Chapter 2: Basic operations and conditional executions

Basic arithmetic operations

Mathematical operations are executed in an order as you get used to in mathematics.

See the precedence order of all Python operators (https://docs.python.org/3/reference/expressions.html#operator-precedence) in the documentation.
Operations with the same precedence are evaluated from left to right.
E.g. `1+2*3` is evaluated as `1+(2*3)` .

Summation

Both numeric and string values can be added together.

For numeric values it works like the mathematical operation, e.g.: `1+2=3` .

For string values they are concatenated, e.g.: `'Hello '+'world'='Hello world'` .

In [1]:

```
print(1+2)
print('Hello '+'world')
```

```
3
Hello world
```

In [2]:

```
x=10
y=20
print(x+y)

z='10'
q='20'
print(z+q)
```

```
30
1020
```

Subtraction

Works only for numeric values:

In [3]:

```
print(10-7)

x=20
print(x-10)
```

```
3
10
```

Multiplication

The multiplication operator can be applied both between 2 numeric values and between a string and a numeric value.

For numeric values it works like the mathematical operation, e.g.: `9*4=36`.

For a string and an integer, the string is repeated and concatenated as many times as we defined, e.g.: `'Hi'*5=HiHiHiHiHi`.

In [4]:

```
print(9*4)
print('Hi'*5)
```

```
36
HiHiHiHiHi
```

In [5]:

```
x=9
y=x*4
print(y)

z='Hi'
w=z*5
print(w)
```

```
36
HiHiHiHiHi
```

Division with floating result

Works only for numeric values.

In [6]:

```
print(17/3)
```

```
5.666666666666667
```

Question: what is the type of the dividend and the divisor? What is the type of the result?

Question: what will be the type of the result if the value is an integer?

In [7]:

```
print(type(17))
print(type(3))
print(type(17/3))
print(type(18/3))
```

```
<class 'int'>
<class 'int'>
<class 'float'>
<class 'float'>
```

Division with integer result

Using the double division operator (`//`) means that the result of the division will be an integer number. If the result has a fractional part, it is dropped.

In [8]:

```
print(18//3)
print(17//3)
```

```
6
5
```

Exponentiation

We can calculate the y^{th} power of x by using the double star (`*`) operator: `x**y` .

In [9]:

```
print(2**3)

x=3
y=4
print(x**y)
```

```
8
81
```

It is effectively the same as calling the `pow` function (the name is short for *power*) with 2 arguments:

In [10]:

```
print(pow(x, y))
```

```
81
```

Remainder (modulo operator)

In computing, the *modulo operation* finds the remainder after division of one number by another (called the *modulus* of the operation).

E.g. `17%3=2` , since 15 is divisible by 3 and the remainder is therefore 2.

Useful scenarios:

- check whether a number is divisible by another (the modulus must be 0);
- get the last digit of a number by calculating the remainder by 10.

In [11]:

```
print(17%3)
```

```
2
```

Summary exercises on basic operations

Exercise: Rectangle

Task: Calculate the area and the perimeter of a rectangle.

Get the width and the height of the rectangle from the user.

In [12]:

```
width = int(input("Width = "))
height = int(input("Height = "))
area = width * height
perimeter = 2 * (width + height)

print("Area = {0}".format(area))
print("Perimeter = {0}".format(perimeter))
```

```
Area = 840
Perimeter = 118
```

Exercise: Circle

Task: Calculate the area and the perimeter of a circle.

Get the radius of circle from the user.

