Chapter 3: Iterations and lists

Data structure: List

A **list** in Python is a *heterogeneous* container for multiple items.

- Container means the list can store multiple values. In case of a list the items are also stored in an ordered way.
- Heterogeneous mean that the elements of a list can be of different types: numbers, strings, etc.

A list is a similar data structure like an array in many other languages (like C++, Java or C#), but since Python does not support arrays, we have lists.

Let's define a list of neighbouring countries for Hungary. The initial items of a list are defined between brackets.

In [59]:

```
neighbours = ['Austria', 'Slovakia', 'Ukraine', 'Romania', 'Serbia', 'Croatia',
'Slovenia']
print(neighbours)
```

```
['Austria', 'Slovakia', 'Ukraine', 'Romania', 'Serbia', 'Croatia',
'Slovenia']
```

Let's check the type of the `neighbours` variable.

In [60]:

```
print(type(neighbours))
```

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

Reading lists

The items of a list can be accessed by the numerical indexes. (The first item is indexed with *zero*.)

In [61]:

```
print(neighbours[0])
print(neighbours[1])
```

```
Austria
Slovakia
```

We can also access a range of elements:

In [62]:

```
print(neighbours[2:5])
```

```
['Ukraine', 'Romania', 'Serbia']
```

The end index is *exclusive*, which means that only the countries with index 2-4 was included in the result above.

By leaving out the *end* index, the range will go on to the end of the list:

In [63]:

```
print(neighbours[2:])
```

```
['Ukraine', 'Romania', 'Serbia', 'Croatia', 'Slovenia']
```

By leaving out the *start* index, the range will start at the first item:

In [64]:

```
print(neighbours[:5])
```

```
['Austria', 'Slovakia', 'Ukraine', 'Romania', 'Serbia']
```

By omitting both the *start* and the *end* index, the range will cover all elements:

In [65]:

```
print(neighbours[:])
```

```
['Austria', 'Slovakia', 'Ukraine', 'Romania', 'Serbia', 'Croatia',
'Slovenia']
```

Note: this is very similar like how we worked with strings, since strings can be treated like list of characters.

The number of items in a list (its length) can also be easily fetched:

In [66]:

```
print(len(neighbours))
```

```
7
```

Manipulating lists

Lists are mutable, meaning the items themselves and the number of items it contains can change dynamically after its initial definition. We can remove elements:

In [67]:

```
neighbours.remove('Slovakia')
print(neighbours)
```

```
['Austria', 'Ukraine', 'Romania', 'Serbia', 'Croatia', 'Slovenia']
```

Add new ones:

In [68]:

```
neighbours.append('Czechoslovakia')
print(neighbours)
```

```
['Austria', 'Ukraine', 'Romania', 'Serbia', 'Croatia', 'Slovenia',
'Czechoslovakia']
```

The elements can also be removed from or inserted to a specific location:

In [69]:

```
neighbours.pop(3) # removes Serbia, as its index is 3
del neighbours[3] # removes Croatia, as its index is 3, after we removed Serbia
neighbours.insert(3, 'Yugoslavia') # adds Yugoslavia on the 3rd index
print(neighbours)
```

```
['Austria', 'Ukraine', 'Romania', 'Yugoslavia', 'Slovenia', 'Czechos
lovakia']
```

Copying a list can be a bit tricky, because assigning a list to a new variable does not make a copy of a list, instead the new variable will just be an alias to the original list.

To make a real copy of a list, we can either use the `copy()` method of the list or use the range accessor to select and copy all elements to a new list.

In [70]:

```
alias_list = neighbours # this is just an alias
copied_list_1 = neighbours.copy() # this is a real copy
copied_list_2 = neighbours[0:len(neighbours)] # this is also real copy
copied_list_3 = neighbours[:] # this also copies all elements

# Clear all elements from the original list
neighbours.clear()

print(neighbours) # this shall be empty
print(alias_list) # this shall also be empty
print(copied_list_1) # this shall contain the elements
print(copied_list_2) # this too
print(copied_list_3) # this too
```

```
[]
[]
['Austria', 'Ukraine', 'Romania', 'Yugoslavia', 'Slovenia', 'Czechos
lovakia']
['Austria', 'Ukraine', 'Romania', 'Yugoslavia', 'Slovenia', 'Czechos
lovakia']
['Austria', 'Ukraine', 'Romania', 'Yugoslavia', 'Slovenia', 'Czechos
lovakia']
```

Lists have further useful methods, we can e.g. *sort* or *reverse* the elements of a list:

In [71]:

```
neighbours = ['Austria', 'Slovakia', 'Ukraine', 'Romania', 'Serbia', 'Croatia',
'Slovenia']
print('Original list: {0}'.format(neighbours))
neighbours.sort()
print('Sorted list: {0}'.format(neighbours))
neighbours.reverse()
print('Reversed list: {0}'.format(neighbours))
```

```
Original list: ['Austria', 'Slovakia', 'Ukraine', 'Romania', 'Serbi
a', 'Croatia', 'Slovenia']
Sorted list: ['Austria', 'Croatia', 'Romania', 'Serbia', 'Slovakia',
'Slovenia', 'Ukraine']
Reversed list: ['Ukraine', 'Slovenia', 'Slovakia', 'Serbia', 'Romani
a', 'Croatia', 'Austria']
```

See the documentation (https://docs.python.org/3/tutorial/datastructures.html) for a complete overview.

