Does Python have a ternary conditional operator?

Ask Question

5196

817

If Python does not have a ternary conditional operator, is it possible to simulate one using other language constructs?

python operators ternary-operator conditional-operator python-2.5

edited Mar 23 at 14:24

community wiki
18 revs, 14 users 43%
Devoted

2 Though Pythons older than 2.5 are slowly drifting to history, here is a list of old pre-2.5 ternary operator tricks: "Python Idioms", search for the text 'Conditional

ジョージ May 26 '11 at 0:48

103 In the Python 3.0 official documentation referenced in a comment above, this is referred to as "conditional_expressions" and is very cryptically defined. That documentation doesn't even include the term "ternary", so you would be hard-pressed to find it via Google unless you knew exactly what to look for. The [version 2 documentation](#) is somewhat more helpful and includes a link to ["PEP 308"](#), which includes a lot of interesting historical context related to this question. – nobar Jan 10 '13 at 5:57

13 "ternary" (having three inputs) is a consequential property of this impelmentation, not a defining property of the concept. eg: SQL has `case [...] { when ... then ...} [ else ... ] end` for a similar effect

Dec 15 '14 at 21:14

6 also ISO/IEC 9899 (the C programming language standard) section 6.5.15 calls it the "the condtitional operator" – user313114 Dec 15 '14 at 21:20

4 Wikipedia covers this thoroughly in the article "?:". – HelloGoodbye Jun 9 '16 at 8:11

21 Answers

6147 ✓

Yes, it was added in version 2.5. The expression syntax is:

```
a if condition e
```

First `condition` is evaluated, then exactly one of either `a` or `b` is evaluated and returned based on the Boolean value of `condition`. If `condition` evaluates to `True`, then `a` is evaluated and returned but `b` is ignored, or else when `b` is evaluated and returned but `a` is ignored.

is true only `a` is evaluated and `b` is not evaluated at all, but when `condition` is false only `b` is evaluated and `a` is not evaluated at all.

For example:

```
>>> 'true' if Tr
'true'
>>> 'true' if Fa
'false'
```

Note that conditionals are an *expression*, not a *statement*. This means you can't use assignment statements or `pass` or other **statements** within a conditional **expression**:

```
>>> pass if Fals
  File "<stdin>"
    pass if Fals
          ^
SyntaxError: inv
```

You can, however, use conditional expressions to assign a variable like so:

```
x = a if True el
```

Think of the conditional expression as switching between two values. It is very useful when you're in a 'one value or another' situation, it but

If you need to use statements, you have to use a normal `if` **statement** instead of a conditional **expression**.

Keep in mind that it's frowned upon by some Pythonistas for several reasons:

- The order of the arguments is different from those of the classic `condition ? a : b` ternary operator from many other languages (such as C, C++, Go, Perl, Ruby, Java, Javascript, etc.), which may lead to bugs when people unfamiliar with Python's "surprising" behaviour use it (they may reverse the argument order).
- Some find it "unwieldy", since it goes contrary to the normal flow of thought (thinking of

- Stylistic reasons. (Although the 'inline `if`' can be *really* useful, and make your script more concise, it really does complicate your code)

If you're having trouble remembering the order, then remember that when read aloud, you (almost) say what you mean. For example, `x = 4 if b > 8 else 9` is read aloud as `x will be 4 if b is greater than 8 otherwise 9`.

Official documentation:

- Conditional expressions
- Is there an equivalent of C's "?:" ternary operator?

ɟited May 18 at 10:26

ɔmmunity wiki
7 revs, 13 users 28%
inko Vrsalovic

212 The order may seems strange for coders however

natural to mathematicians. You may also understand it as do A in most case, except when C then you should do B instead... – yota Jan 25 '16 at 15:07

87 Be careful with order of operations when using this. For example, the line `z = 3 + x if x < y else y`. If `x=2` and `y=1`, you might expect that to yield 4, but it would actually yield 1. `z = 3 + (x if x > y else y)` is the correct usage. – Kal Zekdor Mar 6 '16 at 9:23

6 The point was if you want to perform additional evaluations *after* the conditional is evaluated, like adding a value to the result, you'll either need to add the additional expression to both sides (`z = 3 + x if x < y else 3 + y`), or group the conditional (`z = 3 + (x if`

