Loop control statements

The **break** statement is used to terminate a loop of any type (i.
e. `for` and `while` loops). It may be said that **break** "jumps out" of the loop where it was placed. Let's examine a tiny example:

```
pets = ['dog', 'cat', 'parrot']
for pet in pets:
    print(pet)
    if pet == 'cat':
        break
```

We wanted to stop the loop before it iterated for the last time. For that purpose, we introduced a condition when the loop should be stopped. The output is as follows:

```
dog
cat
```

Be careful where you put print(). If you put it at the loop's end, the output will return only the first value – 'dog'. This happens because **break** exits from the loop immediately.

Often enough, **break** is used to stop endless `while` loops like this one:

```
count = 0
while True:
    print("I am Infinite Loop")
    count += 1
    if count == 13:
        break
```

Let's consider how break works
in `for...else` or `while...else` statements. `else` statements will be executed if no `break` is encountered during the execution of the loop. Look how it works in the examples below!

```
numbers = [1, 2, 3]
for num in numbers:
    if num == 5:
        print("It's 5!")
        break
else:
    print('5 was not listed')
# 5 was not listed
```

In this case, the body of the `else` statement is executed because no `break` was met. Let's change the `number` list.

```
numbers = [1, 2, 3, 4, 5, 6]
for num in numbers:
    if num == 5:
        print("It's 5!")
        break
```

```
else:
    print('5 was not listed')
# It's 5!
```

`break` was met, so the `print()` command after `else` wasn't executed.

Continue

The **continue** operator is commonly used, too. You can stop the iteration if your condition is true and return to the beginning of the loop (that is, jump to the loop's top and continue execution with the next value). Look at the following example:

```
pets = ['dog', 'cat', 'parrot']
for pet in pets:
    if pet == 'dog':
        continue
    print(pet)
```

The output will contain all values except the first one ('dog') since it fulfills the condition:

```
cat
parrot
```

Thus, the loop just skips one value and goes on running.

One nuance is worth mentioning: the **continue** operator should be used moderately. Sometimes you can shorten the code by simply using an `if` statement with the **reversed** condition:

```
pets = ['dog', 'cat', 'parrot']
for pet in pets:
    if pet != 'dog':
        print(pet)
```

In this case, the output will remain the same:

```
cat
parrot
```

Pass

When no action is required (e.g. some condition is met, you need to take it into account in a loop, but do nothing if that's the case), in Python, you can use `pass` statement which does exactly what you need – nothing. Here the program is just waiting to be manually interrupted:

```
while True:
    pass
```

Summary

To sum up, loop control statements represent a useful tool to alter the way a loop works. You can introduce extra conditions using the `break`, `continue`, and `pass` operators. In addition, they allow you to skip a beforehand selected set of values, terminate an endless loop, or even do nothing. Use them wisely and they'll work wonders.