Else Clauses on Loop Statements

Python's loop statements have a feature that some people love (Hi!), some people hate, many have never encountered and many just find confusing: an `else` clause.

This article endeavours to explain some of the reasons behind the frequent confusion, and explore some other ways of thinking about the problem that give a better idea of what is *really* going on with these clauses.

Reasons for Confusion

The major reason many developers find the behaviour of these clauses potentially confusing is shown in the following example:

```
>>> if [1]:
...     print("Then")
... else:
...     print("Else")
...
Then
>>> for x in [1]:
...     print("Then")
... else:
...     print("Else")
...
Then
Else
```

The `if <iterable>` header looks very similar to the `for <var> in <iterable>` header, so it's quite natural for people to assume they're related and expect the `else` clause to be skipped in both cases. As the example shows, this assumption is incorrect: in the second case, the else clauses triggers even though the iterable isn't empty.

If we then look at a common `while` loop pattern instead, it just deepens the confusion because *it* seems to line up with the way we would expect the conditional to work:

```
>>> x = [1]
>>> while x:
...     print("Then")
...     x.pop()
... else:
...     print("Else")
...
Then
Else
>>> if x:
...     print("Then")
... else:
...     print("Else")
...
Else
```

Here, the loop runs until the iterable is empty, and then the `else` clause is executed, just as it is in the `if` statement.

A different kind of `else`

So what's going on? The truth is that the superficial similarity between `if <iterable>` and `for <var> in <iterable>` is rather deceptive. If we call the `else` clause on an `if` statement a "conditional else", then we can look to `try` statements for a different *kind* of `else` clause, a "completion clause":

```
>>> try:
...     pass
... except:
...     print("Then") # The try block threw an exception
... else:
...     print("Else") # The try block didn't throw an exception
...
Else
```

With a completion clause, the question being asked has to do with how an earlier suite of code *finished*, rather than checking the boolean value of an expression. Reaching the `else` clause in a `try` statement means that the try block actually completed successfully - it didn't throw an exception or otherwise terminate before reaching the end of the suite.

This is actually a much better model for what's going on in our `for` loop, since the condition the `else` is checking for is whether or not the loop was explicitly terminated by a `break` statement. While it's not legal syntax, it may be helpful to mentally insert an `except break: pass` whenever you encounter a loop with an associated `else` clause in order to help remember what it means:

```
for x in iterable:
    ...
except break:
    pass # Implied by Python's loop semantics
else:
    ...  # No break statement was encountered

while condition:
    ...
except break:
    pass # Implied by Python's loop semantics
else:
    ...  # No break statement was encountered
```

What possible use is the current behaviour?

The main use case for this behaviour is to implement search loops, where you're performing a search for an item that meets a particular condition, and need to perform additional processing or raise an informative error if no acceptable value is found:

```
for x in data:
    if acceptable(x):
        break
else:
    raise ValueError("No acceptable value in {!r:100}".format(data))

... # Continue calculations with x
```

But how do I check if my loop never ran at all?

The easiest way to check if a `for` loop never executed is to use `None` as a sentinel value:

```
x = None
for x in data:
    ... # process x
if x is None:
    raise ValueError("Empty data iterable: {!r:100}".format(data))
```

If `None` is a legitimate data value, then a custom sentinel object can be used instead:

```
x = _empty = object()
for x in data:
    ... # process x
if x is _empty:
    raise ValueError("Empty data iterable: {!r:100}".format(data))
```

For `while` loops, the appropriate solution will depend on the details of the loop.

But couldn't Python be different?

Backwards compatibility constraints and the general desire not to change the language core without a compelling justification mean that the answer to this question is likely always going to be "No".

The simplest approach for any new language to take to avoid the confusion encountered in relation to this feature of Python would be to just leave it out altogether. Many (most?) other languages don't offer it, and there are certainly other ways to handle the search loop use case, including a sentinel based approach similar to that used to detect whether or not a loop ran at all:

```
result = _not_found = object()
for x in data:
    if acceptable(x):
        result = x
        break
if result is _not_found:
    raise ValueError("No acceptable value in {!r:100}".format(data))

... # Continue calculations with result
```

Closing note: Not so different after all?

Attentive readers may have noticed that the behaviour of `while` loops still makes sense regardless of whether you think of their `else` clause as a conditional else or as a completion clause. We can think of a `while` statement in terms of an infinite loop containing a `break` statement:

```
while True:
    if condition:
        pass # Implied by Python's loop semantics
    else:
        ... # While loop else clause runs here
        break
    ... # While loop body runs here
```

If you dig deep enough, it's also possible to relate the completion clause constructs in `try` statements and `for` loops back to the basic conditional else construct. The thing to remember though, is that it is only `while` loops and `if` statements that are checking the boolean value of an expression, while `for` loops and `try` statements are checking whether or not a section of code was aborted before completing normally.

However, digging to that deeper level doesn't really provide much more enlightenment when it comes to understanding how the two different forms of `else` clause work in practice.

Daniel McCoy • 10 months ago

You could also use 'else' to only check if the loop ran at all when it didn't encounter a 'break', since, obviously, if it encountered a 'break' it wasn't empty.
(Possibly a very slight optimization.)

```
x = None
for x in data:
    ... # process x
else:
     if x is None:
            raise ValueError("Empty data iterable: {!r:100}".format(data))
     else:
           ... # x is not empty, and we ran off the end of it with no break
```

2 • Reply • Share ›

shadow_0359 ➚ Daniel McCoy • 5 months ago

What if last value of a iterable is None??

• Reply • Share ›

Nick Coghlan Mod ➚ shadow_0359 • 5 months ago

If the iterable may contain None as a value, then you can use a custom sentinel:

```
x = no_data = object()
```

• Reply • Share ›

shadow_0359 ➚ Nick Coghlan • 2 months ago

rgt...tnx!!Very well explained..

• Reply • Share ›

Terrence Brannon • 2 months ago

You wrote """Here, the loop runs until the iterable is empty, and then the else clause is executed, just as it is in the if statement.""" - but that is not how the if-else works. Only one of the branches executes in the if-else.

• Reply • Share ›

math2001 • 6 months ago

Thanks nick! Now it's clear!

Thanks Nick. Now it's clear!

• Reply • Share ›

ALSO ON **NICK COGHLAN'S PYTHON NOTES**

Python Language Summit

4 comments • 4 years ago•

sfermigier — Thanks for the writeup, Nick.

Python 3 and ASCII Compatible Binary Protocols

36 comments • 4 years ago•

David Lukeš — I can safely say that Python 3 is single-handedly responsible for me grokking how character encodings work, the

My Current Views on Python Packaging

6 comments • 4 years ago•

Flimm — Thank you for writing this, and for the careful work you're doing to make our lives more enjoyable.

Using the Python Kerberos Module

9 comments • 4 years ago•

Nick Coghlan — A key point to note: some time after I wrote this, support for Kerberos authentication was added directly to …