Eötvös Loránd University

Faculty of Informatics

Máté Cserép

Spatial algorithms with Python

Lecture Notes

Budapest, 2021

Author

Máté Cserép

Assistant Lecturer

Institute of Computer Science

Eötvös Loránd University

Lector

Zsuzsanna Ungvári, PhD

Assistant Professor

Institute of Cartography and Geoinformatics

Eötvös Loránd University

Spatial algorithms with Python

Workbooks

1. Python introduction
2. Basic operations and conditional executions
3. Iterations and lists
4. Functions
5. Basic algorithms
6. Sorting algorithms and complexity
7. Collection data structures
8. Objected-oriented programming
9. Tabular data
10. Plotting and diagram visualization
11. Spatial data management - vector formats
12. Spatial data management - raster formats
13. Graph construction and management in Python
14. Graph algorithms I. - shortest path
15. Graph algorithms II. - minimum spanning tree
16. Spatial indexing
17. Geometric algorithms - Convex Hull
18. Clustering and classification

Appendices

1. Strings
2. Mathematical operations

Exercise books

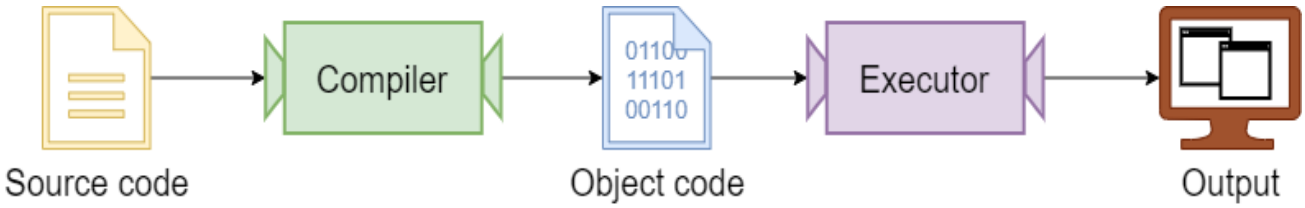
1. Exercise Book 1
2. Exercise Book 2
3. Exercise Book 3
4. Exercise Book 4
5. Exercise Book 5

Chapter 1: Python introduction

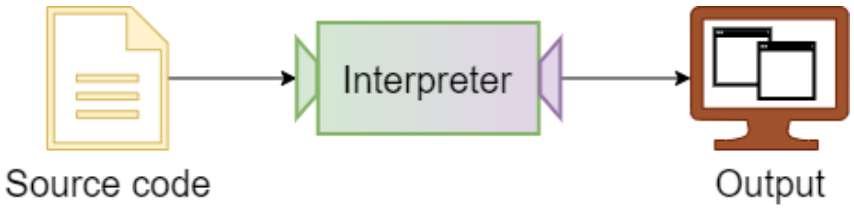
Compiled and interpreted languages

There are two major categories of programming languages: **compiled and interpreted**.

- A compiled language is a language that is turned by a compiler into direct machine code that runs upon the CPU. (Or it might run on a virtual machine stack like the JavaVM or the .NET runtime.)



- An interpreted language is a language that is read in its raw form and executed a statement at a time without being first compiled.



Python is an interpreted language.

How Jupyter Notebook works

A notebook contains cells, each cell is a logically separate and independent information.

There are 2 main type of cells:

- Markdown cells
  - Write documentation and comments
  - Can be formatted with *markdown* syntax
  - See this tutorial on how to use markdown (https://guides.github.com/features/mastering-markdown/)
- Code cells
  - Write code
  - Evaluate code on the fly

Note: you can find a *User Interface Tour* in the *Help* menu.

Let's get Python started! Hello world!

Literals

Literals are constants primitive values. They can be e.g. numbers or strings (texts). Strings are surrounded with quotation marks or apostrophes.

In [1]:

```
"Hello World"
```

Out[1]:

```
'Hello World'
```

In [2]:

```
'Hello Earth'
```

Out[2]:

```
'Hello Earth'
```

In [3]:

```
42
```

Out[3]:

```
42
```

Print

Syntax: `print` is a function, which has an argument. The argument is surrounded by parentheses.

- The argument can be a string literal between quotation marks
- Or a number literal
- Or a variable *(we will cover that later)*

In [4]:

```
print('Hello World')
print("Hello Earth")
print(4)
```

```
Hello World
Hello Earth
4
```

Task: what is the problem with the following codes?

In [5]:

```
print(Hello ELTE!)
```

```
  File "<ipython-input-5-10f8028f182a>", line 1
    print(Hello ELTE!)
                  ^
SyntaxError: invalid syntax
```

In [6]:

```
print "Hello Faculty of Informatics!"
```

```
  File "<ipython-input-6-015d2f017565>", line 1
    print "Hello Faculty of Informatics!"
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean pr
int("Hello Faculty of Informatics!")?
```

Handling outer Python files

Viewing the content of an external file

Lines starting with the `%` symbol are not regular Python instructions, instead they are special commands for the Jupyter Notebook.

We can load the content of an external file with the special `%load` command:

In [7]:

```
# %load ../data/01_outerfile.py
print('Great! This line was print from an external file!')
```

```
Great! This line was print from an external file!
```

Note that after the loading the original `%load` command is commented out and instead the content of the external file is loaded.

Executing an external file

We can execute the content of an external file with the special `%run` command.

Specifying the `-i` flag is important, so the file is executed in the same environment with the Jupyter notebook. Therefore, if we e.g. declare a variable in the external script, it will be accessible in the code cells of the notebook. If the `-i` flag is not specified, the external Python file is evaluated in a separate environment.

In [8]:

```
%run -i ../data/01_outerfile.py
```

```
Great! This line was print from an external file!
```

Variables

Variables can be considered **containers**. You can put anything inside a container, **without specifying the size or type**, which would be needed in e.g. Java, C++ or C#. Note that Python is case-sensitive. Be careful about using letters in different cases.

When assigning values, we put the variable to be assigned to on the left-hand side (LHS), while the value to plug in on the right-hand side (RHS). LHS and RHS are connected by an equal sign (=), meaning assignment.

In [9]:

```
x = 3  # integer
y = 3.1 # floating point number
z = "Hello!" # strings
Z = "Wonderful!" # another string, stored in a variable upper-case z.
print(x)
print(y)
print(z)
print(Z)
```

```
3
3.1
Hello!
Wonderful!
```

You can do operations on numeric values as well as strings.

In [10]:

```
sum_ = x + y # int + float = float
print(sum_)
```

```
6.1
```

In [11]:

```
v = "World!"
sum_string = z + " " + v # concatenate strings
print(sum_string)
```

```
Hello! World!
```

Naming convention

There are two commonly used naming style in programming:

1. **camelCase**
2. **snake_case** or **lower_case_with_underscore**

All variable (function and class) names must start with a letter or underscore (_). You can include numbers.

In [12]:

```
myStringHere = 'my string'
myStringHere
```

Out[12]:

```
'my string'
```

In [13]:

```
x = 3 # valid
x_3 = "xyz" # valid
```

In [14]:

```
3_x = "456" # invalid. Numbers cannot be in the first position.
```

```
  File "<ipython-input-14-520aa7218b05>", line 1
    3_x = "456" # invalid. Numbers cannot be in the first position.
     ^
SyntaxError: invalid decimal literal
```

You can choose either camel case or snake case. Always make sure you use one convention consistently across one project.

Simple Data Structures

In this section, we go over some common primitive data types in Python. While the word *primitive* looks obscure, we can think of it as the most basic data type that cannot be further decomposed into simpler ones.

Numbers

For numbers without fractional parts, we say they are ***integer***. In Python, they are called `int` .

In [15]:

```
x = 3
print(type(x))
```

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

For numbers with fractional parts, they are floating point numbers. They are named `float` in Python.

In [16]:

```
y = 3.0
print(type(y))
```

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

We can apply arithmetic to these numbers. However, one thing we need to be careful about is ***type conversion***. See the example below.

In [17]:

```
z = 2 * x
print(type(z))
```

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

In [18]:

```
z = y + x
print(type(z))
```

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

Text/Characters/Strings

In Python, we use `str` type for storing letters, words, and any other characters.

To initialize a string variable, you can use either double or single quotes.

In [19]:

```
my_word = "see you"
print(type(my_word))
```

```
<class 'str'>
```

Unlike numbers, `str` is an iterable object, meaning that we can iterate through each individual character. You can think of strings as a sequence of characters (or a **list** of characters, which we will cover later). In this case, indices and bracket notations can be used to access specific ranges of characters.

In [20]:

```
print(my_word[1])   # character with index 1; Python starts with 0 NOT 1
print(my_word[2:6]) # [start, end), end is exclusive
print(my_word[-1])  # -1 means the last element
```

```
e
e yo
u
```

We can also use `+` to *concatenate* different strings

In [21]:

```
my_word + ' tomorrow'
```

Out[21]:

```
'see you tomorrow'
```

Formatted strings

Print with formatting with the `format()` function and using placeholders:

In [22]:

```
print("The sum of x and y is {0}".format(sum_))
```

```
The sum of x and y is 6.1
```

In [23]:

```
print("The string `sum_string` is '{0}'".format(sum_string))
```

```
The string `sum_string` is 'Hello! World!'
```

In [24]:

```
print("The sum of x and y is {0} and the string `sum_string` is '{1}''".format(s
um_, sum_string))
```

```
The sum of x and y is 6.1 and the string `sum_string` is 'Hello! Wor
ld!''
```

Boolean

Boolean type comes in handy when we need to check conditions. For example:

In [25]:

```
my_error = 1.6
compare_result = my_error < 0.1
print(compare_result)
```

```
False
```

In [26]:

```
print(type(compare_result))
```

```
<class 'bool'>
```

There are two and only two valid Boolean values: `True` and `False`. We can also think of them as `1` and `0`, respectively.

In [27]:

```
print(my_error > 0)
```

```
True
```

When we use Boolean values for arithmetic operations, they will become `1 / 0` implicitly.

In [28]:

```
print((my_error > 0) + 2)
```

```
3
```

Type Conversion

Since variables in Python are dynamically typed, we need to be careful about type conversion.

When two variables share the same data type, there is not much to be worried about:

In [29]:

```
s1 = "no problem. "
s2 = "talk to you later"
s1 + s2
```

Out[29]:

```
'no problem. talk to you later'
```

But be careful when we are mixing variables up:

In [30]:

```
a = 3 # recall that this is an ____?
b = 2.7 # how about this?
c = a + b # what is the type of `c`?

print(c)
print(type(c))
```

```
5.7
<class 'float'>
```

To make things work between string and numbers, we can explicitly convert numbers into `str` :

In [31]:

```
print(s1 + 3)
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
~/Repos/Lecture/elte_teralg/data/01_outerfile.py in <module>
----> 1 print(s1 + 3)

TypeError: can only concatenate str (not "int") to str
```

In [32]:

```
print(s1 + str(3))
```

```
no problem. 3
```

We may also convert strings to numbers:

In [33]:

```
s3 = "42"
d = int(s3)
print(type(s3), type(d))
```

```
<class 'str'> <class 'int'>
```

User input

User input can be easily requested in Python:

```
k = input('Question')
```

The question text is arbitrary. The user will see a console input prompt. The typed input will be stored in the `k` variable by Python. Example:

In [34]:

```
k = input('What is your name? ')
print('Hello ' + k)
```

```
Hello John
```

Question: what is the type of the value in variable `k` ?

In [35]:

```
print(type(k))
```

```
<class 'str'>
```

Exercise

Task: Query the height of the user.

Print afterward that *Your height is XXX centimeter.*

Print the type of the variable storing the height of the user.

In [36]:

```
height = input("What is your height? ")
print("Your height is {0} centimeters.".format(height))
print(type(height))
```

```
Your height is 186 centimeters.
<class 'str'>
```

Variables assigned by the **input** function will always contain strings. (We will cover error handling later.)

Summary exercise on Python variables, data types and user input

Task: Question the user for the amount of days he/she works a week and his/her gross salary per hour (in euro).

Calculate and print the monthly gross salary of the user.

Calculate and print the monthly net salary of the user, assuming that the net salary is 65% of the gross salary.

Assume that a month consists of 4 weeks and a working day consists of 8 working hours.

In [37]:

```
work_days = int(input("Number of work days per week? "))
salary_hour = int(input("Salary per hour? "))
salary_gross = 4 * work_days * 8 * salary_hour
salary_net = salary_gross * 0.65

print("Gross salary: {0} euros / month".format(salary_gross))
print("Net salary: {0} euros / month".format(salary_net))
```

```
Gross salary: 2560 euros / month
Net salary: 1664.0 euros / month
```

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)
```

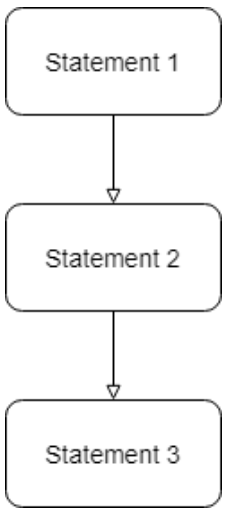
```
3.141592653589793
```

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)))
```

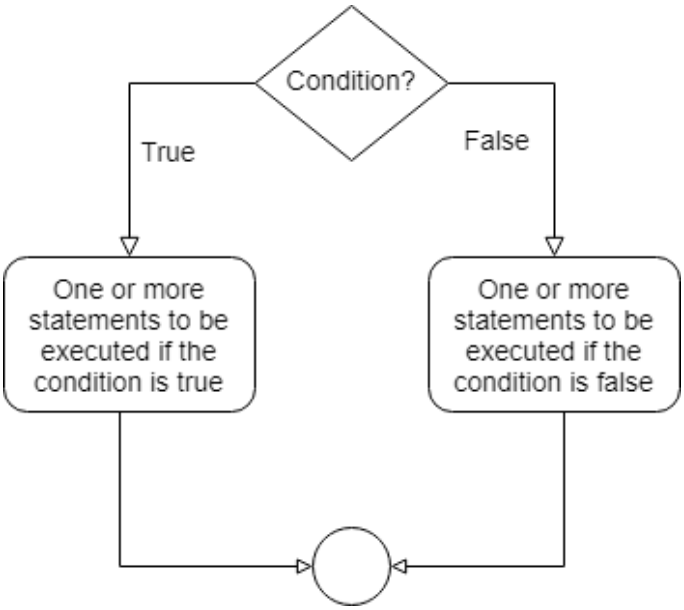
```
Now number is now 42 with type of <class 'int'>
```

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
```

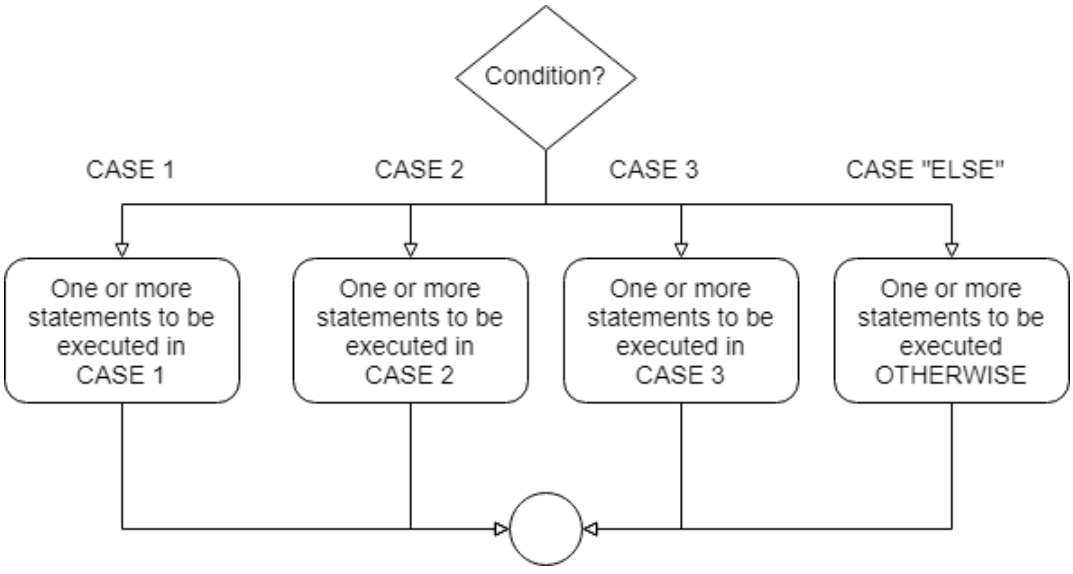
Three (or more) way conditions

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
```



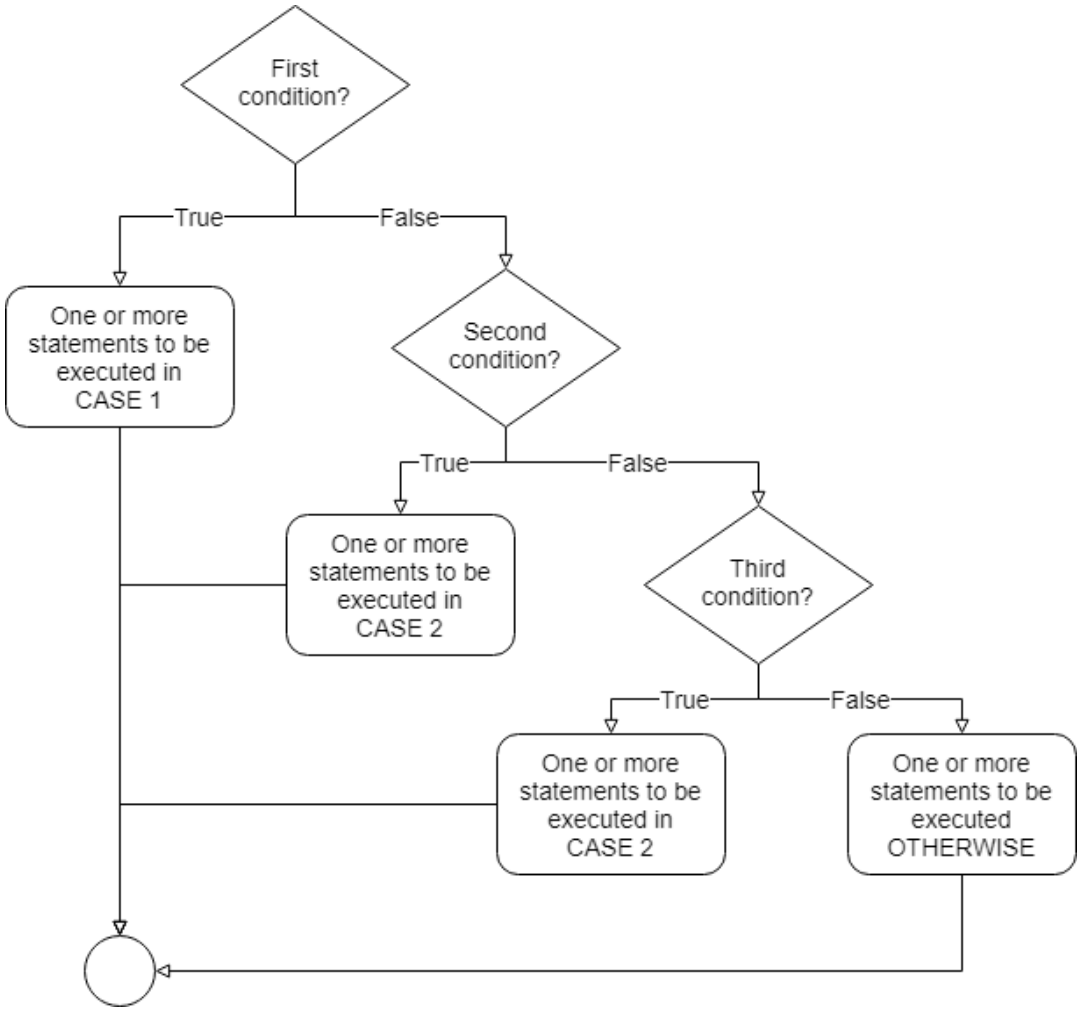
Actually, there are no three (or more) way conditional statements, only two-way conditions. The `elif` keyword is just a little *"syntax sugar"* to provide an easier understandable version of nested, 2-way conditions.

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
```



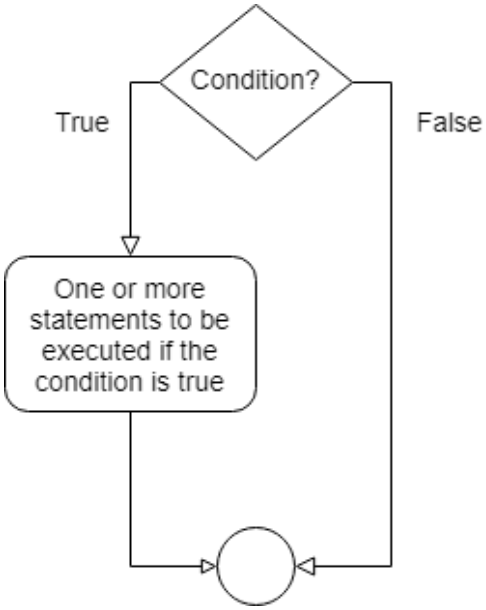
One-way conditions

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 ~= 0.000000000000000000000000000001
```

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
```

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
```

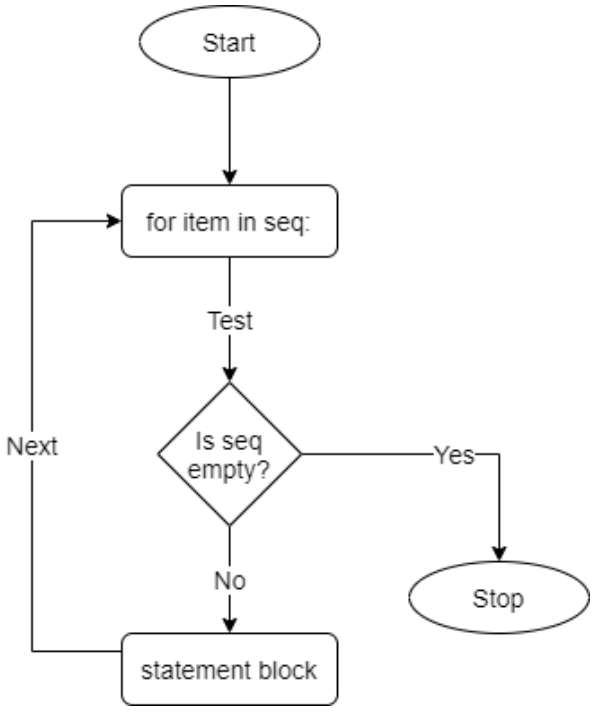
Note the identation of the `print` statement, which means that the `print` statement is *"inside"* the `for` loop and will be executed on each iteration.

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

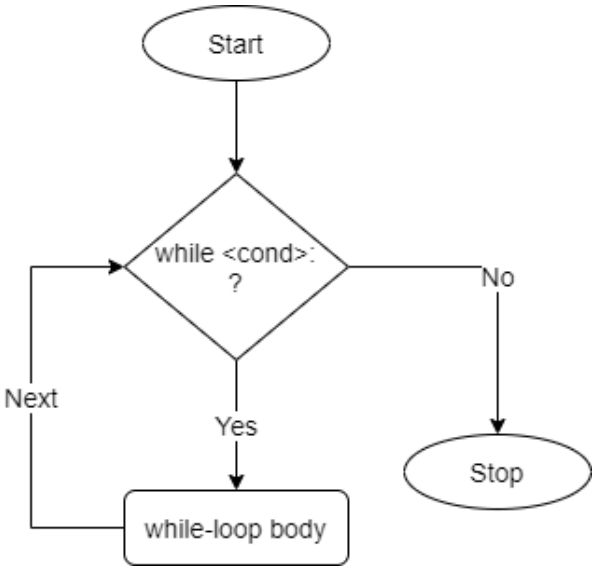
Use `for` when you know **the exact number of iterations**; use `while` when you **do not (e.g., checking convergence)**.

