2 Flow control

Flow control uses logical variables and alternates the flow of the program depending on whether certain conditions are met. Flow control in Python has two forms: conditional statement and loops.

2.1 Boolean expressions

A **boolean expression** is an expression that returns a boolean value: `True` or `False`. In Python following comparison (relational) operators are available:

`<` less than,

`>` greater than,

`<=` less than or equal to,

`>=` greater than on equal to,

`==` equal to,

`!=` not equal to,

`in` membershib check.

Before comparison, numbers of different types are casted to a common type. Strings can also be compared – alphabetical order is a method of string comparison, uppercase letters come before the lowercase letters. It is common to convert all string variables before comparison (e.g. all uppercase or all lowercase, using methods `upper()` and `lower()`). The relational operators work also on lists and tuples. Python starts by comparing the first element of each sequence, and compares them until it finds elements that differs. It is not possible to compare different types (unless then can be implicitly casted to a common class) in one boolean expression.

The most common error when using relational operators is typing `=`, an assignment operator, instead of `==`.

2.2 Logical operators

There are three **logical operators** which enable modyfying the condition and joining more than one boolean expression:

and : `True`, if both conditions are `True`,

or : `True,` if at least one condition is `True`,

not : negates the condition, True when the negated condition is `False`.

The operands of the logical operators should be boolean expressions, but Python is not very strict – any nonzero number is interpreted as `True` (example 2.1).

Example 2.1. Logical operators without boolean expressions

```python
print(2 and True)   # results in True
```

Additionally, Python provides built-in functions that take a sequence of boolean values and return True if all (`all()`) or any (`any()`) of the values are True (example 2.2). Both of the functions work on lists.

Example 2.2. Boolean value of a sequence of boolean values

```python
print(any([True, False, True, True])) # True if at least one value is True
print(all([True, False, True, True])) # True if all values True
```

Boolean operators have lower precedence than the bits of code they are comparing. This means, that the expressions are calculated first and then compared. If in doubt, the easiest way to avoid confusion about the precedence is adding parentheses (Example 2.3).

Example 2.3. Adding parentheses to avoid confusion with operator precedence

```python
5 < x and x < 20 # first, the interpreter checks whether x is greater than five and smaller than
    twenty, then it combines the results with 'and' operator
(5 < x) and (x < 20)  # same as above, but with parentheses for clearance
```

Multiple comparisons with one variable can be shortened, as in Example 2.4.

Example 2.4. Shorter version of multiple comparison

```python
5 < x and x < 20 # two comparisons
5 < x < 20  # same as above
```

2.3 Indentation

Every block of code is signalized with new **indentation** level. **Indentation** is the empty space left to the code. All statements indented the same distance are in the same block of code – a single block of code aligns vertically. The block ends when a less indented line is encountered. Block and statement boundaries are detected automatically, based on the indentation used (no `begin/end` or braces are needed). **Indentation is part of the Python syntax.** More deeply nested blocks are indented further to the right. The top-level code must start in column 1 (no indention).

Indentation may consist of any number of spaces or tabs as long as it is the same number for a single block (consistency!). Usually, indentation of 4 spaces (or a tab) is used. Each line, that represents some alternative flow of the program must be indented. However, do not mix the two (tabs and spaces), stick to one. Some text editors and IDEs allow to insert spaces when a `Tab` key is used; that ensures the consistency of the white space in a file.

A header line, which ends with a colon (:) requires and indented block of code.

2.4 Conditional execution

In order to write useful code, we need to be able to change the behaviour of a program according to some conditions. Using **conditional statements** enables to modify the flow of the program depending on the value of the **condition**. The condition is the boolean expression after a `if` keyword (example 2.5). If it is true, the indented block of code runs, if it is false – nothing happens, the statements following the `if`-header are skipped. You do not need to enclose a condition in parentheses.

Example 2.5. Conditional statement

```python
if x < 0:
    print("x is negative")
```

2.4.1 Alternative execution

A second form of an `if` statement is **alternative execution** – two possibiities are given and dependent on the value of the condition, true or false, only one of them runs (example 2.6). If the condition is true, the first statement following `if`-header runs. If it is false, the block of code following `else` executes. The alternatives are called **branches**.

Example 2.6. Alternative condition

```
if x < 0:
    print("x is negative")
else:
    print("x is not negative")
```

A single-line version of a conditional statement with alternative execution, a so-called **ternary expression** is presented in example 2.7. This version enables executing only one command if the boolean expression is `True` and, alternatively, only one statement if the expression is `False`.

