PEP 308: Conditional Statements in Python

Pradyumna Kikkeri, Madison Rockwell, Aaron Holt

CSCI 3155 Fall 2014

Python is a multipurpose high level programming language with a design that emphasizes code readability. Python syntax is built to express code found in C++ or Java using fewer lines. As with many languages, Python continues to add features and fix bugs in older versions in order to improve the development experience. Development in Python is done primarily using the Python Enhancement Proposal (PEP) process. In any given PEP, various changes to the language are proposed by the community or by Guido van Rossum (it should be noted that Python is Guido van Rossum's language, which means he gets the final decision on whether or not to implement a PEP). These changes are discussed and reviewed before they can become a part of the Python language. One PEP that made it into the language - PEP 308 - was a feature designed to give Python a new ternary operator. PEP 308 came about after many developers in the Python community complained about confusing if/else statements and expressed a desire for a shorter, cleaner method.

In Python version 2.4 (before PEP 308), conditional events were often handled using if/else statements. This led to lots of code like the following[4]:

```
c and x or y   # incorrectly returns y if x is (any kind of) false

(c and [x] or [y])[0]  # reliable, but ugly and churns objects

(x, y)[not c]  # always evaluates both

(y, x)[c]      # only if c is really a bool (or otherwise 0 or 1)
```

While fairly straightforward, this example of an if/else statement is far more verbose than a ternary operator. Having to use this syntax a few times in a file is fine; however, the code becomes difficult to read and understand when many conditional statements or nested conditional statements are needed. Languages such as C++ and Java use the ?: ternary operator to simplify conditional expressions[1][2]. An example syntax in Java is as follows:

```
variable x = (condition) ? value if true : value if false
```

Comparing the two examples above shows that a ternary implementation of the same code is far less verbose than an if/else statement. However, both examples are functionally equivalent.

If the condition is true, then the variable takes on the "value if true". If the condition is false, then the variable takes on the "value if false". Furthermore,

both examples are short-circuited; that is to say, only one of the paths is evaluated. For example, if the condition is true, the "value if false" statement will not be evaluated. It should also be noted that the advantage of a ternary operator over an if/else statement is almost entirely in the readability of code. There is usually not a discernible performance difference[3]. Aside from if/else statements, there was a requirement to create conditional expression using tuples and lists, but it had significant disadvantages.

The following is an example of a conditional expression in Python[5]:

```
sColorName = ("Black","White")[Color == COLOR_WHITE]
```

In this example, the "value if true" is "Black", the "value if false" is white, and the condition is "[Color == COLOR_WHITE]". If the condition evaluates to true, the entire expression simplifies to:

```
sColorName = ("Black","White")[1]
```

which then leads to:

```
sColorName = "White"
```

While this works for simple examples, it comes with drawbacks. The first drawback to this implementation is that it involves unpacking the tuple, which has performance implications. The second drawback is that all of the possible values must be valid, which prevents short circuiting. This can cause problems like the one in the code example below[5]:

```
r = (DEFAULT_RESULT, Database.Results[index])[IsValid(index)]
```

If the "index" is not valid in the above example, the code will fail. The lack of short circuiting means that both the "DEFAULT_RESULT" and the "Database.Results[index]" must be valid. Thus, PEP 308 was proposed in order to simplify code and implement C++-like conditional expressions, but in a flexible and "Pythonesque" way[5].

The initial proposal for PEP 308 was rather open-ended because Guido van Rossum was "neither in favor nor against" the proposal[6]. However he still authored the original PEP and proposed the following syntax:

```
<expression1> if <condition> else <expression2>
```

It should be noted that in this example, `<condition>` is evaluated first! If `<condition>` evaluates to true, then `<expression1>` is evaluated and is the result of the whole example. If `<condition>` evaluates to false, then `<expression2>` is evaluated and is the result of the whole example. Guido van Rossum proposed this syntax for two main reasons. First, only one expression is ever evaluated. A short circuited expression keeps the coding simple and has performance benefits[6]. On a more opinionated note, Guido van Rossum liked that it didn't include any new key words and would be relatively simple to implement. The expression was slightly confusing to some as the conditional statement was in the middle, which meant it can't be read from left to right.

Following the initial proposal, the community split into many factions based on what they thought would be the best ternary operator. Perhaps the most popular, and also first rejected, was to implement the ternary operator exactly like it is in C++ and Java.

```
<condition> ? <expression1> : <expression2>
```

This evaluates the `<condition>` first. If the condition is true, then `<expression1>` is evaluated and becomes the result. Otherwise, `<expression2>` becomes is evaluated and becomes the result. Many of the developers from a C/C++ background voted for this implementation based on familiarity. Other advantages to the C++ ternary operator include being able to read from left to right and the gains of short circuit evaluation. Guido von Rossum rejected this for several reasons: for people not familiar with C/C++, this operator can be difficult to understand. Also, the colon already had multiple uses in Python, and he didn't want to make the colon more confusing. For similar reasons, the following proposed syntax were also vetoed[7]:

```
<condition> ? <expression1> ! <expression2>
<condition> ? <expression1> , <expression2>
<condition> ? <expression1> else <expression2>
```

Many possible options were eliminated based on a lack of short circuiting. The following syntax is one example:

```
ifelse(<condition>,<expression1>,<expression2>)
```

Like many others, it read left to right and was simple to understand. The main advantage here is that it didn't require a new Python built-in, making it easier to add to the Python language. Unfortunately, making it a function meant that both `<expression1>` and `<expression2>` would need to be evaluated and substituted before the function was called. This makes it impossible to short circuit and in certain cases can lead to validity issues as discussed earlier[8]. One possible workaround to this problem involved using the same syntax, but

creating a special form that would evaluate the expressions depending on the value of the first condition. While functional, this idea simply didn't have enough people supporting it and was thus discarded.

