# Emulate a do-while loop in Python?

I need to emulate a do-while loop in a Python program. Unfortunately, the following straightforward code does not work:

```
l = [ 1, 2, 3 ]
i = l.__iter__()
s = None
while True :
  if s :
    print s
  try :
    s = i.next()
  except StopIteration :
    break
print "done"
```

Instead of "1,2,3,done", it prints the following output:

```
[stdout:]1
[stdout:]2
[stdout:]3
None['Traceback (most recent call last):
',  '  File "test_python.py", line 8, in <module>
    s = i.next()
',  'StopIteration
']
```

What can I do in order to catch the 'stop iteration' exception and break a while loop properly?

An example of why such a thing may be needed is shown below as pseudocode.

State machine:

```
s = ""
while True :
  if state is STATE_CODE :
    if "//" in s :
      tokens.add( TOKEN_COMMENT, s.split( "//" )[1] )
      state = STATE_COMMENT
    else :
      tokens.add( TOKEN_CODE, s )
  if state is STATE_COMMENT :
    if "//" in s :
      tokens.append( TOKEN_COMMENT, s.split( "//" )[1] )
    else
      state = STATE_CODE
      # Re-evaluate same line
      continue
  try :
    s = i.next()
  except StopIteration :
    break
```

python   while-loop   do-while

edited Nov 25 '15 at 4:42
Mogsdad
27k 9 64 134

asked Apr 13 '09 at 6:18
Eye of Hell
7,903 32 123 222

1 Um... That's not a proper "do-while"; that's simply a "do-forever". What's wrong with "while True" and "break"? – S.Lott Apr 13 '09 at 15:43

36 S. Lott: I'm pretty sure his question was about *how* to implement do while in python. So, I wouldn't expect his code to be completely correct. Also, he is very close to a do while... he is checking a condition at the end of the "forever" loop to see if he should break out. It's not "do-forever". – Tom Apr 13 '09 at 18:43

1 Can you rather `li` instead of `l` because it looks too similar to `1`. Either that or `L`, or whatever else. – Tshepang May 17 '11 at 11:58

4 so ... your initial example code actually works for me with no problem and i don't get that traceback. that's a proper idiom for a do while loop where the break condition is iterator exhaustion. typically, you'd set `s=i.next()` rather than None and possibly do some initial work rather than just make your first pass through the loop useless though. – underrun Sep 21 '11 at 19:31

3 @underrun Unfortunately, the post is not tagged with which version of Python was being used - the original snippet works for me too using 2.7, presumably due to updates to the Python language itself. – Hannele Oct 2 '12 at 17:55

## 11 Answers

I am not sure what you are trying to do. You can implement a do-while loop like this:

```
while True:
  stuff()
  if fail_condition:
    break
```

Or:

```
stuff()
while not fail_condition:
  stuff()
```

What are you doing trying to use a do while loop to print the stuff in the list? Why not just use:

```
for i in l:
  print i
print "done"
```

Update:

So do you have a list of lines? And you want to keep iterating through it? How about:

```
for s in l:
  while True:
    stuff()
    # use a "break" instead of s = i.next()
```

Does that seem like something close to what you would want? With your code example, it would be:

```
for s in some_list:
  while True :
    if state is STATE_CODE :
      if "//" in s :
        tokens.add( TOKEN_COMMENT, s.split( "//" )[1] )
        state = STATE_COMMENT
      else :
        tokens.add( TOKEN_CODE, s )
    if state is STATE_COMMENT :
      if "//" in s :
        tokens.append( TOKEN_COMMENT, s.split( "//" )[1] )
        break # get next s
      else
        state = STATE_CODE
        # re-evaluate same line
        # continues automatically
```

edited Jan 1 '10 at 10:16
Peter Mortensen
11.1k 15 76 109

answered Apr 13 '09 at 6:28
Tom
11.1k 4 30 42

i need to create a state machine. In state machine it's a normal case to re-evaluate CURRENT statement, so i need to 'continue' without iterating next item. I don't know how to do such thing in 'for s in l:' iteration :(. In do-while loop, 'continue' will re-evaluate current item, iteration at end – Eye of Hell Apr 13 '09 at 6:41

Do you mean you need to keep track of your place in the list? That way when you return the same state, you can pick up where you left off? Give a bit more context. It seems like you might be better off using an index into the list. – Tom Apr 13 '09 at 6:48

pseudocode example added – Eye of Hell Apr 13 '09 at 7:32

Thanks, I commented on your pseudocode... your example seems sort of bad since you seem to handle "//" the same way no matter what state you are in. Also, is this real code where you are processing comments? What if you have strings with slashes? ie: print "blah // <-- does that mess you up?" – Tom Apr 13 '09 at 7:44

3 Also see PEP 315 for the official stance/justification: "Users of the language are advised to use the while-True form with an inner if-break when a do-while loop would have been appropriate." – dtk Aug 15 '16 at 12:47

---

Here's a very simple way to emulate a do-while loop:

```
condition = True
while condition:
    # loop body here
    condition = test_loop_condition()
# end of loop
```