Control structure: Loops

The loop control flow is also called *iteration* or *repetition statement* and provides a way to execute the same code block (call the *core* of the iteration) until a condition is meet.

`for` loop: Iterating through a sequence

When using a `for` loop we introduce a new variable which will iterate over all elements of a list (or later other data structures containing multiple items) and take the value of the next item in each iteration.

Let's iterate over the values of the `europe` list with a variable `country`:

In [72]:

```
europe = ['Albania', 'Andorra', 'Austria', 'Belgium', 'Bosnia and Herzegovina',
'Bulgaria', 'Czech Republic', 'Denmark', 'United Kingdom', 'Estonia', 'Belarus',
 'Finland', 'France', 'Greece', 'Netherlands', 'Croatia', 'Ireland', 'Iceland',
'Kosovo', 'Poland', 'Latvia', 'Liechtenstein', 'Lithuania', 'Luxembourg', 'Maced
onia', 'Hungary', 'Malta', 'Moldova', 'Monaco', 'Montenegro', 'Germany', 'Norwa
y', 'Italy', 'Portugal', 'Romania', 'San Marino', 'Spain', 'Switzerland', 'Swede
n', 'Serbia', 'Slovakia', 'Slovenia', 'Ukraine']

for country in europe:
    print(country)
```

```
Albania
Andorra
Austria
Belgium
Bosnia and Herzegovina
...
Switzerland
Sweden
Serbia
Slovakia
Slovenia
Ukraine
```

This new variable introduced for iterating over the elements can be named anything. (Applying the same rules of course for naming variables discussed in Chapter 1 (01_python_intro.pdf).)

In [73]:

```
for anything in europe:
    print(anything)
```

```
Albania
Andorra
Austria
Belgium
Bosnia and Herzegovina
...
Switzerland
Sweden
Serbia
Slovakia
Slovenia
Ukraine
```

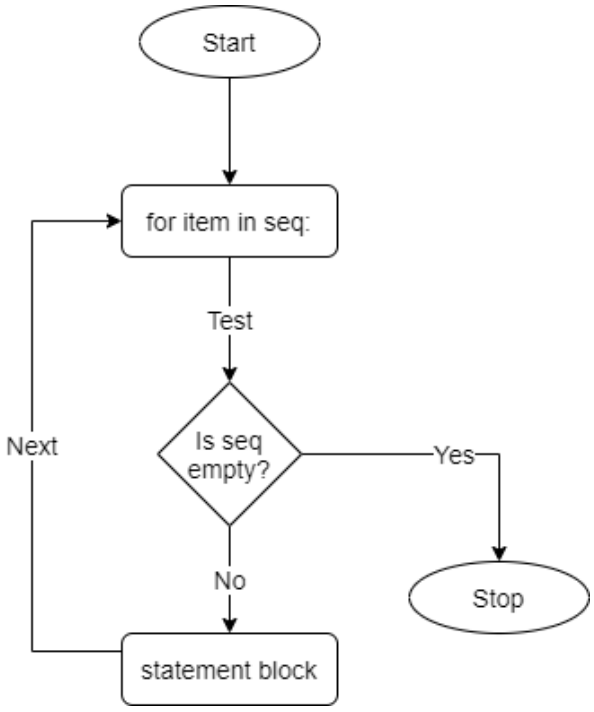
Note the identation of the `print` statement, which means that the `print` statement is *"inside"* the `for` loop and will be executed on each iteration.

In [74]:

```
for country in europe:
    print(country)
    print("This will be printed after each country")
print("This will be printed only once after all countries are printed")
```

```
Albania
This will be printed after each country
Andorra
This will be printed after each country
Austria
This will be printed after each country
Belgium
This will be printed after each country
Bosnia and Herzegovina
...
Switzerland
This will be printed after each country
Sweden
This will be printed after each country
Serbia
This will be printed after each country
Slovakia
This will be printed after each country
Slovenia
This will be printed after each country
Ukraine
This will be printed after each country
This will be printed only once after all countries are printed
```

The workflow diagram of the **for** loop:



The `range` function

`range()` is a function to create integer sequences, which can be converted to lists. We give the start and end value as arguments to the function. The end value is *exclusive*.