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
```

Chapter 4: Functions

We have already used (*called*) functions multiple times, like `print()`, `int()`, `len()` or `randint()`.

In [1]:

```
import random

name = "James Bond"
number = int("007")
print(name)
print(len(name))
print(number)
print(random.randint(1, 100))
```

```
James Bond
10
7
67
```

The concept of a function in programming is very close to the mathematical definition of a function. These functions can:

- accept 0, 1 or multiple parameters;
- return a value or not;
- meanwhile causing *side-effects*, like printing a message on the console output.

Defining custom functions

By defining custom functions, the redundancy in the code can be reduced. A custom function can be defined with the `def` keyword:

```
def function_name ( <parameter_list> ):
    function_statement
```

By *defining* a function we are just "storing" it, be we are not executing it yet. For example:

In [2]:

```
def hello():
    print("Hello World!")
```

Now we may *call* the function even multiple types to execute it:

In [3]:

```
print("First line")
hello()
print("Second line")
hello()
```

```
First line
Hello World!
Second line
Hello World!
```

What will be the type of a function?

In [4]:

```
print(type(hello))
```

```
<class 'function'>
```

Parameters

Functions may have zero, one or multiple parameters, which are given between parentheses as *variables* to the function:

In [5]:

```
def greet(name):
    print("Hello " + name + "!")
```

In [6]:

```
greet("John")
someName = "Jane"
greet(someName)
```

```
Hello John!
Hello Jane!
```

In the above example the variable `name` is a **parameter**. The literal value `John` and the variable `someName` are the **arguments** of the function call. So parameters are the generalized variables in the function definitions, while arguments are the actual, concrete values in a function call.

Return values

Functions can return a value with the `return` statement. When a function reaches a return statement, the execution of the function is stopped and the given value is returned. (A function can contain multiple return statements when using conditions or iterations.)

Let's write the `sum_list` function, which receives a list of numerical values as a parameter and returns the sum of the numbers!

In [7]:

```
def sum_list(numbers): # numbers is assumed to be a list of numerical values
    sum_value = 0
    for num in numbers:
        sum_value += num
    return sum_value
    print("This line will never get printed")
```

Until now, we have only defined the function, now we can call it:

In [8]:

```
nums = [12, 8, 37, 21, 67, 42, 25]
print(sum_list(nums))
```

```
212
```

A function can contain multiple `return` statements. After the first `return` statement reached, the execution of the function is stopped.

Let's write the `average` function, which receives a list of numerical values as a parameter and returns the average of the numbers. If the list is empty, the returned value shall be `None`. Reuse the previous `sum_list` function to produce the sum of the values.

In [9]:

```
def average(numbers): # numbers is assumed to be a list of numerical values
    if len(numbers) == 0:
        return None
    else:
        return sum_list(numbers) / len(numbers)
    print("This line will never get printed")
```

In [10]:

```
nums = [12, 8, 37, 21, 67, 42, 25]
print(average(nums))
```

```
30.285714285714285
```

Remark: the `None` keyword is used to define a no value at all (also called *null value*). `None` is not the same as `0`, `False`, or an empty string. `None` has a data type of its own (`NoneType`) and only `None` can be `None`.

Functions returning a value are called *fruitful* functions. Functions without a return value are called *void* functions. In that case the returned value is *None*.

In [11]:

```
greet("Matthew")
result = greet("Andrew")
print(result)
```

```
Hello Matthew!
Hello Andrew!
None
```

Multiple parameters

Functions may have multiple parameters. In such a case the arguments are matched to the parameters in the same order as they are listed.

In [12]:

```
def add(a, b):
    print("Adding {0} and {1}".format(a,b))
    c = a + b
    return c

result = add(10, 32)
print(result)
result = add(-5, 8)
print(result)
```

```
Adding 10 and 32
42
Adding -5 and 8
3
```

Default arguments

Python allows function parameters to have default values. If the function is called without the argument, the parameter gets its default value.

In [13]:

```
def power(base, exp = 10):
    return base ** exp

print(power(2, 6))
print(power(2, 10))
print(power(2))
```

```
64
1024
1024
```

IMPORTANT: if a parameter has a default value, all other parameters following it must have a default value too! E.g. this is **invalid**:

```
def power(base = 2, exp):
    return base ** exp
```

Passing arguments by their position or name

In Python, we can either pass the arguments by their *position* - as we have seen it so far:

In [14]:

```
print(power(2, 6))
print(power(6, 2))
```

```
64
36
```

Alternatively arguments can be passed by the respective *parameter name*:

In [15]:

```
print(power(base = 2, exp = 6))
print(power(exp = 6, base = 2))
print(power(2, exp = 6))
```

```
64
64
64
```

Note: passing arguments by their name is especially useful when:

- a function has many parameters and the function call is much more *readable* when the parameters are passed by their name;
- a function has many parameters with default values and we would like to override the default value for only a few of them.

Built-in functions

There are many built-in functions in Python for common use cases, e.g. for looking up the maximum/minimum value in a list, or to calculate the sum of a list:

In [16]:

```
print("Maximum value in nums: {0}".format(max(nums)))
print("Minimum value in nums: {0}".format(min(nums)))
print("Sum of the values in nums: {0}".format(sum(nums)))
```

```
Maximum value in nums: 67
Minimum value in nums: 8
Sum of the values in nums: 212
```

A comprehensive list can be found in the documentation:
https://docs.python.org/3/library/functions.html (https://docs.python.org/3/library/functions.html)

Note: defining a variable or function with the same of an existing (even builtin) function will hide it.

In [17]:

```
print("Maximum value in nums: {0}".format(max(nums)))
max = 42
print("Maximum value in nums: {0}".format(max(nums))) # yields error ,as max in
 an integer now, not a function
```

```
Maximum value in nums: 67

---------------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-17-582e3602d77e> in <module>
      1 print("Maximum value in nums: {0}".format(max(nums)))
      2 max = 42
----> 3 print("Maximum value in nums: {0}".format(max(nums))) # yiel
ds error ,as max in an integer now, not a function

TypeError: 'int' object is not callable
```

Modules

In Python a logical unit of definitions (*variables, functions, classes*) shall be put in a standalone file to support the easy reuse of the code. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module.

There are many built-in modules, we have already used the `math` and the `random` module for example. By using modules we can access preinstalled libraries and use them, so our code will be shorter and more compact.

In [18]:

```
import math
print(math.pi) # using a variable definition from module math
print(math.factorial(10)) # using a function definition from module math
```

```
3.141592653589793
3628800
```

You can easily get a documentation for a module, by either looking it up in the reference: https://docs.python.org/3/library/math.html (https://docs.python.org/3/library/math.html)

Or fetching it dynamically with the `help` function:

In [19]:

```
help(math)
```

```
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.

    atan(x, /)
        Return the arc tangent (measured in radians) of x.

    ...

    sqrt(x, /)
        Return the square root of x.

    tan(x, /)
        Return the tangent of x (measured in radians).

    tanh(x, /)
        Return the hyperbolic tangent of x.

    trunc(x, /)
        Truncates the Real x to the nearest Integral toward 0.

        Uses the __trunc__ magic method.

DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
    tau = 6.283185307179586

FILE
    (built-in)
```

Summary exercises on functions

Task 1: Fahrenheit to Celsius

Write a function `fahr2cels`, which computes the temperature in Celcius from Fahrenheit. The formula is the following:

$$C = \frac{5}{9} * (F - 32)$$

Where C is the degree in Celsius and F is the degree in Fahrenheit.

Write a program which prints out the appropriate Celsius values for each degree in Fahrenheit between 0 and 100, using an incremental step of 10.

In [20]:

```
def fahr2cels(f):
    c = 5 / 9 * (f - 32)
    return c

for fahr in range(0, 101, 10):
    cels = fahr2cels(fahr)
    print("Fahr = {0}, Cels = {1:.4f}".format(fahr, cels))
```

```
Fahr = 0, Cels = -17.7778
Fahr = 10, Cels = -12.2222
Fahr = 20, Cels = -6.6667
Fahr = 30, Cels = -1.1111
Fahr = 40, Cels = 4.4444
Fahr = 50, Cels = 10.0000
Fahr = 60, Cels = 15.5556
Fahr = 70, Cels = 21.1111
Fahr = 80, Cels = 26.6667
Fahr = 90, Cels = 32.2222
Fahr = 100, Cels = 37.7778
```

Task 2: Prime check

Write a function `isPrime` which determines whether a number received as a parameter is a prime or not. (You may reuse your algorithm from the previous lecture.)

Wrtite a program which request a number from the user and tests whether it is a prime or not. Check whether the user input is really an integer number or not.

In [21]:

```
import math

def isPrime(number):
    # Handle 0 and 1 as a special case
    if number < 2:
        return False

    # Numbers >= 2 are tested whether they have any divisors
    for i in range(2, int(math.sqrt(number) + 1)):
        #print("Testing divisor %d" % i)
        if number % i == 0:
            # If we found a divisor, we can stop checking, because the number is
 NOT a prime
            return False

    # If no divisors were found, then the number is a prime
    return True

try:
    num = int(input("Number to check: "))
    if isPrime(num):
        print("{0} is a prime".format(num))
    else:
        print("{0} is NOT a prime".format(num))
except:
    print("That was not a number!")
```

```
37 is a prime
```

Task 3: Word count

Request a string input from the user (a sentence). Write a function `wordCount` which count the words in the sentence!

In [22]:

```
def wordCount(sentence):
    spaceCount = 0
    for char in sentence:
        if char == ' ':
            spaceCount += 1
    return spaceCount + 1

userInput = input('Say a sentence: ')
print('Your sentence consisted of {0} words.'.format(wordCount(userInput)))
```

```
Your sentence consisted of 9 words.
```

Hint: count the spaces in the input string.

Task 4: Monotonity

A) Given a list a numbers, write a function `isMonotonous` which decides whether the sequence is monotically increasing or not?

Sample input:

In [23]:

```
list1 = [10, 20, 50, 400, 600]
list2 = [10, 20, 50, 40, 600]
list3 = [1000, 500, 200, 50, 10]
list4 = [10, 20, 50, 50, 300]
```

In [24]:

```
def isMonotonous(numbers):
    # Assume that the list is monotically increasing and search for an index pai
r where it is not true!
    for i in range(1, len(numbers)):
        if numbers[i - 1] > numbers[i]:
            return False
    # If no such errornous index pair was found, then the list was really monoti
cally increasing.
    return True

print("List 1: {0}".format(isMonotonous(list1)))
print("List 2: {0}".format(isMonotonous(list2)))
print("List 3: {0}".format(isMonotonous(list3)))
print("List 4: {0}".format(isMonotonous(list4)))
```

```
List 1: True
List 2: False
List 3: False
List 4: True
```

B) Modify the previous function, so it decides whether the sequence is monotonous or not. (It can be either increasing or decreasing.)

In [25]:

```
def isMonotonous(numbers):
    # Check for monotically increasing
    isIncreasing = True
    for i in range(1, len(numbers)):
        if numbers[i - 1] > numbers[i]:
            isIncreasing = False
            break

    # Check for monotically decreasing
    isDecreasing = True
    for i in range(1, len(numbers)):
        if numbers[i - 1] < numbers[i]:
            isDecreasing = False
            break

    # Return whether either one of the 2 conditions were true!
    return isIncreasing or isDecreasing

print("List 1: {0}".format(isMonotonous(list1)))
print("List 2: {0}".format(isMonotonous(list2)))
print("List 3: {0}".format(isMonotonous(list3)))
print("List 4: {0}".format(isMonotonous(list4)))
```

```
List 1: True
List 2: False
List 3: True
List 4: True
```

Chapter 5: Basic algorithms

Initialize a list of random numbers to work with in the following exercises. Generate 10 numbers between 1 and 100:

In [1]:

```
import random

random.seed(42) # to reproduce the same results
data = []
for i in range(10):
    data.append(random.randint(1, 100))
print(data)
```

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

Same with using Python's list generator expressions:

In [2]:

```
random.seed(42) # to reproduce the same results
data = [random.randint(1, 100) for i in range(10)]
print(data)
```

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

Help: given a list of numbers in variable `numbers`, produce the `halves` list, which contains each number divided by 2:

In [3]:

```
numbers = [10, 20, 30, 40, 50]
print(numbers)

# iteration
halves = []
for x in numbers:
    halves.append(x / 2)
print(halves)

# list generation
halves = [x / 2 for x in numbers]
print(halves)
```

```
[10, 20, 30, 40, 50]
[5.0, 10.0, 15.0, 20.0, 25.0]
[5.0, 10.0, 15.0, 20.0, 25.0]
```

Summation

Let $f : [m..n] \to H$ be a function. Let the addition operator $+$ be defined over the elements of H, which is an associative operation with a left identity element 0. Our task is to summarize the values of f function over the interval. Formally:

$$s = \sum_{i=m}^{n} f(i)$$

Theoretical way

In [4]:

```
result = 0
for i in range(0, len(data)):
    result += data[i]
print("Sum: {0}".format(result))
```

```
Sum: 420
```

Pythonic way

In [5]:

```
result = 0
for value in data:
    result += value
print("Sum: {0}".format(result))
```

```
Sum: 420
```

Built-in function

In [6]:

```
result = sum(data)
print("Sum: {0}".format(result))
```

```
Sum: 420
```

Counting

Given the $[m..n]$ interval, count the number of items inside it. Formally:

$$s = \sum_{i=m}^{n} 1$$

Theoretical way

In [7]:

```
result = 0
for i in range(0, len(data)):
    result += 1
print("Count: {0}".format(result))
```

```
Count: 10
```

Pythonic way

In [8]:

```
result = sum([1 for _ in data])
print("Count: {0}".format(result))
```

```
Count: 10
```

Remark: a single underscore (_) is a valid variable name. We usually name a variable like this to emphasize that this variable will not be used later.

Built-in function

In [9]:

```
result = len(data)
print("Count: {0}".format(result))
```

```
Count: 10
```

Maximum search

Let $f : [m..n] \to H$ be a function, $m \leq n$. Over the elements of H let a total ordering relation be defined (reflexivity, antisymmetry, transitivity and connexity), with a symbol $\leq$, for the strict version $<$. Our task is to determine the greatest value in the interval. Also determine an element of the interval, where function f evaluates to this greatest value. Formally:

$$max = f(ind) \wedge \forall i \in [m..n] : f(i) \leq f(ind)$$

Theoretical way

In [10]:

```
result = data[0]
index = 0
for i in range(1, len(data)): # we don't need to compare the 0th element
    if data[i] > result:
        result = data[i]
        index = i
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

Pythonic way

In [11]:

```
result = data[0]
index = 0
for idx, value in enumerate(data):
    if value > result:
        result = value
        index = idx
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

Little optimization to skip the 0^{th} element:

In [12]:

```
result = data[0]
index = 0
for idx, value in enumerate(data[1:], start = 1):
    if value > result:
        result = value
        index = idx
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

Built-in function

In [13]:

```
result = max(data)
print("Max: {0}".format(result))
```

```
Max: 95
```

If the index of the element is also needed:

In [14]:

```
result = max(data)
index = data.index(result)
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

Note: this will iterate over the list twice, hence the computational cost is also doubled.

Linear search

Let $\beta : [m..n] \to \mathbb{L}$ condition be defined. Determine the first element of the interval which fulfills the condition (if any). Formally:

$$l = (\exists i \in [m..n] : \beta(i))$$
$$l \to (ind \in [m..n] \wedge \beta(ind) \wedge \forall i \in [m..ind-1] : \neg\beta(i))$$

Beta condition

Introduce an `is_odd` function which determines whether a number is odd or not:

In [15]:

```
def is_odd(number):
    return number % 2 != 0
```

Theoretical way

In [16]:

```
result = 0
index = 0
found = False
i = 0
while not found and i < len(data):
    if is_odd(data[i]):
        result = data[i]
        index = i
        found = True
    i += 1
if found:
    print("Linear search: {0}, Index: {1}".format(result, index))
else:
    print("Linear search did not found an appropriate item")
```

```
Linear search: 15, Index: 1
```

Pythonic way

In [17]:

```
result = [x for x in data if is_odd(x)]
if len(result) > 0:
    print("Linear search: {0}".format(result))
else:
    print("Linear search did not found an appropriate item")
```

```
Linear search: [15, 95, 29, 95]
```

Built-in function

In [18]:

```
result = filter(is_odd, data)
print("Linear search: {0}".format(list(result)))
```

```
Linear search: [15, 95, 29, 95]
```

Here `result` is a special filter object which can be either converted to a list to get all results (as above) or dynamically evaluated and step to the next result with the `next()` function:

In [19]:

```
result = filter(is_odd, data)
print("Linear search: {0}".format(next(result, None))) # None is the default val
ue to use if no number was odd.
```

```
Linear search: 15
```

Conditional summation

The algorithm of summation can be further generalized when a $\beta : [m..n] \to \mathbb{L}$ condition is defined to restrict the set of elements.

Let $f : [m..n] \to H$ be a function and $\beta : [m..n] \to \mathbb{L}$ a condition. Let the addition operator $+$ be defined over the elements of H, which is an associative operation with a left identity element 0. Our task is to summarize the values of f function over the interval where the β condition is fulfilled. Formally:

$$s = \sum_{\substack{i=m \\ \beta(i)}}^{n} f(i)$$

Theoretical way

In [20]:

```
result = 0
for i in range(0, len(data)):
    if is_odd(data[i]):
        result += data[i]
print("Sum: {0}".format(result))
```

```
Sum: 234
```

Pythonic way

In [21]:

```
result = 0
for value in data:
    if is_odd(value):
        result += value
print("Sum: {0}".format(result))
```

```
Sum: 234
```

Built-in function

In [22]:

```
result = sum(filter(is_odd, data))
print("Sum: {0}".format(result))
```

```
Sum: 234
```

Conditional counting

Let $\beta : [m..n] \to \mathbb{L}$ be a condition. Count how many items of the interval fulfills the condition! Formally:

$$s = \sum_{\substack{i=m \\ \beta(i)}}^{n} 1$$

Theoretical way

In [23]:

```
result = 0
for i in range(0, len(data)):
    if is_odd(data[i]):
        result += 1
print("Count: {0}".format(result))
```

```
Count: 4
```

Pythonic way

In [24]:

```
result = sum([1 for x in data if is_odd(x)])
print("Count: {0}".format(result))
```

```
Count: 4
```

Built-in function

In [25]:

```
result = len(list(filter(is_odd, data)))
print("Count: {0}".format(result))
```

```
Count: 4
```

Conditional maximum search

The algorithm of maximum search can be further generalized with combining the $\beta : [m..n] \to \mathbb{L}$ condition used in *linear search* as a restriction. Note that now the existence of a maximum value is not guaranteed.

Let $f : [m..n] \to H$ be a function and $\beta : [m..n] \to \mathbb{L}$ a condition. Over the elements of H let a total ordering relation be defined (reflexivity, antisymmetry, transitivity and connexity), with a symbol $\leq$, for the strict version $<$. Our task is to determine the greatest value in the interval which fulfills the β condition. Also determine an element of the interval, where function f evaluates to this greatest value. Formally:

$$l = (\exists i \in [m..n] : \beta(i))$$
$$l \to (\beta(ind) \wedge max = f(ind) \wedge \forall i \in [m..n] : \beta(i) \to f(i) \leq f(ind))$$

Theoretical way

In [26]:

```
found = False
result = 0
index = 0
for i in range(0, len(data)):
    if is_odd(data[i]) and (not found or data[i] > result):
        found = True
        result = data[i]
        index = i
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

The `found` variable can be omitted if initialize the `result` variable with the special `None` value and compare to that:

In [27]:

```
result = None
index = -1
for i in range(0, len(data)):
    if is_odd(data[i]) and (result == None or data[i] > result):
        result = data[i]
        index = i
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

Pythonic way

In [28]:

```
result = None
index = -1
for idx, value in enumerate(data):
    if is_odd(value) and (result == None or value > result):
        result = value
        index = idx
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

Built-in function

In [29]:

```
result = max(filter(is_odd, data))
print("Max: {0}".format(result))
```

```
Max: 95
```

Exercise

Task: the name, area and population data for the neighbouring countries are given in the `countries`, `areas` and `popultions` lists below. Calculate the population density for each neighbouring country and display it. Determine which country has the highest population density.

In [30]:

```
countries = ['Austria', 'Slovakia', 'Ukraine', 'Romania', 'Serbia', 'Croatia',
'Slovenia']
areas = [83871, 49037, 603500, 238397, 88361, 56594, 20273]
populations = [8877036, 5450017, 42010063, 19405156, 6963764, 4130304, 2084301]
```

In [31]:

```
densities = []
for i in range(len(countries)):
    densities.append(populations[i] / areas[i])
    print('{0}: {1:.2f} persons/km2'.format(countries[i], densities[i]))
```

```
Austria: 105.84 persons/km2
Slovakia: 111.14 persons/km2
Ukraine: 69.61 persons/km2
Romania: 81.40 persons/km2
Serbia: 78.81 persons/km2
Croatia: 72.98 persons/km2
Slovenia: 102.81 persons/km2
```

In [32]:

```
result = max(densities)
index = densities.index(result)
print("Max: {0}, Index: {1}, Country: {2}".format(result, index, countries[index
]))
```

```
Max: 111.14091400371149, Index: 1, Country: Slovakia
```

Logarithmic search (optional)

Also called binary search.

Let $f : [m..n] \to H$ be a *monotonically increasing* function. Over the elements of H let a total ordering relation be defined (reflexivity, antisymmetry, transitivity and connexity), with a symbol $\leq$, for the strict version $<$. Determine whether function f evaluates to a given value $h \in H$ at any location. If yes, specify such a location. Formally:

$$l = (\exists i \in [m..n] : f(i) = h) \wedge l \to f(ind) = h$$

In [33]:

```
def log_search(elements, value):
    first = 0
    last = len(elements) - 1
    while first <= last:
        i = (first + last) // 2
        if elements[i] == value:
            return i
        elif elements[i] < value:
            first = i + 1
        else:
            last = i - 1
    return -1

data_sorted = sorted(data)
print("Sorted data: {0}".format(data_sorted))

index = log_search(data_sorted, data[0])
print("Logarithmic search: value={0}, index={1}".format(data[0], index))
```

```
Sorted data: [4, 14, 15, 18, 29, 32, 36, 82, 95, 95]
Logarithmic search: value=82, index=7
```

Chapter 6: Sorting algorithms and complexity

Sorting is one of the most thoroughly studied algorithms in computer science. There are dozens of different sorting implementations, some applicable in general, others efficient in specific circumstances only.

Sorting can be used to solve a variety of problems, to mention a few basic ones:

- **Searching** for an item on a list works much faster if the list is sorted.
- **Selecting** items from a list based on their relationship to the rest of the items is easier with sorted data. For example, finding the k^{th}-largest or smallest value, or finding the median value of the list, is much easier when the values are in ascending or descending order.
- **Finding duplicate** values in a list can be done very quickly when the list is sorted.
- Analyzing the frequency **distribution** of items on a list is very fast if the list is sorted. For example, finding the element that appears most or least often is relatively straightforward with a sorted list.

Generate a list of random numbers to sort:

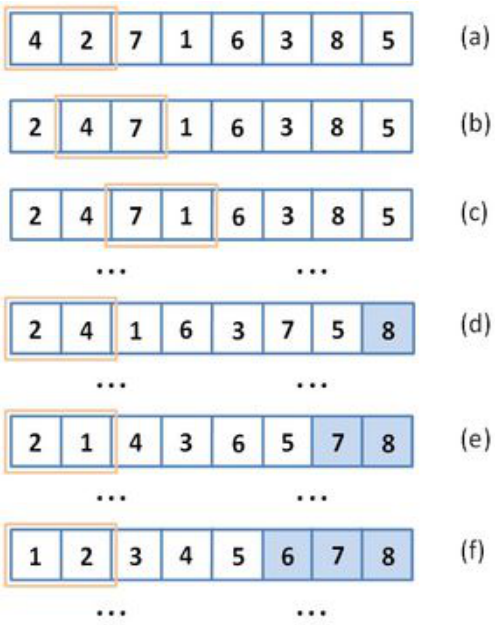
In [1]:

```
import random

originalNumbers = [random.randint(1, 100) for _ in range(20)]
print(originalNumbers)
```

```
[26, 4, 52, 74, 88, 51, 32, 21, 98, 3, 18, 52, 62, 80, 16, 52, 43, 7
1, 90, 17]
```

Bubble sort

Bubble sort is a simple sorting algorithms that works by repeatedly swapping the adjacent elements if they are in wrong order. In one iteration the largest element will be moved to the end of the array, thus reducting the problem to a shorter array.



In [2]:

```
def swap(array, i, j):
    temp = array[i]
    array[i] = array[j]
    array[j] = temp
```

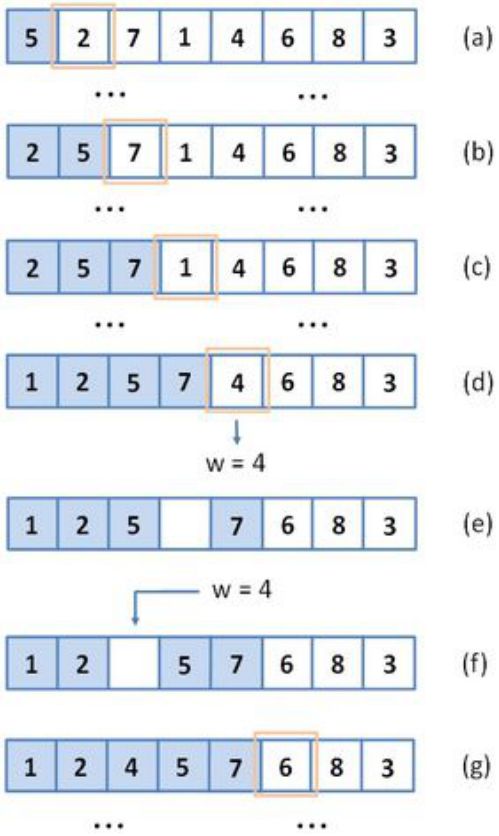
In [3]:

```
def bubbleSort(array):
    for end in range(len(array), 1, -1):
        for i in range(1, end):
            if array[i-1] > array[i]:
                swap(array, i-1, i)

numbers = originalNumbers.copy()
print("Unsorted: {0}".format(numbers))
bubbleSort(numbers)
print("Sorted: {0}".format(numbers))
```

```
Unsorted: [26, 4, 52, 74, 88, 51, 32, 21, 98, 3, 18, 52, 62, 80, 16,
52, 43, 71, 90, 17]
Sorted: [3, 4, 16, 17, 18, 21, 26, 32, 43, 51, 52, 52, 52, 62, 71, 7
4, 80, 88, 90, 98]
```

Insertion sort

Insertion sort is a simple sorting algorithm that maintains a sorted and an unsorted part of the array. Values from the unsorted part are picked and placed at the correct position in the sorted part.