Example 2.7. Single line version of a conditional statement

```
# x = value_if_true if condition else command_if_false
sign = 1 if x > 0 else -1
```

2.4.2 Chained conditionals

If more than one alternative is needed, they can be included in a **chained conditional** (example 2.8). Any number of alternative conditions can be defined using an `elif` statement. `elif` is short for **else if**. Again, only one branch will run (Figure 2.1a). `else` clause signalises the end of conditional statement (no `elif` can follow), but there does not have to be one. Conditions are checked in order, from top to bottom. If the first one is false, then the next is checked, until the true value is found, the corresponding code runs and the conditional statement ends. Even if more than one condition is true, **only the first found true branch runs**. If none of the conditions is true, then the `else` branch, if defined, runs.

Example 2.8. Chained conditions

```
if x < 0:
    print("x is negative")
elif x == 0:
    print("x is equal to 0")
else:
    print("x is positive")
```

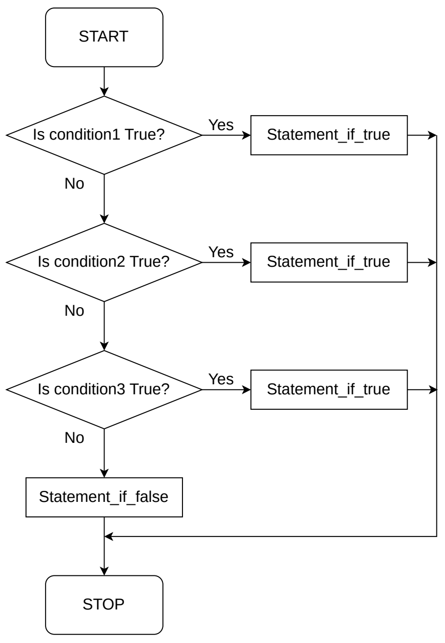
The **multiway branching** is provided with a series of `if/elif` tests – there is no additional `switch` or `case` command known from other programming languages (C, Pascal, etc.).

2.4.3 Nested conditionals

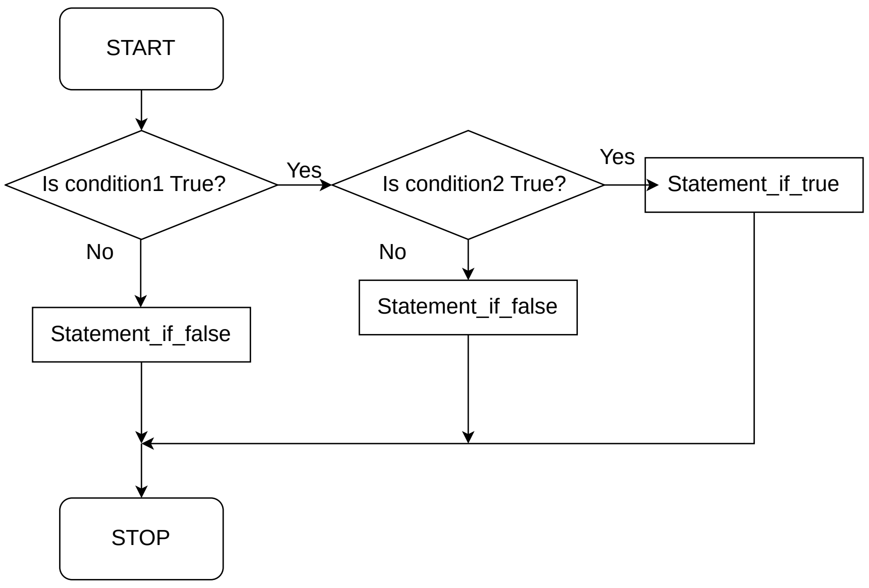
A conditional statement can be nested within another (Figure 2.1b). In example 2.9, the outer conditional contains two branches: the first branch contains another condition, and the second – a single statement.

Example 2.9. Nested conditional

```
if x > 0:
    if x < 10:
        print("x is single-digit positive number")
     else:
        print("x has at least two digits")
else:
     print("x is negative")
```



(a) chained conditional



(b) nested conditional

Figure 2.1. Flowcharts of complex conditional statements

The indentation of the branches makes it clear which statements belong to which branch, however, nested conditionals can become difficult to read and should be avoided if possible. Instead, using logical operators can simplify the structure. Example 2.9 can be rewritten as in example 2.10.

