In [ ]:
```
# get formating done automatically according to style `black`
#%load_ext lab_black
```

Python core language part 2

Content

You will learn how to:

- Copy objects with copy and deepcopy
- Crunch numbers with basic operators and math functions
- Make decisions with conditions
- Let the computer do the work with loops
- Make your life easy with list comprehensions

Copy Objects

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

In [1]:
```
import copy
```

In [2]:
```
a = [1, 2, 3]
b = ['one', 'two', 'three']
c = [a, b]
```

The `id()` method returns a unique integer (identity) of an object.

In [3]:
```
id(c)
```

Out[3]:
```
140737019855040
```

Shallow Copy

In [4]:
```
c_copy = copy.copy(c)
```

In [5]:
```
c_copy
```

Out[5]:
```
[[1, 2, 3], ['one', 'two', 'three']]
```

In [6]:
```
print(id(c_copy)==id(c))
```

```
False
```

In [7]:
```
id(c[0])
```

Out[7]:
```
140737019854464
```

In [8]:
```
id(c_copy[0])
```

Out[8]:
```
140737019854464
```

In [9]:
```
print(id(c_copy[0])==id(c[0]))
```

```
True
```

Deep Copy

In [10]:
```
c_deepcopy = copy.deepcopy(c)
```

In [11]:
```
print(id(c_deepcopy)==id(c))
```

```
False
```

In [12]:
```
print(id(c_deepcopy[0])==id(c[0]))
```

```
False
```

Exercise

1. Can you create a new object with the assign statement (`new_object = old_object`)?
2. Create a list with several items and create a copy with `.copy` and `.deepcopy`
3. Can you illustrate the difference between these two operations?

In [13]:
```
# 1.
new_object = c
id(new_object) == id(c)
```

Out[13]:
```
True
```

No, it's not a new object.

In [14]:
```
# 2.
```

repeat example

In [15]:
```
# 3.

# change the original c
c[1][2] = 'change'

# see how c_copy change
print(c_copy)

# see how c_deepcopy change
print(c_deepcopy)
```

```
[[1, 2, 3], ['one', 'two', 'change']]
[[1, 2, 3], ['one', 'two', 'three']]
```

Basic Operators and Math Functions

Basic mathematical operators are:

- `+` addition
- `-` substraction
- `*` multiplication
- `/` division

Additionally you can use Python's math functions.

```
import math
math.degrees(math.pi)
```

```
180.0
```

All functions can be found in the Python docs.

Exercise

1. Perform an arbitrary calculation, which includes an addition, substraction, mulitplication and division.
2. What happens if you add two string variables?
3. Can you multiply a string with an integer? If yes, what will happen?
4. Have a look at the math functions. Chose three of them and include them in an arbitrary calculation.

In [16]:
```
# 1.
3+4*5/7
```

Out[16]:
```
5.857142857142858
```

In [17]:
```
# 2.
"sting 1 " + "string 2"
```

Out[17]:
```
'sting 1 string 2'
```

In [18]:
```
# 3.
"String" * 3
```

Out[18]:
```
'StringStringString'
```

In [19]:
```
# 4.
import math
```

In [20]:
```
math.cos(45)
```

Out[20]:
```
0.5253219888177297
```

In [21]:
```
math.isfinite(math.sinh(78*math.pi))
```

Out[21]:
```
True
```

Conditions

- Python uses Boolean operators to evaluate conditions
- Logical and comparison operators allow building complex Boolean expressions
- Comparison operators are: `==` , `>=` , `<=` , `>` , `<` and `!=`
- Logical operators are: `not` , `and` and `or`
- If the result of an if expression is True, the condition is fulfilled and the if statement will be executed

Syntax:

```
if(expression):
    statement_1
    statement_2
    [...]
elif(expression):
    statement_3
else:
    statement_4
    […]
```

Exercise

What will happen? Please, only use pen and paper.

```
a = 1.
b = -3.
c = 'Dog'
if(a <= b and not b == c or a >= b):
    print("We are learning Python.")
else:
    print("We are not learning Python.")
```

a <= b and not b == c or a >= b

1. first resolve **boolean** operators

False and not False or True

2. resolve **not**

False and True or True

3. resolve **and**

False or True

4. resolve **or**

True

In [22]:
```
# Confirm by running code

a = 1.
b = -3.
c = 'Dog'
if(a <= b and not b == c or a >= b):
    print("We are learning Python.")
else:
    print("We are not learning Python.")
```

```
We are learning Python.
```

Loops

`for` loops are used when a block of code needs to be repeated for a fixed number of time.