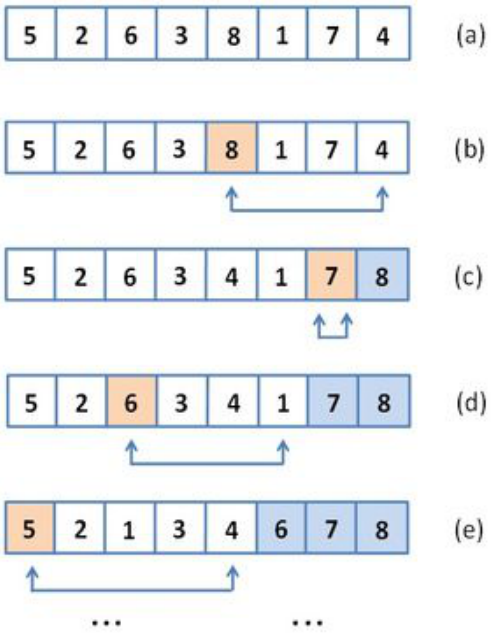
In [4]:

```
def insertionSort(array):
    for i in range(1, len(array)):
        value = array[i]
        j = i - 1
        while j >= 0 and array[j] > value:
            array[j + 1] = array[j]
            j -= 1
        array[j + 1] = value

numbers = originalNumbers.copy()
print("Unsorted: {0}".format(numbers))
insertionSort(numbers)
print("Sorted: {0}".format(numbers))
```

```
Unsorted: [26, 4, 52, 74, 88, 51, 32, 21, 98, 3, 18, 52, 62, 80, 16,
52, 43, 71, 90, 17]
Sorted: [3, 4, 16, 17, 18, 21, 26, 32, 43, 51, 52, 52, 52, 62, 71, 7
4, 80, 88, 90, 98]
```

Maximum sort (a.k.a. Selection sort)

Maximum sort algorithm sorts an array of elements by repeatedly finding the maximum element (considering ascending order) from an unsorted part and putting it at the end of it. Then the length of the unsorted part is reduced by 1.
The algorithm can also formulated as a *Minimum sort* and combined they often they named *Selection sort*.



In [5]:

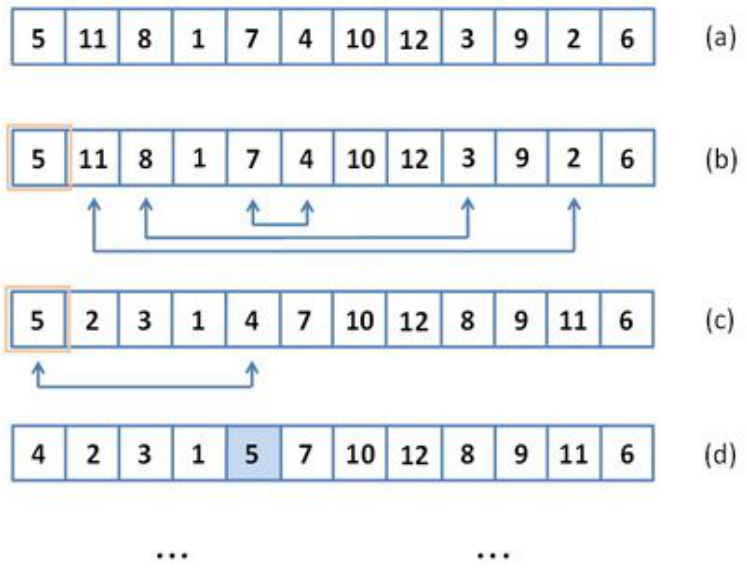
```
def maximumSort(array):
    for end in range(len(array), 1, -1):
        maxIdx = end - 1
        # maximum search algorithm
        for i in range(end):
            if array[i] > array[maxIdx]:
                maxIdx = i
        swap(array, end - 1, maxIdx)


numbers = originalNumbers.copy()
print("Unsorted: {0}".format(numbers))
maximumSort(numbers)
print("Sorted: {0}".format(numbers))
```

```
Unsorted: [26, 4, 52, 74, 88, 51, 32, 21, 98, 3, 18, 52, 62, 80, 16,
52, 43, 71, 90, 17]
Sorted: [3, 4, 16, 17, 18, 21, 26, 32, 43, 51, 52, 52, 52, 62, 71, 7
4, 80, 88, 90, 98]
```

Quicksort

Quicksort is a *Divide and Conquer* algorithm. It picks an element as *pivot* and partitions the given array around the picked pivot. The partitioning is executed that the algorithm puts the smaller element to the left of the pivot and the larger elements to the right of the pivot. Then the algorithm is executed recursively on the partitions.

There are many different versions on how to pick a "good" pivot element, the simplest solution is to always pick the first element.



Divide And Conquer algorithms in general works as follows:

- *Divide*: Divide the problem into more sub problems.
- *Conquer*: Solve the sub problems by calling recursively until sub problem solved is trivially solved.

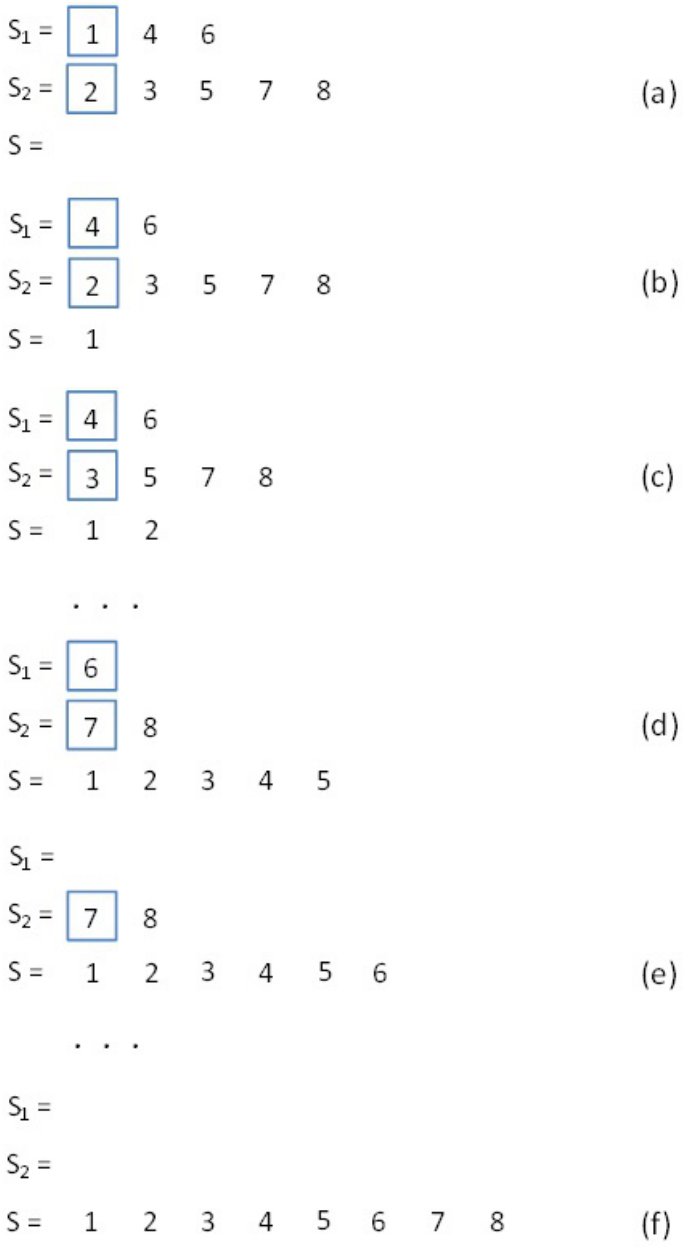
In [6]:

```
# Quicksorting
def quickSort(array):
    n = len(array)
    _quickSort(array, 0, n - 1)

# Quicksorting (partial array)
def _quickSort(array, u, v):
    if u >= v:
        return;

    k = _partition(array, u, v)
    _quickSort(array, u, k - 1)
    _quickSort(array, k + 1, v)

# Partinoning algorithm: move the pivot element to its position
def _partition(array, u, v):
    i = u + 1;
    j = v;
    while i <= j:
        while i <= v and array[i] <= array[u]:
            i += 1
        while j >= u + 1 and array[j] >= array[u]:
            j -= 1

        if i < j:
            swap(array, i , j)
            i += 1
            j -= 1

    swap(array, u, i - 1)
    return i - 1;


# Swap 2 items in a list
def swap(array, i, j):
    temp = array[i]
    array[i] = array[j]
    array[j] = temp

numbers = originalNumbers.copy()
print("Unsorted: %s" % numbers)
quickSort(numbers)
print("Sorted: %s" % numbers)
```

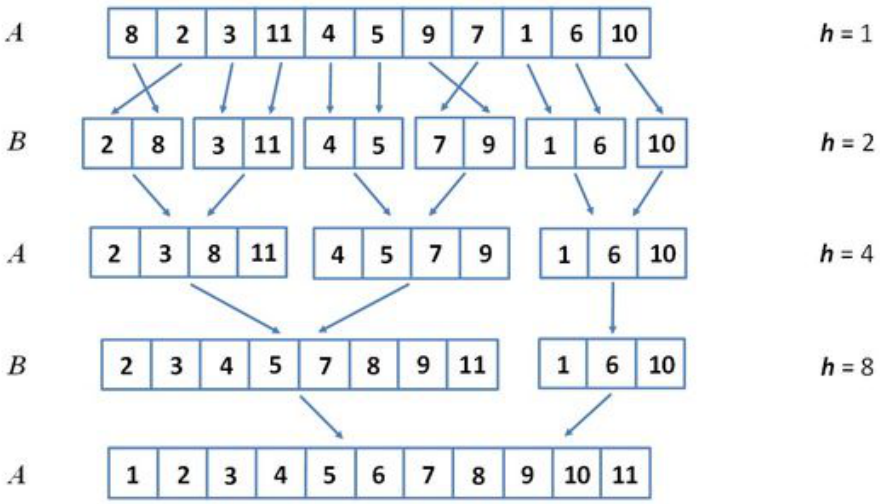
```
Unsorted: [26, 4, 52, 74, 88, 51, 32, 21, 98, 3, 18, 52, 62, 80, 16,
52, 43, 71, 90, 17]
Sorted: [3, 4, 16, 17, 18, 21, 26, 32, 43, 51, 52, 52, 52, 62, 71, 7
4, 80, 88, 90, 98]
```

Merge sort

Two sorted lists of data can be merged together by iterating through their elements only once.



The *Merge Sort* is also a *Divide and Conquer* algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. (We assume that the sorting of 2 elements is trivial.)



Remark: h denotes the (maximum) length of the sorted part of the array.

In [7]:

```
# Merge sort
def mergeSort(array):
    n = len(array)
    _mergeSort(array, 0, n - 1)

# Merge sort (partial array)
def _mergeSort(array, left, right):
    if left < right:
        middle = (left + right) // 2

        # Sort first and second halves
        _mergeSort(array, left, middle)
        _mergeSort(array, middle + 1, right)
        # Merge
        _merge(array, left, right, middle)

# Merges sorted partial arrays
def _merge(array, left, right, middle):
    nLeft = middle - left + 1
    nRight = right - middle

    # create temp arrays
    L = [0] * nLeft
    R = [0] * nRight

    # Copy data to temp arrays L[] and R[]
    for i in range(0 , nLeft):
        L[i] = array[left + i]

    for j in range(0 , nRight):
        R[j] = array[middle + 1 + j]

    # Initialize index positions
    i = 0
    j = 0
    k = left

    # Merge the temp arrays back into array[left..right]
    while i < nLeft and j < nRight:
        if L[i] <= R[j]:
            array[k] = L[i]
            i += 1
        else:
            array[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[]
    while i < nLeft:
        array[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[]
    while j < nRight:
        array[k] = R[j]
        j += 1
        k += 1
```

```
numbers = originalNumbers.copy()
print("Unsorted: %s" % numbers)
mergeSort(numbers)
print("Sorted: %s" % numbers)
```

```
Unsorted: [26, 4, 52, 74, 88, 51, 32, 21, 98, 3, 18, 52, 62, 80, 16,
52, 43, 71, 90, 17]
Sorted: [3, 4, 16, 17, 18, 21, 26, 32, 43, 51, 52, 52, 52, 62, 71, 7
4, 80, 88, 90, 98]
```

Complexity analysis of algorithms

In computer science, the analysis of algorithms is the process of finding the computational complexity of algorithms: the **amount of time, storage**, or other resources needed to execute them. Usually, this involves determining a function that relates the length of an algorithm's input to the number of steps it takes (its **time complexity**) or the number of storage locations it uses (its **space complexity**).

Now we will focus on **time complexity**. Let f and g represent the time complexity of 2 algorithms and we would like to make statements on how they grow compared to each other. (E.g. the time complexity of the *bubble sort* grows no faster than the n^2 function.)

In theoretical analysis of algorithms it is common to estimate their complexity in the *asymptotic sense*, i.e., to estimate the complexity function for arbitrarily large input. We are not concerned with small inputs or constant factors. The following notations are used to this end:

- **Big O (O) notation** describes the *asymptotic upper bound*, meaning that $f(N) = O(g(N))$, if such positive constants c and N_0 exists, that $f(N) \leq c * g(N)$, for all $N \geq N_0$.
- **Big-omega (Ω) notation** describes the *asymptotic lower bound*, meaning that $f(N) = O(g(N))$, if such positive constants c and N_0 exists, that $f(N) \geq c * g(N)$, for all $N \geq N_0$.
- **Big-theta (Θ) notation** describes the *asymptotic tight bound*, meaning that $f(N) = O(g(N))$, if such positive constants c_1, c_2 and N_0 exists, that $c_1 * g(N) \leq f(N) \leq c_2 * g(N)$, for all $N \geq N_0$.

In computer science in most cases we are interested in computing the *Big O* or the *Big-theta* notation, as a lower bound alone would not state much about the complexity.

For the most common complexities, well-known names have also be assigned and used:

Asympthotic complexity	Name
$\Theta(1)$	Constant time
$\Theta(n)$	Linear time
$\Theta(n^2)$	Quadratic time
$\Theta(n^3)$	Cubic time
$\Theta(log(n))$	Logarithmic time
$\Theta(n * log(n))$	Linearithmic time
$\Theta(2^n)$	Exponential time
$\Theta(n!)$	Factorial time

Question: what is the asymptotic computational complexity of the introduced sorting algoirhtms?

Bubbke sort, insertsion sort, maximum sort: $\Theta(n^2))$
Quicksort, merge sort: $\Theta(n * log(n))$

It can be proven that for a general case there is no better time complexity for sorting than $\Theta(n * log(n))$.
There are further algorithms with this complexity, see e.g. **Heap sort** (https://en.wikipedia.org/wiki/Heapsort) or **Tournament sort** (https://en.wikipedia.org/wiki/Tournament_sort).

Chapter 7: Collection data structures

In Python, we have the following built-in data structures:

- Lists
- Dictionaries
- Tuples
- Sets

We have already introduced the **list** data structure in Chapter 3 (03_iterations_lists.pdf). Lists are *heterogeneous* containers for multiple items in Python.

Dictionaries

Unlike sequences (like *lists* in Python), which are indexed by a range of numbers, dictionaries are indexed by *keys*. It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}` . Placing a comma-separated list of *key: value* pairs within the braces adds initial *key: value* pairs to the dictionary; this is also the way dictionaries are written on output.

Dictionaries are sometimes found in other languages as *associative arrays*.

Example: let's create a shopping list with the name of products as keys and quantities as values:

In [1]:

```
shopping_list = {'apple': 6, 'bread': 2, 'milk': 6, 'butter': 1}
print(shopping_list)
```

```
{'apple': 6, 'bread': 2, 'milk': 6, 'butter': 1}
```

Example: let's create a dictionary with the name of countries as keys and their capital cities as values:

In [2]:

```
capitals = {
    'Aland Islands': 'Mariehamn',
    'Albania': 'Tirana',
    'Andorra': 'Andorra la Vella',
    'Armenia': 'Yerevan',
    'Austria': 'Vienna',
    ...
    'Switzerland': 'Bern',
    'Turkey': 'Ankara',
    'Ukraine': 'Kyiv',
    'United Kingdom': 'London',
    'Northern Cyprus': 'North Nicosia'
}
print(capitals)
```

```
{'Aland Islands': 'Mariehamn', 'Albania': 'Tirana', 'Andorra': 'Ando
rra la Vella', 'Armenia': 'Yerevan', 'Austria': 'Vienna', 'Azerbaija
n': 'Baku', 'Belarus': 'Minsk', 'Belgium': 'Brussels', 'Bosnia and H
erzegovina': 'Sarajevo', 'Bulgaria': 'Sofia', 'Croatia': 'Zagreb',
'Cyprus': 'Nicosia', 'Czech Republic': 'Prague', 'Denmark': 'Copenha
gen', 'Estonia': 'Tallinn', 'Faroe Islands': 'Torshavn', 'Finland':
'Helsinki', 'France': 'Paris', 'Georgia': 'Tbilisi', 'Germany': 'Ber
lin', 'Gibraltar': 'Gibraltar', 'Greece': 'Athens', 'Guernsey': 'Sai
nt Peter Port', 'Vatican City': 'Vatican City', 'Hungary': 'Budapes
t', 'Iceland': 'Reykjavik', 'Ireland': 'Dublin', 'Isle of Man': 'Dou
glas', 'Italy': 'Rome', 'Jersey': 'Saint Helier', 'Kosovo': 'Pristin
a', 'Latvia': 'Riga', 'Liechtenstein': 'Vaduz', 'Lithuania': 'Vilniu
s', 'Luxembourg': 'Luxembourg', 'Macedonia': 'Skopje', 'Malta': 'Val
letta', 'Moldova': 'Chisinau', 'Monaco': 'Monaco', 'Montenegro': 'Po
dgorica', 'Netherlands': 'Amsterdam', 'Norway': 'Oslo', 'Poland': 'W
arsaw', 'Portugal': 'Lisbon', 'Romania': 'Bucharest', 'Russia': 'Mos
cow', 'San Marino': 'San Marino', 'Serbia': 'Belgrade', 'Slovakia':
'Bratislava', 'Slovenia': 'Ljubljana', 'Spain': 'Madrid', 'Svalbar
d': 'Longyearbyen', 'Sweden': 'Stockholm', 'Switzerland': 'Bern', 'T
urkey': 'Ankara', 'Ukraine': 'Kyiv', 'United Kingdom': 'London', 'No
rthern Cyprus': 'North Nicosia'}
```

Elements of a dictionary can be accessed through their key:

In [3]:

```
print(capitals['Hungary'])
```

```
Budapest
```

Dictionaries are mutable, so the values can be modified:

In [4]:

```
capitals['Hungary'] = 'Esztergom' # old capital city of Hungary between the 11-13th century
print(capitals['Hungary'])

capitals['Hungary'] = 'Budapest'
print(capitals['Hungary'])
```

```
Esztergom
Budapest
```

Dictionaries can also be extended with new elements (*key: value* pairs):

In [5]:

```
capitals['USA'] = 'Washington'
print(capitals['USA'])
```

```
Washington
```

Removing or deleting existing elements is also possible:

In [6]:

```
del capitals['USA'] # deleting, because not in Europe
print(capitals['USA'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-6-7e32e4eb21a0> in <module>
      1 del capitals['USA'] # deleting, because not in Europe
----> 2 print(capitals['USA'])

KeyError: 'USA'
```

Tuples

Tuples are a sequence of heterogeneous elements, similarly like *lists*. Its initial elements are defined as a comma separated list, surrounded by parentheses.

Note: lists are surrounded by brackets!

In [8]:

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

```
('Austria', 'Slovakia', 'Ukraine', 'Romania', 'Serbia', 'Croatia',
'Slovenia')
```

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

In [9]:

```
print(neighbours[0])
print(neighbours[2:5])
print(len(neighbours))
```

```
Austria
('Ukraine', 'Romania', 'Serbia')
7
```

The elements of tuple can also be fetched by *tuple unpacking*, which means that the items of a tuple are extracted into distinct variables:

In [10]:

```
a, b, c, d, e, f, g = neighbours
print(a, b, c, d, e, f, g)
```

```
Austria Slovakia Ukraine Romania Serbia Croatia Slovenia
```

Through *tuple packing*, we can create a **new** tuple with its elements defined:

In [11]:

```
neighbours2 = (a, b, c, d, e, f, g)
print(neighbours == neighbours2)
```

```
True
```

While lists are mutable, tuples are immutable, meaning that the elements cannot be modified:

In [12]:

```
neighbours[0] = 'Renamed country'
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-12-ee27e9767c8f> in <module>
----> 1 neighbours[0] = 'Renamed country'

TypeError: 'tuple' object does not support item assignment
```

New elements can neither be added to a tuple. Removing existing elements is also not possible.

In [13]:

```
neighbours.append('New country')
```

```
---------------------------------------------------------------------
-------
AttributeError                            Traceback (most recent cal
l last)
<ipython-input-13-1f4e9477ed13> in <module>
----> 1 neighbours.append('New country')

AttributeError: 'tuple' object has no attribute 'append'
```

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements. Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

We could see that lists, tuples and even strings have many common properties, such as indexing and slicing operations. They are **sequence data types**.

Dictionary as a list of tuples

The `dict()` constructor builds dictionaries directly from sequences of key-value pairs. The dictionary in this case is build from a list tuples, each tuple containing precisely two elements: a key and a value, in this order.

In [16]:

```
capitals = dict([
    ('Aland Islands', 'Mariehamn'),
    ('Albania', 'Tirana'),
    ('Andorra', 'Andorra la Vella'),
    ('Armenia', 'Yerevan'),
    ('Austria', 'Vienna'),
    ...
    ('Switzerland', 'Bern'),
    ('Turkey', 'Ankara'),
    ('Ukraine', 'Kyiv'),
    ('United Kingdom', 'London'),
    ('Northern Cyprus', 'North Nicosia')
])
print(capitals)
```

```
{'Aland Islands': 'Mariehamn', 'Albania': 'Tirana', 'Andorra': 'Ando
rra la Vella', 'Armenia': 'Yerevan', 'Austria': 'Vienna', 'Azerbaija
n': 'Baku', 'Belarus': 'Minsk', 'Belgium': 'Brussels', 'Bosnia and H
erzegovina': 'Sarajevo', 'Bulgaria': 'Sofia', 'Croatia': 'Zagreb',
'Cyprus': 'Nicosia', 'Czech Republic': 'Prague', 'Denmark': 'Copenha
gen', 'Estonia': 'Tallinn', 'Faroe Islands': 'Torshavn', 'Finland':
'Helsinki', 'France': 'Paris', 'Georgia': 'Tbilisi', 'Germany': 'Ber
lin', 'Gibraltar': 'Gibraltar', 'Greece': 'Athens', 'Guernsey': 'Sai
nt Peter Port', 'Vatican City': 'Vatican City', 'Hungary': 'Budapes
t', 'Iceland': 'Reykjavik', 'Ireland': 'Dublin', 'Isle of Man': 'Dou
glas', 'Italy': 'Rome', 'Jersey': 'Saint Helier', 'Kosovo': 'Pristin
a', 'Latvia': 'Riga', 'Liechtenstein': 'Vaduz', 'Lithuania': 'Vilniu
s', 'Luxembourg': 'Luxembourg', 'Macedonia': 'Skopje', 'Malta': 'Val
letta', 'Moldova': 'Chisinau', 'Monaco': 'Monaco', 'Montenegro': 'Po
dgorica', 'Netherlands': 'Amsterdam', 'Norway': 'Oslo', 'Poland': 'W
arsaw', 'Portugal': 'Lisbon', 'Romania': 'Bucharest', 'Russia': 'Mos
cow', 'San Marino': 'San Marino', 'Serbia': 'Belgrade', 'Slovakia':
'Bratislava', 'Slovenia': 'Ljubljana', 'Spain': 'Madrid', 'Svalbar
d': 'Longyearbyen', 'Sweden': 'Stockholm', 'Switzerland': 'Bern', 'T
urkey': 'Ankara', 'Ukraine': 'Kyiv', 'United Kingdom': 'London', 'No
rthern Cyprus': 'North Nicosia'}
```

Tuple unpacking

Since we know that each tuple in a dictionary contains a key and a value, *tuple unpacking* can be very useful to extract them into separate variables, e.g.:

In [17]:

```
pair = ('Hungary', 'Budapest')
key, value = pair
print("Key is {0}, value is {1}".format(key, value))
```

```
Key is Hungary, value is Budapest
```

Iterating through a dictionary

Accessing the list of *key-value* tuples can be done with the `items()` function of the dictionary:

In [18]:

```
print("List of key-value pairs:")
print(capitals.items())
```

```
List of key-value pairs:
dict_items([('Aland Islands', 'Mariehamn'), ('Albania', 'Tirana'),
('Andorra', 'Andorra la Vella'), ('Armenia', 'Yerevan'), ('Austria',
'Vienna'), ('Azerbaijan', 'Baku'), ('Belarus', 'Minsk'), ('Belgium',
'Brussels'), ('Bosnia and Herzegovina', 'Sarajevo'), ('Bulgaria', 'S
ofia'), ('Croatia', 'Zagreb'), ('Cyprus', 'Nicosia'), ('Czech Republ
ic', 'Prague'), ('Denmark', 'Copenhagen'), ('Estonia', 'Tallinn'),
('Faroe Islands', 'Torshavn'), ('Finland', 'Helsinki'), ('France',
'Paris'), ('Georgia', 'Tbilisi'), ('Germany', 'Berlin'), ('Gibralta
r', 'Gibraltar'), ('Greece', 'Athens'), ('Guernsey', 'Saint Peter Po
rt'), ('Vatican City', 'Vatican City'), ('Hungary', 'Budapest'), ('I
celand', 'Reykjavik'), ('Ireland', 'Dublin'), ('Isle of Man', 'Dougl
as'), ('Italy', 'Rome'), ('Jersey', 'Saint Helier'), ('Kosovo', 'Pri
stina'), ('Latvia', 'Riga'), ('Liechtenstein', 'Vaduz'), ('Lithuani
a', 'Vilnius'), ('Luxembourg', 'Luxembourg'), ('Macedonia', 'Skopj
e'), ('Malta', 'Valletta'), ('Moldova', 'Chisinau'), ('Monaco', 'Mon
aco'), ('Montenegro', 'Podgorica'), ('Netherlands', 'Amsterdam'),
('Norway', 'Oslo'), ('Poland', 'Warsaw'), ('Portugal', 'Lisbon'),
('Romania', 'Bucharest'), ('Russia', 'Moscow'), ('San Marino', 'San
Marino'), ('Serbia', 'Belgrade'), ('Slovakia', 'Bratislava'), ('Slov
enia', 'Ljubljana'), ('Spain', 'Madrid'), ('Svalbard', 'Longyearbye
n'), ('Sweden', 'Stockholm'), ('Switzerland', 'Bern'), ('Turkey', 'A
nkara'), ('Ukraine', 'Kyiv'), ('United Kingdom', 'London'), ('Northe
rn Cyprus', 'North Nicosia')])
```

