`, and the print statement indented under the `else` clause will get executed when it is not.

Flowchart of a if else statement



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

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

Each statement inside the `if` block of an `if else` statement is executed in order if the boolean expression evaluates to `True`. The entire block of statements is skipped if the boolean expression evaluates to `False`, and instead all the statements 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 else` 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 you haven't written yet). In that case, you 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
```

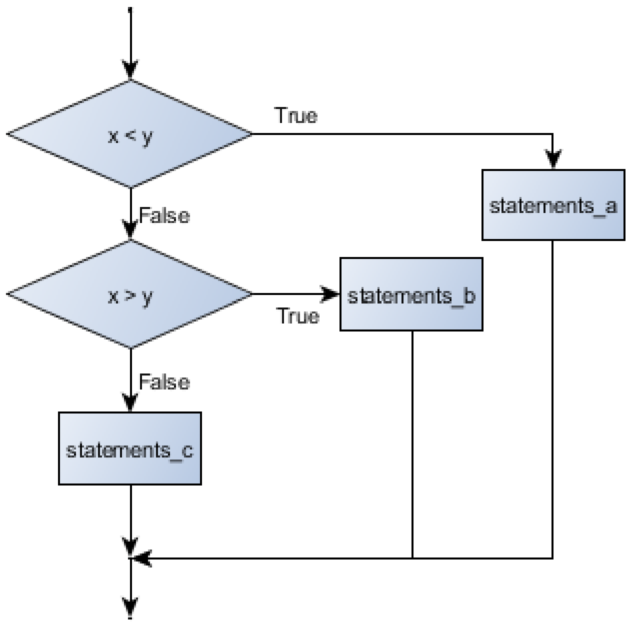
4.2. Chained conditionals

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**:

```
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:

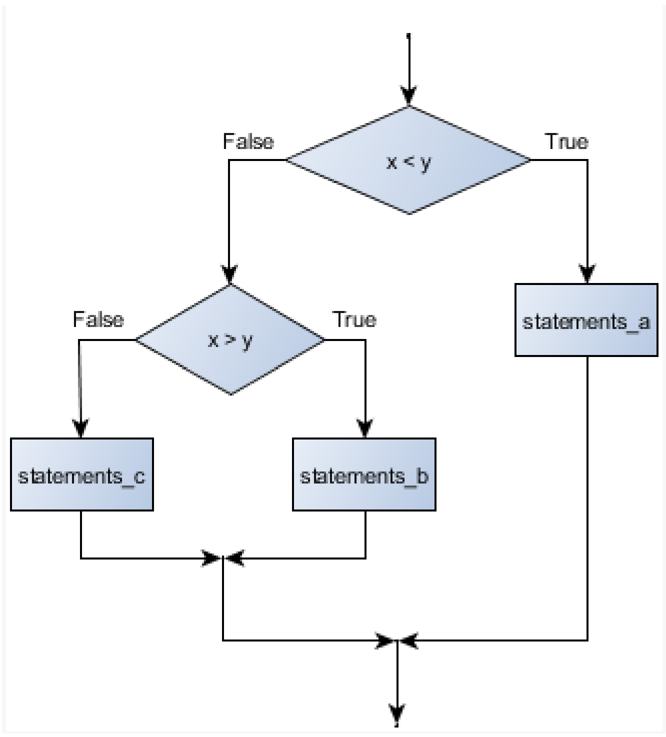
```
if choice == 'a':
    print("You chose 'a'.")
elif choice == 'b':
    print("You chose 'b'.")
elif choice == 'c':
    print("You chose 'c'.")
else:
    print("Invalid choice.")
```

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.

4.3. 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:

Flowchart of this nested conditional



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

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 you can.

