2-Objects

April 21, 2023

1 Object-oriented programming

Object-oriented programming is based on **objects** that have a series of **attributes** and **methods** than can be applied to them.

1.1 Okay, but what is an object?

An object can be anything in the real world that we want to represent in our code. YOU (yes, you), for example, could be an object: you would have attributes such as hair type, height, favourite ice-cream flavor, horoscope sign, etc. You would also have methods, for example, you can walk around, jump, sit down, smile, cry. If you could be simulated in a computer programme, there would be methods to make you move or smile as I mentioned. Unfortunately, people are too complex to be simulated in a programme, so we will have to use a simpler object as an example: the number one.

[314]:
```
print(dir(1))
```

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__getstate__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
'__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'as_integer_ratio',
'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
'numerator', 'real', 'to_bytes']
```

What has been printed here are the methods and attributes that apply to the object `1`. Internal methods have the format `__method__`, whereas attributes and other methods are just listed by their names. External methods are called as `object.method()`, whereas attributes are called as `object.attribute`. The use of internal methods is less straightforward; for example, you call the method `__str__` when you use the `print` **function**. Which brings us to…

1.2 Methods vs functions

Functions are similar to methods, they both perform a series of instructions. However, even though functions can also be applied to objects, they don't have to be linked to an object or a type

of object. So, following this real-life analogy where you are an object, let's say that a function could make it rain. It could make you open an umbrella, or worsen your mood, but rain doesn't fall to annoy you (I hope not, at least). The rain is a good analogy because the purpose of functions is to make it easier for you to run the same code twice and, of course, bad weather is not a one-time event.

Here, you can see how the `print` function can be applied to different types of objects:

[315]:
```
print(type(1), type([1, 1]))
```

```
<class 'int'> <class 'list'>
```

`type` is also a function that gives us the type (called **class**) of each object. The object `1` is an integer, abbreviated as `int` in Python, and the object `[1, 1]` is a list.

1.2.1 Structure of functions

Functions take inputs, referred to as **arguments**, and usually they return outputs. A function doesn't necessarily return an output; sometimes it just performs a task. There are cases in which the function doesn't even need an input either; for example, if you want to write a function that returns you a number from 1 to 10. The structure is, therefore, as follows:

[316]:
```
def print_no(limit): # Function definition
    # Here you write what the function does and what the inputs are
    """This function gives you all the prime numbers between 1 and a number of␣
 ↪your choice
    Input: limit (integer) = upper limit of the range where you screen for␣
 ↪prime numbers
    Outputs: prime_nos = list of prime numbers in the range between 1 and limit
             not_prime = numbers that are not prime in that same range"""

    # Here you write the code for the function
    prime_nos = [] # Defining output variables
    not_prime = []
    if limit > 1: # Code to search for prime numbers
        for n in range(1, limit+1): # Loop through numbers until our number
            check = True # Boolean to check if a number is prime
            for i in range(2, int(n/2)+1): # Loop through numbers until half␣
 ↪the value of our number
                if (n % i) == 0: # Modulo
                    not_prime.append(n) # Add non-prime numbers to non-prime␣
 ↪number list
                    check = False # Boolean set to false (number not prime)
                    break
            if check == True: # If the number is prime
                prime_nos.append(n) # Add prime numbers to prime number list

    # Here you define the outputs (if you have any)
```

```
    return prime_nos, not_prime # Return outputs so that they can be used␣
 ↪outside the function

primes, not_primes = print_no(20) # Get output prime and non-prime numbers␣
 ↪until 20
[print(no, end = " ") for no in primes]; # Print numbers
```

```
1 2 3 5 7 11 13 17 19
```

In this case, the function has only one argument, `limit`, which gives you the integer where you stop looking for more prime numbers. In turn, it returns two outputs, `prime_nos` and `not_prime`, that contain a list of prime numbers and a list of non-prime numbers, respectively. You don't have to understand the code in-between at this stage.

Documenting code Here you can see that I added many **comments**. It is important to **document** your code properly so that other people can re-use it in the future. One of the easiest ways to do this is by commenting what each chunk of code does. You should also write what the function does, how it is used and what arguments you need in the line below the function definition. This way, another user can learn how to use the function. It can even be useful for yourself, if you were sleepy when you wrote the function and can't rely on your memory. Here you have the previous function as an example:

[317]:
```
help(print_no)
```

