E.Tutorial
www.et.ethz.ch

Programming with Python Module 2

Control structures and logic

Theoretical part

Contents

Keywords

Command block	Truth value	while-loops
Command head	Relational operators	Nested loops
Command body	Branch	Dead code
Logical operators	for-loop	Exception handling

Authors:

Lukas Fässler, Markus Dahinden, Dennis Komm, David Sichau

Translation:

Christa Furrer, Robin Schmidiger

E-Mail:

et@ethz.ch

Date:

29 November 2024

Version: 1.1

Hash: 9f51da6

The authors try the best to provide an error free work, however there still might be some errors. The Authors are thankful for your feedback and suggestions.

1 Module overview

A program usually consists of several commands. These commands are executed in an order determined by the programmer. This program sequence (sequence of commands) rarely proceeds linearly. Often, a program sequence branches into two or more program sequences whereby each individual sequence is only executed under certain conditions (**branching**). To simplify an algorithm, certain program sequences are often executed repeatedly (**loops**). The program process can be influenced by means of **control structures** which occur in most programming languages. The decision how to control the process has to be formulated in the **condition**.

1.1 Commands and blocks in Python

As already mentioned in the first module, individual commands are usually placed in a separate line each in Python. Several commands can be combined into one **command block**.

Many programming languages use keywords to mark command blocks (`Begin .. end`) or brackets (`{ . }`). In Python, commands are **indented** with the **tab key** or by **spaces** to structure the program.

The execution of blocks can be controlled by control structures (e.g. branching or loops). These control structures consist of a **head** and a **body**. The command head is marked by a **colon** (`:`) at the end of the line.

```
Command head:
    # Begin command body
    Command 1
    Command 2
    # End command body
```

In the following program, the command block 1 is interrupted by a command block 2:

```
# Begin command block 1

Command head 1:
  Command

  # Begin command block 2
  Command head 2:
    Command
  # End command block 2

  # Continuing command block 1
  Command

# End command block 1
```

2 Operators (part II)

The **arithmetic operators** have already been described in module 1. In connection with control structures, **logical and relational operators** are employed.

2.1 Relational Operators

Relational operators are used to compare values (operands). They provide a logical result **True** or **False**. Values that are connected by relational operators are called **atomic propositions** in the propositional calculus.

Relational operators are summarized in table 1.

Operator	Expression	Description	provides True if ...
>	a > b	larger than	a is larger than b.
<	a < b	smaller than	a is smaller than b.
==	a == b	equal	a and b have the same value.
!=	a != b	unequal	a and b have unequal values.
>=	a >= b	larger or equal	a is larger than or equal to b.
<=	a <= b	smaller or equal	a is smaller than or equal to b.

Table 1: Relational operators in Python.

2.2 Logical operators

Logical operators concatenate atomic propositions. Thereby, **truth values** are compared. The result is also a **truth value**, i.e. **True** or **False**. Since these are the operands of the Boolean propositional logic, their data type is **Boolean**.

The logical operators used in Python are presented in table 2.

Operator	Expression	provides True if...
not	not a	a is wrong (NOT).
and	a and b	both a and b are True (AND).
or	a or b	at least a or b are True (OR).

Table 2: Logical operators in Python.

3 Branches

Branches verify a state of the program. Depending on whether a condition is true or not, the program continues with different blocks of commands. In Python, as in many other programming languages, branches are initiated by the keyword `if`. The meaning of `if` is the same as in the natural language.

If it is raining, then I will take the bus, otherwise I will walk.

This could be written in Python as follows:

```
if rain == True:
    bus
else:
    walk
```

If the condition rain is **True**, the block containing the command `bus` is executed, otherwise the block with the command `walk` is executed.

Generally, it can be decided at run time whether a command or command block should be executed or not by means of an if-command. To formulate conditions, Boolean variables, relations such as equality, larger or smaller as well as logical operators can be used.

Depending on the number of cases to be distinguished, a **one-sided**, **two-sided** or **multilevel branch** needs to be chosen.

3.1 One-sided branch: conditional program execution

A **one-sided** branch consists of a condition check and a command block which is executed or not.

