Fundamentos de Programação

António J. R. Neves
João Rodrigues

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

Summary

- Iteration
- The `while` statement
- The `for` statement
- The `range` function
- The `break` statement
- Other suff: `continue` statement and `else` clause

The `while` statement

- The `while` statement tells Python to repeatedly execute some target statements for as long as a given condition is true.

Syntax	Example

`...`
`while condition:`
`  statements`
`...`
	`n = 3`
	`while n > 0:`
	`  print(n)`
	`  n = n-1`
	`print("Go!")`

- The `condition` may be any expression, which is converted to bool, so any null or empty value means false.
- If the condition is true, the `statements` are executed.
- Then, the condition is evaluated again, and if still true, the statements are repeated.
- When the condition becomes false, execution skips to the line immediately following the loop.

The `break` statement

- The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise, the loop will repeat forever, which is called an *infinite loop*.
- Sometimes only half way through the body is it possible to decide if the loop should stop. In that case you can use the `break` statement to jump out of the loop.

```
while True:
    line = input('Enter text? ')
    if line == 'done':
        break
    print(line)
print('The end')
```

The `for` statement

- Another loop mechanism is the `for` statement.
- It repeats statements once for each item in a sequence, such as a list, a string or a tuple.

Syntax	Example

`...`
`for var in sequence:`
`  statements`
`...`
	`for n in [3, 1, 9]:`
	`  print(n)`
	`print("End")`

- If `sequence` is an expression, it is evaluated first.
- Then, the first item in the sequence is assigned to the iterating variable `var`, and the statements block is executed once.
- Next, the second item is assigned to `var`, the statements are executed again, and so on, until the entire sequence is exhausted.

The `range` function

- The built-in function `range` generates a sequence of numbers in arithmetic progression.

  `list(range(4))  →  [0, 1, 2, 3]`

- The `range` function is often used in `for` loops.

```
for n in range(1, 4):
    print(n)
```

- It may be called with 1, 2 or 3 arguments, as follows:
  - `range(stop)`
  - `range(start, stop)`
  - `range(start, stop, step)`
- All arguments must be **integers**.
- All arguments can be positive or negative.
- Generates integers up to, but **not including**, stop.

Loop control statements

- Loop control statements change the execution from its normal sequence (`break, continue, pass`).
- `break` terminates the loop statement and transfers execution to the statement immediately following the loop.
- The `continue` statement returns the control to the beginning of the current loop. When encountered, the loop starts next iteration without executing the remaining statements in the current iteration.
- `pass` is used when a statement is required syntactically but nothing is needed to be executed. Nothing happens when it is executed. The `pass` statement is also useful in places where the code will eventually go, but has not been written yet.

The `else` clause

- The iteration statements allow an optional `else` clause.

```
count = 0
while count < 5:
    print(count, "is  less than 5")
    count += 1
else:
    print(count, "is not less than 5")
```

- Statements in the `else` clause are executed when the condition evaluates to false.
- They are not executed if a `break` terminates the loop.
- This is <u>unusual</u>, <u>confusing</u> and seldom used.