In [75]:

```
print(list(range(1, 10)))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We may only give one argument to the function, which will be the end value. The start value will be 0 in such a case.

In [76]:

```
print(list(range(8)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

As an optional, third argument, the step value can be passed, which defines the incrementation between the values of the resulted list.

In [77]:

```
print(list(range(1, 20, 2)))
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

We can use the `range()` function to print both the index and the name of each country:

In [78]:

```
print("Number of European countries: {0}".format(len(europe)))
for index in range(len(europe)): # length of a sequence
    print("The {0}th country in the list is: {1}".format(index, europe[index]))
```

```
Number of European countries: 43
The 0th country in the list is: Albania
The 1th country in the list is: Andorra
The 2th country in the list is: Austria
The 3th country in the list is: Belgium
The 4th country in the list is: Bosnia and Herzegovina
...
The 37th country in the list is: Switzerland
The 38th country in the list is: Sweden
The 39th country in the list is: Serbia
The 40th country in the list is: Slovakia
The 41th country in the list is: Slovenia
The 42th country in the list is: Ukraine
```

Remark: ranges are so called "lazy" objects, because they do not generate every number they contain when we create them. Instead they only produce the contained numbers as we need them when looping over them; or when we convert them to lists.

Exercise

Task: Print all the countries in the `europe` list, which start with letter `C`:

In [79]:

```
for country in europe:
    if country[0] == 'C':
        print(country)
```

```
Czech Republic
Croatia
```

Help: you have to place a conditional statement (`if`) inside an iterative statement (`for`) and combine them.

`while` loop: Keep doing until condition no longer holds

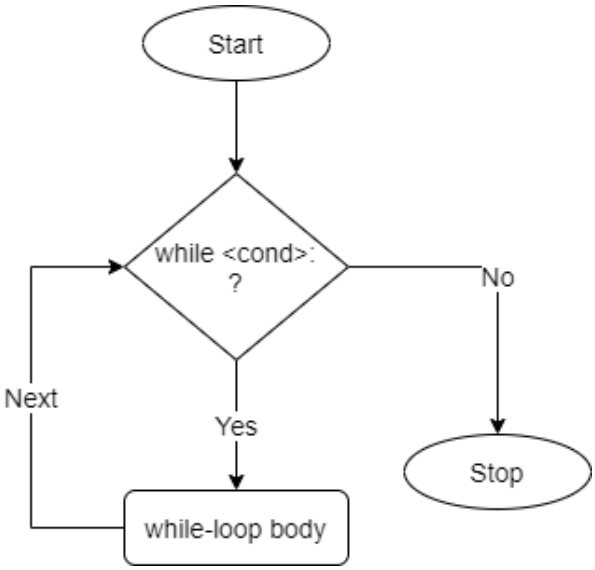
Use `for` when you know **the exact number of iterations**; use `while` when you **do not (e.g., checking convergence)**.

In [80]:

```
x = 123.0
while x > 1:
    print(x)
    x = x / 2
    #x /= 2
```

```
123.0
61.5
30.75
15.375
7.6875
3.84375
1.921875
```

The workflow diagram of the **while** loop:



Exercise

Task: write a program that requests some input from the user until the user types in: *enough*. Then the execution of the program shall stop.

In [81]:

```
user_input = input('Type in something: ')
while user_input != 'enough':
    user_input = input('Type in something: ')
```

Task: write a program that requests number from the user until the user types in: *enough*. Then the program shall list all the previously inputted numbers in the same order. Invalid (not numeric) inputs shall be skipped, also displaying a warning message that an incorrect value was typed in. Sample input and output:

```
Next number: 10
Next number: 8
Next number: apple tree
It is not a number, skipped!
Next number: 42
Next number: -4
Next number: enough
Given numbers: [10, 8, 42, -4]
```

In [82]:

```
numbers = []
user_input = input('Next number: ')

while user_input != 'enough':
    try:
        num = int(user_input)
        numbers.append(num)
    except:
        print('It is not a number, skipped!')

    user_input = input('Next number: ')

print('Given numbers: {0}'.format(numbers))
```

```
It is not a number, skipped!