In [13]:

```
import math

radius = float(input("Radius = "))
area = radius**2 * math.pi
perimeter = 2 * radius * math.pi

print("Area = {0}".format(area))
print("Perimeter = {0}".format(perimeter))
```

```
Area = 50.26548245743669
Perimeter = 25.132741228718345
```

Note: we can get a (finite) representation of pi using the `math.pi` constant after importing the `math` module:

In [14]:

```
import math
print(math.pi)
```

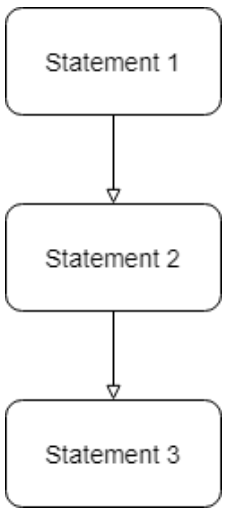
```
3.141592653589793
```

Control structures

There are 3 basic control flows for all *imperative* programming languages: **sequences**, **conditions** and **loops**.

Sequence

When operations are evaluated seqentially one after another, it is called a *sequence statement*.



In [15]:

```
print("First statement")
print("Second statement")
```

```
First statement
Second statement
```

So far, we have worked with sequences.

Conditions

Conditions (or also called *select statements*):

- define multiple branches of the program code;
- it is decided based on logical tests that which branch should be executed.

Two-way conditions

First lets read a number from the user:

In [16]:

```
number = input("Give a number: ")
print("Number is {0} with type of {1}".format(number, type(number)))
```

```
Number is 42 with type of <class 'str'>
```

Convert the `number` to an integer:

In [17]:

```
number = int(number)
print("Now number is now {0} with type of {1}".format(number, type(number)))
```

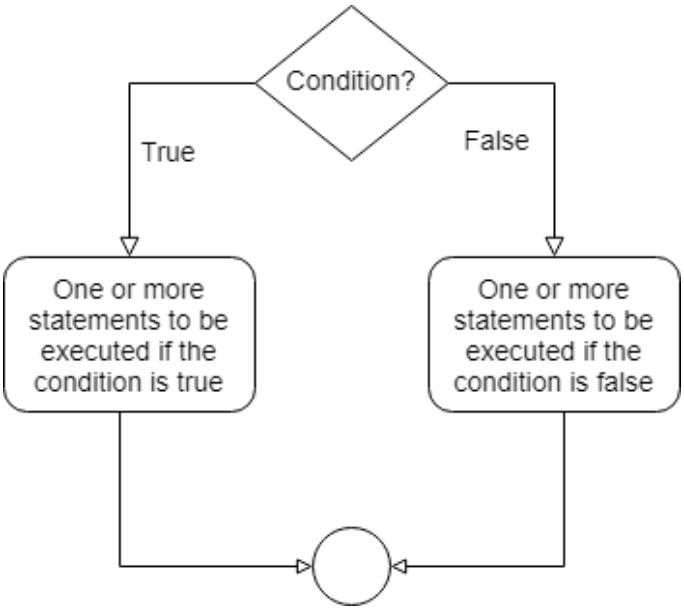
```
Now number is now 42 with type of <class 'int'>
```

Check whether the number is positive or not:

In [18]:

```
if number > 0:
    print("number is positive, its value is " + str(number))
else:
    print("number is non-positive, its value is " + str(number))
```

```
number is positive, its value is 42
```



IMPORTANT: in Python, the indentation of the code is crucial, because it defines the code blocks!

In [19]:

```
if number > 0:
    print("number is positive, its value is " + str(number))
else:
    print("number is non-positive, its value is " + str(number))
    print("Check when this line is printed")
```

```
number is positive, its value is 42
```

In [20]:

```
if number > 0:
    print("number is positive, its value is " + str(number))
else:
    print("number is non-positive, its value is " + str(number))
print("Check when this line is printed")
```

```
number is positive, its value is 42
Check when this line is printed
```

Indentation is done with *whitespace characters*: spaces and tabs. You can either use spaces or tabs to indent and you can decide how many of them you are using. (Typical values are indenting with 2 or 4 whitespaces.)