Example 2.10. Flattenig the nested conditional

```python
if 0 < x < 10:      # available in Python, usually requires a form: (0 < x) and (x < 10)
    print("x is single-digit positive number")
elif x >= 10:
    print("x has at least two digits")
else:
    print("x is negative")
```

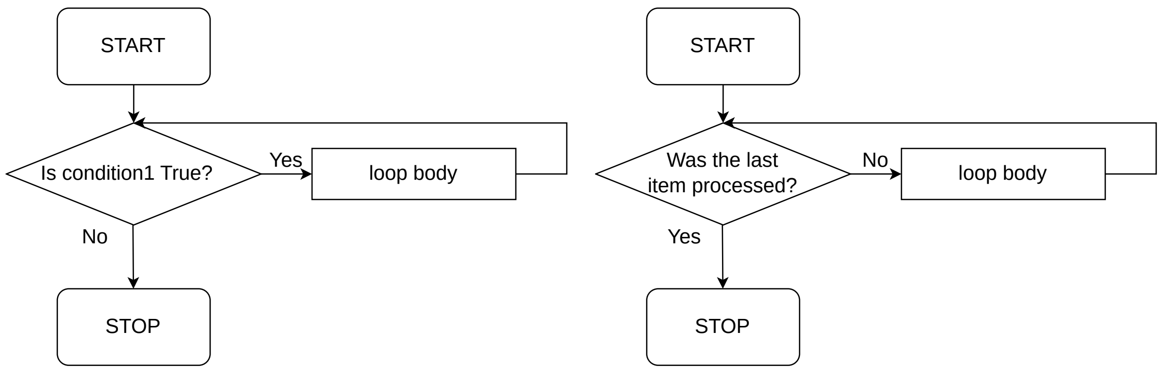
2.5 Iteration

Sometimes it is necessary for a program to repeat a set of instructions for the same or different set of variables. Copying the instructions many times has very little sense and is prone to error, that's when loops come into play. Computers are very good in repeating identical tasks without making errors. Such repetition is called **iteration**. **Iteration** is the ability to run a group of statements repeatedly.

Python provides two features to make iterations easier – one is a `while` statement, and the other is called a `for` loop.

2.5.1 `while` loop

The `while` loop executes a block of code if the specified condition is true (Figure 2.2a) (so a condition needs to be a boolean expression). After the block execution, the condition is evaluated again. If it is still true, the instructions are executed again. The process is continued until the condition becomes `False`. A loop in example 2.11 will run 10 times – until `i` will have a value equal to 10 and condition cease being



(a) `while` loop **(b)** `for` loop

Figure 2.2. Flow charts of loops available in Python

true. If a condition is false, the indented statements are not executed and the next statement after a `while` loop runs.

Example 2.11. While statement

```python
i = 0
print("Counting to 10!")
while i < 10:
    print(i)
    i = i + 1
```

The body of a loop should change the values of one or more variables so that the condition state can change (from true to false) and the loop can terminate. Otherwise the loop will repeat forever in an **infinite loop**. If the specified condition is always `True` and cannot be modified even if a variable inside a function changes, e.g. `while 1`, the loop will run indefinitely.

Example 2.12. While loop with else clause

```python
i = 0
print("Counting to 10!")
while i < 10:
        i = i + 1
        print(i)
else:
        print("10! Stop counting!") # some post-code
```

`else` clause can be used to define a block of code that will be executed if the condition is false (example 2.12). It can be used to catch the ‘other’ (alternate) way out of the loop without using flags or checking conditions. However, the `else` part will be executed also after the `True` part was running. The `else` part is optional and not often used.

2.5.2 `for` loop

`for` loop is usually used to repeat a block of statements a previously known number of times or to iterate through a sequence variable (an iterable variable) (Figure 2.2b):

`for target in sequence:  block`

Looping in Python with a `for` loop is a bit different than in other languages, a role of loop iterator variable is replaced with a sequence variable – a variable that can be iterated, e.g., a list, a tuple, a set, a dictionary or a range object.

else clause can be added to a for loop, the instruction defined will be executed after the for loop has finished.

2.5.3 break statement

The break statement is used to immediately exit the loop. The code that follows is not executed if the break is reached. Sometimes it is not clear when it is time to end a loop, until we get several lines into the body of a loop. Any loop can be interrupted with a break statement. In example 2.13, a loop is an infinite loop and will not terminate unless it hits the break statement. If the break condition was met that statement in a loop that is followed by else clause, that clause is not executed (example 2.14).