Given numbers: [10, 8, 42, -4]
```

Help: define an empty list before requesting user input iteratively. Add the user given numbers to the list (append). Then in the end you only need to display the items of the list.

---

break and continue

`break` means get out of the loop immediately. Any code after the `break` will NOT be executed.

Compare the following 2 solutions for listing *all divisors* versus *only the first divisor* of a number:

In [83]:

```
number = int(input("Input an integer number: "))
for i in range(2, number+1):
    if number % i == 0:
        print("{0} is a divisor of {1}".format(i, number))
    else:
        print("{0} is NOT a divisor of {1}".format(i, number))
```

```
2 is NOT a divisor of 15
3 is a divisor of 15
4 is NOT a divisor of 15
5 is a divisor of 15
6 is NOT a divisor of 15
7 is NOT a divisor of 15
8 is NOT a divisor of 15
9 is NOT a divisor of 15
10 is NOT a divisor of 15
11 is NOT a divisor of 15
12 is NOT a divisor of 15
13 is NOT a divisor of 15
14 is NOT a divisor of 15
15 is a divisor of 15
```

In [84]:

```
number = int(input("Input an integer number: "))
for i in range(2, number+1):
    if number % i == 0:
        print("The first divisor of {0} is {1}".format(number, i))
        break # NOTE the break statement here!
    else:
        print("{0} is NOT a divisor of {1}".format(i, number))
```

```
2 is NOT a divisor of 15
The first divisor of 15 is 3
```

`continue` means get to the next iteration of loop. It will *stop* the current iteration and **continue** to the next.

Compare the following 2 solutions for listing *all divisors* of a number:

In [85]:

```
number = int(input("Input an integer number: "))
for i in range(1, number+1):
    if number % i == 0:
        print("{0} is a divisor of {1}".format(i, number))
```

```
1 is a divisor of 15
3 is a divisor of 15
5 is a divisor of 15
15 is a divisor of 15
```

In [86]:

```
number = int(input("Input an integer number: "))
for i in range(1, number+1):
    if number % i != 0:
        continue # if i NOT a divisor of number, then we continue the iteration
 with the next number
    print("{0} is a divisor of {1}".format(i, number))
```

```
1 is a divisor of 15
3 is a divisor of 15
5 is a divisor of 15
15 is a divisor of 15
```

NOTE: `break` and `continue` can also be used within `while` loops.

Summary exercise on iterations and loops

Task: Test whether a number is a prime

Request an integer number from the user.

Decide whether the number is a prime number or not and display your answer.

In [87]:

```
import math

number = int(input("Number to check: "))
is_prime = True

if number < 2:
    # Handle 0 and 1 as a special case
    is_prime = False
else:
    # Numbers >= 2 are tested whether they have any divisors
    for i in range(2, int(math.sqrt(number) + 1)):
        print("Testing divisor {0}".format(i))
        if number % i == 0:
            # If we found a divisor, we can stop checking, because the number is
 NOT a prime
            is_prime = False
            break

if is_prime:
    print("{0} is a prime".format(number))
else:
    print("{0} is NOT a prime".format(number))
```

```
Testing divisor 2
Testing divisor 3
Testing divisor 4
Testing divisor 5
Testing divisor 6
37 is a prime
```

Random generation

Random numbers can be generated with the `random` module.

Execute the code cell below multiple times to get different results:

In [88]:

```
import random
number = random.randint(1, 10)
print(number)
```

```
5
```

Generate 10 numbers between 1 and 100:

In [89]:

```
numbers = []
for i in range(10):
    numbers.append(random.randint(1, 100))
print(numbers)
```

```
[76, 55, 40, 73, 80, 8, 79, 95, 13, 98]
```

Advanced: pseudo-random numbers (optional)

This is advanced level remark, which is interesting, but not mandatory on this introductory course.

There is no real randomicity in computer science (unless you build a device which measures e.g. cosmic radiation). These numbers generated are so called pseudo-random numbers. They are generated by a deterministic mathematical algorithm along a uniform distribution.

The following algorithm will always generate the same numbers regardless how many times you execute it, because we seed the algorithm a fix initial value before each random generation:

In [90]:

```
random.seed(42)
numbers = []
for i in range(10):
    numbers.append(random.randint(1, 100))
print(numbers)
```

```
[82, 15, 4, 95, 36, 32, 29, 18, 95, 14]
```

By default the algorithm is only seeded once, with a value related to the milliseconds passed since the start of the computer. Hence it will look like real "random" numbers.

Exercise

Task: Dice roll

Write a loop which generates a dice roll (1-6) in every iteration. Run the loop 100 times and calculate the average value of the dice rolls! What is the difference against the expected value?
How does it change if you execute the dice roll 1000 times?

Hint: instead of adding the generated numbers together one-by-one, you may use the `sum(my_list)` function to calculate the accumulated value of a list of numbers.
Built-in functions like `sum` will be further discussed in Chapter 4 (04_functions.pdf).

In [91]:

```
import random

numbers = []
for i in range(0, 1000):
    dice_roll = random.randint(1, 6)
    numbers.append(dice_roll)

avg = sum(numbers) / len(numbers)
print('Average value: {0}'.format(avg))
print('Expected value: 3.5')
```

```
Average value: 3.519
Expected value: 3.5
```