Note: a single tab is just 1 whitespace even if displayed as multiple in your text editor! **Therefore you shall not mix spaces and tabs when indenting, use only one of them!**

Logical operations

Boolean values and expressions can also be combined with the logical, binary `and` and `or` operators. Negation can be done with the unary `not` operator.

In [21]:

```
print("True and False is {0}".format(True and False))
print("True or False is {0}".format(True or False))
print("not True is {0}".format(not True))
```

```
True and False is False
True or False is True
not True is False
```

Just like with the arithmetic operators, there is also a precedence order for the logical operators: `not`, `and`, `or`. Use parentheses to "override" the default precedence order.

In [22]:

```
print("True or True and False is {0}".format(True or True and False))
print("(True or True) and False is {0}".format((True or True) and False))
```

```
True or True and False is True
(True or True) and False is False
```

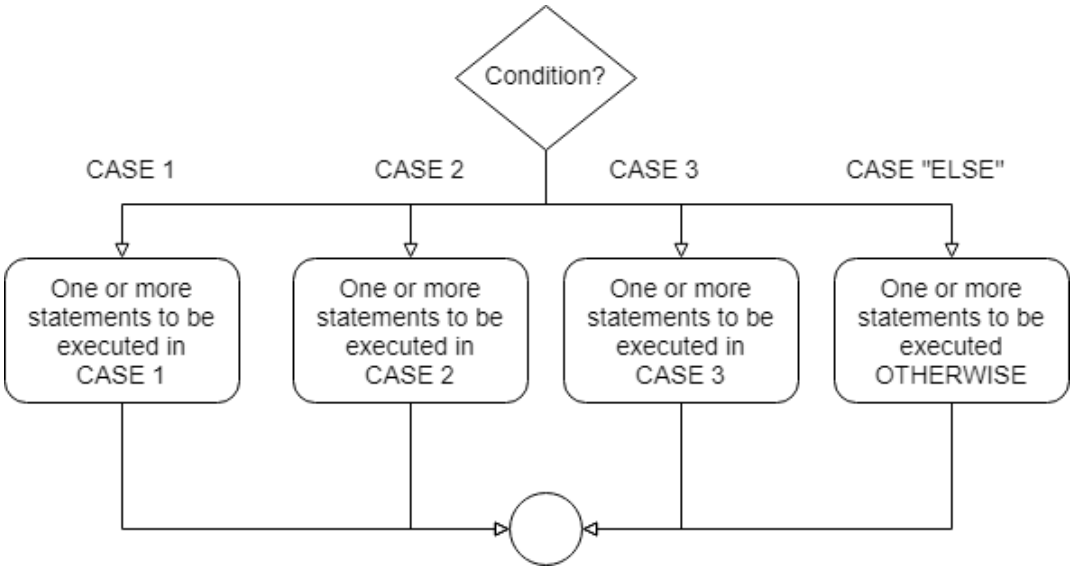
Three (or more) way conditions

We can define multiple logical expression (*elif*) to test. These conditions are tested in the order they are defined and the body for the **first one** to be *True* will be executed. We can still define an *else* branch in case none of the conditions was *True*.

In [23]:

```
if number > 0:
    print("number is positive, its value is " + str(number))
elif number < 0:
    print("number is negative, its value is " + str(number))
else:
    print("number is zero")
```

```
number is positive, its value is 42
```