```
Help on function print_no in module __main__:

print_no(limit)
    This function gives you all the prime numbers between 1 and a number of your
choice
    Input: limit (integer) = upper limit of the range where you screen for prime
numbers
    Outputs: prime_nos = list of prime numbers in the range between 1 and limit
             not_prime = numbers that are not prime in that same range
```

1.2.2 Task: Write a function that takes two numbers as inputs and sums the numbers

You can write the code in the cell below, with some basic indications of the structure of a function.

OBS! Don't forget to document your function and to test-run it.

[318]:
```
def my_function(input1, input2):
    """My documentation"""
    # Here goes your code
    return output
```

1.3 Indentation

As you can see in the cell above, where you tried to create your own function, the blocks of code that are inside the function are **indented**. This just means that there are spaces at the beginning

of the line. In Python, when you have a line that ends with `:`, you need to add indented lines below. These indented lines go inside the line that ends with `:`, in this case, the function definition. If you forget to indent a line, it will be executed outside the function that you defined, which might lead to **errors**. In a similar way, if you don't add any indented line after a statement ending with `:`, you will get an error.

1.4 Objects vs variables

We have looked into objects and what they are, but there is a similar concept that is often confused with objects, and it is that of **variables**. Variables are names that we give to objects. By calling the variable, we have access to information about the object. Going back to the analogy where you are an object, your full name could be a variable: in Sweden, if someone googles your full name, they can get some information about you, such as your birthday or your age. But perhaps you have also included that information in your Twitter profile, where you are using the fake name Lax Salmonsson so that your boss doesn't fire you because of your strong opinions on the price of salmon.

As an example, we will define three variables and assign the same object, the integer `1`, to two of them.

[319]:
```
var1 = 1
var2 = 1
var3 = 3.5
```

Now, let's check if var1 and var2 are the same object. To do so, we will print the id of each variable.

[320]:
```
print(id(var1))
print(id(var2))
print(id(var3))
```

```
4340138232
4340138232
4563845072
```

As you can see, `var1` and `var2` have the same id! Did you expect this?

`var3` doesn't have the same id, since it is a different object, the number `3.5`. This number is not an integer (`int`), but a different datatype. Can you guess which type? (Hint: you can use the `type` function on `var3`).

[ ]:
```
# Write your code here
```

This brings us to the next question: What are the data types that you can use in Python? Let's look into it.

2 Classes of objects

2.1 Numbers

In the example above, you could see (hopefully) that there are two different data types for numbers, **integers** and **floats**. Integers are any numbers, positive or negative, without a decimal, whereas

floats always include a decimal, even if it is 0. This means that `1` and `1.0` are different objects, even though they have the same numerical value. You can see it here:

[321]:
```
1 == 1.0
```

[321]: `True`

The `==` operator is used to check if two objects are equal. Its counterpart is the `=!`operator, that allows you to check if they are different. In this case, what we see is that the **value** of the objects is equal. If we want to check if the objects themselves are the same, we can compare the ids:

[322]:
```
id(1) == id(1.0)
```

[322]: `False`

Here, we can see how these are different objects! Now, let's play around a bit with them.

[323]:
```
var_int = 1
var_float = 1.0
print(var_int + var_float)
print(var_int - var_float)
```

```
2.0
0.0
```

In this context, the `+` and `-` operators are used to **sum and substract values**, respectively, like in a typical calculator. However, as you can note, the output of these calculations is always a float! Let's try multiplication, performed with the `*` character, and division, with the `/` character.

[324]:
```
print(var_int * var_float)
print(var_int/var_float)
```

```
1.0
1.0
```

Again, we got floats! This behavior is defined by the internal methods assigned to integer and float objects. It's possible to perform calculations involving both floats and integers, but then the result will always be a float, unless... we force it to be an integer with the function `int()`.

[325]:
```
print(int(var_int/var_float))
```

```
1
```

Now we got an integer! This shows how you can change the datatype of certain objects. However, keep in mind that, if you convert a float into an integer, you might be losing information. Try to apply the `int()` function to `var3` and see what happens! You can also try to convert it back to float using the `float()`function.

[ ]:
```
# Write your code here
```

Even if you have two integers, **you will always get a float result from a division**. If you want to divide two integers and get an integer output, you can use the `//` operand. Let's take a look at it:

[326]:
```
print(var_int/4) # Standard division
print(var_int//4) # Integer division
```