Notation:

```
if condition:
    Command block
```

Example:

```
if rain == True:
    print("It is raining. ")
```

The sentence `"It is raining. "` is only output if the variable `rain` has the value *True*.

3.2 Two-sided branch

In a **two-sided branch**, we can also specify what should be executed in the other case (*else*), i.e. if the condition does not apply.

Notation:

```
if condition:
    Command block 1
else:
    Command block 2
```

Example:

```
if rain == True:
    print("It is raining. ")
else:
    print("It is not raining. ")
```

The sentence `"It is raining. "` is output if the variable `rain` has the value *True* and the sentence `"It is not raining. "` is output in the other case (*False*).

An `if...else...` with an empty `else` block is the same as an `if...` (see one-sided branch).

3.3 Multilevel branches

By using a **multilevel branch**, several comparisons can be performed. This can be necessary for checking different possibilities in a certain order.

Notation:

```
if condition 1:
    Command block 1

elif condition 2:
     Command block 2

elif condition 3:
     Command block 3

else:
     Command block 4
```

Example:

```
if rain == True:
    print("It is raining. ")

elif snow == True:
    print("It is snowing. ")

elif sun == True:
    print("The sun is shining. ")

else:
    print("The weather conditions are not clear. ")
```

The sentence `"It is raining. "` is output if the variable `rain` has the value *True*. If it has the value *False*, the variable `snow` is checked in a next step. If the variable `snow` has the value *True*, the sentence `"It is snowing. "` is output. If it has the value *False*, the variable `sun` is checked in a next step. If the variable `sun` has the value *True*, the sentence `"The sun is shining. "` is output. If it has the value *False*, the sentence `"The weather conditions are not clear. "` is output.

Note that in an `elif` structure, only a single branch can be taken at a time. So when executing the code above only once, it is impossible to print both `"It is raining. "` and `"The sun is shining"`.

3.4 Nested branches

Branches can be **nested** in each other as desired.

Example: The following example with nested `if-else` has the same result as the last example with `if-elif-else`:

```
if rain == True:
    print("It is raining.")
else:
    if snow == True:
        print("It is not raining, but it is snowing.")
    else:
        if sun == True:
            print("The sun is shining.")
        else:
            print("The weather conditions are not clear.")
```

4 Loops

Commands can be executed repeatedly by means of **loops**. As in other programming languages, there are different kinds of loops in Python. A loop consists of a loop head and a loop body. The loop body consists of the command block to be repeated. The loop head controls the loop. It indicates how often or under which conditions the commands of the loop body should be repeated.

4.1 for-loops

In a counter-controlled **for-loop**, a **counter variable** is counted in the loop head in a **range** from a **start value** to the **end value**.

Notation:

```
for counter variable in range(start value, end value):
    Command
```

Example:

The following command outputs the values 0 to 4 on the screen.

```
for i in range(0, 5):
    print(i)
```

First, the counter variable `i` is set to the start value `0`, the loop body is run through and `i` is increased by `1`. Then the loop is run through once again and `i` is again increased by `1`. The counter variable `i` runs through the *range* from `0` (included) to `5` (excluded).

Since the initial value is often set to 0, it can also be omitted in the instruction. The following notation thus results in the same screen output:

```
for i in range(5):
    print(i)
```

The range can optionally be further specified by a **step size**, The following program outputs the values 2, 4, 6, 8 on the screen.

```
for i in range(2, 10, 2):
    print(i)
```

4.2 while-loops

The frequency of instruction repetition is often difficult to predict because it depends on the events within the loop body. In this context, counter-controlled loops show their limitations. In contrast, **conditional loops** bypass the need for a counter and instead rely on a **condition** to determine the number of repetitions.

In the **while-loop**, the loop head checks a **condition**. If the condition is *True*, the loop body is executed. If the condition is *False*, the loop is aborted and the commands are no longer executed. It is important to check the condition again after each execution of the loop body. Within the loop body, the values must be changed so that the condition is finally fulfilled (**update**). Otherwise there is the danger of an endless loop, which leads to an infinite wait for the desired result.

