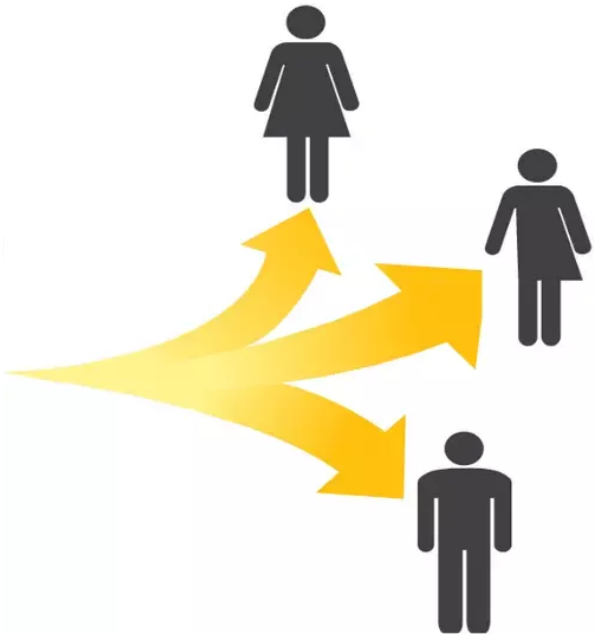
CONDITIONAL STATEMENTS

DECISION MAKING



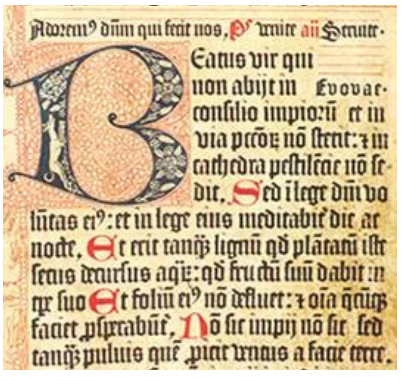
Do you belong to those people who find it hard to make decisions? In real life, I mean. In certain circumstances, some decisions in daily life are unavoidable, e.g. people have to go their separate ways from time to time, as our picture shows. We all have to make decisions all the time, ranging from trivial issues like choosing what to wear in the morning, what to have for breakfast, right up to life-changing decisions like taking or changing a job, deciding what to study, and many more. However, there you are, having decided to learn more about the conditional statements in Python.

Conditions - usually in the form of if statements - are one of the key features of a programming language, and Python is no exception. You will hardly find a programming language without an if statement.[1] There is hardly a way to program without branches in the code flow, at least if the code needs to solve a useful problem.

A decision must be made when the script or program comes to a point where there is a selection of actions, i.e. different calculations from which to choose.

The decision, in most cases, depends on the value of variables or arithmetic expressions. These expressions are evaluated using the Boolean True or False values. The instructions for decision making are called conditional statements. In a programming language, therefore, the parts of the conditional statements are executed under the given conditions. In many cases there are two code parts: One which will be executed, if the condition is True, and another one, if it is False. In other words, a branch determines which of two (or even more) program parts (alternatives) will be executed depending on one (or more) conditions. Conditional statements and branches belong to the control structures of programming languages, because with their help a program can react to different states that result from inputs and calculations.

CONDITIONAL STATEMENTS IN REAL LIFE



The principle of the indentation style is also known in natural languages, as we can deduce from the following text:

```
 If it rains tomorrow, I'll clean up the basemen
t.
 After that, I will paint the walls. If there is
 any time left, I will file my tax return.
```

Of course, as we all know, there will be no time left to file the tax return. Jokes aside: in the previous text, as you see, we have a sequence of actions that must be performed in chronological order. If you take a closer look at the text, you will find some ambiguity in it: is 'the painting of the walls' also related to the event of the rain? Will the tax declaration be done with or without rain? What will this person do, if it does not