Accessing ONLY the list of *keys* or *values* is also possible with the `keys()` and `values()` functions of the dictionary:

In [19]:

```
print("List of keys:")
print(capitals.keys())
```

```
List of keys:
dict_keys(['Aland Islands', 'Albania', 'Andorra', 'Armenia', 'Austri
a', 'Azerbaijan', 'Belarus', 'Belgium', 'Bosnia and Herzegovina', 'B
ulgaria', 'Croatia', 'Cyprus', 'Czech Republic', 'Denmark', 'Estoni
a', 'Faroe Islands', 'Finland', 'France', 'Georgia', 'Germany', 'Gib
raltar', 'Greece', 'Guernsey', 'Vatican City', 'Hungary', 'Iceland',
'Ireland', 'Isle of Man', 'Italy', 'Jersey', 'Kosovo', 'Latvia', 'Li
echtenstein', 'Lithuania', 'Luxembourg', 'Macedonia', 'Malta', 'Mold
ova', 'Monaco', 'Montenegro', 'Netherlands', 'Norway', 'Poland', 'Po
rtugal', 'Romania', 'Russia', 'San Marino', 'Serbia', 'Slovakia', 'S
lovenia', 'Spain', 'Svalbard', 'Sweden', 'Switzerland', 'Turkey', 'U
kraine', 'United Kingdom', 'Northern Cyprus'])
```

In [20]:

```
print("List of values:")
print(capitals.values())
```

```
List of values:
dict_values(['Mariehamn', 'Tirana', 'Andorra la Vella', 'Yerevan',
'Vienna', 'Baku', 'Minsk', 'Brussels', 'Sarajevo', 'Sofia', 'Zagre
b', 'Nicosia', 'Prague', 'Copenhagen', 'Tallinn', 'Torshavn', 'Helsi
nki', 'Paris', 'Tbilisi', 'Berlin', 'Gibraltar', 'Athens', 'Saint Pe
ter Port', 'Vatican City', 'Budapest', 'Reykjavik', 'Dublin', 'Dougl
as', 'Rome', 'Saint Helier', 'Pristina', 'Riga', 'Vaduz', 'Vilnius',
'Luxembourg', 'Skopje', 'Valletta', 'Chisinau', 'Monaco', 'Podgoric
a', 'Amsterdam', 'Oslo', 'Warsaw', 'Lisbon', 'Bucharest', 'Moscow',
'San Marino', 'Belgrade', 'Bratislava', 'Ljubljana', 'Madrid', 'Long
yearbyen', 'Stockholm', 'Bern', 'Ankara', 'Kyiv', 'London', 'North N
icosia'])
```

We can also use a *for* loop to iterate through the items of a dictionary:

In [21]:

```
for item in capitals.items():
    print(item)
```

```
('Aland Islands', 'Mariehamn')
('Albania', 'Tirana')
('Andorra', 'Andorra la Vella')
('Armenia', 'Yerevan')
('Austria', 'Vienna')
...
('Switzerland', 'Bern')
('Turkey', 'Ankara')
('Ukraine', 'Kyiv')
('United Kingdom', 'London')
('Northern Cyprus', 'North Nicosia')
```

Here we iterate through the *key: value* tuples of a dictionary with the `item` variable.
The *key* is the element with the index 0, the *value* is the element with the index 1 in `item`:

In [22]:

```
for item in capitals.items():
    key = item[0]
    value = item[1]
    print("{0}: {1}".format(key, value))
```

```
Aland Islands: Mariehamn
Albania: Tirana
Andorra: Andorra la Vella
Armenia: Yerevan
Austria: Vienna
...
Switzerland: Bern
Turkey: Ankara
Ukraine: Kyiv
United Kingdom: London
Northern Cyprus: North Nicosia
```

By creating a list of tuples from the dictionary, we can fetch a single tuple by its numerical index and then extract the *key* and the *value* into separate variables with *tuple unpacking*:

In [23]:

```
item_10 = list(capitals.items())[10]
print(item_10)
key, value = item_10
print("Key is {0}, value is {1}".format(key, value))
```

```
('Croatia', 'Zagreb')
Key is Croatia, value is Zagreb
```

We can also use a *for* loop to iterate through the *key: value* pairs in a dictionary with *tuple unpacking*:

In [24]:

```
for item in capitals.items():
    key, value = item
    print("{0}: {1}".format(key, value))
```

```
Aland Islands: Mariehamn
Albania: Tirana
Andorra: Andorra la Vella
Armenia: Yerevan
Austria: Vienna
...
Switzerland: Bern
Turkey: Ankara
Ukraine: Kyiv
United Kingdom: London
Northern Cyprus: North Nicosia
```

We do not even need a temporary `item` variable, the unpacking can be done directly in the `for` statement:

In [25]:

```
for key, value in capitals.items():
    print("{0}: {1}".format(key, value))
```

```
Aland Islands: Mariehamn
Albania: Tirana
Andorra: Andorra la Vella
Armenia: Yerevan
Austria: Vienna
...
Switzerland: Bern
Turkey: Ankara
Ukraine: Kyiv
United Kingdom: London
Northern Cyprus: North Nicosia
```

Now compare the 2 versions of iterating through the items of a dictionary and observe how *tuple unpacking* makes accessing the key and the value easier:

```
for item in capitals.items():
    key = item[0]
    value = item[1]
    print("{0}: {1}".format(key, value))

for key, value in capitals.items():
    print("{0}: {1}".format(key, value))
```

Note: keys in a dictionary can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified.

Note: the same tuple unpacking were utilized when using the `enumerate()` function introduced in Chapter 5 (05_basic_algorithms.pdf#Maximum-search):

```
data = [ ... ]
for idx, value in enumerate(data):
    ...
```

Summary exercise on dictionaries

The dictionaries `population_2008` and `population_2018` store the population of some European countries in the according years (2008 and 2010):

In [26]:

```
population_2008 = { 'Belgium': 10666866, 'Bulgaria': 7518002, 'Czechia': 1034342
2, 'Denmark': 5475791, 'Germany': 82217837, 'Estonia': 1338440, 'Ireland': 44577
65, 'Greece': 11060937, 'Spain': 45668939, 'France': 64007193, 'Croatia': 431196
7, 'Italy': 58652875, 'Cyprus': 776333, 'Latvia': 2191810, 'Lithuania': 3212605,
 'Luxembourg': 483799, 'Hungary': 10045401, 'Malta': 407832, 'Netherlands': 1640
5399, 'Austria': 8307989, 'Poland': 38115641, 'Portugal': 10553339, 'Romania': 2
0635460, 'Slovenia': 2010269, 'Slovakia': 5376064, 'Finland': 5300484, 'Sweden':
 9182927, 'United Kingdom': 61571647, 'Iceland': 315459, 'Liechtenstein': 35356,
 'Norway': 4737171, 'Switzerland': 7593494, 'Montenegro': 615543, 'North Macedon
ia': 2045177, 'Albania': 2958266, 'Serbia': 7365507, 'Turkey': 70586256, 'Andorr
a': 83137, 'Belarus': 9689770, 'Bosnia and Herzegovina': 3843846, 'Kosovo': 2153
139, 'Moldova': 3572703, 'San Marino': 32054, 'Ukraine': 46192309, 'Armenia': 32
30086, 'Azerbaijan': 8629900, 'Georgia': 4382070 }
population_2018 = { 'Belgium': 11398589, 'Bulgaria': 7050034, 'Czechia': 1061005
5, 'Denmark': 5781190, 'Germany': 82792351, 'Estonia': 1319133, 'Ireland': 48303
92, 'Greece': 10741165, 'Spain': 46658447, 'France': 66926166, 'Croatia': 410549
3, 'Italy': 60483973, 'Cyprus': 864236, 'Latvia': 1934379, 'Lithuania': 2808901,
 'Luxembourg': 602005, 'Hungary': 9778371, 'Malta': 475701, 'Netherlands': 17181
084, 'Austria': 8822267, 'Poland': 37976687, 'Portugal': 10291027, 'Romania': 19
530631, 'Slovenia': 2066880, 'Slovakia': 5443120, 'Finland': 5513130, 'Sweden':
10120242, 'United Kingdom': 66273576, 'Iceland': 348450, 'Liechtenstein': 38114,
 'Norway': 5295619, 'Switzerland': 8484130, 'Montenegro': 622359, 'North Macedon
ia': 2075301, 'Albania': 2870324, 'Serbia': 7001444, 'Turkey': 80810525, 'Andorr
a': 74794, 'Belarus': 9491823, 'Bosnia and Herzegovina': 3502550, 'Kosovo': 1798
506, 'Moldova': 3547539, 'San Marino': 34453, 'Ukraine': 42386403, 'Armenia': 29
72732, 'Azerbaijan': 9898085, 'Georgia': 3729633 }

print("Population of European countries in 2008:")
print(population_2008)

print()
print("Population of European countries in 2018:")
print(population_2018)
```

```
Population of European countries in 2008:
{'Belgium': 10666866, 'Bulgaria': 7518002, 'Czechia': 10343422, 'Den
mark': 5475791, 'Germany': 82217837, 'Estonia': 1338440, 'Ireland':
4457765, 'Greece': 11060937, 'Spain': 45668939, 'France': 64007193,
'Croatia': 4311967, 'Italy': 58652875, 'Cyprus': 776333, 'Latvia': 2
191810, 'Lithuania': 3212605, 'Luxembourg': 483799, 'Hungary': 10045
401, 'Malta': 407832, 'Netherlands': 16405399, 'Austria': 8307989,
'Poland': 38115641, 'Portugal': 10553339, 'Romania': 20635460, 'Slov
enia': 2010269, 'Slovakia': 5376064, 'Finland': 5300484, 'Sweden': 9
182927, 'United Kingdom': 61571647, 'Iceland': 315459, 'Liechtenstei
n': 35356, 'Norway': 4737171, 'Switzerland': 7593494, 'Montenegro':
615543, 'North Macedonia': 2045177, 'Albania': 2958266, 'Serbia': 73
65507, 'Turkey': 70586256, 'Andorra': 83137, 'Belarus': 9689770, 'Bo
snia and Herzegovina': 3843846, 'Kosovo': 2153139, 'Moldova': 357270
3, 'San Marino': 32054, 'Ukraine': 46192309, 'Armenia': 3230086, 'Az
erbaijan': 8629900, 'Georgia': 4382070}

Population of European countries in 2018:
{'Belgium': 11398589, 'Bulgaria': 7050034, 'Czechia': 10610055, 'Den
mark': 5781190, 'Germany': 82792351, 'Estonia': 1319133, 'Ireland':
4830392, 'Greece': 10741165, 'Spain': 46658447, 'France': 66926166,
'Croatia': 4105493, 'Italy': 60483973, 'Cyprus': 864236, 'Latvia': 1
934379, 'Lithuania': 2808901, 'Luxembourg': 602005, 'Hungary': 97783
71, 'Malta': 475701, 'Netherlands': 17181084, 'Austria': 8822267, 'P
oland': 37976687, 'Portugal': 10291027, 'Romania': 19530631, 'Sloven
ia': 2066880, 'Slovakia': 5443120, 'Finland': 5513130, 'Sweden': 101
20242, 'United Kingdom': 66273576, 'Iceland': 348450, 'Liechtenstei
n': 38114, 'Norway': 5295619, 'Switzerland': 8484130, 'Montenegro':
622359, 'North Macedonia': 2075301, 'Albania': 2870324, 'Serbia': 70
01444, 'Turkey': 80810525, 'Andorra': 74794, 'Belarus': 9491823, 'Bo
snia and Herzegovina': 3502550, 'Kosovo': 1798506, 'Moldova': 354753
9, 'San Marino': 34453, 'Ukraine': 42386403, 'Armenia': 2972732, 'Az
erbaijan': 9898085, 'Georgia': 3729633}
```

Data source: EuroStat (https://ec.europa.eu/eurostat/)

Exercise 1

Task: What was the population of Hungary in 2008 and in 2018?

In [27]:

```
print("Population of Hungary in 2008: {0}".format(population_2008["Hungary"]))
print("Population of Hungary in 2018: {0}".format(population_2018["Hungary"]))
```

```
Population of Hungary in 2008: 10045401
Population of Hungary in 2018: 9778371
```

Exercise 2

Task: What was the population change between 2008 and 2018 in Hungary? What is the average change per year?

In [28]:

```
diff = population_2018["Hungary"] - population_2008["Hungary"]
print("Population difference for Hungary: {0}".format(diff))

diff_avg = diff // 10
print("Average population change for Hungary per year: {0}".format(diff_avg))
```

```
Population difference for Hungary: -267030
Average population change for Hungary per year: -26703
```

Exercise 3

Task: Display for all countries the population change between 2008 and 2018!

In [29]:

```
for key in population_2008.keys():
    diff = population_2018[key] - population_2008[key]
    print("{0}: {1}".format(key, diff))
```

```
Belgium: 731723
Bulgaria: -467968
Czechia: 266633
Denmark: 305399
Germany: 574514
...
San Marino: 2399
Ukraine: -3805906
Armenia: -257354
Azerbaijan: 1268185
Georgia: -652437
```

Exercise 4

Task: Which country had the largest population growth in the given timespan? Which one had the largest population decline?

In [30]:

```
max_country = "Hungary"
max_diff = population_2018[max_country] - population_2008[max_country]
min_country = "Hungary"
min_diff = population_2018[min_country] - population_2008[min_country]

for key in population_2008.keys():
    diff = population_2018[key] - population_2008[key]
    if diff > max_diff:
        max_diff = diff
        max_country = key
    if diff < min_diff:
        min_diff = diff
        min_country = key

print("Largest growth: {0} ({1})".format(max_country, max_diff))
print("Largest decline: {0} ({1})".format(min_country, min_diff))
```

```
Largest growth: Turkey (10224269)
Largest decline: Ukraine (-3805906)
```

Sets

Python also includes a data type for *sets*. A set is an unordered collection with no duplicate elements. Basic usage include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the `set()` function can be used to create sets.
Note: to create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary and not a set.

In [31]:

```
neighbours = {'Austria', 'Slovakia', 'Ukraine', 'Romania', 'Serbia', 'Croatia',
'Slovenia'}
print(neighbours)
```

```
{'Slovakia', 'Romania', 'Austria', 'Ukraine', 'Croatia', 'Serbia',
'Slovenia'}
```

Since sets are unordered theoretically, Python does not support indexing for sets, meaning we cannot access an item with a numerical index:

In [32]:

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

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-32-a5f7753d713b> in <module>
----> 1 print(neighbours[0])

TypeError: 'set' object is not subscriptable
```

However, we can perform membership testing, evaluating whether an item is in the list or not:

In [33]:

```
print('Serbia' in neighbours)
print('Germany' in neighbours)
```

```
True
False
```

New element can be added with the `add()` method to a set, existing element can be removed with the `remove()` method from a set. Sets also guarantee to contain no duplicate entries:

In [34]:

```
neighbours.add('Ukraine') # already in the set
print(neighbours)
```

```
{'Slovakia', 'Romania', 'Austria', 'Ukraine', 'Croatia', 'Serbia',
'Slovenia'}
```

Demonstration of basic set operations:

In [35]:

```
german_speakers = {'Germany', 'Austria', 'Switzerland'}
```

In [36]:

```
print("Union: {0}".format(neighbours | german_speakers))
print("Intersection: {0}".format(neighbours & german_speakers))
print("Difference: {0}".format(neighbours - german_speakers))
print("Symmetric difference: {0}".format(neighbours ^ german_speakers))
```

```
Union: {'Slovakia', 'Romania', 'Switzerland', 'Austria', 'Ukraine',
'Croatia', 'Serbia', 'Slovenia', 'Germany'}
Intersection: {'Austria'}
Difference: {'Slovakia', 'Romania', 'Ukraine', 'Croatia', 'Serbia',
'Slovenia'}
Symmetric difference: {'Slovakia', 'Romania', 'Switzerland', 'Ukrain
e', 'Croatia', 'Serbia', 'Slovenia', 'Germany'}
```

Exercise 5

Task: Verify whether the dictionaries `population_2008` and `population_2018` contain exactly the same countries as keys.

In [37]:

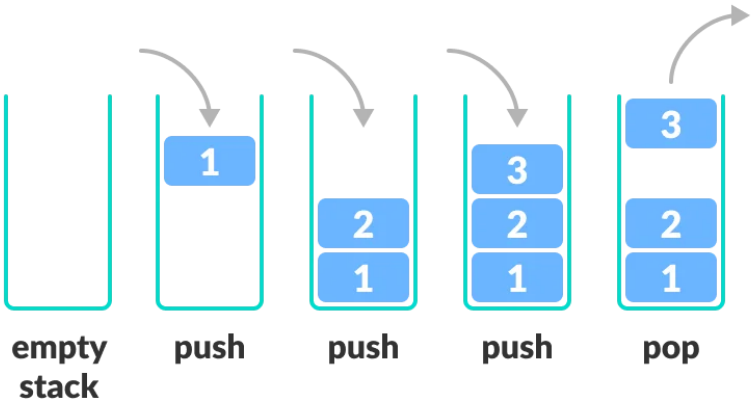
```
keyset_2008 = set(population_2008.keys())
keyset_2018 = set(population_2018.keys())
nomatch = keyset_2008 ^ keyset_2018

if len(nomatch) == 0:
    print("The 2 dictionaries contains the same countries.")
else:
    print("There are some countries only present in one of the dictionaries: {0}
".format(nomatch))
```

```
The 2 dictionaries contains the same countries.
```

Stacks

A stack is an abstract data structure that follows the "*last-in-first-out*" or *LIFO* model: elements are added to the top of the stack and only the top element of a stack can be removed.
Some well-known real world examples for usage of stacks:

- undoing the operations in a text editor;
- going back to the previous web page in a browser.



The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (*last-in, first-out*). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index.

In [38]:

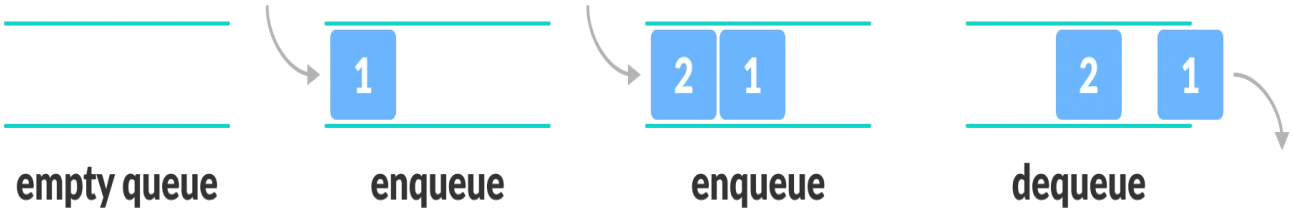
```
stack = [1, 2, 3, 4, 5]
stack.append(6)
stack.append(7)
print(stack)
print(stack.pop())
print(stack.pop())
print(stack)
stack.append(8)
stack.append(9)
print(stack)

print("Process all the elements of the stack:")
while len(stack) > 0:
    print(stack.pop())
```

```
[1, 2, 3, 4, 5, 6, 7]
7
6
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 8, 9]
Process all the elements of the stack:
9
8
5
4
3
2
1
```

Queues

A queue is an abstract data structure that follows "*first-in-first-out*" or *FIFO model*: new elements are added to the back of queue and only the front element of the queue can be removed.
Some well-known real world examples for usage of queues:

- waiting in a line (in a polite way);
- a printer machine's queue of files to be printed.



It is also possible to use a list as a queue, where the first element added is the first element retrieved (*first-in, first-out*); however, **lists are not efficient for this purpose**. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends.

In [39]:

```
from collections import deque

queue = deque([1, 2, 3, 4, 5])
queue.append(6)
queue.append(7)
print(queue)
print(queue.popleft())
print(queue.popleft())
print(queue)
queue.append(8)
queue.append(9)
print(queue)

print("Process all the elements of the stack:")
while len(queue) > 0:
    print(queue.popleft())
```

```
deque([1, 2, 3, 4, 5, 6, 7])
1
2
deque([3, 4, 5, 6, 7])
deque([3, 4, 5, 6, 7, 8, 9])
Process all the elements of the stack:
3
4
5
6
7
8
9
```

Note: `deque` is short for *double ended queue*, because we can manage both ends of the data structure (add or remove elements).

Chapter 8: Object-oriented programming

Python is an *object-oriented* programming language (OOP). **Objects** are an encapsulation of variables and functions into a single entity.

Let's assume we have a data type for rectangles. Without objects we could write the following code:

In [1]:

```
rec1_bl = (0, 2) # bl = bottom-left
rec1_ur = (6, 8) # ur = upper-right

rec2_bl = (4, 3)
rec2_ur = (7, 5)

def area(bl, ur):
    width = ur[0] - bl[0]
    height = ur[1] - bl[1]
    return width * height

print("Area of rectangle #1: {0}".format(area(rec1_bl, rec1_ur)))
print("Area of rectangle #2: {0}".format(area(rec2_bl, rec2_ur)))
```

```
Area of rectangle #1: 36
Area of rectangle #2: 6
```

As we can observe the data (`rec1_bl` , `rec1_ur` , etc.) and the functions (`area()` and possible further functions) are defined separately, not encapsulated together.

Classes and objects

Simply put, an object is a collection of data (variables) and methods (functions) that act on those data. A **class** is a blueprint for the object. Classes introduce new data types in Python, describing real-world things and situations. Objects are the *instances* of classes, the creation process of objects are also called *instantiation*.

Let's create the `Rectangle` class now:

In [2]:

```
class Rectangle():
    name = 'Rectangle'

    def area(self):
        width = self.ur[0] - self.bl[0]
        height = self.ur[1] - self.bl[1]
        return width * height

rec1 = Rectangle()
rec1.bl = (0, 2)
rec1.ur = (6, 8)
rec2 = Rectangle()
rec2.bl = (4, 3)
rec2.ur = (7, 5)

print("Area of rectangle {0}".format(rec1.area()))
print("Area of rectangle {0}".format(rec2.area()))
```

```
Area of rectangle 36
Area of rectangle 6
```

In this example the `Rectangle` class has 4 attributes: `name`, `ur`, `bl` and `area()`. Attributes may be data or method: the `name` is a simple string, `bl` and `ur` are tuples while `area()` is a function. Functions in a class are called *methods* more specifically.

The `rec1 = Rectangle()` statement creates a new instance object named `rec1` from the class `Rectangle`. We can access the attributes of objects using the object name prefix, e.g. `rec1.area()`.

Remember how we used list and dictionary functions:

```
numbers = [1, 4, 5, -2, 8]
numbers.sort()

shopping_list = {'apple': 6, 'bread': 2, 'milk': 6, 'butter': 1}
for item in shopping_list.items():
    print(item)
```

This is the same syntax, we are calling methods on objects.

`self` parameter

There is a `self` parameter in the `area()` function definition inside the `Rectangle` class but, we called the method simply as `rec1.area()`, without any arguments. It still worked.

This is because, whenever an object calls its method, the object itself is passed as the first argument. So, `rec1.area()` translates into `Rectangle.area(rec1)`.

In [3]:

```
print("Area of rectangle #1: {0}".format(Rectangle.area(rec1)))
print("Area of rectangle #2: {0}".format(Rectangle.area(rec2)))
print("Name of all rectangles: {0}".format(Rectangle.name))
```

```
Area of rectangle #1: 36
Area of rectangle #2: 6
Name of all rectangles: Rectangle
```

In general, calling a method with a list of arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

For these reasons, the first argument of the function in class must be the object itself. This is conventionally called `self`. It can be named otherwise but it is highly discouraged to follow the convention.

Constructors

In our previous example we deliberately gave a value to the `bl` and `ur` attributes of `rec1` and `rec2` before calling the `area()` method on them, so it can process those values. What happens if we e.g. forget to initialize those attributes beforehand?

In [4]:

```
rec3 = Rectangle()
print("Area of rectangle #3: {0}".format(rec3.area()))
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent cal
l last)
<ipython-input-4-a77eecff7f61> in <module>
      1 rec3 = Rectangle()
----> 2 print("Area of rectangle #3: {0}".format(rec3.area()))

<ipython-input-2-1afbe4a6155f> in area(self)
      3
      4     def area(self):
----> 5         width = self.ur[0] - self.bl[0]
      6         height = self.ur[1] - self.bl[1]
      7         return width * height

AttributeError: 'Rectangle' object has no attribute 'ur'
```

This issue can be addressed with a special *constructor method*, which is always executed when a new object is instantiated from a class.

In Python, class functions that begins with double underscore (`__`) are called special functions as they have special meaning. The `__init__()` function has particular interest for us now. This special function gets called whenever a new object of that class is instantiated. This type of function is also called a **constructor** in object-oriented programming. We normally use it to initialize all the variables.

In [5]:

```
class Rectangle():
    name = 'Rectangle'

    def __init__(self, bl_x, bl_y, ur_x, ur_y):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def area(self):
        width = self.ur[0] - self.bl[0]
        height = self.ur[1] - self.bl[1]
        return width * height

rec1 = Rectangle(0, 2, 6, 8)
rec2 = Rectangle(4, 3, 7, 5)

print("Area of rectangle #1: {0}".format(rec1.area()))
print("Area of rectangle #2: {0}".format(rec2.area()))
```