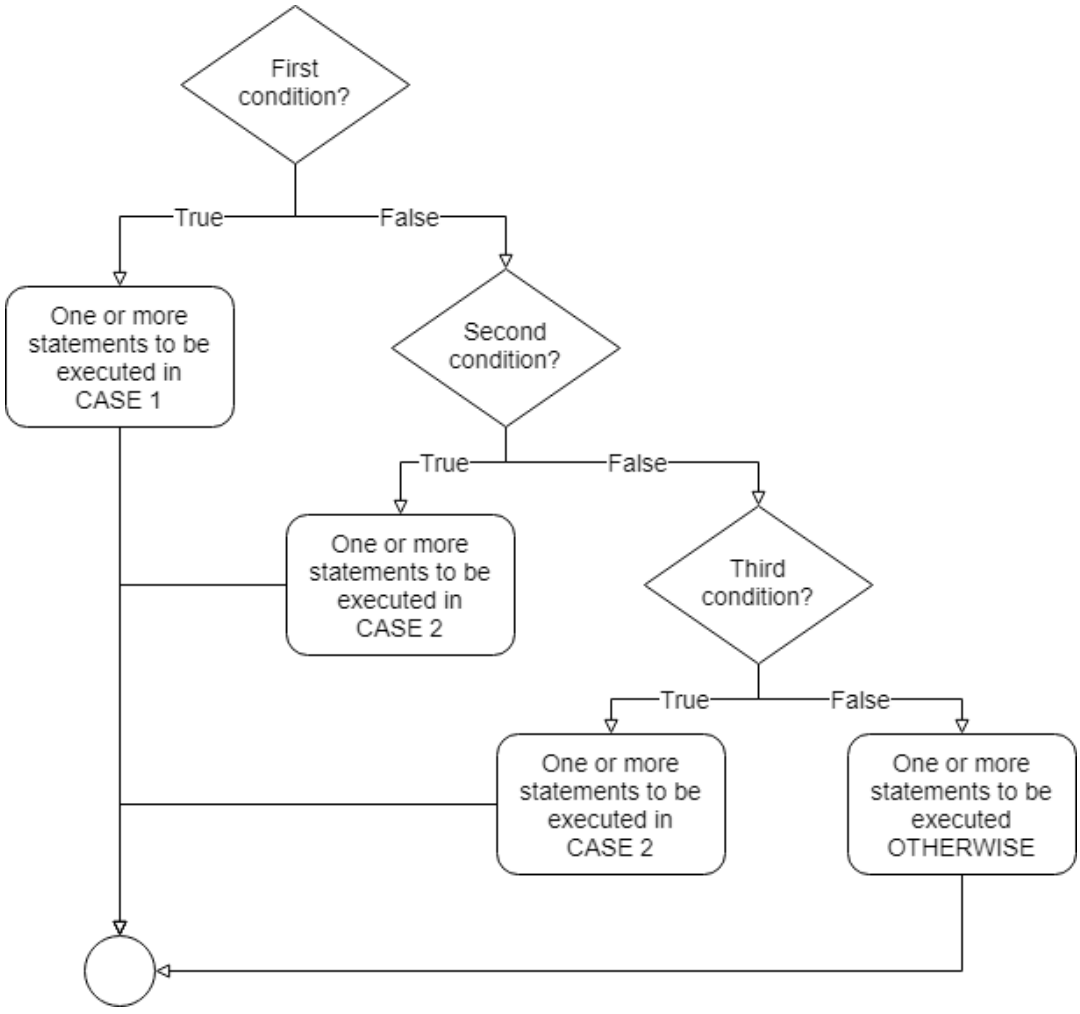
Actually, there are no three (or more) way conditional statements, only two-way conditions. The `elif` keyword is just a little *"syntax sugar"* to provide an easier understandable version of nested, 2-way conditions.

Task: Can you write the above 3-way condition with just 2-way conditions?

In [24]:

```
if number > 0:
    print("number is positive, its value is " + str(number))
else:
    if number < 0:
        print("number is negative, its value is " + str(number))
    else:
        print("number is zero")
```

```
number is positive, its value is 42
```