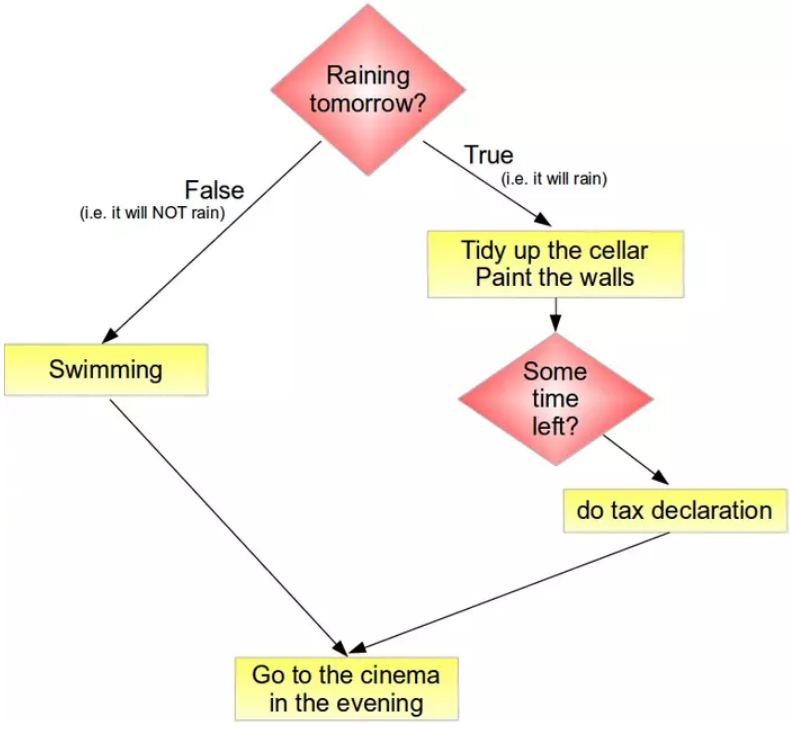
rain? Therefore, we extend the text with additional alternatives:

```
If it rains tomorrow, I'll clean up the basement.
After that I will paint the walls. If there is
any time left, I will file my tax return. Otherwise,
i.e. if it will not rain, I will go swimming.
In the evening, I will go to the cinema with my wife!
```

It is still nor clear. Can his wife hope to be invited to the cinema? Does she have to pray or hope for rain? To make the text clear, we can phrase it to be closer to the programming code and the Python coding style, and hopefully there will be a happy ending for his wife:

```
If it rains tomorrow, I will do the following:
     - Clean up the basement
     - Paint the walls
     - If there is any time left, I will make my
       tax declaration
Otherwise I will do the following:
     - Go swimming
Going to the movies in the evening with my wife
```

We can also graphically depict this. Such a workflow is often referred to in the programming environment as a so-called flowchart or program schedule (PAP):



To encode conditional statements in Python, we need to know how to combine statements into a block. So this seems to be the ideal moment to introduce the Python Block Principle in combination with the conditional statements.

COMBINING STATEMENTS INTO BLOCKS

In programming, blocks are used to treat sets of statements as if they were a single statement. A block consists of one or more statements. A program can be considered a block consisting of statements and other nested blocks. In programming languages, there exist various approaches to syntactically describing blocks: ALGOL 60 and Pascal, for example, use "begin" and "end", while C and similar languages use curly braces "{" and "}". Bash has another design that uses 'do ... done' and 'if ... fi' or 'case ... esac' constructs. All these languages have in common is that the indentation style is not binding. The problems that may arise from this will be shown in the next subchapter. You can safely skip this, if you are only interested in Python. For inexperienced programmers, this chapter may seem a bit difficult.