3) – Kal Zekdor Apr 15 '16 at 0:36

1 what if there are multiple conditions ? – MrGeek May 26 '17 at 15:31

2 @MrGeek, I see what you mean, so you would basically be nesting the operations: ` "foo" if Bool else ("bar" if Bool else "foobar") ` – Dimesio Aug 11 '17 at 0:04

678

You can index into a tuple:

```
(falseValue, true
```

`test` needs to return *True* or *False*.
It might be safer to always implement it as:

```
(falseValue, true
```

or you can use the built-in `bool()` to assure a Boolean value:

```
(falseValue, true
```

ommunity wiki
andon Kuhn

522 Note that this one always evaluates everything, whereas the if/else construct only evaluates the winning expression. – SilverbackNet Feb 4 '11 at 2:25

97 `(lambda: print("a"), lambda: print("b"))[test==true]()` – Dustin Getz Mar 8 '12 at 19:31

14 It should be noted that what's within the `[]`s can be an arbitrary expression. Also, for safety you can explicitly test for truthiness by writing `[bool(<expression>)]`. The `bool()` function has been around since v2.2.1. – martineau May 31 '12 at 18:20

12 This is great for code-golf, not so much for actual code. Although I have gotten so used to it that I do use it sometimes for conciseness when doing

string constants. – Claudiu Dec 5 '14 at 17:52

9 I've done a similar trick -- only once or twice, but done it -- by indexing into a dictionary with `True` and `False` as the keys: `{True:trueValue, False:falseValue}[test]` I don't know whether this is any less efficient, but it does at least avoid the whole "elegant" vs. "ugly" debate. There's no ambiguity that you're dealing with a boolean rather than an int. – JDM Mar 1 '16 at 18:43

281

For versions prior to 2.5, there's the trick:

```
[expression] and
```

It can give wrong results when `on_true` has a false boolean value.[1] Although it does have the benefit of evaluating expressions left to right, which is clearer in my opinion.

1. Is there an equivalent

dited Jan 13 '14 at 7:16

ommunity wiki
ames Brady

56 The remedy is to use (test and [true_value] or [false_value])[0], which avoids this trap. – ThomasH Oct 21 '09 at 15:33

3 Ternary operator usually executes faster(sometimes by 10-25%). – volcano Jan 13 '14 at 7:52

5 @volcano Do you have source for me? – OrangeTux Aug 5 '14 at 12:30

2 @OrangeTux Here's the disassembled code. Using the method ThomasH suggested would be even slower. – mbomb007 Mar 19 '18 at 20:59

190

```
<expression 1> if <condition> else <expression 2>
```

```
a = 1
b = 2

1 if a > b else -
# Output is -1

1 if a > b else -
# Output is -1
```

ommunity wiki
revs, 2 users 62%
imon Zimmermann

13 What's the difference between this and the top answer? – kennytm May 27 '10 at 7:59

70 This one emphasizes the primary intent of the ternary operator: value selection. It also shows that more than one ternary can be chained together into a single expression. – Roy Tinker Oct 4 '10 at 21:14

4 @Craig , I agree, but it's also helpful to know what will happen when there are no parentheses. In real code, I too would tend to insert explicit parens. – Jon Coombs Dec 1 '14 at 21:30

4 Somehow, I'm able to understand this better than the top answer. – Abhishek Divekar Mar 23 '18 at 5:46

124

From the documentation:

> Conditional expressions (sometimes

> Python operations.
>
> The expression `x if C else y` first evaluates the condition, *C* (*not x*); if *C* is true, *x* is evaluated and its value is returned; otherwise, *y* is evaluated and its value is returned.
>
> See PEP 308 for more details about conditional expressions.

New since version 2.5.

dited Oct 17 '15 at 7:43

ommunity wiki
ichael Burr

95

An operator for a conditional expression in Python was added in 2006 as part of Python Enhancement Proposal 308. Its form differ from common `?:` operator and it's:

```
<expression1> if
```

which is equivalent to:

Here is an example:

```
result = x if a >
```

Another syntax which can be used (compatible with versions before 2.5):

```
result = (lambda:
```

where operands are [lazily evaluated](#).