Example 2.13. break statement

```
i = 0
while 1:
    i += 1
    if i == 10:
        break    # stop loop when i reaches a value of 10
```

Example 2.14. break statement with else clause

```
list = ['Jack', 'Jill', 'Tim', 'Dave']
name = input('Type a name: ')

for i in range(len(list)):
    if list[i] == name:
        print(name,'is number',i + 1,'on the list')
        break
else:
    print(name,'is not on the list')
```

This way of writing loops is quite common, because the condition can be checked anywhere in the loop and the stop condition can be expressed affirmatively ("stop when something happens") rather than negatively ("keep going until something happens").

2.5.4 continue statement

The continue statement allows to skip an iteration when a certain conditon is met (example 2.15) and returning immediately to the beginning of the next iteration (top of the loop).

Example 2.15. continue statement

```
x = []
for i in range(10):
    if i%2 == 0:    # if divisible by 2, skip
        continue
    x.append(i)    # add an element to a list
```

2.6 Empty blocks of code

Empty blocks of code are not permitted – they must contain at least one statement. However, we can use a placeholder, pass, if a statement is syntatically required. The pass statement performs no action (example 2.16). The block of code can be filled out later in the programming process.

Example 2.16. pass statement

```python
if x < 0:
   print("x is negative")
elif x == -2:
   pass # we want the code to do nothing in this case
else:
   pass    # not sure what the code will be doing in that case, nothing happens
```

An ellipsis (. . .) can also be used as an alternative to `pass` (in Python 3.X version, but not 2.X) (example 2.17).

Example 2.17. Using ellilpsis to do nothing

```python
if x < 0:
   print("x is negative")
elif x == -2:
   ... # we want the code to do nothing in this case
```

2.7 Iterations vs accuracy

Throughout this course, loops will be used to compute roots of the function or solutions to linear equations. Usually, the number of steps needed to reach the right answer will not be known in advance – loop will run until a solution is found or rather, until the estimate will stop changing. So, it seems natural, that one would want to check, whether the current estimation and previous estimation are the same using an equality operator (`y == x`) (example 2.18). However, testing float equality is very risky and definitely not recommended. Floating-points are only approximately correct, most of the numbers, like 1/3, cannot be represented exactly with a float.

Example 2.18. Exit condition for float equality

```python
while True:
   print(x)
   y = (x + a/x) / 2
   if y == x:
      break
   x = y
```

Instead of testing equality, it is safer to test if the result has acceptable accuracy or magnitude. In other words, we check if the difference between the previous and current estimate is lower than the accepted difference level, ε (example 2.19).

Example 2.19. Exit condition for float equality

```python
epsilon = 0.000001
while True:
   print(x)
   y = (x + a/x) / 2
   if abs(y-x) < epsilon:
      break
   x = y
```

2.8 Exercises

Exercise 2.1. Sort three numbers `a`, `b` and `c` defined by the user in an ascending order.

Exercise 2.2. Given three values, `a`, `b` and `c`, containing the lengths of the edges, check if these values can create a triangle. If possible, decide what kind of triangle it is.

Exercise 2.3. Compute the roots of a quadratic equation ($P(x) = 0$), base on the polynomial coefficients 'a', 'b' and 'c' given by the user (or defined in code):

$$P(x) = ax^2 + bx + c$$

Exercise 2.4. Execute calculations for Celsius-Fahrenheit for every value in `0, 10, 20, ..., 100`.

Exercise 2.5. Write a `while` loop creating a list of odd numbers from 1 to n.

Exercise 2.6. Find prime numbers lower than 100.

2.9 Useful links

[R] – recommended read, [A] – additional read;

- [R] Control flow: `https://docs.python.org/3/tutorial/controlflow.html`
- [R] Conditionals (Think Python): `http://greenteapress.com/thinkpython2/html/thinkpython2006.html`
- Flow control Cheat sheet:
  - `https://www.codecademy.com/learn/learn-python-3/modules/learn-python3-loops/cheatsheet`
  - `https://www.codecademy.com/learn/learn-python-3/modules/learn-python3-control-flow/cheatsheet`
- `https://python.swaroopch.com/control_flow.html`

References

Ceder, N. (2018). *The quick Python book*. Simon and Schuster.

Downey, A. B. (2016). *Think Python. How to Think Like a Computer Scientist*. O'Reilly Media, 2nd edition.

Lubanovic, B. (2014). *Introducing Python: Modern Computing in Simple Packages*. O'Reilly Media, Inc.

Lutz, M. (2013). *Learning Python*. O'Reilly Media, 5th edition.

Python (2020). Python 3.8.5 documentation. `https://docs.python.org/3/` [Accessed 10 December 2019].