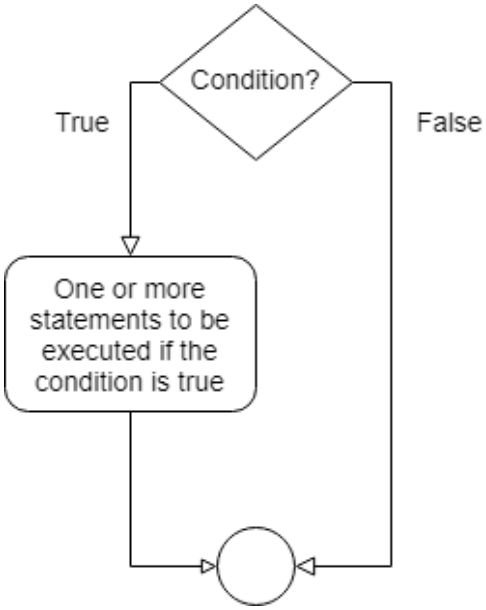
One-way conditions

You do not have to use `if-else` or `if-elif-...-else` . You can use `if` without other clauses following that. The *else* branch can be omitted if not required.

In [25]:

```
if number > 0:
    print("number is positive, its value is " + str(number))
```

```
number is positive, its value is 42
```



Comparison

Python syntax for comparison is the same as our hand-written convention:

1. Larger (or equal): `>` (>=)
2. Smaller (or equal): `<` (<=)
3. Equal to: `==` (**Note here that there are double equal signs**)
4. Not equal to: `!=`

In [26]:

```
print(3 == 5)
```

```
False
```

In [27]:

```
print(72 >= 2)
```

```
True
```

In [28]:

```
store_name = 'Auchan'
#store_name = 'Tesco'
```

In [29]:

```
print(store_name)
```

```
Auchan
```

In [30]:

```
print(store_name == "Tesco") # Will return a boolean value True or False
```

```
False
```

In [31]:

```
if store_name == 'Auchan':
    print("The store is an Auchan.")
else:
    print("The store is not an Auchan. It's " + store_name + ".")
```

```
The store is an Auchan.
```

Floating point comparison

IMPORTANT: Note that floating point precision and therefore comparisons between floating point numbers can be tricky.

What will these floating point mathematical operations result?

In [32]:

```
print(0.1 + 0.1 + 0.1)
print(0.1 + 0.1 + 0.1 == 0.3)

print(1.0 - 0.83)
print(1.0 - 0.83 == 0.17)

print(2.2 * 3.0)
print(2.2 * 3.0 == 6.6)

print(3.3 * 2.0)
print(3.3 * 2.0 == 6.6)
```

```
0.30000000000000004
False
0.17000000000000004
False
6.6000000000000005
False
6.6
True
```

In [33]:

```
a = 1000.0
b = 0.000000001
print(a + b == a)

a = 100000000.0
b = 0.000000001
print(a + b == a)
```

```
False
True
```

Therefore, calculated floating point numbers shall never be checked for precise equality, instead a small error threshold shall be allowed.

In [34]:

```
a = 2.2 * 3.0
b = 6.6
print(abs(a - b) < 1e-5) # 1e-5 == 10^-5 == 0.00001 (scientific number notation)
```

```
True
```

Theoretical background for floating point calculation errors *(optional)*

Signed integers representation

Signed integers are usually represented with the *Two's complement interpretation* (https://en.wikipedia.org/wiki/Two%27s_complement):

```
00000000 -> 0
00000001 -> 1
00000010 -> 2
00000101 -> 5
...
10000000 -> -128
10000001 -> -127
11111110 -> -2
11111111 -> -1
```

Floating point representation

Floating point numbers are usually represented according to the *standard IEEE-754* (https://en.wikipedia.org/wiki/IEEE_754).

Since hardware can only work with integers, numbers are represented in a form of `[mantissa|exponent]`, where $value = mantissa * 2^{exponent}$. Both mantissa and exponent are a *two's complement interpretation* of signed integers.