The next seriously considered option came in the form of

```
if <condition> then <expression1> else <expression2>
```

It had most of the main components that developers were looking for: left to right evaluation, short circuiting, and was easy to work through. The main problem with this form was the creation of a new keyword "then". Guido van Rossum was hesitant to create a new keyword with an arbitrary meaning for a PEP that he was uncertain about in the first place. Some developers also thought that it could be difficult to add because the parser would believe that it's an 'if' statement. The following example represents such a case.[6]

```
if verbose then sys.stdout.write("hello") else None
```

In order to prevent a parser error, parentheses would be necessary. This led to the final contender:

```
(if <condition>: <expression1> else: <expression2>)
```

Unlike the previous case, this form doesn't require a new keyword and prevents any sort of parser difficulty by adding parentheses around the expression. Beyond that, it follows standard Python syntax, has left to right evaluation, and is short circuited. The drawbacks are minor: the parentheses make it look a little odd and the colon is typically at the end of a line.

After much deliberation, Guido van Rossum called for a vote with sixteen possible choices to vote on. The winner would be the option with a clear majority. The vote was uneventful and no option gained a clear majority. Thus, Guido picked the original form he proposed stating that it was still his language after all.

```
<expression1> if <condition> else <expression2>
```

The new ternary operator was given a priority lower than that of 'or' but higher than that of 'lambda'[6]. The following code examples explain how it works.

```
<expression1> if <condition1> else <expression2> if <condition2> else <expression3>
```

can be read as

```
<expression1> if <condition1> else (<expression2> if <condition2> else <expression3>)
```

where the code inside the parentheses is evaluated first[9]. It should be noted that code like this is slightly frowned upon as it makes the code difficult to read, which negates the one of the main points of the ternary operator.

```
<expression1> or <expression2> if <condition2> else <expression3>
```

can be read as

“‘python (or) if else Again, the ternary operator has lower precedence than the ‘or’ operator.

```
lambda: <expression1> if <condition2> else <expression2>
```

can be interpreted as

```
lambda: (<expression1> if <condition2> else <expression2>)
```

This makes sense as the ternary operator has a higher order of precedence than that of lambda.

The new syntax nearly created minor backwards incompatibility with Python versions before Python 2.5. In versions 2.4 and earlier, note the following operation:

```
[f for f in lambda x: x, lambda x: x**2 if f(1) == 1]
```

which is effectively a list comprehension where a sequence following ‘in’ is a series of lambdas without parentheses. For this reason, Python version 3.0 requires a series of lambdas to be parenthesized[9]. For example:

```
[f for f in (lambda x: x, lambda x: x**2) if f(1) == 1]
```

This is due to the fact that lambda binds less tightly than the if-else expression, but could already be followed by an ‘if’ keyword (which binds even less tightly). On the other hand, Python 2.5 uses a slightly different grammar that prevents lambda from being used in this position by forbidding the lambda body from containing a conditional expression without parentheses[9]. Several examples are as follows:

```
[f for f in (1, lambda x: x if x >= 0 else -1)]    # OK
[f for f in 1, (lambda x: x if x >= 0 else -1)]    # OK
[f for f in 1, lambda x: (x if x >= 0 else -1)]    # OK
[f for f in 1, lambda x: x if x if x>= 0 else -1]  #Invalid
```

The addition of a ternary operator to Python came from community desire and error-prone attempts to achieve the same effect using 'or' and 'and'. A ternary operator can also clean up simple if/else statements by condensing them to one line. After debating what syntax the ternary operator should take, there was a general consensus that it should read left to right, include short circuiting, be 'Pythonesque', and not include any new key words. Many options were presented by various Python developers, but no one option was clearly better than the others. After an indecisive vote, Guido decided to implement his personal choice, which had the above characteristics aside from not reading left to right. In the end, the addition of a ternary operator to the Python language was well received by the community and allowed for cleaner, simpler code.

Bibliography

Citations in the above paper are denoted by numbers with square brackets around them, e.g. [4]

1. Ternary Operators in C++
2. Ternary Operators in Java
3. Ternary operators vs. if/else performance
4. Advantages of ternary operators in Python
5. Ways of implementing ternary ops in Python before PEP 308
6. Initial PEP 308 proposal
7. Voting options proposed for this change
8. Vetoing of "ifelse" function
9. Official PEP 308 Documentation