Python → Control flow statements → Loop control statements

Theory: Loop control statements

0 users solved this topic.

0 / 0 problems solved

Start practicing

§1. Modifying loops

Loop control statements are nested inside loops and designed to change their typical behavior. In this topic, we'll find out how they work and what they are used for.

§2. How to break

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
```

§3. How to 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:

Current topic:

Loop control statements ···

Table of contents:

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

In this case, the output will remain the same:

```
cat
parrot
```

§4. Loop else clause

If the loop didn't encounter the break statement, an **else clause** can be used to specify a block of code to be executed after the loop.

```
pets = ['dog', 'cat', 'parrot']
for pet in pets:
    print(pet)
else:
    print('We need a turtle!')
```

So after the loop body, the **else** statement will execute:

```
dog
cat
parrot
We need a turtle!
```

Importantly, loop `else` runs if and only if the loop is exited normally (without hitting **break**). Also, it is run when the loop is never executed (e. g. the condition of the `while` loop is false right from the start). Consider an example:

```
pancakes = 2
while pancakes > 0:
    print("I'm the happiest human being in the world!")
    pancakes -= 1
    if pancakes == 0:
        print("Now I have no pancakes!")
        break
else:
    print("No pancakes...")
```

When we run the code for the first time we'll get this output:

```
I'm the happiest human being in the world!
I'm the happiest human being in the world!
Now I have no pancakes!
```

Execution of the code snippet for the second time (when the condition is not met, for `pancakes = 0`) will end up with another message:

```
No pancakes...
```

§5. In conclusion

To sum up, loop control statements represent a useful tool to alter the way a loop works. You can introduce extra conditions using `break`, `continue` and `else` operators. In addition, they allow you to print a message after the successful code execution, skip a beforehand selected set of values, or even terminate an endless loop. Use them wisely and they'll work wonders.

Report a typo

604 users liked this theory. 5 didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Comments (17) Hints (1) Useful links (0) Show discussion