```
0.25
0
```

You can also do other operations, such as **powers** (`**` operator) and **roots**. Powers are quite intuitive:

[327]:
```
var3**2
```

[327]:
```
12.25
```

If you want to do a square root, you can either use the power of 0.5 or the `sqrt` function in the `math` package. If you want to use this function, you have to **import** the required package.

[328]:
```
from math import sqrt
sqrt(var3)
```

[328]:
```
1.8708286933869707
```

[329]:
```
var3**0.5
```

[329]:
```
1.8708286933869707
```

You can also calculate the **modulo** (outputs the remainder of a division) using the `%` operand. Here is an example of how it is used:

[330]:
```
3%2
```

[330]:
```
1
```

2.1.1 In-place operations

When you do math calculations, it's generally faster to use **in-place operations**. This is how in-place operations look vs **implicit-copy operations**:

[331]:
```
test = 3

# In-place multiplication
test *= 2

# Implicit-copy multiplication
test = test*2
```

In the in-place operation, the value of `test` is overwritten with the result of the calculation, whereas in the implicit-copy operation, the value of the multiplication is calculated and then assigned to a new variable with the same name, `test`. In general, in-place operations are faster than implicity-copy operations, because the latter require to create a copy of the variable (`test` in this case), whereas the former just assigns a new value to the same variable.

Extra: Example in numpy **numpy** is a Python package that speeds numerical computing. Here I use it as an example of how in-place operations are generally faster than implicit-copy operations.

[332]:
```
import numpy as np
```

[333]:
```
%%time
# Implicit-copy multiplication
var4 = np.ones(10**6) # Create matrix of ones of size 10e6
var4 = var4*2 # Multiply all the numbers by two (implicit-copy)
```

```
CPU times: user 4.39 ms, sys: 15 ms, total: 19.3 ms
Wall time: 37.9 ms
```

[334]:
```
%%time
# In-place multiplication
var4 = np.ones(10**6)
var4 *= 2 # Multiply all the numbers by two (in-place)
```

```
CPU times: user 3.64 ms, sys: 906 µs, total: 4.55 ms
Wall time: 2.59 ms
```

With the **magic command %%time**, we can calculate how long the cells takes to run. You can run it multiple times and check how the in-place multiplication is faster most of the times.

[335]:
```
# Implicit-copy multiplication
var4 = np.ones(10**6)
aid = id(var4) # Get id of matrix
var4 = var4*2
bid = id(var4) # Get id of matrix after multiplication
aid == bid # Check if ids are equal
```

[335]:
```
False
```

In this case, the objects ids are different when we perform the implicit-copy multiplication, since we are copying the matrix object during the multiplication step.

[336]:
```
# In-place multiplication
var4 = np.ones(10**6)
aid = id(var4)
var4 *= 2
bid = id(var4)
aid == bid
```

[336]:
```
True
```

In this case, the ids are identical, because we are modifying the values that are stored in the object, but we are not creating a new object in the process.

OBS! Note that this is not applicable to numbers, where each number is an object with its own id.

2.2 Booleans

As you have probably noticed, when you use the `==` operator, you get either `True` or `False` as an answer. these values are **booleans**. `True` also has the integer value `1`, and will display that value in sums or multiplications, whereas `False` has the integer value `0`:

[337]:
```
print(int(True))
print(int(False))
```

```
1
0
```

[338]:
```
True + False
```

[338]: 1

Booleans represent **logical values** and, as such, you can perform logical operations on them, such as `or` and `and`.

[339]:
```
True or False
```

[339]: True

[340]:
```
True and False
```

[340]: False

You can use the `bool()` function to evaluate if values are `True` or `False`. Most values will be true, except for empty values, `None` and, of course, `False`.

2.3 Characters and strings

Strings represent text, and they can be displayed with the `print()`function. Strings can also be defined as an array of **characters**. Any letter or symbol can be a character. To define variables as character of string, you need to use **single (`''`) or double (`""`) quotation marks**. The type of quotation mark doesn't matter.

[341]:
```
my_string = "(Why don't my results make sense?)\n  \\"
print(my_string)

my_string2 = '( ° °     '
print(my_string2)
```

```
(Why don't my results make sense?)
  \
( ° °
```

As you can see, both of the attempts at printing worked! (Unlike my experiments…).

Like for numbers and booleans, there are several operations that you can perform on strings. For example, you can sum characters of strings. In addition, using a backslash (\) allows you to scape special characters (such as the quotation marks themselves) and to write different special characters

(\n adds a linebreak and \t adds a tabulation). In the example above, we used /n to add a line break between the text bubble and the backlash. Now, we will see an example of how to sum two strings:

[342]:
```
"Hello " + "world"
```

[342]:
```
'Hello world'
```

Keep in mind that, even though it is possible to **add** strings, you can't **substract** them, nor divide them, for that matter. But **string multiplication** is possible, as long as you **multiply the string by an integer**:

[343]:
```
print("Hello "*2)
```