Another way is by indexing a tuple (which isn't consistent with the conditional operator of most other languages):

```
result = (y, x)[a
```

or explicitly constructed dictionary:

```
result = {True: x
```

Another (less reliable), but simpler method is to use `and` and `or` operators:

```
result = (a > b)
```

however this won't work if `x` would be `False`.

A possible workaround is to make `x` and `y` lists or tuples as in or:

```
result = ((a > b)
```

If you're working with dictionaries, instead of using a ternary conditional, you can take advantage of [`get(key, default)`](), for example:

```
shell = os.enviro
```

Source: [?: in Python at Wikipedia]()

lited Aug 7 '17 at 14:22

ommunity wiki
revs, 2 users 98%
enorb

1 `result = {1: x, 0: y}[a > b]` is another possible variant (`True` and `False` are actually integers with values `1` and `0`) – Walter Tross Feb 9 at 18:07

86

Unfortunately, the

```
(falseValue, true
```

solution doesn't have short-circuit behaviour; thus both `falseValue` and `trueValue` are evaluated suboptimal or even buggy (i.e. both `trueValue` and `falseValue` could be methods and have side-effects).

One solution to this would be

```
(lambda: falseVal
```

(execution delayed until the winner is known ;)), but it introduces inconsistency between callable and non-callable objects. In addition, it doesn't solve the case when using properties.

And so the story goes - choosing between 3 mentioned solutions is a trade-off between having the short-circuit feature, using at least 3ython 2.5 (IMHO not a problem anymore) and not being prone to " `trueValue` - evaluates-to-false" errors.

dited May 9 at 9:45

ommunity wiki
revs, 4 users 76%
orsky

1 While the tuple of lambdas trick

It's only likely to be a reasonable idea if it can replace a long chain of `if else if`. – Perkins Oct 11 '18 at 17:34

56

For Python 2.5 and newer there is a specific syntax:

```
[on_true] if [con
```

In older Pythons a ternary operator is not implemented but it's possible to simulate it.

```
cond and on_true
```

Though, there is a potential problem, which if `cond` evaluates to `True` and `on_true` evaluates to `False` then `on_false` is returned instead of `on_true`. If you want this behavior the method is OK, otherwise use this:

```
{True: on_true, F
```

which can be wrapped by:

```
def q(cond, on_tr
    return {True:
```

and used this way:

```
q(cond, on_true,
```

dited Apr 25 '12 at 12:02

ommunity wiki
aolo

2 The behaviour is not identical - `q("blob", on_true, on_false)` returns `on_false`, whereas `on_true if cond else on_false` returns `on_true`. A workaround is to replace `cond` with `cond is not None` in these cases, although that is not a perfect solution. – user3317 Sep 26 '12 at 9:09

4 Why not `bool(cond)` instead of `cond is True`? The former checks the truthiness of `cond`, the latter checks for pointer-equality with the `True` object. As highlighted by @AndrewCecil, `"blob"` is truthy but it `is not True`. – Jonas Kölker Nov 11 '13 at 16:11

Wow, that looks really hacky! :) Technically, you can even write `[on_false, on_True][cond is True]` so the expression becomes shorter.

circuit in this answer. If on_true and on_false are expensive to call this is a bad answer. – Hucker Mar 28 at 14:08

56

Ternary Operator in different programming Languages

Here I just try to show some important difference in `ternary operator` between a couple of programming languages.

> *Ternary Operator in Javascript*

```
var a = true ? 1
# 1
var b = false ? 1
# 0
```

> *Ternary Operator in Ruby*

```
a = true ? 1 : 0
# 1
b = false ? 1 : 0
# 0
```

> *Ternary operator in Scala*

```
val a = true ? 1
```

> *Ternary operator in R programming*

```
a <- if (TRUE) 1 (
# 1
b <- if (FALSE) 1
# 0
```

> *Ternary operator in Python*

```
a = 1 if True els
# 1
b = 1 if False el
# 0
```

dited Jan 26 at 14:05

ommunity wiki
revs, 3 users 96%
implans

11 This blogger found python's ternary operator to be unnecessarily different than most other languages. – JamesThomasMo Feb 15 '17 at 23:08