EXCURSION TO C

For all these approaches, there is a big drawback: The code may be fine for the interpreter or compiler of the language, but it can be written in a way that is poorly structured for humans. We want to illustrate this in the following "pseudo" C-like code snippet:

```
if (raining_tomorrow) {
    tidy_up_the_cellar();
    paint_the_walls();
    if (time_left)
          do_taxes();
} else
    enjoy_swimming();
go_cinema();
```

We will add a couple of spaces in front of calling 'go_cinema'. This puts this call at the same level as 'enjoy_swimming':

```
if (raining_tomorrow) {
    tidy_up_the_cellar();
    paint_the_walls();
    if (time_left)
          do_taxes();
} else
    enjoy_swimming();
    go_cinema();
```

The execution of the program will not change: you will go to the cinema, whether it will be raining or not! However, the way the code is written falsely suggests that you will only go to the cinema if it does not rain. This example shows the possible ambiguity in the interpretation of C code by humans. In other words, you can inadvertently write the code, which leads to a procedure that does not "behave" as intended.
In this example lurks even another danger. What if the programmer had just forgotten to put the last statement out of the curly braces?

Should the code look like this:

```
if (raining_tomorrow) {
    tidy_up_the_cellar();
    paint_the_walls();
    if (time_left)
          do_taxes();
} else {
    enjoy_swimming();
    go_cinema();
}
```

In the following program you will only go to the cinema, if it is not a rainy day. That only makes sense if it's an open-air cinema.

The following code is the right code to go to the movies regardless of the weather:

```
if (raining_tomorrow) {
    tidy_up_the_cellar();
    paint_the_walls();
    if (time_left)
          do_taxes();
} else {
    enjoy_swimming();
}
go_cinema();}
```

The problem with C is that the way the code is indented has no meaning for the compiler, but it can lead to misunderstanding, if misinterpreted.

This is different in Python. Blocks are created with indentations, in other words, blocks are based on indentation. We can say "What you see is what you get!"
The example above might look like this in a Python program:

```
if raining_tomorrow:
    tidy_up_the_cellar()
    paint_the_walls()
    if time_left:
          do_taxes()
else:
    enjoy_swimming()
go_cinema()
```

There is no ambiguity in the Python version. The cinema visit of the couple will take place in any case, regardless of the weather. Moving the go_cinema() call to the same indentation level as enjoy_swimming() immediately changes the program logic. In that case, there will be no cinema if it rains.

We have just seen that it is useful to use indentations in C or C++ programs to increase the readability of a program. For the compilation or execution of a program they are but irrelevant. The compiler relies exclusively on the structuring determined by the brackets. Code in Python, on the other hand, can only be structured with indents. That is, Python forces the programmer to use indentation that should be used anyway to write nice and readable codes. Python does not allow you to obscure the structure of a program by misleading indentations.

Each line of a code block in Python must be indented with the same number of spaces or tabs. We will continue to delve into this in the next subsection on conditional statements in Python.

CONDITIONAL STATEMENTS IN PYTHON

The if statement is used to control the program flow in a Python program. This makes it possible to decide at runtime whether certain program parts should be executed or not.
The simplest form of an if statement in a Python program looks like this:

```
if condition:
    statement
```

```
        statement
        # further statements, if necessary
```

The indented block is only executed if the condition 'condition' is True. That is, if it is logically true.

```
person = input("Nationality? ")
if person == "french":
    print("Préférez-vous parler français?")
```

```
Nationality? french
Préférez-vous parler français?
```

Please note that if "French" is entered in the above program, nothing will happen. The check in the condition is case-sensitive and in this case only responds when "french" is entered in lower case. We now extend the condition with an "or"

```
person = input("Nationality? ")
if person == "french" or person == "French":
    print("Préférez-vous parler français?")
```

```
Nationality? french
Préférez-vous parler français?
```

The Italian colleagues might now feel disadvantaged because Italian speakers were not considered. We extend the program by another if statement.

```
person = input("Nationality? ")
if person == "french" or person == "French":
    print("Préférez-vous parler français?")
if person == "italian" or person == "Italian":
    print("Preferisci parlare italiano?")
```

```
Nationality? italian
Preferisci parlare italiano?
```

This little script has a drawback. Suppose that the user enters "french" as a nationality. In this case, the block of the first "if" will be executed. Afterwards, the program checks if the second "if" can be executed as well. This doesn't make sense, because we know that the input "french" does not match the condition "italian" or "Italian". In other words: if the first 'if' matched, the second one can never match as well. In the end, this is an unnecessary check when entering "french" or "French".