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

```python
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 we can use the `and` operator:

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

Note

Python actually allows a short hand form for this, so the following will also work:

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

4.4. Iteration

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

Repeated execution of a set of statements is called iteration. Python has two statements for iteration – the `for` statement, which we met last chapter, and the `while` statement.

4.5. The `for` loop

The `for` loop processes each item in a sequence, so it is used with Python's sequence data types – strings, lists, and tuples.

Each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed.

The general form of a `for` loop is:

```
for LOOP_VARIABLE in SEQUENCE:
    STATEMENTS
```

This is another example of a compound statement in Python, and like the branching statements, it has a header terminated by a colon (`:`) and a body consisting of a sequence of one or more statements indented the same amount from the header.

The loop variable is created when the `for` statement runs, so you do not need to create the variable before then. Each iteration assigns the the loop variable to the next element in the sequence, and then executes the statements in the body. The statement finishes when the last element in the sequence is reached.

This type of flow is called a **loop** because it loops back around to the top after each iteration.

```
for friend in ['Margot', 'Kathryn', 'Prisila']:
    invitation = "Hi " + friend + ".  Please come to my party on Saturday!"
    print(invitation)
```

Running through all the items in a sequence is called **traversing** the sequence, or **traversal**.

You should run this example to see what it does.

Tip

As with all the examples you see in this book, you should try this code out yourself and see what it does. You should also try to anticipate the results before you do, and create your own related examples and try them out as well.

If you get the results you expected, pat yourself on the back and move on. If you don't, try to figure out why. This is the essence of the scientific method, and is essential if you want to think like a computer programmer.

Often times you will want a loop that iterates a given number of times, or that iterates over a given sequence of numbers. The `range` function come in handy for that.

```
>>> for i in range(5):
...     print('i is now:', i)
...
i is now 0
i is now 1
i is now 2
i is now 3
i is now 4
>>>
```

4.7. The `while` statement

The general syntax for the while statement looks like this:

```
while BOOLEAN_EXPRESSION:
    STATEMENTS
```

Like the branching statements and the `for` loop, the `while` statement is a compound statement consisting of a header and a body. A `while` loop executes an unknown number of times, as long at the BOOLEAN EXPRESSION is true.

Here is a simple example:

```
number = 0
prompt = "What is the meaning of life, the universe, and everything? "

while number != "42":
    number =  input(prompt)
```

Notice that if `number` is set to `42` on the first line, the body of the `while` statement will not execute at all.

Here is a more elaborate example program demonstrating the use of the `while` statement

```
name = 'Harrison'
guess = input("So I'm thinking of person's name. Try to guess it: ")
pos = 0

while guess != name and pos < len(name):
    print("Nope, that's not it! Hint: letter ", end='')
    print(pos + 1, "is", name[pos] + ". ", end='')
    guess = input("Guess again: ")
    pos = pos + 1

if pos == len(name) and name != guess:
    print("Too bad, you couldn't get it.  The name was", name + ".")
else:
    print("\nGreat, you got it in", pos + 1,  "guesses!")
```

The flow of execution for a `while` statement works like this:

1. Evaluate the condition (`BOOLEAN EXPRESSION`), yielding `False` or `True`.
2. If the condition is false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, execute each of the `STATEMENTS` in the body and then go back to step 1.

The body consists of all of the statements below the header with the same indentation.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**.

An endless source of amusement for computer programmers is the observation that the directions on shampoo, lather, rinse, repeat, are an infinite loop.

In the case here, we can prove that the loop terminates because we know that the value of `len(name)` is finite, and we can see that the value of `pos` increments each time through the loop, so eventually it will have to equal `len(name)`. In other cases, it is not so easy to tell.

What you will notice here is that the `while` loop is more work for you — the programmer — than the equivalent `for` loop. When using a `while` loop one has to control the loop variable yourself: give it an initial value, test for completion, and then make sure you change something in the body so that the loop terminates.

4.13. The `continue` statement

This is a control flow statement that causes the program to immediately skip the processing of the rest of the body of the loop, *for the current iteration*. But the loop still carries on running for its remaining iterations:

```
for i in [12, 16, 17, 24, 29, 30]:
    if i % 2 == 1:      # if the number is odd
        continue        # don't process it
    print(i)
print("done")
```

This prints:

```
12
16
24
30
done
```