2 Ruby works also with `a = true ? 1 : 0` – rneves May 15 '17 at 17:50

7 "Now you can see the beauty of python language. its highly readable and maintainable." I don't see the relevance of this sentence, nor how the ternary operator syntax

Dec 8 '17 at 15:38

2 It may sound opinionated; but what it essentially says is that it the Python syntax is likely to be understood by a person who never saw a ternary operator, while very few people will understand the more usual syntax unless they have been told first what it means. – fralau Jan 10 '18 at 17:12

1 Algol68: a=.if. .true. .then. 1 .else. 0 .fi. This may be expressed also a=(.true.|1|0) As usual Algol68 is an improvement over its successors. – Albert van der Hor Jun 17 '18 at 12:55

36

You might often find

```
cond and on_true
```

but this lead to problem when on_true == 0

```
>>> x = 0
>>> print x == 0
1
>>> x = 1
>>> print x == 0
1
```

where you would

```
>>> x = 0
>>> print 0 if x
0
>>> x = 1
>>> print 0 if x
1
```

ıswered Jan 14 '13 at 15:56

ɔmmunity wiki
enoit Bertholon

29

Does Python have a ternary conditional operator?

Yes. From the grammar file:

```
test: or_test ['i
```

The part of interest is:

```
or_test ['if' or_
```

So, a ternary conditional operation is of the form:

```
expression1 if ex
```

`expression3` will be lazily evaluated (that is, evaluated only if `expression2` is false in a boolean context). And it may considered bad style.)

```
expression1 if ex
and so on
```

A note on usage:

Note that every `if` must be followed with an `else`. People learning list comprehensions and generator expressions may find this to be a difficult lesson to learn - the following will not work, as Python expects a third expression for an else:

```
[expression1 if e
#
```

which raises a `SyntaxError: invalid syntax`. So the above is either an incomplete piece of logic (perhaps the user expects a no-op in the false condition) or what may be intended is to use expression2 as a filter - notes that the following is legal Python:

```
[expression1 for
```

`expression2` works as a filter for the list comprehension, and is *not* a ternary

Alternative syntax for a more narrow case:

You may find it somewhat painful to write the following:

```
expression1 if ex
```

`expression1` will have to be evaluated twice with the above usage. It can limit redundancy if it is simply a local variable. However, a common and performant Pythonic idiom for this use-case is to use `or`'s shortcutting behavior:

```
expression1 or ex
```

which is equivalent in semantics. Note that some style-guides may limit this usage on the grounds of clarity - it does pack a lot of meaning into very little syntax.

dited Aug 8 '16 at 18:56

ommunity wiki
revs
aron Hall

1 `expression1`
`or`

ves as `expression1 || expression2` in javascript – JSDBroughton Feb 18 '16 at 13:05

1 Thanks, @selurvedu - it can be confusing until you get it straight. I learned the hard way, so your way might not be as hard. ;) Using if without the else, at the end of a generator expression or list comprehension will filter the iterable. In the front, it's a ternary conditional operation, and requires the else. Cheers!! – Aaron Hall ♦ May 27 '16 at 4:37

@AaronHall Although your use of metasyntactic `expressionN` for all instances is consistent, it might be easier to understand with naming that distinguished the conditional test expression from the two result expressions; eg, `result1 if condition else result2`. This is especially evident when nesting (aka chaining): `result1 if condition1 else result2 if condition2` tchrist Jan 26 at 14:12

@tchrist thanks for the review - if you look at the revision history, this post currently has two revisions. Most of my other answers, especially the top ones, have been revisited again and again. This answer never gets my attention because the community wiki status gives me no credit for the content, and so I never see any votes on it. As I don't really have time for an edit on this right now, frog knows when it will come to my attention again in the future. I can see you've edited the top answer, so feel free to borrow/quote my material from this post in that one (and cite me if apropos!) – Aaron Hall ♦ Jan 26 at 18:24

18

Simulating the python ternary operator.

For example

```
a, b, x, y = 1, 2
result = (lambda:
```

output:

```
'b greater than a
```

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

ɔmmunity wiki
asikiran Vaddi

Why not simply `result = (y, x)[a < b]` Why do you uses `lambda` function **?** – Grijesh Chauhan Dec 27 '13 at 5:50

5 @GrijeshChauhan Because on "compliated" expressions, e. g. involving a function call etc., this would be executed in both cases. This might not be wanted. – glglgl Feb 13 '14 at 8:14

16

you can do this :-

```
[condition] and [expression_1] or [expression_2] ;
```

Example:-

```
print(number%2 and "odd" or "even")
```

This would print "odd" if the number is odd or "even" if the number is even.