The key features of a do-while loop are that the loop body always executes at least once, and that the condition is evaluated at the bottom of the loop body. The control structure show here accomplishes both of these with no need for exceptions or break statements. It does introduce one extra Boolean variable.

edited Oct 2 '12 at 17:23
martineau
46.8k 6 66 106

answered Mar 14 '10 at 0:09
powderflask
1,797 1 9 2

10 It doesn't always add an extra boolean variable. Often there's something(s) that already exist whose state can be tested. – martineau Oct 2 '12 at 17:32

8 The reason I like this solution the most is that it doesn't add another condition, it still is just one cycle, and if you pick a good name for the helper variable the whole structure is quite clear. – Roberto Oct 8 '13 at 21:04

3 NOTE: While this does address the original question, this approach is less flexible than using `break`. Specifically, if there is logic needed AFTER `test_loop_condition()`, that should not be executed once we are done, it has to be wrapped in `if condition:`. BTW, `condition` is vague. More descriptive: `more` or `notDone`. – ToolmakerSteve Dec 15 '13 at 0:30

5 @ToolmakerSteve I disagree. I rarely use `break` in loops and when I encounter it in code that I maintain I find that the loop, most often, could have been written without it. The presented solution is, IMO, the *clearest* way to represent a do while construct in python. – nonsensickle Sep 24 '15 at 23:48

1 Ideally, condition will be named something descriptive, like `has_no_errors` or `end_reached` (in which case the loop would start `while not end_reached` – Josiah Yoder Sep 28 '15 at 20:27

---

Exception will break the loop, so you might as well handle it outside the loop.

```
try:
  while True:
    if s:
      print s
    s = i.next()
except StopIteration:
  pass
```

I guess that the problem with your code is that behaviour of `break` inside `except` is not defined. Generally `break` goes only one level up, so e.g. `break` inside `try` goes directly to `finally` (if it exists) an out of the `try`, but not out of the loop.

Related PEP: http://www.python.org/dev/peps/pep-3136
Related question: Breaking out of nested loops

edited Apr 13 '09 at 7:48

answered Apr 13 '09 at 7:06
vartec
80.6k 24 156 204

6 It's good practice though to only have inside the try statement what you expect to throw your exception, lest you catch unwanted exceptions. – Paggas Nov 2 '09 at 18:10

5 @PiPeep: RTFM, search for EAFP. – vartec Nov 4 '10 at 9:35

1 @vartec My apologies, I am new to Python – bgw Nov 7 '10 at 19:47

2 @PiPeep: no problem, just keep in mind, that what's true for some languages, may not be true for other. Python is optimized for intensive use of exceptions. – vartec Nov 10 '10 at 16:30

4 break and continue are perfectly well-defined in any clause of a try/except/finally statement. They simply ignore them, and either break out of or move on to the next iteration of the containing while or for loop as appropriate. As components of the looping constructs, they're only relevant to while and for statements, and trigger a syntax error if they run into a class or def statement before reaching the innermost loop. They ignore if, with and try statements. – ncoghlan Feb 18 '11 at 6:41

---

```
do {
  stuff()
} while (condition())
```

->

```
while True:
  stuff()
  if not condition():
    break
```

You can do a function:

```
def do_while(stuff, condition):
  while condition(stuff()):
    pass
```

But 1) It's ugly. 2) Condition should be a function with one parameter, supposed to be filled by stuff (it's the only reason *not* to use the classic while loop.)

edited Jan 1 '10 at 10:18
Peter Mortensen
**11.1k** 15 76 109

answered Apr 13 '09 at 13:57
ZeD
**393** 1 3

2 Writing `while True: stuff(); if not condition(): break` is a very good idea. Thank you! – Noctis Skytower Sep 11 '12 at 20:05

---

My code below might be a useful implementation, highlighting the main difference between `do-while` vs `while` as I understand it.

So in this one case, you always go through the loop at least once.

```
firstPass = True
while firstPass or Condition:
    firstPass = False
    do_stuff()
```

edited Nov 27 '15 at 2:03
compski
**320** 5 19

answered Nov 23 '14 at 23:37
evan54
**1,138** 12 25

1 Correct answer, I'de argue. Plus it avoids **break**, for safe use in try/except blocks. – Zv_oDD Feb 26 '16 at 19:41

1 Lovely, elegant solution; tarnished only by the unfortunate intermingling of snake and camel case, yet not sufficiently as to dissuade my upvote. – Apollys Apr 25 at 2:17

the good old days when I didn't even know what that meant. I have to say I'm much more snake_case friendly these days hehehe – evan54 Apr 26 at 2:37

