

# For loops

## Usage in Python

- When do I use for loops?

*for* loops are traditionally used when you have a block of code which you want to repeat a fixed number of times. The Python *for* statement iterates over the members of a sequence in order, executing the block each time. Contrast the *for* statement with the ["while" loop](#), used when a condition needs to be checked each iteration, or to repeat a block of code 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 forever.*

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

As you can see, these loop constructs serve different purposes. The *for* loop runs for a fixed amount - in this case, 3, while the *while* loop runs until the loop condition changes; in this example, the condition is the boolean *True* which will never change, so it could theoretically run 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, you have undoubtedly come across a *for* loop or an equivalent to it. Many languages have conditions in the syntax of their *for* loop, such as a relational expression to determine if the loop is done, and an increment expression to determine the next loop value. In Python this is controlled instead by generating the appropriate sequence. Basically, any object with an iterable method can be used in a *for* loop. 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 are 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 block of code you want to run **x** number of times, then a block of 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.

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

- 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* statement, which will immediately drop out of the loop and continue execution at the first statement after the block. You can also have an optional *else* clause, which will run should the *for* loop exit cleanly - that is, without breaking.

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

## Examples

### *For..Else*

```
for x in range(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)
```

### *Loop over 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
```

### *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)
```

## A note on `range`

The "[range](#)" function is seen so often in *for* statements that you might think *range* is part of the *for* syntax. It is not: it is a Python built-in function which returns a sequence following a specific pattern (most often sequential integers), which thus meets the requirement of providing a sequence for the *for* statement to iterate over. Since *for* can operate directly on sequences, and there is often no need to count. This is a common beginner construct (if they are coming from another language with different loop syntax):

```
mylist = ['a', 'b', 'c', 'd']
for i in range(len(mylist)):
    # do something with mylist[i]
```

It can be replaced with this:

```
mylist = ['a', 'b', 'c', 'd']
for v in mylist:
    # do something with v
```

Consider `for var in range(len(something)):` to be a flag for possibly non-optimal Python coding.