Example for converting a float representation to decimal value:

```
representation = [000000000101|11111100]
value = 5 * 2^-2 = 1.25
```

Example for converting a decimal value to float representation:

```
value = 179.375
binary value = 10110011.011
normal form = 0.10110011011 * 2^8
mantissa = 10110011011
exponent = 1000
representation = [010110011011|00001000]
```

Problem 1

The number base problem: not all numbers can be exactly represented in all bases. Neither 0.17 or 0.83 can be represented in base 2, so:

```
0.17000000000000001221245327087672194465994834899902343750 -> 0.0010101110
000101000111101011100001010001111010111000011
0.82999999999999996003197111349436454474925994873046875000 -> 0.1101010001
111010111000010100011110101110000101000111100

1.0 - 0.83 = 0.00101011100001010001111010111000010100011110101110000011
0.17       = 0.00101011100001010001111010111000010100011110101110000100
```

Problem 2

The floating point problem: all representations have restricted range by the exponent, performing operations on numbers with very large and small exponents could result in the ignorance of the smaller one, as it would be shifted out of range.

```
decimal = 100000000
binary  = 10111110101111000010000000
normal  = 0.10111110101111000010000000 * 2^27

decimal = 0.000000001
binary ~= 0.000000001101111101011100001000
```

Summary exercise on conditions

Exercise: Parity

Task: Decide whether an integer number is even or odd!

Request the number from the user.

In [35]:

```
number = int(input("Number to test: "))
if number % 2 == 0:
    print("{0} is even".format(number))
else:
    print("{0} is odd".format(number))
```

```
43 is odd
```

Exercise: Body Mass Index

Task: Calculate the Body Mass Index (BMI) of the user and categorize it.

The BMI is defined as the body mass (in kilogramms) divided by the square of the body height (in meters), and is universally expressed in units of kg/m^2.

$$BMI = \frac{Weight}{Height^2}$$

Request the weight and the height and calculate the BMI value for the user!

Categorize the user based on the BMI value:

Category	BMI value
Underweight	BMI < 18.5
Normal	18.5 <= BMI < 25
Overweight	25 <= BMI < 30
Obese	30 <= BMI

Note: this is just a simplified categorization.

In [36]:

```
weight_kg = int(input("Weight of the user (in kg): "))
height_cm = int(input("Height of the user (in cm): "))
height_m = height_cm / 100
bmi = weight_kg / (height_m**2)

print("BMI of the user is {0:.2f}".format(bmi))

if bmi < 18.5:
    print("Category: underweight")
elif bmi < 25:
    print("Category: normal")
elif bmi < 30:
    print("Category: overweight")
else:
    print("Category: obese")
```

```
BMI of the user is 23.99
Category: normal
```

Note how the conditions are tested in the order they are defined. The body of the first one to be *True* gets executed and the further ones are omitted.

Exception handling: Try and Except

Python code raises so called *exceptions* in exceptional cases, typically when an error occurred. We can use the `try-except` block to handle these errors (exceptions), so our code will not stop and abort because of the error.

E.g.: lets consider we would like to request a number from the user, but the user can type in any value, even a string. Then converting this string to an integer with `int()` would raise an exception. Not handling this exception will abort the program. By handling the exception we can print out an error message, set a default value or even request the number a second time.

Format:

- **TRY** block: look for exception to be raised in the code block.
- **EXCEPT** block: if an exception was detected, stop the execution of the *TRY* block at that point and continue with the *EXCEPT* block.

Example **without** exception handling:

In [37]:

```
age = input('What is your age?')
age = int(age)
print("The given age is: {0}".format(age))
```

```
---------------------------------------------------------------------
-------
ValueError                                Traceback (most recent cal
l last)
<ipython-input-37-8b0cdbac8dc1> in <module>
      1 age = input('What is your age?')
----> 2 age = int(age)
      3 print("The given age is: {0}".format(age))

ValueError: invalid literal for int() with base 10: 'Twenty'
```

Test what will happen if you type in a string instead of a number? Will the value of the `age` variable printed out?

