Conditionals All Versions

Conditional expressions, involving keywords such as if, elif, and else, provide Python programs with the ability to perform different actions depending on a boolean condition: True or False. This section covers the use of Python conditionals, boolean logic, and ternary statements.

Examples

Conditional Expression (or "The Ternary Operator")

The ternary operator is used for inline conditional expressions. It is best used in simple, concise operations that are easily read.

- The order of the arguments is different from many other languages (such as C, Ruby, Java, etc.), which may lead to bugs when people unfamiliar with Python's "surprising" behaviour use it (they may reverse the order).
- Some find it "unwieldy", since it goes contrary to the normal flow of thought (thinking of the condition first and then the effects).

```
n = 5

"Greater than 2" if n > 2 else "Smaller than or equal to 2"
# Out: 'Greater than 2'
```

The result of this expression will be as it is read in English - if the conditional expression is True, then it will evaluate to the expression on the left side, otherwise, the right side.

Tenary operations can also be nested, as here:

```
n = 5
"Hello" if n > 10 else "Goodbye" if n > 5 else "Good day"
```

They also provide a method of including conditionals in lambda functions .

if, elif, and else

In Python you can define a series of conditionals using `if` for the first one, `elif` for the rest, up until the final (optional) `else` for anything not caught by the other conditionals.

```
number = 5

if number > 2:
    print("Number is bigger than 2.")
elif number < 2:  # Optional clause (you can have multiple elifs)
    print("Number is smaller than 2.")
else:  # Optional clause (you can only have one else)
    print("Number is 2.")
```

Outputs `Number is bigger than 2`

Using `else if` instead of `elif` will trigger a syntax error and is not allowed.

Truth Values

The following values are considered falsey, in that they evaluate to `False` when applied to a boolean operator.

- None
- False
- `0` , or any numerical value equivalent to zero, for example `0L` , `0.0` , `0j`
- Empty sequences: `''` , `""` , `()` , `[]`
- Empty mappings: `{}`
- User-defined types where the `__bool__` or `__len__` methods return 0 or `False`

All other values in Python evaluate to `True` .

Boolean Logic Expressions

Boolean logic expressions, in addition to evaluating to `True` or `False`, return the *value* that was interpreted as `True` or `False`. It is Pythonic way to represent logic that might otherwise require an if-else test.

And operator

The `and` operator evaluates all expressions and returns the last expression if all expressions evaluate to `True`. Otherwise it returns the first value that evaluates to `False`:

```
>>> 1 and 2
2

>>> 1 and 0
0

>>> 1 and "Hello World"
"Hello World"

>>> "" and "Pancakes"
""
```

Or operator

The `or` operator evaluates the expressions left to right and returns the first value that evaluates to `True` or the last value (if none are `True`).

```
>>> 1 or 2
1

>>> None or 1
1

>>> 0 or []
[]
```

Lazy evaluation

When you use this approach, remember that the evaluation is lazy. Expressions that are not required to be evaluated to determine the result are not evaluated. For example:

```
>>> def print_me():
        print('I am here!')
>>> 0 and print_me()
0
```

In the above example, `print_me` is never executed because Python can determine the entire expression is `False` when it encounters the `0` (`False`). Keep this in mind if `print_me` needs to execute to serve your program logic.

Testing for multiple conditions

A common mistake when checking for multiple conditions is to apply the logic incorrectly.

This example is trying to check if two variables are each greater than 2. The statement is evaluated as - `if (a) and (b > 2)`. This produces an unexpected result because `bool(a)` evaluates as `True` when `a` is not zero.

```
>>> a = 1
>>> b = 6
>>> if a and b > 2:
...     print('yes')
... else:
...     print('no')

yes
```

Each variable needs to be compared separately.

```
>>> if a > 2 and b > 2:
...     print('yes')
... else:
...     print('no')

no
```

Another, similar, mistake is made when checking if a variable is one of multiple values. The statement in this example is evaluated as - `if (a == 3) or (4) or (6)`. This produces an unexpected result because `bool(4)` and `bool(6)` each evaluate to `True`

```
>>> a = 1
>>> if a == 3 or 4 or 6:
...     print('yes')
... else:
```

```
...     print('no')

yes
```