The result :- If condition is true exp_1 is executed else exp_2 is executed.

evaluates as False. And any data other than 0 evaluates to True.

Here's how it works:

if the condition [condition] becomes "True" then , expression_1 will be evaluated but not expression_2 . If we "and" something with 0 (zero) , the result will always to be fasle .So in the below statement ,

```
0 and exp
```

The expression exp won't be evaluated at all since "and" with 0 will always evaluate to zero and there is no need to evaluate the expression . This is how the compiler itself works , in all languages.

In

```
1 or exp
```

the expression exp won't be evaluated at all since "or" with 1 will always be 1. So it won't bother to evaluate the expression exp since the result will be 1 anyway .

But in case of

```
True and exp1 or
```

The second expression exp2 won't be evaluated since `True and exp1` would be True when exp1 isn't false .

Similarly in

```
False and exp1 or
```

The expression exp1 won't be evaluated since False is equivalent to writing 0 and doing "and" with 0 would be 0 itself but after exp1 since "or" is used, it will evaluate the expression exp2 after "or" .

Note:- This kind of branching using "or" and "and" can only be used when the expression_1 doesn't have a Truth value of False (or 0 or None or emptylist [ ] or emptystring ' '.) since if expression_1 becomes False , then the expression_2 will be evaluated because of the presence "or" between exp_1 and exp_2.

cases regardless of what exp_1 and exp_2 truth values are, do this :-

```
[condition] and
([expression_1] or
1) or
[expression_2] ;
```

dited Aug 20 '17 at 7:48

ommunity wiki
revs
atesh bhat

If you want to use that in the context of `x = [condition] and ([expression_1] or 1) or [expression_2]` and `expression_1` evaluates to false, `x` will be `1`, not `expression_1`. Use the accepted answer. – moi Oct 20 '17 at 6:37

15

Ternary conditional operator simply allows testing a condition in a single line replacing the multiline if-else making the code compact.

Syntax :

> [on_true] if [expression] else [on_false]

ternary operator:

```
# Program to demo
a, b = 10, 20
# Copy value of a
min = a if a < b
print(min)  # Out
```

2- Direct Method of using tuples, Dictionary, and lambda:

```
# Python program
a, b = 10, 20
# Use tuple for s
print( (b, a) [a
# Use Dictionary
print({True: a, F
# lamda is more e
# because in lamb
# only one expres
# tuple and Dicti
print((lambda: b,
```

3- Ternary operator can be written as nested if-else:

```
# Python program
a, b = 10, 20
print ("Both a an
        if a > b
```

Above approach can be written as:

```
# Python program
a, b = 10, 20
if a != b:
    if a > b:
        print("a
    else:
        print("b
else:
    print("Both a
# Output: b is gr
```

ıswered Apr 4 '18 at 14:02

li Hallaji

1 Note that the ternary operator is smaller (in memory) and faster than the nested if. Also, your nested `if-else` isn't actually a rewrite of the ternary operator, and will produce different output for select values of a and b (specifically if one is a type which implements a weird `__ne__` method). – Perkins Oct 11 '18 at 17:28

13

More a tip than an answer (don't need to repeat the obvious for the hundreth time), but I sometimes use it as a oneliner shortcut in such constructs:

```
if conditionX:
    print('yes')
else:
    print('nah')
```

, becomes:

```
print('yes') if c
```

Some (many :) may frown upon it as unpythonic (even, ruby-ish :), but I personally find it more natural - i.e. how you'd express it normally, plus a ɟited May 23 '16 at 19:16

ɔmmunity wiki
revs, 2 users 96%
ɔdor

3 I prefer `print( 'yes' if conditionX else 'nah' )` over your answer. :-) – frederick99 Aug 20 '17 at 6:07

That is if you want to `print()` in both cases - and it looks a bit more pythonic, I have to admit :) But what if the expressions/functions are not the same - like `print('yes') if conditionX else True` - to get the `print()` only in truthy `conditionX` – Todor Minakov Oct 26 '17 at 11:40