Notation:

```
Initializing the variables
while condition:
    Command block
    Update
```

Example:

The following command outputs the values 1 to 4 on the screen.

```
i = 1
while i < 5:
    print(i)
    i += 1
```

We first initialize the variable `i` to 1. Then, at the beginning of each iteration of the loop, we check if `i` is less than 5. If this condition is *True*, indicating that `i` is indeed smaller than 5, we execute the loop body. However, if the condition is *False*, meaning that `i` is not smaller than 5, we terminate the loop. Regardless, after each iteration, we increment the value of `i` by 1.

4.3 Nested loops

When programming, we often want to **nest two loops**. This means that an outer loop controls an inner one.

This can be done as follows:

```
Outer loop:
    Inner loop:
        Command block
```

Example:

We can write a program to display the days and hours of a non-leap year using the following nested loop:

```
for days in range(1,366):
    for hours in range(0,24):
        print(days, hours)
```

The outer loop waits until the inner loop finishes its execution. Then, the run variable of the outer loop is incremented by 1, and the inner loop starts again from the beginning. The total number of iterations of the inner loop instructions is equal to the product of the number of iterations of the outer loop and the number of iterations of the inner loop.

The first three outputs are:

```
1 0
1 1
1 2
...
```

The last three outputs are:

```
...
365 21
365 22
365 23
```

4.4 Influence loop sequence

There are situations when you want to end a loop before the condition to break the loop occurs. Maybe you want to skip a part of the loop before continuing or you want to ignore a check completely. The keywords `break`, `continue` and `pass` let you influence the **control flow** of a loop.

Control flow refers to how a program's instructions are executed, determining the order and path of execution. It includes loops and conditionals that allow for making choices, repeating tasks, or skipping code. It's about controlling the program's flow based on conditions and decision points.

4.4.1 break-statement

The `break` statement ends the current loop immediately and goes directly to the line after the loop.

Example:

```
for a in range(0,4):
    if a == 2:
        break
    print(a)

# Output:
# 0
# 1
```

4.4.2 continue-statement

With the `continue` statement only the current iteration is suspended, the loop is not aborted but continued with the next iteration.

Example:

```
for a in range(0,4):
    if a == 2:
        continue
    print(a)

# Output:
# 0
# 1
# 3
```

4.4.3 pass-statement

The `pass` statement tells the program to ignore a condition and continue executing the loop. This is useful, for example, if you have not yet written a part of the program but want to test it anyway.

Example:

```
for a in range(0,4):
    if a == 2:
        pass
    print(a)

# Output:
# 0
# 1
# 2
# 3
```

5 Redundant and dead code

Dead code refers to lines of code that cannot be reached by any possible control flow, for example, because they are no longer needed. Dead code should be avoided, since it loads the memory unnecessarily and costs time to test and maintain the program. Also **redundant code**, which is superfluous in the further course of the program, should be avoided for the same reason.

Example: The following program contains redundant and dead code:

```
x = int(input("Input:"))
y = 10 * x          # y is no longer used.
                    # Redundant Code.
z = x * x
if z >= 0:
    print(x * x)
else:
    x += 1          # Line is never reached.
                    # Dead code.
```

6 Exception handling

A Python program terminates as soon as it encounters an error. An error can be a *syntax error* or a *exception error*. Syntax errors occur when the rules of the language are violated.

Example:

```
for i in range(1,4))

# Error message:
# SyntaxError: invalid syntax
```

An exception error occurs if a syntactically correct code still leads to an error message.

Example:

```
x = int(input("Enter an integer: "))
# input: 0.5

# Error message:
# ValueError: invalid literal for int()
```

The `try` and `except` block in Python is used to catch and handle **exceptions**. Python executes code after the `try` block as a "normal" part of the program. The code in the `except` block is the program's response to all exceptions.

Example:

```
try:
  x = int(input("Enter an integer:\n"))
except ValueError:
  print("Wrong input! This is not an integer")
  x = 0
print("Value of x:", x)

# Enter an integer: 0.5
# Wrong input! This is not an integer
# Value of x: 0
```