```
Hello Hello
```

Now, an example on how to add a tabulation:

[344]:
```
print("Hello\tworld")
```

```
Hello	world
```

Here you can also see that, even though the string was written with double quotation marks, the ending result had the default single quotation marks.

By default, every time you run a print statement, \n is added at the end of the statement. This means that these two cells do the same:

[345]:
```
print("Hello")
print("world")
```

```
Hello
world
```

[346]:
```
print("Hello\nworld")
```

```
Hello
world
```

You can modify the end character that you print by feeding the `print()` function the argument `end`:

[347]:
```
print("Hello", end = "\t")
print("world")
```

```
Hello	world
```

Now, as you can see, we printed the two words separated by a tabulation!

Since strings are made up of characters, we can also **loop** through them and access individual characters.

[348]:
```
test_string = "On the price of Salmon, by Lax Salmonsson"
[print(character, end = " ") for character in test_string];
```

```
O n  t h e  p r i c e  o f  S a l m o n ,  b y  L a x  S a l m o n s s o
n
```

Here I went through every character of the string and printed it, but, because I added a space as the end of the `print` statement, we have double-spaced characters. We will cover the concept of loops in the following section:

2.3.1 f-strings

There are different ways to put strings together (such as sums as I showed you above), but this one is my favorite so, unfortunately, you're stuck with this one. Keep in mind that it is only supported in Python 3.

This is how and **f-string** looks like:

[394]:
```
num1 = 1.8375865
num2 = num1*2
f"{num1} times 2 equals {num2}"
```

[394]:
```
'1.8375865 times 2 equals 3.675173'
```

Esentially, you add an `f` before your string and `{}` brackets around the names of your variables, and Python replaces your variables by their values. In this case, you can adjust the number of decimals that are displayed adding `:.nf`, where you replace `n` with the number of digits that you want.

[395]:
```
f"{num1:.2f} times 2 equals {num2:.2f}"
```

[395]:
```
'1.84 times 2 equals 3.68'
```

You can use placeholders in your string to replace them with your variables, which is quite similar to f-strings:

[401]:
```
"%s times 2 equals %s" % (num1, num2)
```

[401]:
```
'1.8375865 times 2 equals 3.675173'
```

However, I find f-strings more intuitive and easy to use.

2.4 Lists, dictionaries and tuples

Python **lists** are one-dimensional arrays that can contain any type of data, or any combination of types. You can put anything in a list, it depends entirely on your reasoning (or lack thereof).

[349]:
```
list_a = [1, 2, 3, 4, 5, 6, 7, 8]
print(list_a)
l15t_Aaaah = ["hi", 5.4, "bye", "h3h3", True]
print(l15t_Aaaah)
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
['hi', 5.4, 'bye', 'h3h3', True]
```

See? No errors! 3v3ryth1ng is f1n3.

Python **dictionaries** consist of key/value pairs, which each key is assigned a value. Both the keys and the values can be any Python data type (class), and you can also combine different classes of keys and assign them different classes of values:

[350]:
```
dict_a = {1: "cow", 2: "spider", 3: "octopus", 4: "amoeba"}
print(dict_a)
d1ct_Aaaah = {1: "c0w", "sp1d3r": 2, 3.5: "o", False: l15t_Aaaah}
print(d1ct_Aaaah)
```

```
{1: 'cow', 2: 'spider', 3: 'octopus', 4: 'amoeba'}
{1: 'c0w', 'sp1d3r': 2, 3.5: 'o', False: ['hi', 5.4, 'bye', 'h3h3', True]}
```

Again, no errors! G00d g00d g00d.

Why don't you try creating a dictionary with the numbers from 1 to 12 as keys and the names of the months as values?

[ ]:
```
# Write your code here
```

Tuples can be understood as **immutable lists**. This means that they are **not modifiable**, but they are **more memory efficient**. They are useful to represent certain types of data; for example, coordinates. Like for lists and dictionaries, you can mix any types of data:

[380]:
```
tuple_a = (1, 2, 3, 4, 5, 6, 7, 8)
print(tuple_a)
tup13_Aaaah = ("hi", 5.4, "bye", "h3h3", True)
print(tup13_Aaaah)
```

