



- » [ForLoop](#)
- » [ForLoop](#)
- » [FrontPage](#)
- » [RecentChanges](#)
- » [FindPage](#)
- » [HelpContents](#)
- » [ForLoop](#)

## Page

- » Immutable Page
- » [Info](#)
- » [Attachments](#)
- » More Actions:

## User

- » [Login](#)

## For loops

### Usage in Python

- » When do I use for loops?

*For* loops are traditionally used when you have a piece of code which you want to repeat  $n$  number of times. As an alternative, there is the [WhileLoop](#), however, *while* is used when a condition is to be met, or if you want a piece of code to repeat forever, for example -

*For loop from 0 to 2, therefore running 3 times.*

```
for x in range(0, 3):  
    print "We're on time %d" % (x)
```

*While loop from 1 to infinity, therefore running infinity times.*

```
x = 1
```

```
while True:
    print "To infinity and beyond! We're getting close, on %d now!"
    % (x)
    x += 1
```

As you can see, they serve different purposes. The *for* loop runs for a fixed amount - in this case, 3, while the *while* loop theoretically runs forever. You could use a *for* loop with a huge number in order to gain the same effect as a *while* loop, but what's the point of doing that when you have a construct that already exists? As the old saying goes, "why try to reinvent the wheel?".

#### » How do they work?

If you've done any programming before, there's no doubt you've come across a *for* loop or an equivalent to it. In Python, they work a little differently. Basically, any object with an iterable method can be used in a *for* loop in Python. Even strings, despite not having an iterable method - but we'll not get on to that here. Having an iterable method basically means that the data can be presented in list form, where there's multiple values in an orderly fashion. You can define your own iterables by creating an object with `next()` and `iter()` methods. This means that you'll rarely be dealing with raw numbers when it comes to *for* loops in Python - great for just about anyone!

#### » Nested loops



When you have a piece of code you want to run *x* number of times, then code within that code which you want to run *y* number of times, you use what is known as a "nested loop". In Python, these are heavily used whenever someone has a list of lists - an iterable object within an iterable object.

#### » Early exits

Like the *while* loop, the *for* loop can be made to exit before the given object is finished. This is done using the *break* keyword, which will stop the code from executing any further. You can also have an optional *else* clause, which will run should the *for* loop exit cleanly - I.E., without breaking.

## Things to remember

#### » range vs xrange

The  *range* function creates a list containing numbers defined by the input. The  *xrange* function creates a number generator. You will often see that *xrange* is used much more frequently than *range*. This is for one reason only - resource usage. The *range* function generates a list of numbers all at once, where as *xrange* generates them as needed. This means that less memory is used, and should the *for* loop exit early, there's no need to waste time creating the unused numbers. This effect is tiny in smaller lists, but increases rapidly in larger lists as you can see in the examples below.

## Examples

### *Nested loops*

```
for x in xrange(1, 11):
    for y in xrange(1, 11):
        print '%d * %d = %d' % (x, y, x*y)
```

### *Early exit*

```
for x in xrange(3):
    if x == 1:
        break
```

### *For..Else*

```
for x in xrange(3):
    print x
else:
    print 'Final x = %d' % (x)
```

### *Strings as an iterable*

```
string = "Hello World"
for x in string:
    print x
```

### *Lists as an iterable*

```
collection = ['hey', 5, 'd']
for x in collection:
    print x
```

### *Lists of lists*

```
list_of_lists = [ [1, 2, 3], [4, 5, 6], [7, 8, 9]]
for list in list_of_lists:
    for x in list:
        print x
```

### *Creating your own iterable*

```
class Iterable(object):

    def __init__(self, values):
        self.values = values
        self.location = 0
```

```
def __iter__(self):
    return self

def next(self):
    if self.location == len(self.values):
        raise StopIteration
    value = self.values[self.location]
    self.location += 1
    return value
```

### *range vs xrange*

```
import time

#use time.time() on Linux

start = time.clock()
for x in range(10000000):
    pass
stop = time.clock()

print stop - start

start = time.clock()
for x in xrange(10000000):
    pass
stop = time.clock()

print stop - start
```

### *Time on small ranges*

```
import time

#use time.time() on Linux

start = time.clock()

for x in range(1000):
    pass
stop = time.clock()

print stop-start
```

```
start = time.clock()
for x in xrange(1000):
    pass
stop = time.clock()

print stop-start
```

### *Your own range generator using yield*

```
def my_range(start, end, step):
    while start <= end:
        yield start
        start += step

for x in my_range(1, 10, 0.5):
    print x
```

ForLoop (last edited 2013-04-14 21:11:23 by [HunterSelsor](#))

- » [MoinMoin Powered](#)
- » [Python Powered](#)
- » [GPL licensed](#)
- » [Valid HTML 4.01](#)