Syntax:

```
for item in sequence
    statement_1
    statement_2
    [...]
```

In [23]:
```
my_list = ["Hello", "world", "!"]

# Example 1
for thing in my_list:
    print(thing)
```

```
Hello
world
!
```

In [24]:
```
# Example 2
for number in range(0, 4):
    print(number)
```

```
0
1
2
3
```

In [25]:
```
number
```

Out[25]: 3

Python's `zip()` function allows you to iterate in parallel over two or more iterables.

In [26]:
```
fruit = ["apple", "banana", "peach", "mango"]
vegetable = ["carrot", "potato", "cabage", "tomato"]

for f, v in zip(fruit, vegetable):
    print('Fruit: ', f)
    print('Vegetable: ', v)
```

```
Fruit:  apple
Vegetable:  carrot
Fruit:  banana
Vegetable:  potato
Fruit:  peach
Vegetable:  cabage
Fruit:  mango
Vegetable:  tomato
```

With `enumerate()` Python gives you the counter and the value of the iterable at the same time.

In [27]:
```
for count, value in enumerate(fruit):
    print(count, value)
```

```
0 apple
1 banana
2 peach
3 mango
```

Exercise

Sum up all numbers from 1 to 100 by using a `for` loop. This problem is also know as Gauß formula.

In [28]:
```
# Exercise

sum1=0

for number in range(1, 101):
    sum1=sum1+number

print(sum1)
```

```
5050
```

The `while` loop is another type of loop.

```
while (expression):
    statement_1
    statement_2
    ...
```

The block of code will be executed as long the expression is true.

Hint:

Make sure that the condition can be fulfilled. Otherwise the loop will run forever.

In [29]:
```
x = 0
while x<=10:
    print(x)
    x += 1 # same as x = x +1
```

```
0
1
2
3
4
5
6
7
8
9
10
```

List comprehensions

List comprehensions are an easy way to apply a function to each entry of your list or to filter your list, the result is again a list. The general syntax looks like this:

```
new_list = [ expression for item in old_iterable if condition ]
```

The advantage of this construct is its brevity, but it impedes the code readability. The important thing about list comprehension is to be able to recognise it.

In [30]:
```
a = [1,2,3,4]
print(a)
```

```
[1, 2, 3, 4]
```

Add 1 to each list item.

In [31]:
```
b = [xz + 1 for xz in a]
print(b)
```

```
[2, 3, 4, 5]
```

Get a filtered list, in which are only odd values from *a*

In [32]:
```
c = [x for x in a if x % 2 == 1]
print(c)
```

```
[1, 3]
```

List comprehensions can be nested to reflect a nested loop. It requires less typing but also becomes harder to read.

In [33]:
```
d1 = []
for x in c:
    for y in a:
        if y % 2 == 0:
            d1.append(10*x+y)
print(d1)
```

```
[12, 14, 32, 34]
```

In [34]:
```
d2 = [10*x+y for x in c for y in a if y%2 == 0]
print(d2)
```

```
[12, 14, 32, 34]
```

Using the analogy of the nested loop we can use nested list comprehension to get a flat list of a nested list.

In [35]:
```
nested_list = [[1,2,3],[4,5],[6]]
print(nested_list)
```

```
[[1, 2, 3], [4, 5], [6]]
```

In [36]:
```
flat_list = [x for sublist in nested_list for x in sublist]
print(flat_list)
```

```
[1, 2, 3, 4, 5, 6]
```

Exercise

Create a list with a few numbers and square each entry via list comprehension.

In [37]:
```
# Exercise variant a

example=[2,4,6,8,10,12]

output_list=[ff*ff for ff in example]
print(output_list)
```

```
[4, 16, 36, 64, 100, 144]
```

In [38]:
```
# Exercise variant b
import math
my_list = [1, 2, 3, 4, 5]
new_list = [math.pow(gg,2) for gg in my_list]
new_list
```

Out[38]:
```
[1.0, 4.0, 9.0, 16.0, 25.0]
```

In [39]:
```
# Exercise variant c
example2 = [3, 6, 7, 9]
output2 = [x**2 for x in example2]
output2
```

Out[39]:
```
[9, 36, 49, 81]
```