We solve the problem with an "elif" condition. The 'elif' expression is only checked if the expression of a previous "elif" or "if" was false.

```
person = input("Nationality? ")
if person == "french" or person == "French":
    print("Préférez-vous parler français?")
elif person == "italian" or person == "Italian":
    print("Preferisci parlare italiano?")
else:
    print("You are neither French nor Italian.")
    print("So, let us speak English!")
```

```
Nationality? german
You are neither French nor Italian.
So, let us speak English!
```

As in our example, most if statements also have "elif" and "else" branches. This means, there may be more than one "elif" branches, but only one "else" branch. The "else" branch must be at the end of the if statement. Other "elif" branches can not be attached after an 'else'. 'if' statements don't need either 'else' nor 'elif' statements.

The general form of the if statement in Python looks like this:

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2

...

elif another_condition:
    another_statement_block
else:
    else_block
```

If the condition "condition_1" is True, the statements of the block statement"_block_1" will be executed. If not, condition_2 will be evaluated. If "condition_2" evaluates to True, statement"_block_2" will be executed, if condition_2 is False, the other conditions of the following elif conditions will be checked, and finally if none of them has been evaluated to True, the indented block below the else keyword will be executed.

EXAMPLE WITH IF CONDITION

For children and dog lovers it is an interesting and frequently asked question how old their dog would be if it was not a dog, but a human being. To calculate this task, there are various scientific and pseudo-scientific approaches. An easy approach can be:

```
 A one-year-old dog is roughly equivalent to a 14-year-old human bein
g
 A dog that is two years old corresponds in development to a 22 year
old person.
 Each additional dog year is equivalent to five human years.
```

The following example program in Python requires the age of the dog as the input and calculates the age in human years according to the above rule. 'input' is a statement, where the program flow stops and waits for user input printing out the message in brackets:

```
age = int(input("Age of the dog: "))
print()
if age < 0:
    print("This cannot be true!")
elif age == 0:
    print("This corresponds to 0 human years!")
elif age == 1:
    print("Roughly 14 years!")
elif age == 2:
    print("Approximately 22 years!")
else:
    human = 22 + (age -2) * 5
    print("Corresponds to  " + str(human) + " human years!")
```

```
Age of the dog: 1

Roughly 14 years!
```

In [ ]:
```
We will use this example again in our tutorial
```

Using if statements within programs can easily lead to complex decision trees, i.e. every if statements can be seen like the branches of a tree.

We will read in three float numbers in the following program and will print out the largest value:

```
x = float(input("1st Number: "))
y = float(input("2nd Number: "))
z = float(input("3rd Number: "))

if x > y and x > z:
    maximum = x
elif y > x and y > z:
    maximum = y
else:
    maximum = z

print("The maximal value is: " + str(maximum))
```

```
1st Number: 1
2nd Number: 2
3rd Number: 3
The maximal value is: 3.0
```

There are other ways to write the conditions like the following one:

```
x = float(input("1st Number: "))
y = float(input("2nd Number: "))
z = float(input("3rd Number: "))

if x > y:
    if x > z:
        maximum = x
    else:
        maximum = z
else:
    if y > z:
        maximum = y
    else:
        maximum = z

print("The maximal value is: " + str(maximum))
```

```
1st Number: 4.4
2nd Number: 1
3rd Number: 8.3
The maximal value is: 8.3
```

Another way to find the maximum can be seen in the following example. We are using the built-in function max, which calculates the maximum of a list or a tuple of numerical values:

```
x = float(input("1st Number: "))
y = float(input("2nd Number: "))
z = float(input("3rd Number: "))

maximum = max((x,y,z))

print("The maximal value is: " + str(maximum))
```

```
1st Number: 1
2nd Number: 4
3rd Number: 5.5
The maximal value is: 5.5
```

TERNARY IF STATEMENT

Let's look at the following English sentence, which is valid in Germany :

The maximum speed is 50 if we are within the city, otherwise it is 100.

This can be translated into ternary Python statements:

```
inside_city_limits = True
maximum_speed = 50 if inside_city_limits else 100
print(maximum_speed)
```

```
50
```

This code looks like an abbreviation for the following code:

```
im_ort = True
if im_ort:
    maximale_geschwindigkeit = 50
else:
    maximale_geschwindigkeit = 100
print(maximale_geschwindigkeit)
```

```
50
```

But it is something fundamentally different. "50 if inside_city_limits else 100" is an expression that we can also use in function calls.

EXERCISES

EXERCISE 1

- A leap year is a calendar year containing an additional day added to keep the calendar year synchronized with the astronomical or seasonal year. In the Gregorian calendar, each leap year has 366 days instead of 365, by extending February to 29 days rather than the common 28. These extra days occur in years which are multiples of four (with the exception of centennial years not divisible by 400). Write a Python program, which asks for a year and calculates, if this year is a leap year or not.

EXERCISE 2

- Body mass index (BMI) is a value derived from the mass (weight) and height of a person. The BMI is defined as the body mass divided by the square of the body height, and is universally expressed in units of kg/m2, resulting from mass in kilograms and height in metres. $ BMI = \frac{m}{l^{2}}$ Write a program, which asks for the length and the weight of a person and returns an evaluation string according to the following table:

Category	BMI (kg/m²)		BMI Prime	
	from	to	from	to
Very severely underweight		15		0.60
Severely underweight	15	16	0.60	0.64
Underweight	16	18.5	0.64	0.74
Normal (healthy weight)	18.5	25	0.74	1.0
Overweight	25	30	1.0	1.2
Obese Class I (Moderately obese)	30	35	1.2	1.4
Obese Class II (Severely obese)	35	40	1.4	1.6
Obese Class III (Very severely obese)	40	45	1.6	1.8
Obese Class IV (Morbidly Obese)	45	50	1.8	2
Obese Class V (Super Obese)	50	60	2	2.4
Obese Class VI (Hyper Obese)	60		2.4	

SOLUTIONS

EXERCISE 1:

We will use the modulo operator in the following solution. 'n % m' returns the remainder, if you divide (integer division) n by m. '5 % 3' returns 2 for example.

```
year = int(input("Which year? "))

if year % 4:
    # not divisible by 4
    print("no leap year")
elif year % 100 == 0 and year % 400 != 0:
    print("no leap year")
else:
    print("leap year")
```

```
Which year? 2020
leap year
```

EXERCISE 2:

```
height = float(input("What is your height? "))
weight = float(input("What is your weight? "))

bmi = weight / height ** 2
print(bmi)
if bmi < 15:
    print("Very severely underweight")
elif bmi < 16:
    print("Severely underweight")
elif bmi < 18.5:
    print("Underweight")
elif bmi < 25:
    print("Normal (healthy weight)")
elif bmi < 30:
    print("Overweight")
elif bmi < 35:
    print("Obese Class I (Moderately obese)")
elif bmi < 40:
    print("Obese Class II (Severely obese)")
else:
    print("Obese Class III (Very severely obese)")
```

```
What is your height? 180
What is your weight? 74
0.002283950617283951
Very severely underweight
```

FOOTNOTES

[1] LOOP is a programming language without conditionals, but this language is purely pedagogical. It has been designed by the German computer scientist Uwe Schöning. The only operations which are supported by LOOP are assignments, additions and loopings. But LOOP is of no practical interest and besides this, it is only a proper subset of the computable functions.