Again each comparison must be made separately

```
>>> if a == 3 or a == 4 or a == 6:
...     print('yes')
... else:
...     print('no')

no
```

Using the in operator is the canonical way to write this.

```
>>> if a in (3, 4, 6):
...     print('yes')
... else:
...     print('no')

no
```

Else statement

```
if condition:
    body
else:
    body
```

The else statement will execute it's body only if preceding conditional statements all evaluate to False.

```
if True:
    print "It is true!"
else:
    print "This won't get printed.."

# Output: It is true!

if False:
    print "This won't get printed.."
else:
    print "It is false!"

# Output: It is false!
```

Using the cmp function to get the comparison result of two objects

Python 2 includes a `cmp` function which allows you to determine if one object is less than, equal to, or greater than another object. This function can be used to pick a choice out of a list based on one of those three options.

Suppose you need to print `'greater than'` if `x > y` , `'less than'` if `x < y` and `'equal'` if `x == y` .

```
['equal', 'greater than', 'less than', ][cmp(x,y)]

# x,y = 1,1 output: 'equal'
# x,y = 1,2 output: 'less than'
# x,y = 2,1 output: 'greater than'
```

`cmp(x,y)` returns the following values

Comparison	Result
x < y	-1
x == y	0
x > y	1

This function is removed on Python 3. You can use the `cmp_to_key(func)` helper function located in `functools` in Python 3 to convert old comparison functions to key functions.

If statement

```
if condition:
    body
```

The `if` statements checks the condition. If it evaluates to `True`, it executes the body of the `if` statement. If it evaluates to `False`, it skips the body.

```
if True:
    print "It is true!"
>> It is true!

if False:
    print "This won't get printed.."
```

The condition can be any valid expression:

```
if 2 + 2 == 4:
    print "I know math!"
>> I know math!
```

Testing if an object is None and assigning it

You'll often want to assign something to an object if it is `None`, indicating it has not been assigned. We'll use `aDate`.

The simplest way to do this is to use the `is None` test.

```
if aDate is None:
    aDate=datetime.date.today()
```

(Note that it is more Pythonic to say `is None` instead of `== None`.)

But this can be optimized slightly by exploiting the notion that `not None` will evaluate to `True` in a boolean expression. The following code is equivalent:

```
if not aDate:
    aDate=datetime.date.today()
```

But there is a more Pythonic way. The following code is also equivalent:

```
aDate=aDate or datetime.date.today()
```

This does a Short Circuit evaluation. If `aDate` is initialized and is `not None`, then it gets assigned to itself with no net effect. If it `is None`, then the `datetime.date.today()` gets assigned to `aDate`.

Conditional Expression Evaluation Using List Comprehensions

Python allows you to hack list comprehensions to evaluate conditional expressions.

For instance,

```
[value_false, value_true][<conditional-test>]
```

Example:

```
>> n = 16
>> print [10, 20][n <= 15]
10
```

Here `n<=15` returns `False` (which equates to 0 in Python). So what Python is evaluating is:

```
[10, 20][n <= 15]
==> [10, 20][False]
==> [10, 20][0]     #False==0, True==1 (Check Boolean Equivalencies in Python)
==> 10
```

Python 2.x ≤2.7

The inbuilt `__cmp__` method returned 3 possible values: 0, 1, -1, where cmp(x,y) returned 0: if both objecs were the same 1: x > y -1: x < y

This could be used with list comprehensions to return the first(ie. index 0), second(ie. index 1) and last(ie. index -1) element of the list. Giving us a conditional of this type:

last(ie. index -1) element of the list. Giving us a conditional of this type:

```
[value_equals, value_greater, value_less][<conditional-test>]
```

Finally, in all the examples above Python evaluates both branches before choosing one. To only evaluate the chosen branch:

```
[lambda: value_false, lambda: value_true][<test>]()
```

where adding the `()` at the end ensures that the lambda functions are only called/evaluated at the end. Thus, we only evaluate the chosen branch.

Example:

```
count = [lambda:0, lambda:N+1][count==N]()
```

Syntax

```
<expression> if <conditional> else <expression> # Ternary Operator
```

Parameters

Remarks