7. while loops (indefinite interations)

Overview:

- Construct basic and complex (nested) while loops;
- Interrupt loop execution with **break** and **continue**;
- Use the **else** clause with a while loop;
- Deal with **infinite loops**.

In programming, there are **2 types of iteration**:

- **Indefinite iteration:** the number of times the loop is executed isn't specified explicitly in advance. Rather, the designated block is executed repeatedly **as long as some condition is met**;
- **Definite iteration:** the number of times a block will be executed is specified explicitly at the time when the loop defined.

1 while-loop

A rudimentary **while** loop:

```
while <expr>:
    <statement(s)>
```

where:
`<statement(s)>` is a block to be repeatedly executed, aka **body of the loop**; `<expr>` is a **controlling expression**; typically involves one or more variables that are initialized prior the loop and then **modified somewhere in the loop body**.

The `<expr>` is first evaluated in Boolean context. If it is `True`, the loop body is executed. Then `<expr>` is checked again, and if still `True`, the body is executed again. This continues until `<expr>` becomes `False`, so that program goes to the first statement after the loop body.

Example:

```
n = 5
while n > 0: # checking the condition
    n -= 1
    print(n)
>> 4 3 2 1 0
```

Note that the **controlling expression of the while loop is tested first, before anything else happens**:

```
n = 0
while n > 0: # checking the condition - nothing will happen!
    n -= 1
    print(n)
```

When a list is evaluated in Boolean context, it is truthy if it has elements in it and falsy if it is empty:

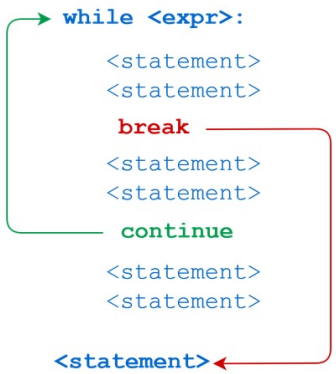
```
a = ['foo', 'bar', 'baz']
while a:
     print(a.pop()) # .pop() method also returns a popping element
>> baz bar foo
```

2 break and continue statements

Python provides 2 keywords that **terminate a loop iteration prematurely**:

(1) `break` statement **immediately terminates a loop entirely**. Program execution proceeds to the first statement following the loop body.

(2) `continue` statement **immediately terminates the current loop iteration**. Execution jumps to the top of the loop, and the controlling expression is re-evaluated.



3 else-clause

This is a *unique feature of Python.* The syntax is shown below:

```
while <expr>:
    <statement(s)>
else:
    <additional_statement(s)>
```

The <additional_statement(s)> specified in the else clause will be executed when the while loop terminates.

You might think that you could accomplish the same by putting those statements immediately after the while loop, without the else:

```
while <expr>:
    <statement(s)>
<additional_statement(s)>
```

What's the difference?

When <additional_statement(s)> are placed in an `else` clause, **they will be executed only if the loop terminates "by exhaustion"** i.e. if the loop iterates until the controlling condition becomes false. **If the loop is exited by a `break`, the `else` clause won't be executed**:

```
n = 5
while n > 0:
    n -= 1
    print(n)
    if n == 2:
        break
else:
    print('Loop done.')
>> 4 3 2
```

This loop is terminated prematurely with break, so the else clause isn't executed.

When the else-clause be useful?

(1) E.g. searching in a list for a specific item. You can use **break** to exit the loop if the item is found, and the **else**-clause can contain code that is meant to be executed if the item isn't found:

```
a = ['foo', 'bar', 'baz', 'qux']
s = 'corge'

i = 0
while i < len(a):
    if a[i] == s:
        # Processing for item found
        break
    i += 1
else:
    # Processing for item not found
    print(s, 'not found in list.')

>> 'corge not found in list.'
```

Note: The code shown above is useful to illustrate the concept, but you'd actually be **very unlikely to search a list that way**.

An **else**-clause with a **while**-loop is a bit of an oddity, not often seen. But don't shy away from it if you find a situation in which you feel it adds clarity to your code!

4 Infinite loops

Suppose you write a while loop that **theoretically never ends**.

```
while True:
    <statement(s)>
```

1. This pattern is actually quite common e.g. you might write code for a service that runs accepting service requests.
2. Loops can be broken out of with the break statement. **It may be more straightforward to terminate a loop based on conditions recognized within the loop body**.

Example of loop that successively removes items from a list using `.pop()` until it is empty:

```
a = ['foo', 'bar', 'baz']
while True:
    if not a:
        break
    print(a.pop(-1))
>> baz bar foo
```

You can also specify **multiple break statements in a loop**:

```
while True:
    if <expr1>:  # One condition for loop termination
        break
    ...
    if <expr2>:  # Another termination condition
        break
    ...
    if <expr3>:  # Yet another
        break
```

In cases like this, with multiple reasons to end the loop, it is often **cleaner to break out from several different locations, rather than try to specify all the termination conditions in the loop header. Infinite loops can be very useful.**

5 Nested while loops

```
a = ['foo', 'bar']
while len(a):
    print(a.pop(0))
    b = ['baz', 'qux']
    while len(b):
        print('>', b.pop(0))

foo
> baz
> qux
bar
> baz
> qux
```

A **break** or **continue** statement found within nested loops **applies to the nearest enclosing loop**:

```
while <expr1>:
    statement
    statement

    while <expr2>:
        statement
        statement
        break  # Applies to while <expr2>: loop

    break  # Applies to while <expr1>: loop
```

In fact, all the Python control structures can be intermingled with one another to whatever extent you need.

6 Single-line while loops

As with an if statement, a **while** loop can be specified on one line. If there are multiple statements in the block that makes up the loop body, they can be separated by semicolons **;** :

```
n = 5
while n > 0: n -= 1; print(n)

>> 4 3 2 1 0
```

This **only works with simple statements** though. You can't combine two compound statements into one line.