```
(1, 2, 3, 4, 5, 6, 7, 8)
('hi', 5.4, 'bye', 'h3h3', True)
```

Now we know how to create lists, tuples and dictionaries, but how do we access the data that is stored in them?

In lists, you can access any value by providing its **index**. Note that the **indexes in Python start at 0**, not at 1. For example,...

[351]:
```
list_a[1]
```

[351]:
```
2
```

...returns the second item in the list, not the first.

As for strings, you can add several lists (even if they don't have compatible data types). You can do:

[352]:
```
list_a + l15t_Aaaah
```

[352]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 'hi', 5.4, 'bye', 'h3h3', True]
```

You can also **multiply strings by integers**, which is the same as adding the list to itself a number of times that is given by the integer:

[353]:
```
l15t_Aaaah*2
```

[353]:
```
['hi', 5.4, 'bye', 'h3h3', True, 'hi', 5.4, 'bye', 'h3h3', True]
```

Tuples behave essentially like lists. You can access the elements in the tuple using an index that starts at 0:

[378]:
```
print(tuple_a[3])
```

```
0.5
7.4
```

You can also sum them or multiply them by integers:

[379]:
```
print(tuple_a + tup13_Aaaah)
print(tuple_a * 2)
```

```
(0.5, 7.4, 'hi', True)
(0.5, 7.4, 0.5, 7.4)
```

In dictionaries, you access values by **key**, instead of by index, even though, of course, a key can look like a list index. For example:

[354]:
```
dict_a[1]
```

[354]:
```
'cow'
```

Another property of dictionaries is that you can get a list of the keys or a list of the values using the functions `dict.keys()` or `dict.values()`:

[355]:
```
keys = d1ct_Aaaah.keys()
values = d1ct_Aaaah.values()
print(keys)
print(values)
```

```
dict_keys([1, 'sp1d3r', 3.5, False])
dict_values(['c0w', 2, 'o', ['hi', 5.4, 'bye', 'h3h3', True]])
```

These keys and values can easily be converted into lists with the `list()` function, but it is not necessary to loop through the items. Looping through keys or values is quite useful if you want to modify the data in the dictionary or pass it as an input for the next chunk of your code. We haven't yet introduced loops, but it's about time!

2.4.1 Loops

When we loop through data, the underlying structure of these data is usually an array of items. You can iterate through lists, dictionaries and strings which, as I mention, are an array of characters. The first line gives you the most common sintax of a **loop**:

[357]:
```
for element in list_a:
    print(element, end = " ")
```

```
1 2 3 4 5 6 7 8
```

Note that the code that goes inside the loop has to be indented.

Of course, you don't always iterate over lists. It is quite common to iterate over **ranges**, defined with `range(n, m)`, where both `n` and `m` are integers. For example, if we want to iterate from 1 to 10:

[358]:
```
for number in range(1, 11):
    print(number, end = " ")
```

```
1 2 3 4 5 6 7 8 9 10
```

Note that ranges go from `n` to `m-1`. In this case, the range was defined from 1 to 11, but it only printed integers up until 10. When we define a range, we only need to specify `m`, but we have to keep in mind that `n` is 0 by default.

Why using ranges and not just lists? The advantage of using ranges is that the elements in the range are created as you iterate over them, unlike in lists, where we have values that we can access any time. Of course, depending on the situation that might be what we want.

And what about dictionaries? If we want to iterate over a dictionary, we usually iterate over the keys using the `dict.keys()` method:

[360]:
```
for key in dict_a.keys():
    print(key, end = "\t")
    print(dict_a[key])
```

```
1       cow
2       spider
3       octopus
4       amoeba
```

To access the values, we use the keys as indexes.

We have seen several examples of loops, now it's time for you to write a loop that goes from 1 to 7 in the box below. You can use a list or a range for that purpose. Perhaps you also want to take this chance to loop through the months in the dictionary you created.

[ ]:
```
# Write your code
```

The most common syntax for loops is a loop that starts with `for <name> in <iterable>:`, followed by indented chunks of code, but there are other possibilities. Let's take a look into them!