Example **with** exception handling:

In [38]:

```
age = input('What is your age?')
try:
    age = int(age)
except:
    age = -1

print("The given age is: {0}".format(age))
```

```
The given age is: -1
```

Test again what will happen if you type in a string instead of a number? Will the value of the `age` variable printed out?

Modify the code above by displaying an error message if not a number was given on the first attempt. Also request the age of the user a second time.

In [39]:

```
age = input('What is your age?')
try:
    age = int(age)
except:
    print('That was not a number, try again!')
    age = input('What is your age?')
    age = int(age)

print("The given age is: {0}".format(age))
```

```
That was not a number, try again!
The given age is: 20
```

Both the *TRY* and the *EXCEPT* block can contain multiple statements. Test what will happen here if you comment out the erroneous assignment of the `y` variable?

In [40]:

```
x = 'Ten'
try:
    print('Line 1 in TRY block')
    y = int(x) # this will raise an exception
    print('Line 2 in TRY block')
except:
    print('Line in EXCEPT block')
print('END')
```

```
Line 1 in TRY block
Line in EXCEPT block
END
```

Multiple Except blocks

Different errors have different types which can be checked on the *EXCEPT* blocks.

IMPORTANT: *EXCEPT* blocks are tested in the order they are defined, so more specific error types MUST precede more general types.

In [41]:

```
x = 'Ten'
try:
    print('Line 1 in TRY block')
    y = int(x) # this will raise a ValueError
    y = 10 / 0 # this will raise a ZeroDivisionError
    print('Line 2 in TRY block')
except ValueError as e:
    print("ValueError was raised: " + str(e))
except ZeroDivisionError:
    print("ZeroDivisionError was raised")
except:
    print("Unknown error was raised.")
print('END')
```

```
Line 1 in TRY block
ValueError was raised: invalid literal for int() with base 10: 'Ten'
END
```

Finally

The `try-except` structure can be extended with a `finally` block. The code inside this block is always evaluated:

- Even if the *TRY* block was executed without an exception.
- Even if an exception was raised and handled by an *EXCEPT* block. (After the *EXCEPT* block.)
- Even if an exception was raised, but not handled by any *EXCEPT* block.

In [42]:

```
x = 'Ten'
try:
    print('Line 1 in TRY block')
    y = int(x) # this will raise a ValueError
    y = 10 / 0 # this will raise a ZeroDivisionError
    print('Line 2 in TRY block')
except ValueError as e:
    print("ValueError was raised: " + str(e))
finally:
    print('This line always gets printed')
print('END')
```

```
Line 1 in TRY block
ValueError was raised: invalid literal for int() with base 10: 'Ten'
This line always gets printed
END
```

The *finally* block can be especially when some operations must be performed in all cases; e.g. an opened file must be closed even if an error occurred during its processing.

Summary exercise on exception handling and conditions

Task: Check whether a certain year is a leap year or not?

According to the Gregorian calendar, every year that is exactly divisible by 4 is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400.

Also make sure that the user input is a positive number.

In [43]:

```
try:
    year = int(input('Which year to check? '))

    if year > 0:
        if year % 4 == 0:
            if year % 100 == 0:
                if year % 400 == 0:
                    print('{0} is a leap year'.format(year))
                else:
                    print('{0} is NOT leap year'.format(year))
            else:
                print('{0} is a leap year'.format(year))
        else:
            print('{0} is NOT leap year'.format(year))
    else:
        print('That was not a positive number!')
except:
    print('That was not an integer number!')
```

```
2000 is a leap year
```

The solution can be simplified by constructing a combined condition with the logical operators `and` and `or`:

In [44]:

```
try:
    year = int(input('Which year to check? '))

    if year > 0:
        if year % 400 == 0 or year % 4 == 0 and year % 100 != 0:
            print('{0} is a leap year'.format(year))
        else:
            print('{0} is NOT leap year'.format(year))
    else:
        print('That was not a positive number!')
except:
    print('That was not an integer number!')
```

```
2020 is a leap year
```