To add to Frederick99's remark, another reason to avoid `print('yes') if conditionX else print('nah')` is that it gives a SyntaxError in Python2. – Thierry Lathuille Oct 21 '18 at 21:51

The only reason it gives a syntax error is because in Python 2 print is a statement -

That can be resolved by either using it as a statement, or better - `from future import print_function` . – Todor Minakov Oct 22 '18 at 4:09

9

```
a if condition el
```

Just memorize this pyramid if you have trouble remembering:

```
     condition
  if          el
a
```

ıswered Dec 6 '18 at 14:45

ommunity wiki
ıivtej

5

YES, python have a ternary operator, here is the syntax and an example code to demonstrate the same :)

```
#[On true] if [ex
# if the expressi
false


a= input("Enter t
b= input("Enter t

print("A is Bigge
```

ıited Oct 21 '18 at 20:46

ommunity wiki
revs
ythonLover

I have added a one line statement example to check which number is big to elaborate it further – PythonLover Oct 21 '18 at 20:45

1 `print` is really not a good choice, as this will give a SyntaxError in Python2. – Thierry Lathuille Oct 21 '18 at 21:52

@Thierry Lathuille here I used print() function not print statement, print function is for Python 3 while print statement is for Python 2 – PythonLover Oct 21 '18 at 21:54

The question has already been asked on SO, just try it with Python 2 and you will see by yourself. 'print('hello') is a perfectly valid syntax in Python 2.7, but the way it is parsed makes your code above throw a SyntaxError. – Thierry Lathuille Oct 21 '18 at 21:58

What is the correct one please tell – PythonLover Oct

4

Many programming languages derived from `C` usually have the following syntax of ternary conditional operator:

```
<condition> ? <ex
```

> At first, the `Python` **B**enevolent **D**ictator **F**or **L**ife (I mean Guido van Rossum, of course) rejected it (as non-Pythonic style), since it's quite hard to understand for people not used to `C` language. Also, the colon sign **`:`** already has many uses in `Python`. After **PEP 308** was approved, `Python` finally received its own shortcut conditional expression (what we use now):

```
<expression1> if
```

So, firstly it evaluates the condition. If it returns `True`, **expression1** will be evaluated to give the result, otherwise **expression2** will one expression will be executed.

Here are some examples (conditions will be evaluated from left to right):

```
pressure = 10
print('High' if p

# Result is 'High
```

Ternary operators can be chained in series:

```
pressure = 5
print('Normal' if

# Result is 'Norm
```

The following one is the same as previous one:

```
pressure = 5

if pressure < 20:
    if pressure <
        print('No
    else:
        print('Hi
else:
    print('Critic

# Result is 'Norm
```

Hope this helps.

dited Jan 4 at 22:02

ommunity wiki
revs, 2 users 99%
RGeo

3

One of the alternatives to Python's

```
{True:"yes", Fals
```

which has the following nice extension:

```
{True:"yes", Fals
```

The shortest alterative remains:

```
("no", "yes")[boo
```

but there is no alternative if you want to avoid the evaluation of both `yes()` and `no()` in the following:

```
yes() if [conditi
```

ıswered Feb 9 at 18:23

ɔmmunity wiki
/alter Tross

0

A neat way to chain multiple operators:

```
f = lambda x,y: '

array = [(0,0),(0

for a in array:
  x, y = a[0], a[
  print(f(x,y))

# Output is:
#   equal,
#   less,
#   greater,
#   equal
```

ıswered May 12 at 13:03

-1

if variable is defined and you want to check if it has value you can just `a or b`

```
def test(myvar=No
    # shorter tha
    print myvar o

test()
test([])
test(False)
test('hello')
test(['Hello'])
test(True)
```

will output

```
no Input
no Input
no Input
hello
['Hello']
True
```

dited Apr 26 '18 at 16:22

ommunity wiki
revs
wwink

1 While useful for similar problems, this is not a ternary conditional. It works to replace `x if x else y`, but not `x if z else y`. – Perkins Oct 11 '18 at 17:13

protected by NullPoиteя Jun 10 '13 at 5:15

Thank you for your

answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?