2.4.2 List and dictionary comprehensions

Until now, we have seen how the useful syntax of loops is, but there is a way to **speed up loops**. Especially when you have **nested loops** (loops inside loops inside loops inside loops... you get the idea). **List comprehensions** can speed up the processing of your data, and you can do something similar for dictionaries, which is called **dictionary comprehension**. Hooray!

This is how the basic syntax of a list comprehension looks like:

[362]:
```
[print(element, end = " ") for element in l15t_Aaaah];
```

```
hi 5.4 bye h3h3 True
```

It is even possible to have two loops in a list comprehension! For example, if we want to print any possible combination of keys and values in a dictionary:

[364]:
```
[print(key, value) for key in dict_a.keys() for value in dict_a.values()];
```

```
1 cow
1 spider
1 octopus
1 amoeba
2 cow
2 spider
2 octopus
2 amoeba
3 cow
3 spider
3 octopus
3 amoeba
4 cow
4 spider
4 octopus
4 amoeba
```

And, speaking of dictionaries, the syntax of a dictionary comprehension can be identical to that of a list comprehension:

[365]:
```
{print(key, value) for (key, value) in dict_a.items()};
print("\n") # Line break between outputs
[print(key, value) for (key, value) in dict_a.items()];
```

```
1 cow
2 spider
3 octopus
4 amoeba


1 cow
2 spider
3 octopus
4 amoeba
```

As you can see, we got the same result from the two. Why use dictionary comprehensions then? The syntax can also look a bit different. You can also use dictionary comprehensions to create new key/value pairs. For example, here I replaced the animal names with the squared value of the keys:

[366]:
```
new_dict = {key: key**2 for key in dict_a.keys()}
{print(key, value) for (key, value) in new_dict.items()};
```

```
1 1
2 4
```

```
3 9
4 16
```

2.4.3 While loops

Unlike lists comprehensions, which are a form of **for loops**, the most common type of loop in Python, these loops make use of the concept of **booleans**. The idea is that the **while loop** will keep running as long as a certain condition is true. For example:

[367]:
```
i = 1
while i < 11:
    print(i, end = " ")
    i += 1
```

```
1 2 3 4 5 6 7 8 9 10
```

This code returns all numbers between 1 and 10, and stops before the number becomes bigger than 11. However, there is an obvious downside to these types of loops. Beware... the **infinite loop**!:

[ ]:
```
# You better stop this before it's too late...
i = 1
while i < 11:
    print(i, end = " ")
```

Because we forgot to increase the value of the variable `i` in every iteration, the condition defined in the while loop was always fulfilled. Therefore, the loop can go on forever, until you put an end to it. So, remember: if you ever use a while loop, **make sure that it will stop at some point**, hopefully not too late.

2.5 Conditional statements

When we introduced while loops, we also introduced the concept of **conditions**. The idea of conditional statements is to execute a chunk of code if a condition is fulfilled. Unlike while loops, they are only executed once –unless you include them in a loop. Let's look at the basic syntax:

[389]:
```
n = int(input("Give a number: ")) # Ask for a number (integer)

if n%2 == 0: # If the number is even
    print("Even number") # Print this
elif n%5 == 0: # If the number is not even, check if it is a multiple of 5
    print("Odd number multiple of 5")
else:  # If the number doesn't fit any of the two conditions above
    print("Odd number, not multiple of 5")
```

```
Give a number: 15
Odd number multiple of 5
```

In this case, you can test the conditional statement by inputting the number that you want with the `input()` function, but note that this function is not part of conditional statements. The statement is made of of the three following parts:

1. `if` statement: The first condition that you want to test.
2. `elif` statement: If the first condition is not fulfilled, the next condition, defined by an `elif` statement, will be assessed. These statements are optional and only used if we want to test more than one condition.
3. `else` statement: If none of the conditions that were defined are fulfilled, the code under this statement will be executed. This is also optional.

OBS! The code that goes inside every statement has to be indented.

Perhaps now you want to try to write a conditional statement? You can also use the `input()` function to test the different conditions.

[ ]:
```
# Write your code here
```

2.6 What do I do now?

Try to create a small script where you use a **function** that includes **loops** and **conditional statements**. Keep in mind that you can have **nested loops** and **nested conditional statements**, meaning that you can have a loop inside a loop and a conditional statement inside a conditional statement inside a loop. You can write your code here or create a script (`<filename>.py`) and test a Python IDE!

[ ]:
```
# Write your code here... or not
```