---

Here is a crazier solution of a different pattern -- using coroutines. The code is still very similar, but with one important difference; there are no exit conditions at all! The coroutine (chain of coroutines really) just stops when you stop feeding it with data.

```
def coroutine(func):
    """Coroutine decorator

    Coroutines must be started, advanced to their first "yield" point,
    and this decorator does this automatically.
    """
    def startcr(*ar, **kw):
        cr = func(*ar, **kw)
        cr.next()
        return cr
    return startcr

@coroutine
```

```
def collector(storage):
    """Act as "sink" and collect all sent in @storage"""
    while True:
        storage.append((yield))

@coroutine
def state_machine(sink):
    """ .send() new parts to be tokenized by the state machine,
    tokens are passed on to @sink
    """
    s = ""
    state = STATE_CODE
    while True:
        if state is STATE_CODE :
            if "//" in s :
                sink.send((TOKEN_COMMENT, s.split( "//" )[1] ))
                state = STATE_COMMENT
            else :
                sink.send(( TOKEN_CODE, s ))
        if state is STATE_COMMENT :
            if "//" in s :
                sink.send(( TOKEN_COMMENT, s.split( "//" )[1] ))
            else
                state = STATE_CODE
                # re-evaluate same line
                continue
        s = (yield)

tokens = []
sm = state_machine(collector(tokens))
for piece in i:
    sm.send(piece)
```

The code above collects all tokens as tuples in `tokens` and I assume there is no difference between `.append()` and `.add()` in the original code.

answered Nov 2 '09 at 17:32
u0b34a0f6ae
**25.1k** 8 71 87

2 How would you write this in Python 3.x today? – Noctis Skytower Sep 11 '12 at 20:07

---

for a do - while loop containing try statements

```
loop = True
while loop:
    generic_stuff()
    try:
        questionable_stuff()
#       to break from successful completion
#       loop = False
    except:
        optional_stuff()
#       to break from unsuccessful completion -
#       the case referenced in the OP's question
        loop = False
   finally:
        more_generic_stuff()
```

alternatively, when there's no need for the 'finally' clause

```
while True:
    generic_stuff()
    try:
        questionable_stuff()
#       to break from successful completion
#       break
    except:
        optional_stuff()
#       to break from unsuccessful completion -
#       the case referenced in the OP's question
        break
```

edited Oct 2 '12 at 18:12
martineau
**46.8k** 6 66 106

answered May 10 '11 at 22:03
Mark
**1,025** 7 5

---

```
while condition is True:
  stuff()
else:
  stuff()
```

edited Nov 30 '10 at 1:43

answered Nov 30 '10 at 1:38

MuSheng
**189** 1 5

7 Ew. That seems significantly uglier than using a break. – mattdm Jan 26 '12 at 14:42

3 That is clever, but it requires `stuff` to be a function or for the code body to be repeated. – Noctis Skytower Sep 11 '12 at 20:08

9 All that's needed is `while condition:` because `is True` is implied. – martineau Oct 2 '12 at 18:15

1 this fails if `condition` depends on some inner variable of `stuff()`, because that variable is not defined at that moment. – yo' Feb 25 '14 at 20:23

2 Not the same logic, because on the last iteration when condition != True : It calls the code a final time. Where as a ***Do While***, calls the code once first, then checks condition before re-running. Do While : **execute block once; then check and re-run**, this answer: **check and re-run; then execute code block once**. Big difference! – Zv_oDD Feb 26 '16 at 19:34

---

Quick hack:

```
def dowhile(func = None, condition = None):
    if not func or not condition:
        return
    else:
        func()
        while condition():
            func()
```

Use like so:

```
>>> x = 10
>>> def f():
...     global x
...     x = x - 1
>>> def c():
        global x
        return x > 0
>>> dowhile(f, c)
>>> print x
0
```

answered Apr 21 '13 at 21:42
Naftuli Kay
**24.3k** 57 176 272

---

Why don't you just do

```
for s in l :
    print s
print "done"
```

?

edited Jan 1 '10 at 10:19
Peter Mortensen
**11.1k** 15 76 109

answered Apr 13 '09 at 6:23
Martin
**4,137** 3 19 43

i need to create a state machine. In state machine it's a normal case to re-evaluate CURRENT statement, so i need to 'continue' without iterating next item. I don't know how to do such thing in 'for s in l:' iteration :(. In do-while loop, 'continue' will re-evaluate current item, iteration at end. – Eye of Hell Apr 13 '09 at 6:26

then, can you define some pseudo-code for your state machine, so we can hint you towards the best pythonic solution ? I don't know much about state machines(and am probably not the only one), so if you tell us a bit about your algorithm, this will be easier for us to help you. – Martin Apr 13 '09 at 6:48

pseudocode example added – Eye of Hell Apr 13 '09 at 7:29

For loop does not work for things like: a = fun() while a == 'zxc': sleep(10) a = fun() – harry Sep 19 '13 at 7:26

---

See if this helps :

Set a flag inside the exception handler and check it before working on the s.

```
flagBreak = false;
while True :

    if flagBreak : break
    if s :
        print s
    try :
        s = i.next()
    except StopIteration :
        flagBreak = true

print "done"
```

edited Oct 2 '12 at 18:26
martineau
46.8k 6 66 106

answered Apr 13 '09 at 8:17
Nrj
3,751 5 33 45

2 Could be simplified by using `while not flagBreak:` and removing the `if (flagBreak) : break`. – martineau Oct 2 '12 at 18:23

1 I avoid variables named `flag` --I am unable to infer what a True value or False value mean. Instead, use `done` or `endOfIteration`. The code turns into `while not done: ...`. – IceArdor Mar 11 '14 at 20:03