Full Stack Development
Sesi 2

PYTHON: CONDITIONS, CONTROL FLOW LOOPING

Conditional

Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b
- These conditions can be used in several ways, most commonly in "if statements" and loops.

Introduction to the if Statement

We'll start by looking at the most basic type of if statement. In its simplest form, it looks like this:

```
if <expr>:
    <statement>
```

In the form shown above:

- <expr> is an expression evaluated in Boolean context.
- <statement> is a valid Python statement, which must be indented.

If <expr> is true (evaluates to a value that is "truthy"), then <statement> is executed. If <expr> is false, then <statement> is skipped over and not executed.

Grouping Statements

In a Python program, contiguous statements that are indented to the same level are considered to be part of the same block. Thus, a compound if statement in Python looks like this:

```
if <expr>:
    <statement>
    <statement>
    ...
    <statement>
<following_statement>
```

The else and elif Clauses

Sometimes, you want to evaluate a condition and take one path if it is true but specify an alternative path if it is not. This is accomplished with an else clause:

```
if <expr>:
    <statement(s)>
else:
    <statement(s)>
```

If <expr> is true, the first suite is executed, and the second is skipped. If is false, the first suite is skipped and the second is executed. Either way, execution then resumes after the second suite. Both suites are defined by indentation, as described above.

The else and elif Clauses

There is also syntax for branching execution based on several alternatives. For this, use one or more elif (short for else if) clauses:

```
if <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
    ...
else:
    <statement(s)>
```

One-Line if Statements

It is customary to write if <expr> on one line and <statement> indented on the following line like this:

```
if <expr>:
    <statement>
```

But it is permissible to write an entire if statement on one line:

```
if <expr>: <statement>
```

There can even be more than one <statement> on the same line, separated by semicolons:

```
if <expr>: <statement_1>; <statement_2>; ...; <statement_n>
```

Conditional Expressions

Python supports one additional decision-making entity called a conditional expression or ternary operator. (It is also referred to as a conditional operator or ternary operator in various places in the Python documentation.) Conditional expressions were proposed for addition to the language in PEP 308 and green-lighted by Guido in 2005.

In its simplest form, the syntax of the conditional expression is as follows:

<expr1> if <conditional_expr> else <expr2>

In the above example, <conditional_expr> is evaluated first. If it is true, the expression evaluates to <expr1>. If it is false, the expression evaluates to <expr2>.

The Python pass Statement

Occasionally, you may find that you want to write what is called a code stub: a placeholder for where you will eventually put a block of code that you haven't implemented yet. If you try to run the code below, you will get an error:

```
if True:

        print('foo')
```

The Python pass statement solves this problem and the code can run without error.

```
if True:

    pass


print('foo')
```

Looping

Iteration means executing the same block of code over and over, potentially many times. A programming structure that implements iteration is called a loop.

In programming, there are two types of iteration, indefinite and definite:

- With 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.
- With definite iteration, the number of times the designated block will be executed is specified explicitly at the time the loop starts.

The Python while Loops

The format of a rudimentary while loop is shown below:

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

<statement(s)> represents the block to be repeatedly executed, often referred to as the body of the loop.

When a while loop is encountered, <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.

The break and continue Statements

Python provides two keywords that terminate a loop iteration prematurely:

- The Python break statement immediately terminates a loop entirely. Program execution proceeds to the first statement following the loop body.
- The Python continue statement immediately terminates the current loop iteration. Execution jumps to the top of the loop, and the controlling expression is re-evaluated to determine whether the loop will execute again or terminate.

The break and continue Statements

Example of a break statements:

```
n = 5
while n > 0:
    n -= 1
    if n == 2:
        break # Break Statement
    print(n)
print('Loop ended.')
```

The break and continue Statements

Example of a continue statements:

```
n = 5
while n > 0:
    n -= 1
    if n == 2:
        continue
    print(n)
print('Loop ended.')
```

The else Clause

Python allows an optional else clause at the end of a while loop. This is a unique feature of Python, not found in most other programming languages. 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.

Nested while Loops

In general, Python control structures can be nested within one another. For example:

```
a = ['foo', 'bar']

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

    while len(b):
        print('>', b.pop(0))
```

One-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)
```

The Python for Loop

Python's for loop looks like this:

```
for <var> in <iterable>:
    <statement(s)>
```

<iterable> is a collection of objects—for example, a list or tuple. The <statement(s)> in the loop body are executed once for each item in <iterable>. The loop variable <var> takes on the value of the next element in <iterable> each time through the loop. Here is a representative example:

```
a = ['foo', 'bar', 'baz']

for i in a:
    print(i)
```

The range() Function

Python provides a better option—the built-in range() function, which returns an iterable that yields a sequence of integers. range(<end>) returns an iterable that yields integers starting with 0, up to but not including <end>:

```
x = range(5)

for n in x:

    print(n)
```

Altering for Loop Behavior

break terminates the loop completely and proceeds to the first statement following the loop:

```
for i in ['foo', 'bar', 'baz', 'qux']:
    if 'b' in i:
        break
    print(i)
```

continue terminates the current iteration and proceeds to the next iteration:

```
for i in ['foo', 'bar', 'baz', 'qux']:
    if 'b' in i:
        continue
    print(i)
```

External References

Control Flow - Visit Here

Looping - Visit Here