```
Area of rectangle #1: 36
Area of rectangle #2: 6
```

Now we cannot "forget" to pass all the required data to the object upon instatiation, because Python will raise a `TypeError` .

In [6]:

```
rec3 = Rectangle()
print("Area of rectangle #3: {0}".format(rec3.area()))
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-6-a77eecff7f61> in <module>
----> 1 rec3 = Rectangle()
      2 print("Area of rectangle #3: {0}".format(rec3.area()))

TypeError: __init__() missing 4 required positional arguments: 'bl_
x', 'bl_y', 'ur_x', and 'ur_y'
```

Default arguments

Alternatively we could use default values for the parameters, so a `Rectangle` could be constructed without defining its dimensions, but still giving value to the *instance attributes*.

In [7]:

```
class Rectangle():
    name = 'Rectangle'

    def __init__(self, bl_x = 0, bl_y = 0, ur_x = 0, ur_y = 0):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def area(self):
        width = self.ur[0] - self.bl[0]
        height = self.ur[1] - self.bl[1]
        return width * height
```

In [8]:

```
rec3 = Rectangle()
print("Area of rectangle #3: {0}".format(rec3.area()))
```

```
Area of rectangle #3: 0
```

Class and instance attributes

Generally speaking, instance attributes are for data unique to each instance and class attributes are for variables and methods shared by all instances of the class.

In the example `Rectangle` class, the `name` attribute is a class variable, because it is defined as an attribute of the class.

In [9]:

```
print(Rectangle.name)
```

```
Rectangle
```

The `bl` and `ur` are instance attributes (because they are accessed through the `self` object). This means that each rectangle can have its own bottom-left and upper-right position, but all rectangles share the same name.

In [10]:

```
rec1.bl = (-2, 1)
print(rec1.bl) # has no effect on rec2
print(rec2.bl)
```

```
(-2, 1)
(4, 3)
```

String representation of an object

By default, the string representation of an object consists of the type name and memory address:

In [11]:

```
print(rec1)
```

```
<__main__.Rectangle object at 0x7fbcfd0291f0>
```

As we discussed, methods that begins with double underscore (`__`) are called special functions in Python. The `__str__()` method is another special function, which can compute and return the "informal" or nicely printable string representation of an object. The return value must be a string object.

In [12]:

```
class Rectangle():
    def __init__(self, bl_x, bl_y, ur_x, ur_y):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def __str__(self):
        return "Rectangle ({0}, {1}, {2}, {3})".format(self.bl[0], self.bl[1], s
elf.ur[0], self.ur[1])

    def area(self):
        width = self.ur[0] - self.bl[0]
        height = self.ur[1] - self.bl[1]
        return width * height

rec1 = Rectangle(0, 2, 6, 8)
rec2 = Rectangle(4, 3, 7, 5)

print(rec1)
print(rec2)
```

```
Rectangle (0, 2, 6, 8)
Rectangle (4, 3, 7, 5)
```

Summary exercises on object-oriented programming

Task 1: Perimeter

Extend the `Rectangle` class with a `perimeter()` method.

Sample usage:

```
result = rec1.perimeter()
# result is an integer value
```

In [13]:

```
class Rectangle():
    def __init__(self, bl_x, bl_y, ur_x, ur_y):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def __str__(self):
        return "Rectangle ({0}, {1}, {2}, {3})".format(self.bl[0], self.bl[1], s
elf.ur[0], self.ur[1])

    def area(self):
        width = self.ur[0] - self.bl[0]
        height = self.ur[1] - self.bl[1]
        return width * height

    def perimeter(self):
        width = self.ur[0] - self.bl[0]
        height = self.ur[1] - self.bl[1]
        return 2 * (width + height)
```

The computation of the width and height of the rectangle is now redundantly given in the `area()` and the `perimeters()` methods. Eliminate the redundancy by extracting a new `width()` and `height()` function in the `Rectangle` class.

In [14]:

```
class Rectangle():
    def __init__(self, bl_x, bl_y, ur_x, ur_y):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def __str__(self):
        return "Rectangle ({0}, {1}, {2}, {3})".format(self.bl[0], self.bl[1], s
elf.ur[0], self.ur[1])

    def width(self):
        return self.ur[0] - self.bl[0]

    def height(self):
        return self.ur[1] - self.bl[1]

    def area(self):
        return self.width() * self.height()

    def perimeter(self):
        return 2 * (self.width() + self.height())
```

Task 2: Translation

Extend the `Rectangle` class with a `translate()` method, which moves it in the Euclidean space in the given direction.

Sample usage:

```
print(rec1)
rec1.translate(3, 4)
print(rec1)
```

In [15]:

```
class Rectangle():
    def __init__(self, bl_x, bl_y, ur_x, ur_y):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def __str__(self):
        return "Rectangle ({0}, {1}, {2}, {3})".format(self.bl[0], self.bl[1], s
elf.ur[0], self.ur[1])

    def width(self):
        return self.ur[0] - self.bl[0]

    def height(self):
        return self.ur[1] - self.bl[1]

    def area(self):
        return self.width() * self.height()

    def perimeter(self):
        return 2 * (self.width() + self.height())

    def translate(self, x, y):
        self.bl = (self.bl[0] + x, self.bl[1] + y)
        self.ur = (self.ur[0] + x, self.ur[1] + y)
```

Task 3: Overlap

Extend the `Rectangle` class with an `overlap()` method, which can decide whether 2 rectangles overlap each other.

Sample usage:

```
result = rec1.overlap(rec2)
# result is a boolean value
```

In [16]:

```
class Rectangle():
    def __init__(self, bl_x, bl_y, ur_x, ur_y):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def __str__(self):
        return "Rectangle ({0}, {1}, {2}, {3})".format(self.bl[0], self.bl[1], s
elf.ur[0], self.ur[1])

    def width(self):
        return self.ur[0] - self.bl[0]

    def height(self):
        return self.ur[1] - self.bl[1]

    def area(self):
        return self.width() * self.height()

    def perimeter(self):
        return 2 * (self.width() + self.height())

    def translate(self, x, y):
        self.bl = (self.bl[0] + x, self.bl[1] + y)
        self.ur = (self.ur[0] + x, self.ur[1] + y)

    def overlap(self, other):
        overlap_x = ((self.bl[0] < other.bl[0] and self.ur[0] > other.bl[0]) or
                     (other.bl[0] < self.bl[0] and other.ur[0] > self.bl[0]))
        overlap_y = ((self.bl[1] < other.bl[1] and self.ur[1] > other.bl[1]) or
                     (other.bl[1] < self.bl[1] and other.ur[1] > self.bl[1]))

        return overlap_x and overlap_y
```

Hint: Two *axis-parallel* rectangles overlap if they overlap either by the X or Y dimensions.
They overlap by the X dimension if `A_X1 < B_X1` and `A_X2 > B_X1` ; or `B_X1 < A_X1` and `B_X2 > A_X1` .
Similar inequality condition apply on the Y dimension.

Data classes *(optional)*

A relatively new feature available since Python 3.7 (released in 2018) is the *data class*. A data class is a class typically containing mainly data, although there aren't really any restrictions. It is created using the `@dataclass` decorator, as follows:

In [17]:

```
from dataclasses import dataclass

@dataclass
class Country:
    name: str
    capital: str
    area: int
    population: int
    gdp: int
    literacy: float
    region: str = 'Unknown'

    def population_density(self):
        return self.population / self.area
```

The benefit of using data classes is that some special methods, e.g. the `__init__()` constructor method will be automatically generated and added to the class, initiating all instance variables. The generated constructor will look like:

```
    def __init__(self, name, capital, area, population, gdp, literacy, region
     = 'Unknown'):
        self.name = name
        self.capital = capital
        self.area = area
        self.population = population
        self.gdp =gdp
        self.literacy = literacy
        self.region = region
```

Since data classes is just a new syntactical approach in Python for defining classes, we can instantiate objects from data classes and use them just like before:

In [18]:

```
hungary = Country('Hungary', 'Budapest', 93030, 9981334, 13900, 99.4, 'Central-E
urope')
```

In [19]:

```
print(hungary.capital)
print(hungary.population_density())
```

```
Budapest
107.29156186176502
```

The string representational `__str__()` method is also predefined for data classes:

In [20]:

```
print(hungary)
```

```
Country(name='Hungary', capital='Budapest', area=93030, population=9
981334, gdp=13900, literacy=99.4, region='Central-Europe')
```

Type annotations

As you have may noticed we also defined the type of the instance variables in the data class, e.g. `population: int` . This is called *type hinting* or *type annotations* and is mandatory when defining a data class. Type hinting is available in Python since version 3.5 and can also be used elsewhere: for local variables, function parameters, return types, etc.

Note that the Python runtime does not enforce function and variable type annotations, so whether you use them or not, they will not affect how your code is executed. However they can be used by third party tools such as type checkers (see mypy (http://mypy-lang.org/)) or integrated development environments (IDEs), to early detect potential errors in your code.

We will not use type hinting further in this course, as *Jupyter Notebook* itself does not perform type checking based on them.

Inheritance *(advanced, optional)*

We do not always have to start from scratch when writing a class. If the class is a specialized version of another already existing class, we can use **inheritance**.

When one class *inherits* from another, it automatically takes on all the attributes and methods of the first class. The original class is called the *parent* class, and the new class is the *child* class. The child class inherits every attribute and method from its parent class but is also free to define new attributes and methods of its own.

Let's inherit the `Square` class from the `Rectangle` class:

In [21]:

```
class Square(Rectangle):
    def __init__(self, bl_x, bl_y, width):
        self.bl = (bl_x, bl_y)
        self.ur = (bl_x + width, bl_y + width)

s1 = Square(5, 10, 3)
print("Area of square #1: {0}".format(s1.area()))
```

```
Area of square #1: 9
```

The `__init__()` function in the child class

Often we would like to reuse the original `__init__` function of the parent class in the child class.

This can be done with the `super()` function inside the child class constructor. This is a special function that helps Python make connections between the parent and child class.

Note: The name super comes from a convention of calling the parent class a *superclass* and the child class a *subclass*.

In [22]:

```
class Square(Rectangle):
    def __init__(self, bl_x, bl_y, width):
        super().__init__(bl_x, bl_y, bl_x + width, bl_y + width)

s1 = Square(5, 10, 3)
print("Area of square #1: {0}".format(s1.area()))
```

```
Area of square #1: 9
```

Overiding methods from the *parent* class

Let's see what happens if we print our `s1` object:

In [23]:

```
print(s1)
```

```
Rectangle (5, 10, 8, 13)
```

It shows the text "Rectangle", because the `__str__()` special function was defined this way in the `Rectangle` parent class and now the `Square` child class inherited it.

We can override any method from the parent class that do not fit into the model of the child class. To achieve this, we can simply redefine the method in the child class with the same name as the method we want to override in the parent class. Python will disregard the parent class method and only pay attention to the method you define in the child class.

In [24]:

```
class Square(Rectangle):
    def __init__(self, bl_x, bl_y, width):
        self.bl = (bl_x, bl_y)
        self.ur = (bl_x + width, bl_y + width)

    def __str__(self):
        return "Square ({0}, {1}, width = {2})".format(self.bl[0], self.bl[1], s
elf.ur[0] - self.bl[0])

s1 = Square(5, 10, 3)
print("Area of square #1: {0}".format(s1.area()))
print(s1)
```

```
Area of square #1: 9
Square (5, 10, width = 3)
```

Note: the `super()` function can be used in any overriding child class methods.

Extend the functionality of the parent class

Child classes may also extend the functionality of their parent class by adding new methods to themselves.

In [25]:

```
class Square(Rectangle):
    def __init__(self, bl_x, bl_y, width):
        self.bl = (bl_x, bl_y)
        self.ur = (bl_x + width, bl_y + width)

    def __str__(self):
        return "Square ({0}, {1}, width = {2})".format(self.bl[0], self.bl[1], s
elf.ur[0] - self.bl[0])

    def side(self):
        return self.bl[1] - self.ur[0]
```

In [26]:

```
s1 = Square(5, 10, 3)
print(s1.side())
```

```
2
```

The `Rectangle` class does not have this new `side()` method:

In [27]:

```
print(rec1.side())
```

```
---------------------------------------------------------------------
-------
AttributeError                            Traceback (most recent cal
l last)
<ipython-input-27-63985b0fc09e> in <module>
----> 1 print(rec1.side())

AttributeError: 'Rectangle' object has no attribute 'side'
```

Chapter 9: Tabular data

The **pandas** package is a high-level data manipulation tool for Python.

The name *pandas* is derived from the term "panel data", an econometrics term for data sets that include observations over multiple time periods for the same individuals.

How to install pandas?

If you have Anaconda installed, then pandas was already installed together with it.

If you have a standalone Python3 and Jupyter Notebook installation, open a command prompt / terminal and type in:

```
pip3 install pandas xlrd openpyxl
```

The `xlrd` and `openpyxl` packages are required for managing Microsoft Excel files.
If you have Anaconda installed, these packages were also included in the default installation.

Support OpenDocument format

The `xlrd` package is used for older format MS Excel files (`.xls`), while `openpyxl` is used for newer format MS Excel files (`.xlsx`).

In case support for *OpenDocument* format is required (`.ods` files), e.g. for compatibility with OpenOffice or LibreOffice, the `odf` package can be used:

- Install with Anaconda Command Prompt: `conda install odf`
- Install with Python Package Installer: `pip3 install odf`

How to use pandas?

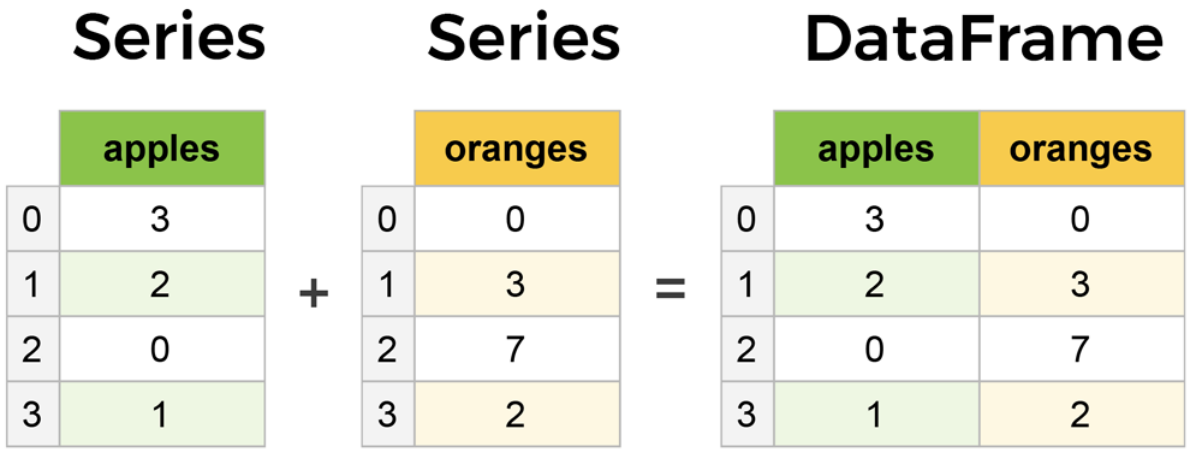
The pandas package is a module which you can simply import. It is usually aliased with the `pd` abbreviation:

```
import pandas as pd
```

Series and Dataframes

The primary two components of pandas are the **Series** and **DataFrame**.

A *Series* is essentially a column, and a *DataFrame* is a multi-dimensional table made up of a collection of *Series*.



We can work with DataFrames and Series in a similar way, since many operations support both of them. (E.g. calculating the mean value of a Series or a DataFrame.)

Create a DataFrame from scratch

A Python *dictionary* can easily be converted to a pandas *DataFrame*. Each `(key, value)` tuple in the dictionary corresponds to a column in the resulting DataFrame. The values in the rows for each column are given as lists.

In [1]:

```
import pandas as pd

data = {
    'apples': [3, 2, 0, 1, 5, 0, 4],
    'oranges': [0, 3, 7, 2, 1, 6, 2]
}

df = pd.DataFrame(data)
display(df)
```

	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2
4	5	1
5	0	6
6	4	2

By default the rows in the DataFrame are *indexed* numerically from 0, but we can set a custom **Index**:

In [2]:

```
df = pd.DataFrame(data, index = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday', 'Sunday'])
display(df)
```

	apples	oranges
Monday	3	0
Tuesday	2	3
Wednesday	0	7
Thursday	1	2
Friday	5	1
Saturday	0	6
Sunday	4	2

Remark: the `display()` function is a special Jupyter Notebook function to provide "prettier" output. Where `display()` is used, `print()` could also be used.

In [3]:

```
print(df)
```

```
           apples  oranges
Monday          3        0
Tuesday         2        3
Wednesday       0        7
Thursday        1        2
Friday          5        1
Saturday        0        6
Sunday          4        2
```

Access the columns of a DataFrame

In [4]:

```
print('Apples purchased over the week:')
print(df['apples']) # df is a DataFrame, df['apples'] is a Series
```

```
Apples purchased over the week:
Monday       3
Tuesday      2
Wednesday    0
Thursday     1
Friday       5
Saturday     0
Sunday       4
Name: apples, dtype: int64
```

Values of single cells can also be accessed:

In [5]:

```
print('Apples purchased on Monday:')
print(df['apples']['Monday'])
print(df['apples'][0])
```

```
Apples purchased on Monday:
3
3
```

Access the rows of a DataFrame

Rows can be accessed through the `loc` property with their textual indexes:

In [6]:

```
print('Fruits purchased on Monday:')
print(df.loc['Monday'])
```

```
Fruits purchased on Monday:
apples     3
oranges    0
Name: Monday, dtype: int64
```

Rows can also be accessed through the `iloc` property with ther numerical indexes:

In [7]:

```
print('Fruits purchased on Monday:')
print(df.iloc[0])
```

```
Fruits purchased on Monday:
apples     3
oranges    0
Name: Monday, dtype: int64
```

Accessing single cells:

In [8]:

```
print('Apples purchased on Monday:')
print(df.loc['Monday']['apples'])
print(df.loc['Monday'][0])
print(df.iloc[0]['apples'])
print(df.iloc[0][0])
```

```
Apples purchased on Monday:
3
3
3
3
```

Iterate over the rows of a DataFrame

Manually iterating through all the rows with the `iterrows()` method:

In [9]:

```
for index, row in df.iterrows():
    print("Index: {0}, Apples: {1}, Oranges: {2}".format(index, row["apples"], r
ow["oranges"]))
```

```
Index: Monday, Apples: 3, Oranges: 0
Index: Tuesday, Apples: 2, Oranges: 3
Index: Wednesday, Apples: 0, Oranges: 7
Index: Thursday, Apples: 1, Oranges: 2
Index: Friday, Apples: 5, Oranges: 1
Index: Saturday, Apples: 0, Oranges: 6
Index: Sunday, Apples: 4, Oranges: 2
```

Alternatively, if only the index values are required, the `df.index` list can be iterated:

In [10]:

```
print(df.index)
```

```
Index(['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Satu
rday',
       'Sunday'],
      dtype='object')
```

Reading external files

The *pandas* library has a great support for reading (and writing) external data files, like CSV files, Excel files, JSON files, etc.

Let's use the *European countries datatset*. The dataset contains the country name, capital city name, area (in km^2), population (in millions) and the region data for 43 European countries respectively.
Data source: [EuroStat (https://ec.europa.eu/eurostat/)](https://ec.europa.eu/eurostat/)

The dataset is given in the `data/countries_europe.csv` file, which we can read with th `read_csv()` method, passing the file path and the delimiter symbol (latter will be discussed soon).

In [11]:

```
import pandas as pd

countries = pd.read_csv('../data/countries_europe.csv', delimiter = ';')
display(countries)
```

	Country	Capital	Area (km2)	Population (millions)	Region
0	Albania	Tirana	28748	3.20	Southern
1	Andorra	Andorra la Vella	468	0.07	Western
2	Austria	Vienna	83857	7.60	Western
3	Belgium	Brussels	30519	10.00	Western
4	Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern
...	...	...	...	...	...
38	Sweden	Stockholm	449964	8.50	Northern
39	Serbia	Belgrade	66577	7.20	Southern
40	Slovakia	Bratislava	49035	5.30	Central
41	Slovenia	Ljubljana	20250	2.00	Southern
42	Ukraine	Kiev	603700	51.80	Eastern

A **comma-separated values (CSV) file** is a delimited text file that uses a **comma** to separate values. The separator (also called the *delimiter*) can be another character than a comma, often a **semicolon** is used. The default delimiter is the comma, but we can easily configure it int the `read_csv()` method call.

A CSV file stores tabular data (numbers and text) in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format.

For example:

```
Country;Capital;Area (km2);Population (millions);Region
Albania;Tirana;28748;3.2;Southern
Andorra;Andorra la Vella;468;0.07;Western
Austria;Vienna;83857;7.6;Western
Belgium;Brussels;30519;10;Western
Bosnia and Herzegovina;Sarajevo;51130;4.5;Southern
Bulgaria;Sofia;110912;9;Southern
Czech Republic;Prague;78864;10.4;Central
Denmark;Copenhagen;43077;5.1;Northern
United Kingdom;London;244100;57.2;Western
...
```

Overwrite the used column names:

In [12]:

```
countries.columns = ['country', 'capital', 'area', 'population', 'region']
display(countries)
```

	country	capital	area	population	region
0	Albania	Tirana	28748	3.20	Southern
1	Andorra	Andorra la Vella	468	0.07	Western
2	Austria	Vienna	83857	7.60	Western
3	Belgium	Brussels	30519	10.00	Western
4	Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern
...	...	...	...	...	...
38	Sweden	Stockholm	449964	8.50	Northern
39	Serbia	Belgrade	66577	7.20	Southern
40	Slovakia	Bratislava	49035	5.30	Central
41	Slovenia	Ljubljana	20250	2.00	Southern
42	Ukraine	Kiev	603700	51.80	Eastern

Use further reading functions, like `read_excel`, `read_json` to easily load other file formats into a pandas DataFrame.

The same European country dataset is given in the `data/countries_europe.xls` and `data/countries_europe.xlsx` MS Excel files:

In [13]:

```
countries = pd.read_excel('../data/countries_europe.xls')
countries.columns = ['country', 'capital', 'area', 'population', 'region']
display(countries)
```

	country	capital	area	population	region
0	Albania	Tirana	28748	3.20	Southern
1	Andorra	Andorra la Vella	468	0.07	Western
2	Austria	Vienna	83857	7.60	Western
3	Belgium	Brussels	30519	10.00	Western
4	Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern
...	...	...	...	...	...
38	Sweden	Stockholm	449964	8.50	Northern
39	Serbia	Belgrade	66577	7.20	Southern
40	Slovakia	Bratislava	49035	5.30	Central
41	Slovenia	Ljubljana	20250	2.00	Southern
42	Ukraine	Kiev	603700	51.80	Eastern

In [14]:

```
countries = pd.read_excel('../data/countries_europe.xlsx')
countries.columns = ['country', 'capital', 'area', 'population', 'region']
display(countries)
```

	country	capital	area	population	region
0	Albania	Tirana	28748	3.20	Southern
1	Andorra	Andorra la Vella	468	0.07	Western
2	Austria	Vienna	83857	7.60	Western
3	Belgium	Brussels	30519	10.00	Western
4	Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern
...	...	...	...	...	...
38	Sweden	Stockholm	449964	8.50	Northern
39	Serbia	Belgrade	66577	7.20	Southern
40	Slovakia	Bratislava	49035	5.30	Central
41	Slovenia	Ljubljana	20250	2.00	Southern
42	Ukraine	Kiev	603700	51.80	Eastern

Working with DataFrames

Query the row and column count of a DataFrame:

In [15]:

```
print('Number of rows: {0}'.format(len(countries)))
print('Number of rows: {0}'.format(countries.shape[0]))
print('Number of columns: {0}'.format(countries.shape[1]))
```

```
Number of rows: 43
Number of rows: 43
Number of columns: 5
```

In [16]:

```
print('Number of rows by columns:')
print(countries.count())
```

```
Number of rows by columns:
country       43
capital       43
area          43
population    43
region        43
dtype: int64
```

Remark: if a column contains empty cells, we would see different numbers here.

The first or last few rows of a DataFrame can be fetched with the `head(n)` and `tail(n)` methods. They are especially useful with large DataFrames.

In [17]:

```
display(countries.head(3))
```

	country	capital	area	population	region
0	Albania	Tirana	28748	3.20	Southern
1	Andorra	Andorra la Vella	468	0.07	Western
2	Austria	Vienna	83857	7.60	Western

In [18]:

```
display(countries.tail(3))
```

	country	capital	area	population	region
40	Slovakia	Bratislava	49035	5.3	Central
41	Slovenia	Ljubljana	20250	2.0	Southern
42	Ukraine	Kiev	603700	51.8	Eastern

Add a new column to a DataFrame

Calculate the population density of each country and add it as a new column to the `countries` *DataFrame*.

First, create a list of the density values:

In [19]:

```
density = []
for i in range(len(countries)):
    density.append(countries['population'][i] * 1e6 / countries['area'][i])

print(density)
```

```
[111.31209127591485, 149.57264957264957, 90.63047807577185, 327.6647
3344473934, 88.01095247408567, 81.14541257934218, 131.8725907892067
3, 118.39264572741834, 234.33019254403933, 35.47671840354767, 49.614
643545279385, 14.490824941962767, 103.3154706644729, 75.782262403661
8, 436.1535967936817, 83.1858407079646, 49.79867108689157, 2.9126213
59223301, 202.07587030403232, 120.88920728021671, 40.81632653061224
4, 187.5, 55.214723926380366, 154.67904098994586, 81.67075020417687,
111.78468549808676, 949.367088607595, 130.56379821958458, 15000.0, 4
3.440486533449175, 220.14216814828507, 12.967885956705786, 190.85426
36842507, 113.6498933855762, 97.6842105263158, 491.8032786885246, 7
6.86486443652903, 162.25510377061488, 18.890400120898562, 108.145455
63783288, 108.08606097685326, 98.76543209876543, 85.80420738777539]
```

Then add a new column to the DataFrame:

In [20]:

```
countries['density'] = density
display(countries)
```

	country	capital	area	population	region	density
0	Albania	Tirana	28748	3.20	Southern	111.312091
1	Andorra	Andorra la Vella	468	0.07	Western	149.572650
2	Austria	Vienna	83857	7.60	Western	90.630478
3	Belgium	Brussels	30519	10.00	Western	327.664733
4	Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern	88.010952
...	...	...	...	...	...	...
38	Sweden	Stockholm	449964	8.50	Northern	18.890400
39	Serbia	Belgrade	66577	7.20	Southern	108.145456
40	Slovakia	Bratislava	49035	5.30	Central	108.086061
41	Slovenia	Ljubljana	20250	2.00	Southern	98.765432
42	Ukraine	Kiev	603700	51.80	Eastern	85.804207

Calculating aggregated values

Summation, mean and median values:

In [21]:

```
print("Sum population: {0:.2f} million".format(countries["population"].sum()))
print("Mean population: {0:.2f} million".format(countries["population"].mean()))
print("Median population: {0:.2f} million".format(countries["population"].median
()))
```

```
Sum population: 572.16 million
Mean population: 13.31 million
Median population: 5.10 million
```

Maximum and minimum values and their indexes:

In [22]:

```
print("Max population:{0:.2f} million".format(countries["population"].max()))
print("Max population index: {0}".format(countries["population"].idxmax()))
print("Min population:{0:.2f} million".format(countries["population"].min()))
print("Min population index: {0}".format(countries["population"].idxmin()))
```

```
Max population:78.60 million
Max population index: 30
Min population:0.03 million
Min population index: 21
```

Standard deviation:

In [23]:

```
print("Standard deviation of population: {0:.2f} million".format(countries["popu
lation"].std()))
```

```
Standard deviation of population: 19.42 million
```

Quantile calculation:

In [24]:

```
print("The 90% quantile for the European countries:")
print(countries.quantile(0.9))
```

```
The 90% quantile for the European countries:
area          353262.600000
population        49.200000
density          308.997825
Name: 0.9, dtype: float64
```

Which means that e.g. the top 5 countries (top 10%) has a higher population than 49.2 million.

Calculate all basic statistical data for all columns at once:

In [25]:

```
display(countries.describe())
```

	area	population	density
count	43.000000	43.000000	43.000000
mean	136551.162791	13.306047	489.478549
std	163143.708680	19.415679	2271.195151
min	2.000000	0.030000	2.912621
25%	29633.500000	2.150000	76.323563
50%	65200.000000	5.100000	108.086061
75%	222550.000000	10.400000	158.467072
max	603700.000000	78.600000	15000.000000

Sorting the DataFrame

A DataFrame can be sorted into a **new** DataFrame through the `sort_values` function. The original DataFrame remains intact.

In [26]:

```
bypopulation = countries.sort_values(by = 'population', ascending = False)
display(bypopulation)
```

	country	capital	area	population	region	density
30	Germany	Berlin	357042	78.60	Western	220.142168
32	Italy	Rome	301277	57.50	Southern	190.854264
8	United Kingdom	London	244100	57.20	Western	234.330193
12	France	Paris	543965	56.20	Western	103.315471
42	Ukraine	Kiev	603700	51.80	Eastern	85.804207
36	Spain	Madrid	504782	38.80	Southern	76.864864
...	...	...	...	...	...	...
17	Iceland	Reykjavik	103000	0.30	Northern	2.912621
1	Andorra	Andorra la Vella	468	0.07	Western	149.572650
28	Monaco	Monaco	2	0.03	Southern	15000.000000
35	San Marino	San Marino	61	0.03	Southern	491.803279
21	Liechtenstein	Vaduz	160	0.03	Western	187.500000

Note that the row indices remained intact.
Task: which countries will display the following code cell?

In [27]:

```
print(bypopulation.loc[0])
print()
print(bypopulation.iloc[0])
```

```
country          Albania
capital           Tirana
area               28748
population           3.2
region           Southern
density       111.312091
Name: 0, dtype: object

country          Germany
capital           Berlin
area              357042
population          78.6
region           Western
density       220.142168
Name: 30, dtype: object
```

Now we can e.g. verify that the top 5 countries (top 10%) has a higher population than 49.2 million:

In [28]:

```
print("Population of the 4th country: {0:.1f} million".format(bypopulation.iloc[
4]["population"]))
print("Population of the 5th country: {0:.1f} million".format(bypopulation.iloc[
5]["population"]))
```

```
Population of the 4th country: 51.8 million
Population of the 5th country: 38.8 million
```

Note: a DataFrame can be sorted by modifying it (and without creating a new one) by passing the `inplace = True` argument to the `sort_values()` method:

```
countries.sort_values(by = 'population', ascending = False, inplace = Tru
e)
```

Sorting by multiple columns

A *DataFrame* can be sorted using multiple columns, by passing a list of columns to the `by` parameter. (And a list of boolean values to the `ascending` parameter.)

In [29]:

```
byregion = countries.sort_values(by = ['region', 'population'], ascending = [True, False])
display(byregion)
```

	country	capital	area	population	region	density
19	Poland	Warsaw	312683	37.80	Central	120.889207
6	Czech Republic	Prague	78864	10.40	Central	131.872591
25	Hungary	Budapest	93036	10.40	Central	111.784685
40	Slovakia	Bratislava	49035	5.30	Central	108.086061
42	Ukraine	Kiev	603700	51.80	Eastern	85.804207
...	...	...	...	...	...	...
37	Switzerland	Berne	41293	6.70	Western	162.255104
16	Ireland	Dublin	70283	3.50	Western	49.798671
23	Luxembourg	Luxembourg	2586	0.40	Western	154.679041
1	Andorra	Andorra la Vella	468	0.07	Western	149.572650
21	Liechtenstein	Vaduz	160	0.03	Western	187.500000

Indexing the DataFrame

We can assign one of the columns as the index column. The indexer column must contain unique values.

In [30]:

```
countries_indexed = countries.set_index('country')
display(countries_indexed)
```

country	capital	area	population	region	density
Albania	Tirana	28748	3.20	Southern	111.312091
Andorra	Andorra la Vella	468	0.07	Western	149.572650
Austria	Vienna	83857	7.60	Western	90.630478
Belgium	Brussels	30519	10.00	Western	327.664733
Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern	88.010952
...	...	...	...	...	...
Sweden	Stockholm	449964	8.50	Northern	18.890400
Serbia	Belgrade	66577	7.20	Southern	108.145456
Slovakia	Bratislava	49035	5.30	Central	108.086061
Slovenia	Ljubljana	20250	2.00	Southern	98.765432
Ukraine	Kiev	603700	51.80	Eastern	85.804207

In the indexed *DataFrame*, rows can be accessed through the values of the index column with the already used `loc` property:

In [31]:

```
print(countries_indexed.loc["Hungary"])
```

```
capital         Budapest
area               93036
population          10.4
region           Central
density       111.784685
Name: Hungary, dtype: object
```

Remark: by default the `set_index()` method call removes the indexer column. We can keep the indexer column also as a "normal" column through setting the `drop` parameter to `False` .

In [32]:

```
countries_indexed = countries.set_index('country', drop = False)
display(countries_indexed)
```

country	country	capital	area	population	region	density
Albania	Albania	Tirana	28748	3.20	Southern	111.312091
Andorra	Andorra	Andorra la Vella	468	0.07	Western	149.572650
Austria	Austria	Vienna	83857	7.60	Western	90.630478
Belgium	Belgium	Brussels	30519	10.00	Western	327.664733
Bosnia and Herzegovina	Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern	88.010952
...	...	...	...	...	...	...
Sweden	Sweden	Stockholm	449964	8.50	Northern	18.890400
Serbia	Serbia	Belgrade	66577	7.20	Southern	108.145456
Slovakia	Slovakia	Bratislava	49035	5.30	Central	108.086061
Slovenia	Slovenia	Ljubljana	20250	2.00	Southern	98.765432
Ukraine	Ukraine	Kiev	603700	51.80	Eastern	85.804207

Filtering the DataFrame

Configure a condition as a boolean expression:

In [33]:

```
condition = countries["region"] == "Central"
print(type(condition))
```

```
<class 'pandas.core.series.Series'>
```

As we have seen, the result is a *Series*, so a new column! Evaluate it:

In [34]:

```
print(condition)
```

```
0     False
1     False
2     False
3     False
4     False
5     False
6      True
7     False
...
39    False
40     True
41    False
42    False
Name: region, dtype: bool
```

The series in the `condition` variable stores the boolean `True` / `False` value for each row, whether the profit was positive for that row or not.

Now filter the DataFrame by the condition. As we can observe, only the rows with the logical `True` value in the `condition` series remained:

In [35]:

```
centralEuropean = countries[condition]
display(centralEuropean)
```

	country	capital	area	population	region	density
6	Czech Republic	Prague	78864	10.4	Central	131.872591
19	Poland	Warsaw	312683	37.8	Central	120.889207
25	Hungary	Budapest	93036	10.4	Central	111.784685
40	Slovakia	Bratislava	49035	5.3	Central	108.086061

The whole workflow can be achieved in a single statement:

In [36]:

```
display(countries[countries["region"] == "Central"])
```

	country	capital	area	population	region	density
6	Czech Republic	Prague	78864	10.4	Central	131.872591
19	Poland	Warsaw	312683	37.8	Central	120.889207
25	Hungary	Budapest	93036	10.4	Central	111.784685
40	Slovakia	Bratislava	49035	5.3	Central	108.086061

Summary exercise on DataFrames

Task 1

Display the name of the countries which have a population of less than 1 million.

In [37]:

```
small_countries = countries[countries["population"] < 1]
print(small_countries["country"])
```

```
1           Andorra
17          Iceland
21    Liechtenstein
23       Luxembourg
26            Malta
28           Monaco
29       Montenegro
35       San Marino
Name: country, dtype: object
```

Task 2

Write a program that calculates which Western-European country has the largest area?

In [38]:

```
max_index = countries[countries["region"] == "Western"]["area"].idxmax()
print(countries.iloc[max_index])
```

```
country           France
capital            Paris
area              543965
population          56.2
region           Western
density       103.315471
Name: 12, dtype: object
```

Chapter 10: Plotting and diagram visualization

Matplotlib is the most popular 2D plotting library in Python. Using matplotlib, you can create pretty much any type of plot.

Pandas has **tight integration** with *matplotlib*.

How to install matplotlib?

If you have Anaconda installed, then matplotlib was already installed together with it.

If you have a standalone Python3 and Jupyter Notebook installation, open a command prompt / terminal and type in:

```
pip3 install matplotlib
```

How to use matplotlib?

We will use the *pyplot module* inside the matlplotlib package for plotting. You can simply import this module as usual. It is usually aliased with the `plt` abbreviation:

```
import matplotlib.pyplot as plt
```

The dataset

Let's use the *World Countries datatset*. For each country the following information is given:

1. country name,
2. region name,
3. population,
4. area (in mi^2),
5. GDP ($ per capita),
6. Literacy (%)

The dataset is given in the `data/countries_world.csv` file. The used delimiter is the semicolon (`;`) character.

In [1]:

```
import pandas as pd
import matplotlib.pyplot as plt

# Special Jupyter Notebook command, so the plots by matplotlib will be display i
nside the Jupyter Notebook
%matplotlib inline

countries = pd.read_csv('../data/countries_world.csv', delimiter = ';')
countries.columns = ['country', 'region', 'population', 'area', 'gdp', 'literac
y']
display(countries)
```

	country	region	population	area	gdp	literacy
0	Afghanistan	ASIA (EX. NEAR EAST)	31056997	647500	700.0	36.0
1	Albania	EASTERN EUROPE	3581655	28748	4500.0	86.5
2	Algeria	NORTHERN AFRICA	32930091	2381740	6000.0	70.0
3	American Samoa	OCEANIA	57794	199	8000.0	97.0
4	Andorra	WESTERN EUROPE	71201	468	19000.0	100.0
...	...	...	...	...	...	...
222	West Bank	NEAR EAST	2460492	5860	800.0	NaN
223	Western Sahara	NORTHERN AFRICA	273008	266000	NaN	NaN
224	Yemen	NEAR EAST	21456188	527970	800.0	50.2
225	Zambia	SUB-SAHARAN AFRICA	11502010	752614	800.0	80.6
226	Zimbabwe	SUB-SAHARAN AFRICA	12236805	390580	1900.0	90.7

227 rows × 6 columns

Data source: [US Government (https://gsociology.icaap.org/dataupload.html)](https://gsociology.icaap.org/dataupload.html)

Lets take just the top 50 countries by area, so visualization will be easier to overview in the following tasks:

In [2]:

```
countries50 = countries.sort_values(by = 'area', ascending = False).head(50)
display(countries50)
```

	country	region	population	area	gdp	literacy
169	Russia	C.W. OF IND. STATES	142893540	17075200	8900.0	99.6
36	Canada	NORTHERN AMERICA	33098932	9984670	29800.0	97.0
214	United States	NORTHERN AMERICA	298444215	9631420	37800.0	97.0
42	China	ASIA (EX. NEAR EAST)	1313973713	9596960	5000.0	90.9
27	Brazil	LATIN AMER. & CARIB	188078227	8511965	7600.0	86.4
...	...	...	...	...	...	...
124	Madagascar	SUB-SAHARAN AFRICA	18595469	587040	800.0	68.9
107	Kenya	SUB-SAHARAN AFRICA	34707817	582650	1000.0	85.1
69	France	WESTERN EUROPE	60876136	547030	27600.0	99.0
224	Yemen	NEAR EAST	21456188	527970	800.0	50.2
201	Thailand	ASIA (EX. NEAR EAST)	64631595	514000	7400.0	92.6

Plotting

Plots can be generated with the `plot()` function of a Pandas *DataFrame* (table) or *Series* (column). The most important parameter of the function is the `kind` parameter, which defines the type of plot to be generated. Supported kinds are (non-exhaustive list):

- `line`
- `bar` (vertical bar)
- `barh` (horizontal bar)
- `scatter`
- `hist` (histogram)
- `box` (boxplot
- `pie`

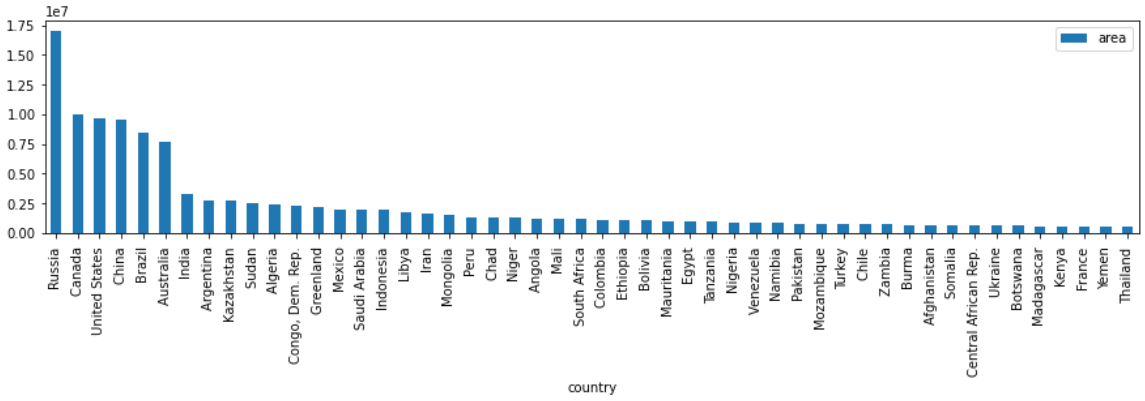
After a plot is generated, it can be displayed by the `show()` function of the `matplotlib.pyplot` module.

Vertical bar plot

Display a bar plot on the area of the selected 50 largest countries.

In [3]:

```
countries50.plot(kind='bar', x='country', y='area', figsize = [15, 3])
plt.show() # matplotlib.pyplot was imported as plt
```

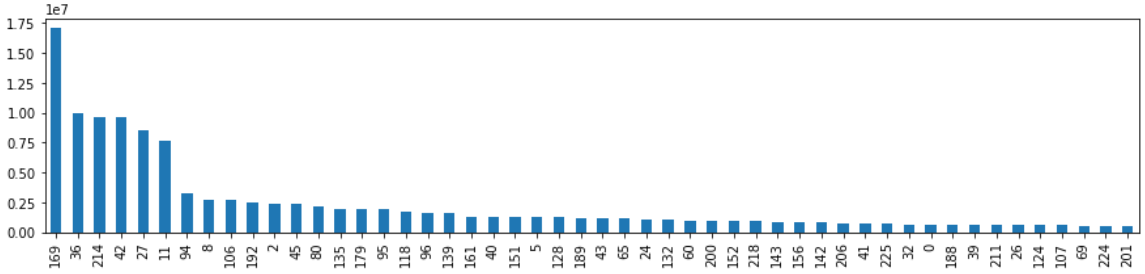


The size of the diagram can be configured with the `figsize` parameter. The size is given in inches (1 inch = 2.54 centimeters).
The default size is `[6.4, 4.8]` .

The bar diagram can be created directly on the selected *Series* (column of data). In this case the Series will be placed along axis Y, while the horizontal axis X will become the index of the *DataFrame*.

In [4]:

```
countries50['area'].plot(kind='bar', figsize = [15, 3])
plt.show()
```



The index column can be modified through the `set_index` function (see Chapter 7 for more details) of the *DataFrame* and a **new** *DataFrame* is created so:

In [5]:

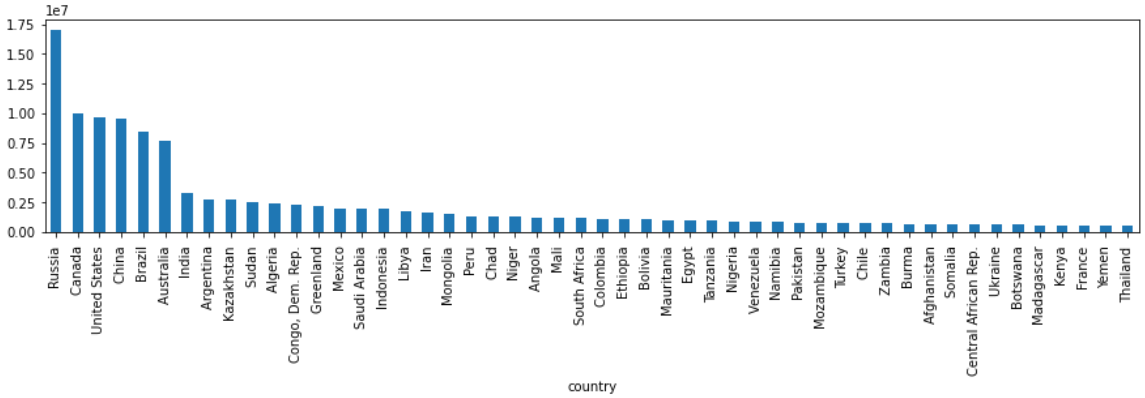
```
countries50_indexed = countries50.set_index('country')
display(countries50_indexed)
```

country	region	population	area	gdp	literacy
Russia	C.W. OF IND. STATES	142893540	17075200	8900.0	99.6
Canada	NORTHERN AMERICA	33098932	9984670	29800.0	97.0
United States	NORTHERN AMERICA	298444215	9631420	37800.0	97.0
China	ASIA (EX. NEAR EAST)	1313973713	9596960	5000.0	90.9
Brazil	LATIN AMER. & CARIB	188078227	8511965	7600.0	86.4
...	...	...	...	...	...
Madagascar	SUB-SAHARAN AFRICA	18595469	587040	800.0	68.9
Kenya	SUB-SAHARAN AFRICA	34707817	582650	1000.0	85.1
France	WESTERN EUROPE	60876136	547030	27600.0	99.0
Yemen	NEAR EAST	21456188	527970	800.0	50.2
Thailand	ASIA (EX. NEAR EAST)	64631595	514000	7400.0	92.6

Creating the bar plot from the `countries50_indexed` *DataFrame* will display the country names as labels correctly.

In [6]:

```
countries50_indexed['area'].plot(kind='bar', figsize = [15, 3])
plt.show()
```

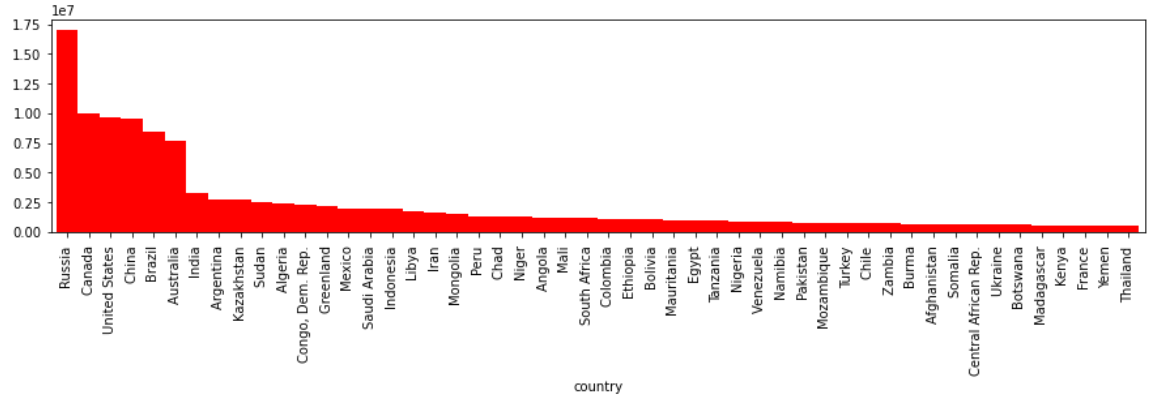


Visual tuning

The color of the bars can be defined with the `color` parameter. The width of the bars is set by the `width` parameter, 1.0 meaning *100%*.

In [7]:

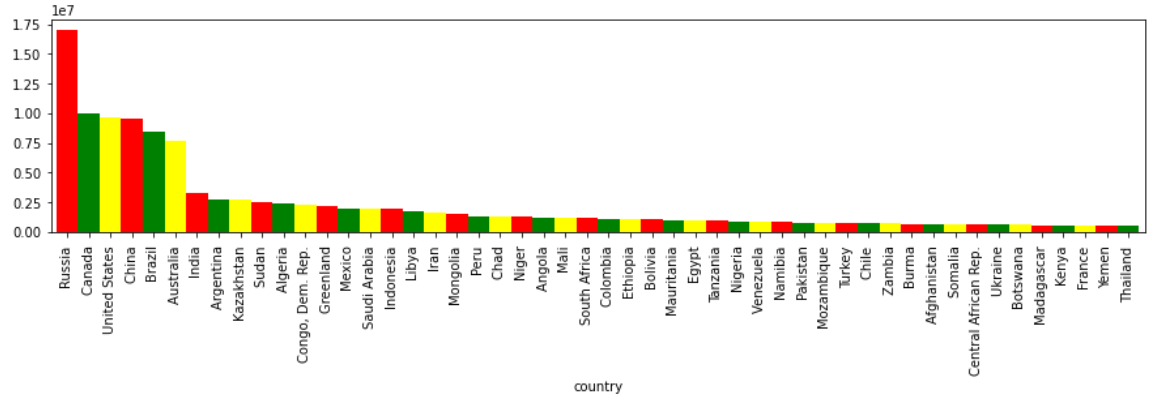
```
countries50_indexed['area'].plot(kind='bar', figsize = [15, 3], color = 'red', w
idth = 1.0)
plt.show()
```



Multiple colors can be passed in a list.

In [8]:

```
countries50_indexed['area'].plot(kind='bar', figsize = [15, 3], color = ['red',
'green', 'yellow'], width = 1.0)
plt.show()
```

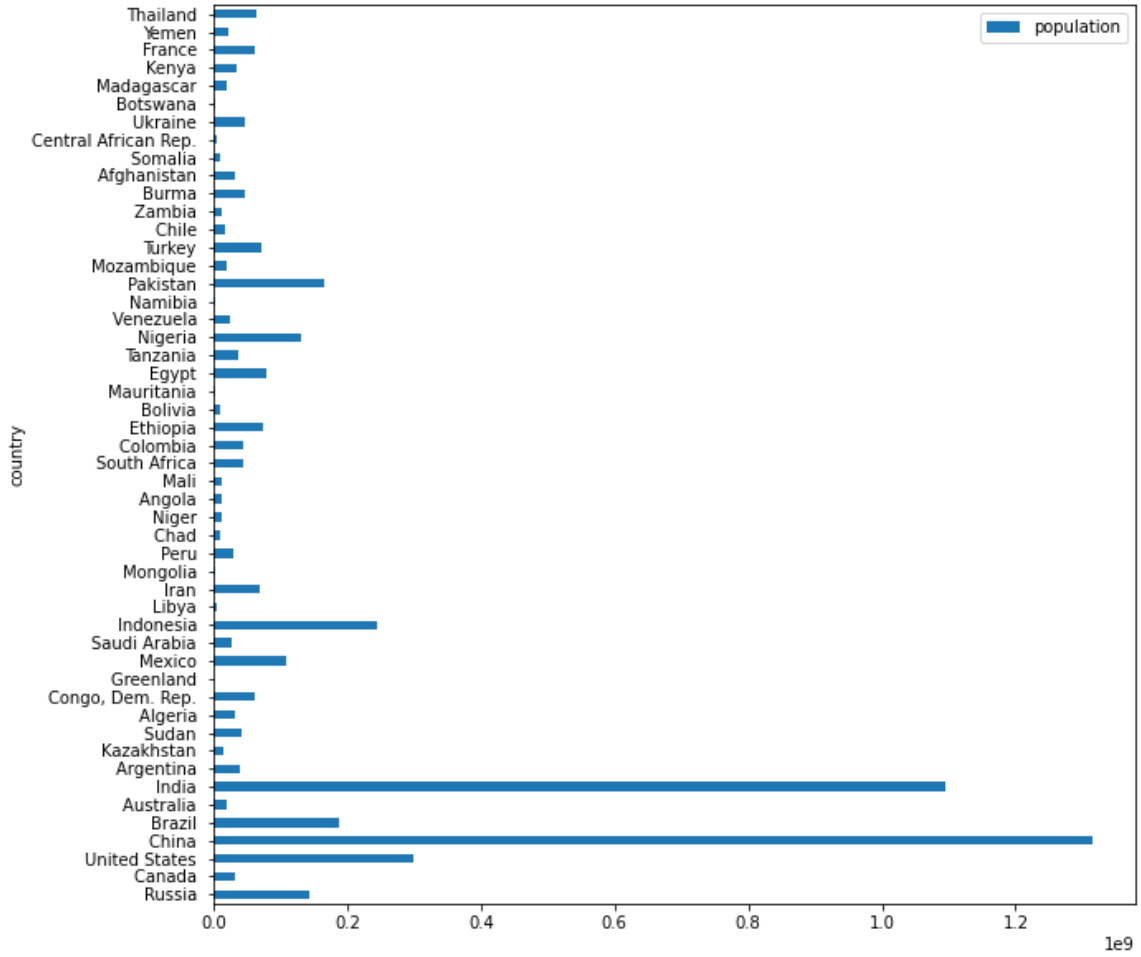


Horizontal bar plot

Display a horizontal bar plot on the population of the selected 50 largest countries.

In [9]:

```
countries50.plot(kind='barh', x='country', y='population', figsize = [10, 10])
plt.show()
```

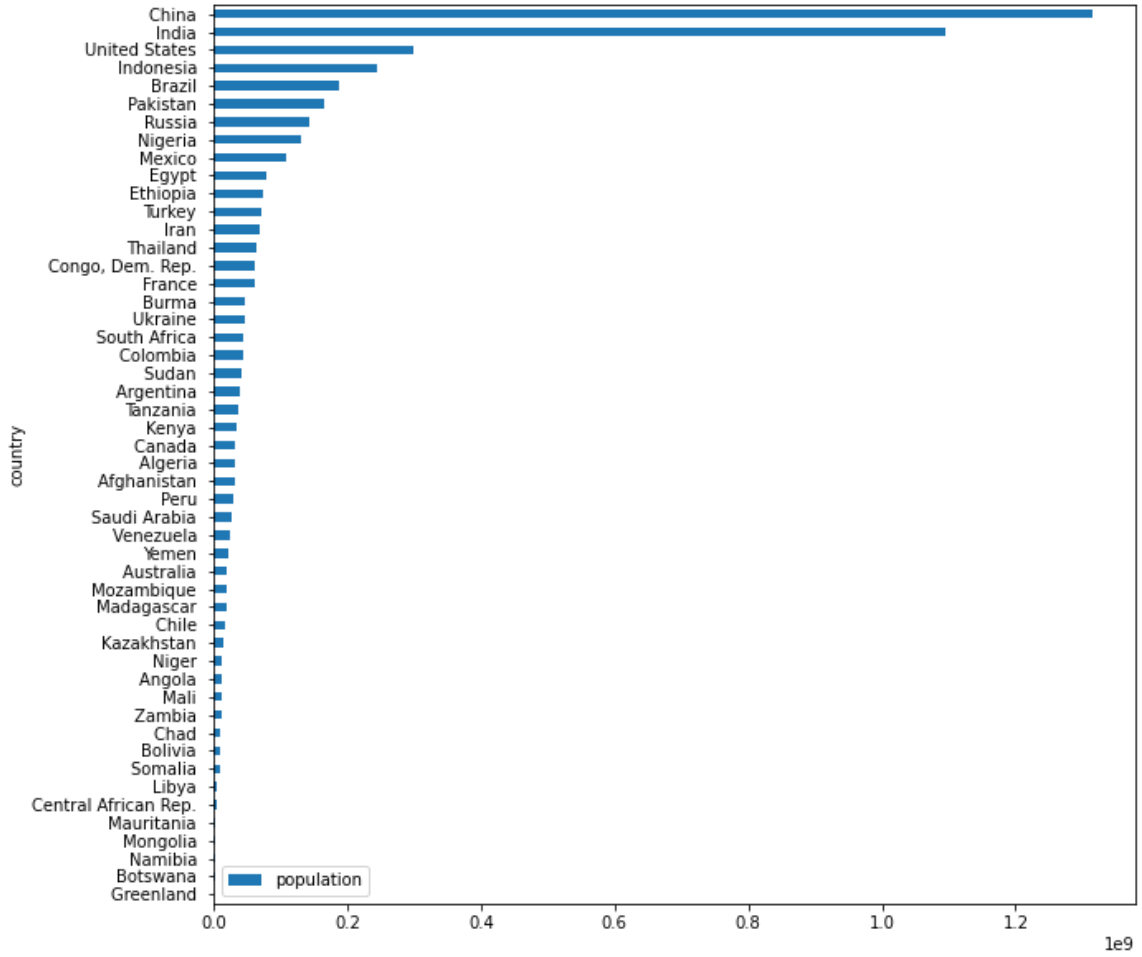


Note that for the horizontal bar plot, the *axis X* is the vertical axis, and *axis Y* is the horizontal axis. It is defined by this was, so only the `kind` parameter of the `plot()` function has to be changed when switching to a different type of diagram.

Before visualizing the data, sort it by the column population, instead of the default area.

In [10]:

```
countries50.sort_values(by = 'population').plot(kind='barh', x='country', y='population', figsize = [10, 10])
plt.show()
```



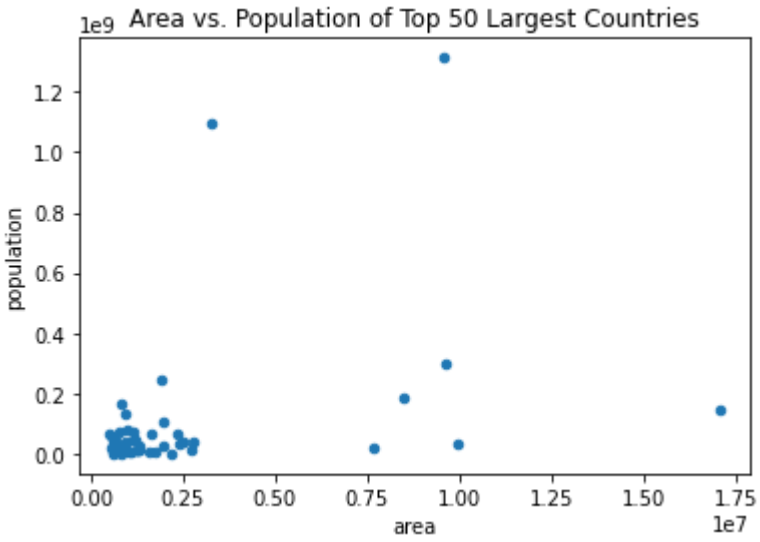
Scatter plot

Display a scatter plot on the correlation of the area and the population columns of the selected 50 largest countries.

Question: What correlation can be expected between these 2 attributes of countries?

In [11]:

```
countries50.plot(kind='scatter', x='area', y='population', title='Area vs. Popul
ation of Top 50 Largest Countries')
plt.show()
```

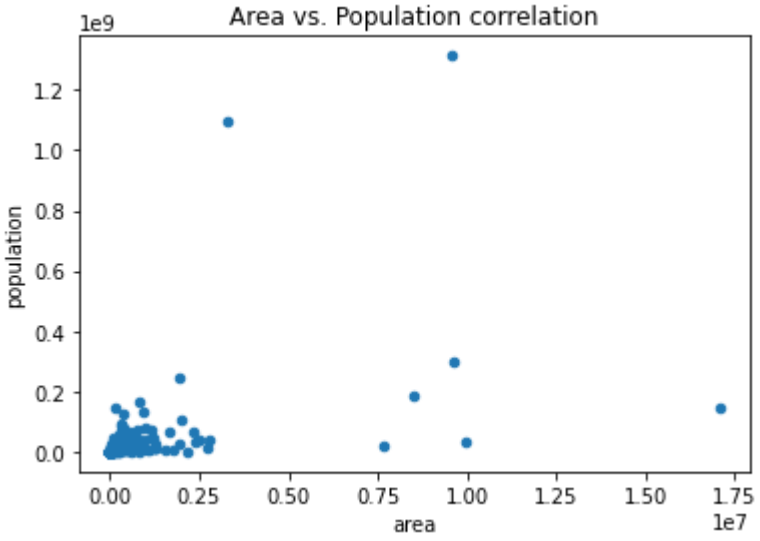


A title can be given to be displayed above the generated diagram with the `title` parameter.

Extend the scatter plot for all countries in the dataset.

In [12]:

```
countries.plot(kind='scatter', x='area', y='population', title='Area vs. Populat
ion correlation')
plt.show()
```

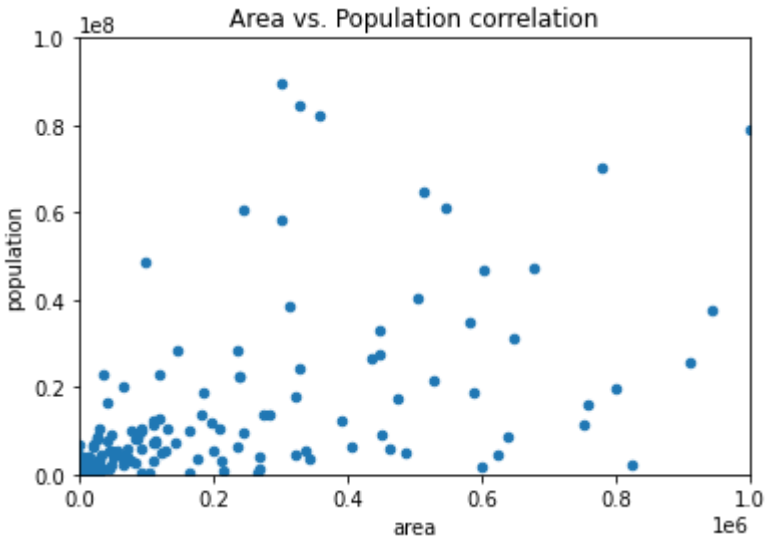


As we can observe there is a moderately strong correlation between area and population, which matches our expectation.

The limits of the X and Y axes can be configured with the `xlim` and `ylim` parameters, so the *outliers* can be excluded from the visualization. Both a minimum and a maximum boundary can be given, as a tuple.

In [13]:

```
countries.plot(kind='scatter', x='area', y='population', title='Area vs. Populat
ion correlation', xlim=(0, 1e6), ylim=(0, 1e8))
plt.show()
```



Short outlook on correlation *(optional)*

The correlation matrix between *Series* of a *Pandas DataFrame* can be generated with the `corr()` function:

In [14]:

```
display(countries.corr())
```

	population	area	gdp	literacy
population	1.000000	0.469985	-0.039324	-0.043481
area	0.469985	1.000000	0.072185	0.035994
gdp	-0.039324	0.072185	1.000000	0.513144
literacy	-0.043481	0.035994	0.513144	1.000000

Or just for 2 selected Series:

In [15]:

```
print(countries['area'].corr(countries['population']))
```

```
0.46998508371848174
```

Every correlation has two qualities: *strength* and *direction*. The direction of a correlation is either positive or negative. When two variables have a positive correlation, it means the variables move in the same direction. This means that as one variable increases, so does the other one. In a negative correlation, the variables move in inverse, or opposite, directions. In other words, as one variable increases, the other variable decreases.

We determine the strength of a relationship between two correlated variables by looking at the numbers. A correlation of 0 means that no relationship exists between the two variables, whereas a correlation of 1 indicates a perfect positive relationship. It is uncommon to find a perfect positive relationship in the real world.

The further away from 1 that a positive correlation lies, the weaker the correlation. Similarly, the further a negative correlation lies from -1, the weaker the correlation. The following guidelines are useful when determining the strength of a positive correlation:

- 1: perfect positive correlation
- .70 to .99: very strong positive relationship
- .40 to .69: strong positive relationship
- .30 to .39: moderate positive relationship
- .20 to .29: weak positive relationship
- .01 to .19: no or negligible relationship
- 0: no relationship exists

Question: which series of the dataframe show strong correlation?

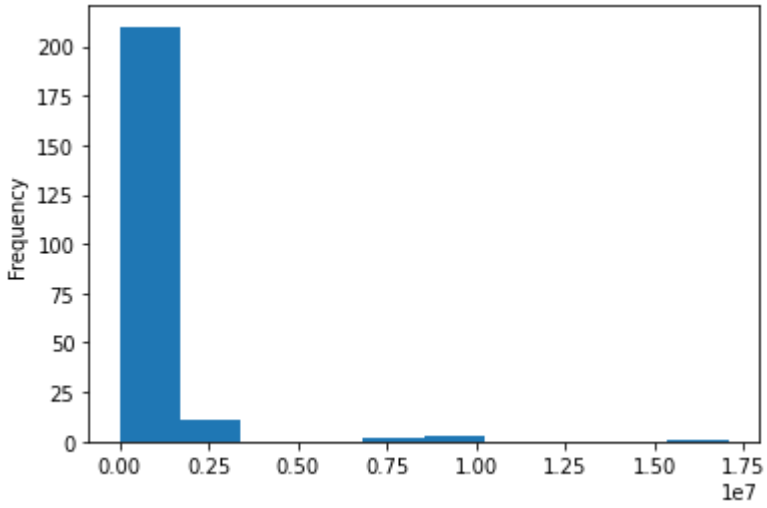
Histogram

A histogram is an accurate representation of the distribution of numerical data. It differs from a bar graph, in the sense that a bar graph relates two variables, but a histogram relates only one.

Display a histogram on the area of the selected 50 countries.

In [16]:

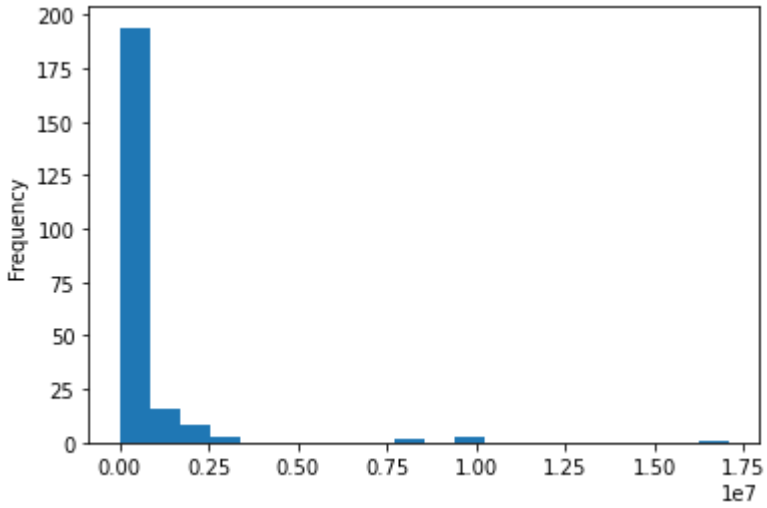
```
countries['area'].plot(kind='hist')
plt.show()
```



The number of columns (called *bins* or *buckets*) in the histrogram can be configured with the `bins` parameter.

In [17]:

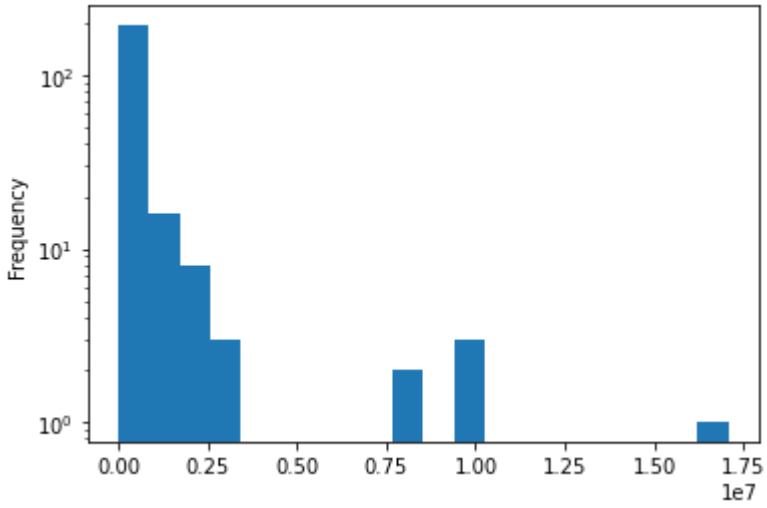
```
countries['area'].plot(kind='hist', bins=20)
plt.show()
```



Extend the histogram to cover all countries in the dataset. Apply a logarithmic scale with the `logx / logy` parameter.

In [18]:

```
countries['area'].plot(kind='hist', bins=20, logy=True)
plt.show()
```



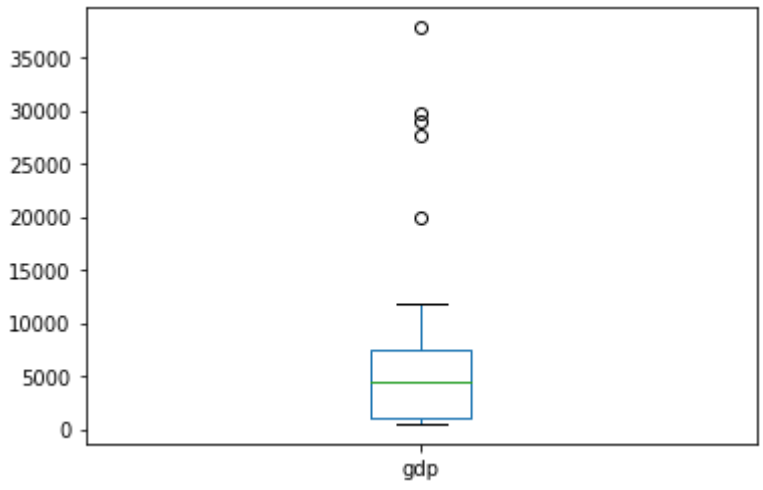
Boxplot

In descriptive statistics, a *boxplot* is a method for graphically depicting groups of numerical data through their quartiles.

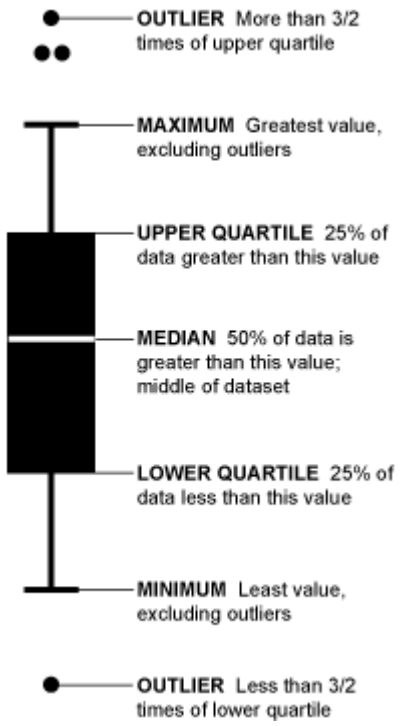
Display a boxplot on the GDP of the selected 50 largest countries.

In [19]:

```
countries50['gdp'].plot(kind='box')
plt.show()
```



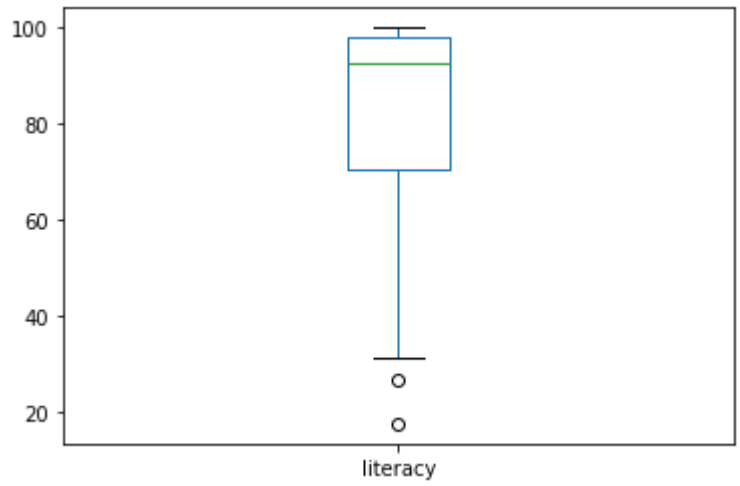
Explaining the graphical visualization of a boxplot:



Task: Display a boxplot on the literacy of all countries! What can we state based on the diagram?

In [20]:

```
countries['literacy'].plot(kind='box')
plt.show()
```

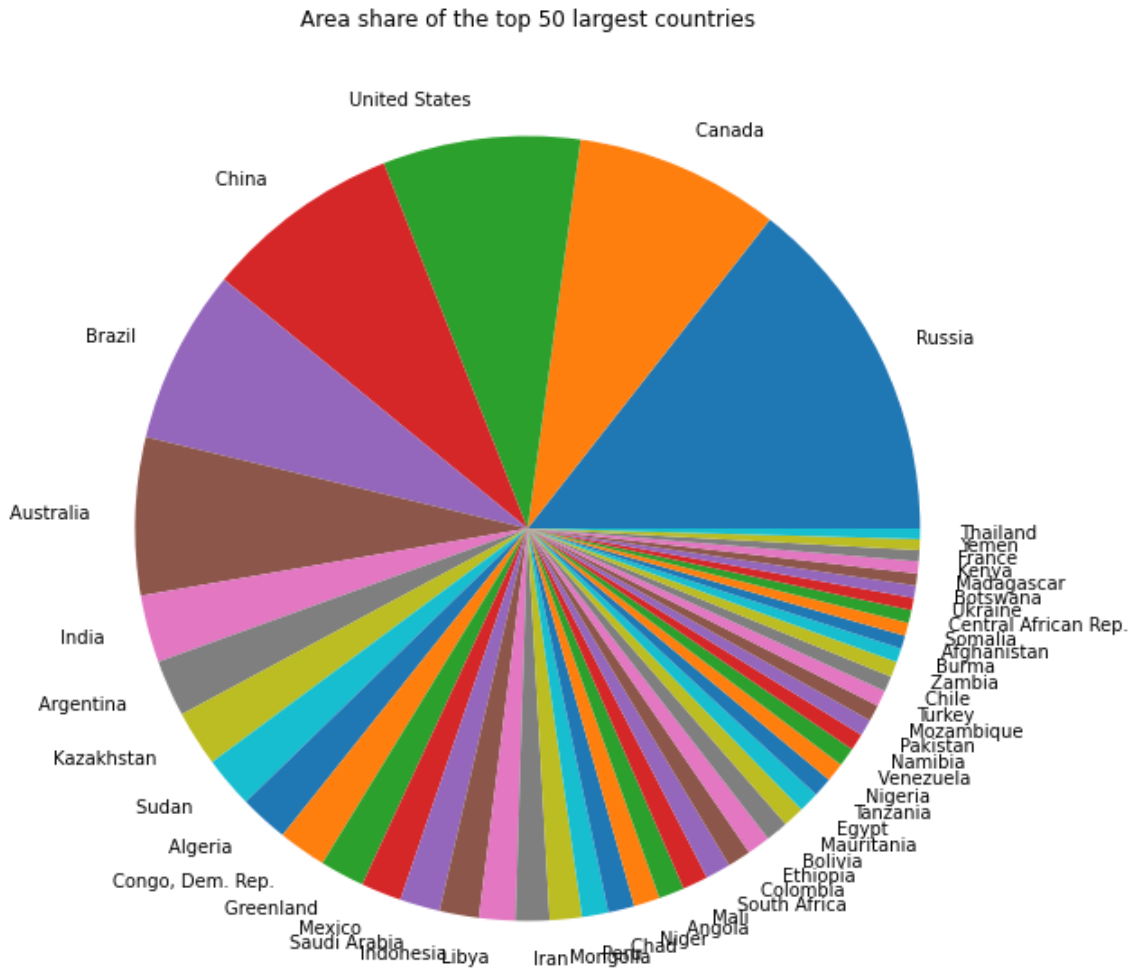


Pie chart

Display a pie chart on the area share of the selected 50 largest countries. Since we are creating this plot on the `area` *Series*, we use the `countries50_indexed` DataFrame, which was indexed with the country names, so the labels will contain them instead of numerical indices.

In [21]:

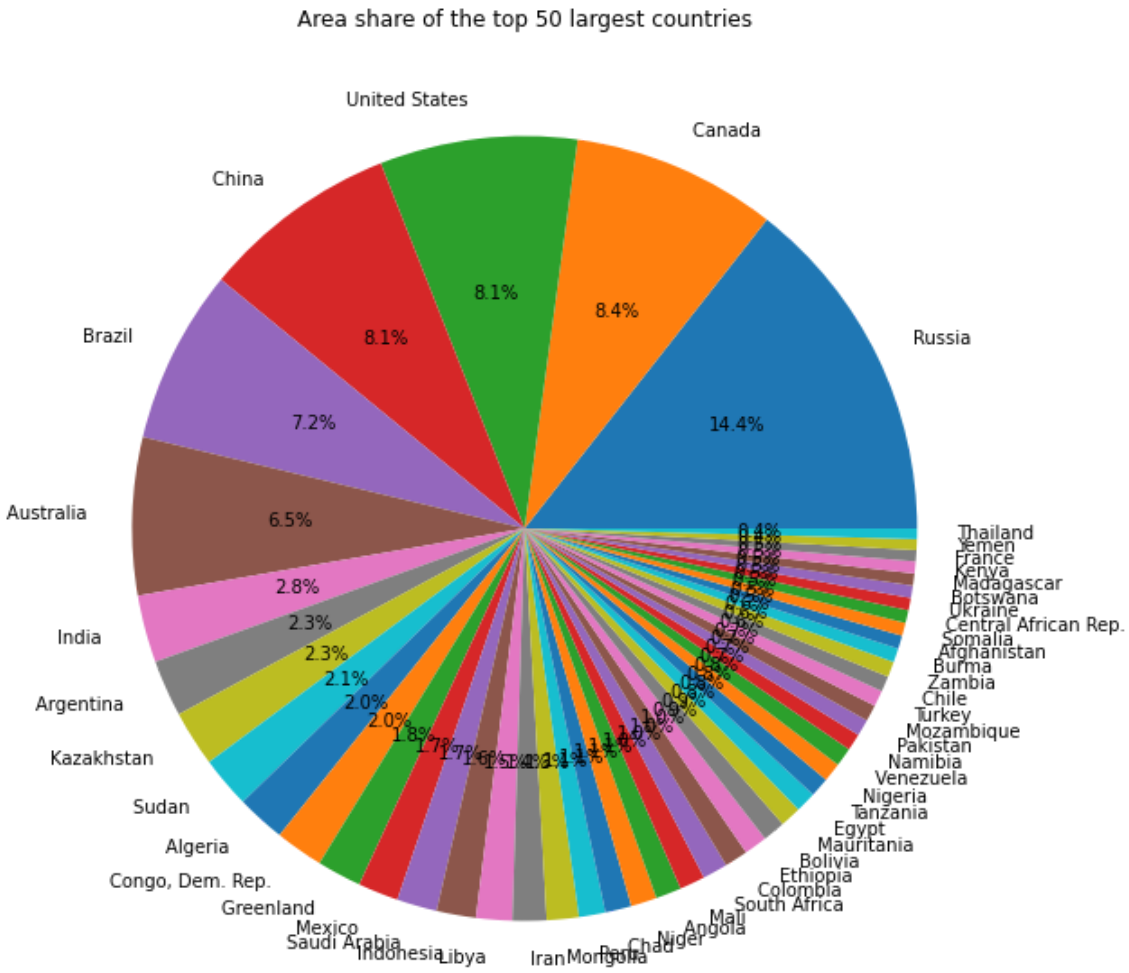
```
countries50_indexed['area'].plot(kind='pie', figsize=[10,10], label="", title="Area share of the top 50 largest countries")
plt.show()
```



Percentages for each slice can be displayed with the `autopct` parameter:

In [22]:

```
countries50_indexed['area'].plot(kind='pie', figsize=[10,10], autopct='%.1f%%',
label="", title="Area share of the top 50 largest countries")
plt.show()
```

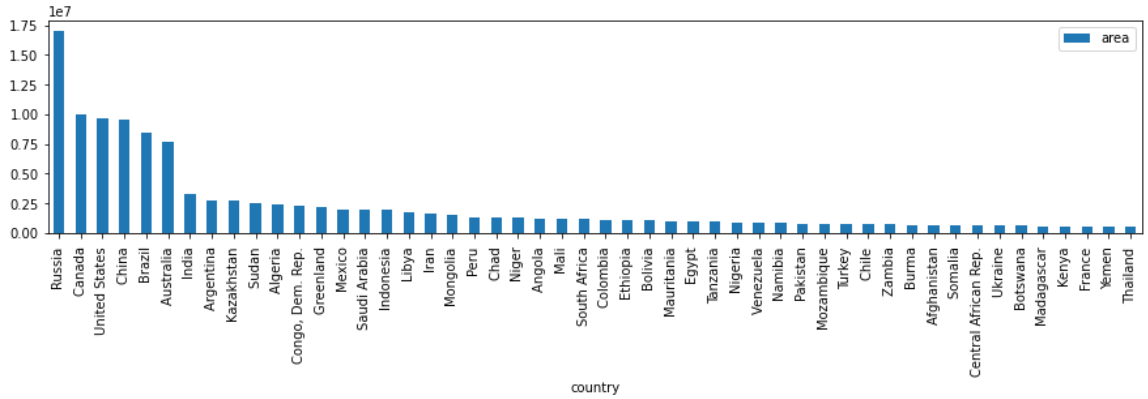


Saving a plot to file

Intead of using the `show()` function of the `matplotlib.pyplot` module, the `savefig()` function can also be used to export and save a created plot to an external file.

In [23]:

```
countries50.plot(kind='bar', x='country', y='area', figsize = [15, 3])
plt.savefig('10_country_area.png')
```



Hint: look for the created file right next this Jupyter Notebook file on your computer.

Time Series Analysis

Read the *Population History dataset* from the `data/population_world.csv` file, which contains the population data for all countries between the years 1950 and 2019. All together the dataset contains 239 countries (or territories), 70 years of data, so all together 16,730 rows of data.

Each row stores the following data:

1. location (country or region),
2. year,
3. male population,
4. female population,
5. total population,
6. population density.

The used delimiter is the semicolon (`;`) character.

In [24]:

```
population_history = pd.read_csv('../data/population_history.csv', delimiter =
';')
display(population_history)
```

	Country	Year	PopMale	PopFemale	PopTotal	PopDensity
0	Afghanistan	1950	4099.243	3652.874	7752.117	11.874
1	Afghanistan	1951	4134.756	3705.395	7840.151	12.009
2	Afghanistan	1952	4174.450	3761.546	7935.996	12.156
3	Afghanistan	1953	4218.336	3821.348	8039.684	12.315
4	Afghanistan	1954	4266.484	3884.832	8151.316	12.486
...	...	...	...	...	...	...
16725	Zimbabwe	2015	6568.778	7245.864	13814.642	35.711
16726	Zimbabwe	2016	6674.206	7356.132	14030.338	36.268
16727	Zimbabwe	2017	6777.054	7459.545	14236.599	36.801
16728	Zimbabwe	2018	6879.119	7559.693	14438.812	37.324
16729	Zimbabwe	2019	6983.353	7662.120	14645.473	37.858

16730 rows × 6 columns

Data source: United Nations (https://www.un.org/development/desa/pd/)

Display the countries in the dataset:

In [25]:

```
print(population_history['Country'].unique())
```

['Afghanistan' 'Albania' 'Algeria' 'American Samoa' 'Andean Community'
 'Andorra' 'Angola' 'Anguilla' 'Antigua and Barbuda' 'Argentina' 'Armenia'
 'Aruba' 'Australia' 'Australia/New Zealand' 'Austria' 'Azerbaijan'
 'Bahamas' 'Bahrain' 'Bangladesh' 'Barbados' 'Belarus' 'Belgium' 'Belize'
 'Benin' 'Bermuda' 'Bhutan' 'Bolivia (Plurinational State of)'
 'Bonaire, Sint Eustatius and Saba' 'Bosnia and Herzegovina' 'Botswana'
 'Brazil' 'British Virgin Islands' 'Brunei Darussalam' 'Bulgaria'
 'Burkina Faso' 'Burundi' "Côte d'Ivoire" 'Cabo Verde' 'Cambodia'
 'Cameroon' 'Canada' 'Cayman Islands' 'Central African Republic' 'Chad'
 'Channel Islands' 'Chile' 'China' 'China, Hong Kong SAR'
 'China, Macao SAR' 'China, Taiwan Province of China' 'Colombia' 'Comoros'
 'Congo' 'Cook Islands' 'Costa Rica' 'Croatia' 'Cuba' 'Curaçao' 'Cyprus'
 'Czechia' "Dem. People's Republic of Korea"
 'Democratic Republic of the Congo' 'Denmark' 'Djibouti' 'Dominica'
 'Dominican Republic' 'Ecuador' 'Egypt' 'El Salvador' 'Equatorial Guinea'
 'Eritrea' 'Estonia' 'Eswatini' 'Ethiopia' 'Falkland Islands (Malvinas)'
 'Faroe Islands' 'Fiji' 'Finland' 'France' 'French Guiana'
 'French Polynesia' 'Gabon' 'Gambia' 'Georgia' 'Germany' 'Ghana'
 'Gibraltar' 'Greece' 'Greenland' 'Grenada' 'Guadeloupe' 'Guam'
 'Guatemala' 'Guinea' 'Guinea-Bissau' 'Guyana' 'Haiti' 'Holy See'
 'Honduras' 'Hungary' 'Iceland' 'India' 'Indonesia'
 'Iran (Islamic Republic of)' 'Iraq' 'Ireland' 'Isle of Man' 'Israel'
 'Italy' 'Jamaica' 'Japan' 'Jordan' 'Kazakhstan' 'Kenya' 'Kiribati'
 'Kuwait' 'Kyrgyzstan' "Lao People's Democratic Republic" 'Latvia'
 'Lebanon' 'Lesotho' 'Liberia' 'Libya' 'Liechtenstein' 'Lithuania'
 'Luxembourg' 'Madagascar' 'Malawi' 'Malaysia' 'Maldives' 'Mali' 'Malta'
 'Marshall Islands' 'Martinique' 'Mauritania' 'Mauritius' 'Mayotte'
 'Melanesia' 'Mexico' 'Micronesia' 'Micronesia (Fed. States of)' 'Monaco'
 'Mongolia' 'Montenegro' 'Montserrat' 'Morocco' 'Mozambique' 'Myanmar'
 'Namibia' 'Nauru' 'Nepal' 'Netherlands' 'New Caledonia' 'New Zealand'
 'Nicaragua' 'Niger' 'Nigeria' 'Niue' 'North Macedonia'
 'Northern Mariana Islands' 'Norway' 'Oman' 'Pakistan' 'Palau' 'Panama'
 'Papua New Guinea' 'Paraguay' 'Peru' 'Philippines' 'Poland' 'Polynesia'
 'Portugal' 'Puerto Rico' 'Qatar' 'Réunion' 'Republic of Korea'
 'Republic of Moldova' 'Romania' 'Russian Federation' 'Rwanda'
 'Saint Barthélemy' 'Saint Helena' 'Saint Kitts and Nevis' 'Saint Lucia'
 'Saint Martin (French part)' 'Saint Pierre and Miquelon'
 'Saint Vincent and the Grenadines' 'Samoa' 'San Marino'
 'Sao Tome and Principe' 'Saudi Arabia' 'Senegal' 'Serbia' 'Seychelles'
 'Sierra Leone' 'Singapore' 'Sint Maarten (Dutch part)' 'Slovakia'
 'Slovenia' 'Solomon Islands' 'Somalia' 'South Sudan' 'Spain' 'Sri Lanka'
 'State of Palestine' 'Sudan' 'Suriname' 'Sweden' 'Switzerland'

```
 'Syrian Arab Republic' 'Tajikistan' 'Thailand' 'Timor-Leste' 'Togo'
 'Tokelau' 'Tonga' 'Trinidad and Tobago' 'Tunisia' 'Turkey' 'Turkmen
istan'
 'Turks and Caicos Islands' 'Tuvalu' 'Uganda' 'Ukraine'
 'United Arab Emirates' 'United Kingdom' 'United Republic of Tanzani
a'
 'United States of America' 'United States Virgin Islands' 'Uruguay'
 'Uzbekistan' 'Vanuatu' 'Venezuela (Bolivarian Republic of)' 'Viet N
am'
 'Wallis and Futuna Islands' 'Western Sahara' 'Yemen' 'Zambia' 'Zimb
abwe']
```

Line plot

Line diagrams works best with a series of data, assuming continuous change between the known discrete values.
Let's visualize the total and male population of *Hungary* between the years 1950 an 2019.

First filter the rows based on the country for *Hungary* and set the year as the index column.

In [26]:

```
hungary = population_history[population_history['Country'] == 'Hungary']
hungary.set_index('Year', drop=False, inplace=True)
display(hungary)
```

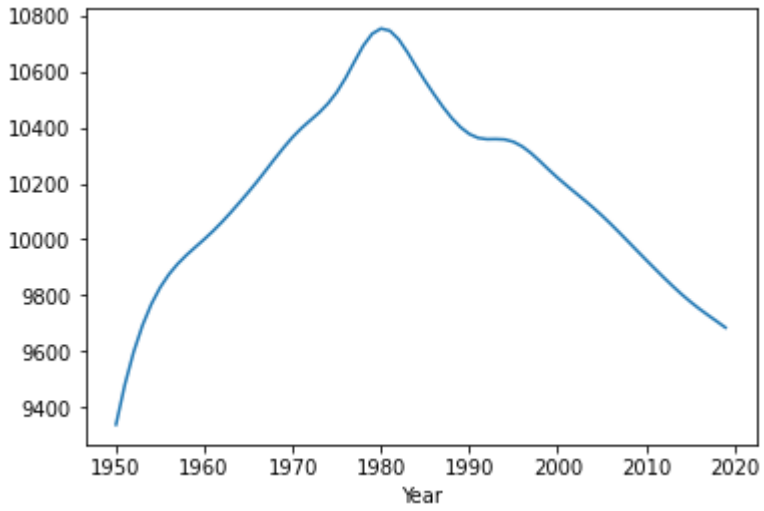
Year	Country	Year	PopMale	PopFemale	PopTotal	PopDensity
1950	Hungary	1950	4494.406	4843.312	9337.718	103.145
1951	Hungary	1951	4573.710	4906.897	9480.607	104.723
1952	Hungary	1952	4637.570	4960.372	9597.942	106.019
1953	Hungary	1953	4687.602	5005.300	9692.902	107.068
1954	Hungary	1954	4725.599	5043.080	9768.679	107.905
...	...	...	...	...	...	...
2015	Hungary	2015	4646.716	5131.209	9777.925	108.008
2016	Hungary	2016	4636.375	5116.595	9752.970	107.732
2017	Hungary	2017	4626.816	5103.006	9729.822	107.476
2018	Hungary	2018	4617.623	5089.879	9707.502	107.230
2019	Hungary	2019	4608.250	5076.430	9684.680	106.978

70 rows × 6 columns

Now a line plot on the total population change of Hungary between 1950 and 2019 can be displayed.

In [27]:

```
hungary['PopTotal'].plot(kind='line')
plt.show()

# same:
#hungary.plot(kind='line', x='Year', y='PopTotal')
#plt.show()
```



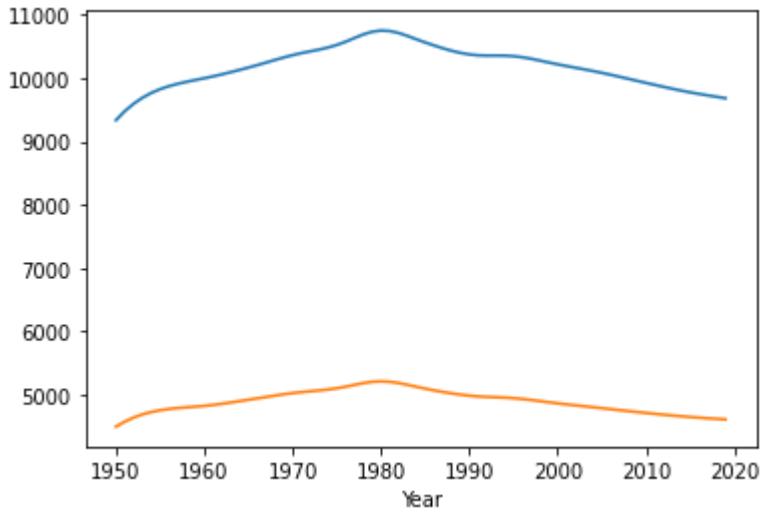
Multiple column diagrams

Let's use multiple columns in the previous line plot, and add the male population to the diagram as a second line.

Multiple plot data can be generated with the `plot()` method of Pandas *Series*. Calling the `show()` function of the `matplotlib.pyplot` module will visualize them on a single diagram.

In [28]:

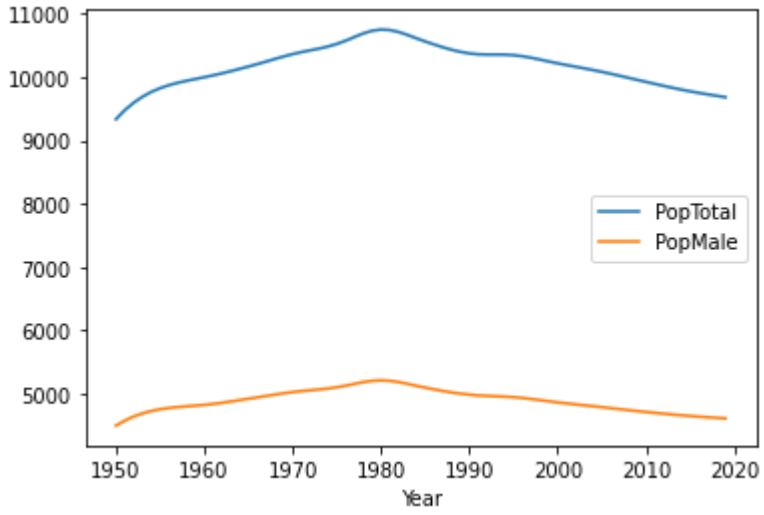
```
hungary['PopTotal'].plot(kind='line')
hungary['PopMale'].plot(kind='line')
plt.show()
```



Add legend to the diagram:

In [29]:

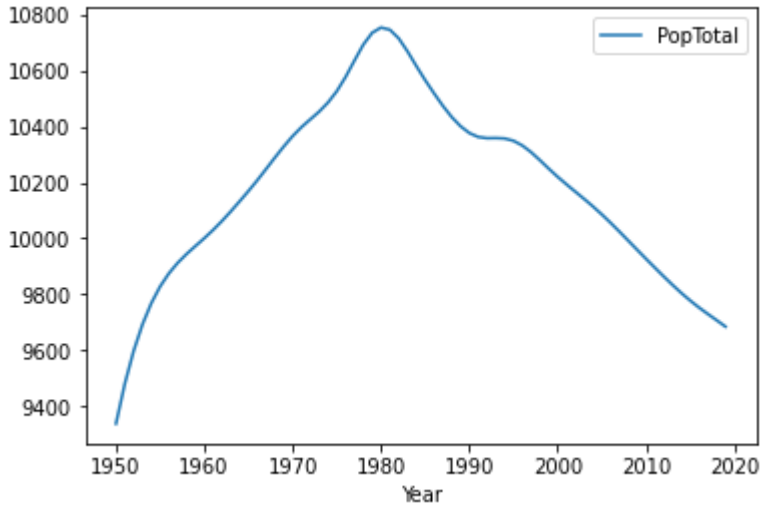
```
hungary['PopTotal'].plot(kind='line', legend=True)
hungary['PopMale'].plot(kind='line', legend=True)
plt.show()
```

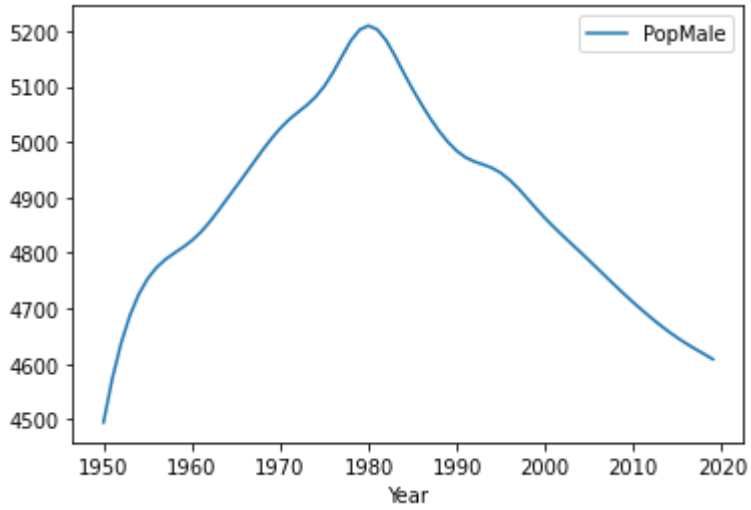


The same can be done by calling the `plot()` method of a *Pandas DataFrame*. Be aware though, that in this case each plot will be displayed in an individual diagram:

In [30]:

```
hungary.plot(kind='line', x='Year', y='PopTotal', legend=True)
hungary.plot(kind='line', x='Year', y='PopMale', legend=True)
plt.show()
```

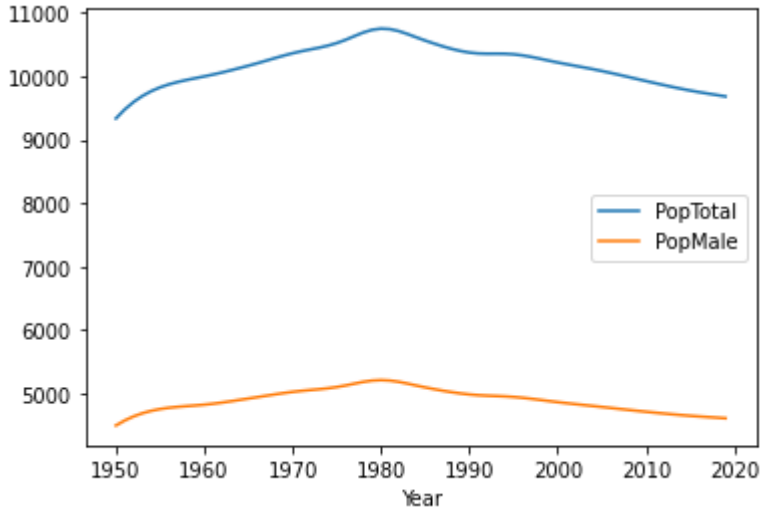




This can be fixed by explicitly configuring matplotlib to use the same *axis object* for visualization for both diagrams:

In [31]:

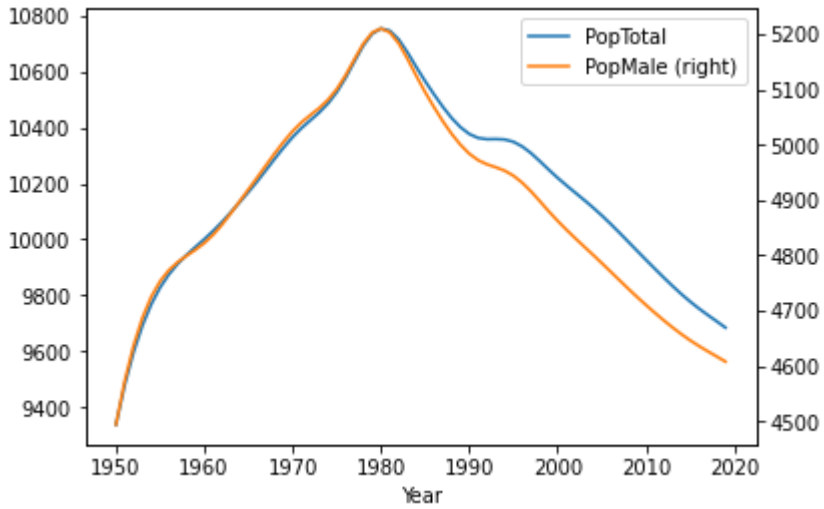
```
ca = plt.gca() # gca = get current axis configuration object
hungary.plot(kind='line', x='Year', y='PopTotal', ax=ca, legend=True) # use the
 ca axis configuration object
hungary.plot(kind='line', x='Year', y='PopMale', ax=ca, legend=True) # use the c
a axis configuration object
plt.show()
```



Use a different, secondary scale for the male population.

In [32]:

```
hungary['PopTotal'].plot(kind='line', legend=True)
hungary['PopMale'].plot(kind='line', secondary_y=True, legend=True)
plt.show()
```



Data grouping

Pandas supports the grouping of data by the given column(s), which then can be used also for visualization.

Select 10 countries by your choice.

In [33]:

```
selected_countries = pd.Series(['Hungary', 'Germany', 'France', 'United Kingdom'
, 'Romania', 'Oman', 'Libya', 'Turkey', 'Chile', 'Viet Nam'])
display(selected_countries)
```

```
0           Hungary
1           Germany
2            France
3    United Kingdom
4           Romania
5              Oman
6             Libya
7            Turkey
8             Chile
9          Viet Nam
dtype: object
```

Select the rows of the original DataFrame for these selected countries.

In [34]:

```
selected_history = population_history[population_history['Country'].isin(selecte
d_countries)]
display(selected_history)
```

	Country	Year	PopMale	PopFemale	PopTotal	PopDensity
3150	Chile	1950	3335.670	3262.848	6598.518	8.875
3151	Chile	1951	3398.318	3331.262	6729.580	9.051
3152	Chile	1952	3465.497	3404.217	6869.714	9.239
3153	Chile	1953	3535.877	3480.588	7016.465	9.437
3154	Chile	1954	3608.433	3559.476	7167.909	9.640
...	...	...	...	...	...	...
16375	Viet Nam	2015	46197.466	46479.616	92677.082	298.891
16376	Viet Nam	2016	46696.272	46944.163	93640.435	301.998
16377	Viet Nam	2017	47193.015	47407.628	94600.643	305.094
16378	Viet Nam	2018	47680.864	47865.095	95545.959	308.143
16379	Viet Nam	2019	48151.352	48310.756	96462.108	311.098

700 rows × 6 columns

The `selected_history` *DataFrame* now contains all historical data for the selected 10 countries.

Visualize the population change of the selected 10 countries for the time period 1950-2019 in a line diagram. To achieve this, we first group the `selected_history` *DataFrame* by the `Country` *Series*:

In [35]:

```
selected_history.groupby('Country')
```

Out[35]:

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f148327a2
80>
```

We have got a `DataFrameGroupBy` object, which can be converted to a list:

In [36]:

```
grouped_history = list(selected_history.groupby('Country'))
print("Length: {0}".format(len(grouped_history)))
```

```
Length: 10
```

Each item of the list contains all records for a given *country* (the column used for groupping):

In [37]:

```
print(grouped_history[0])
```

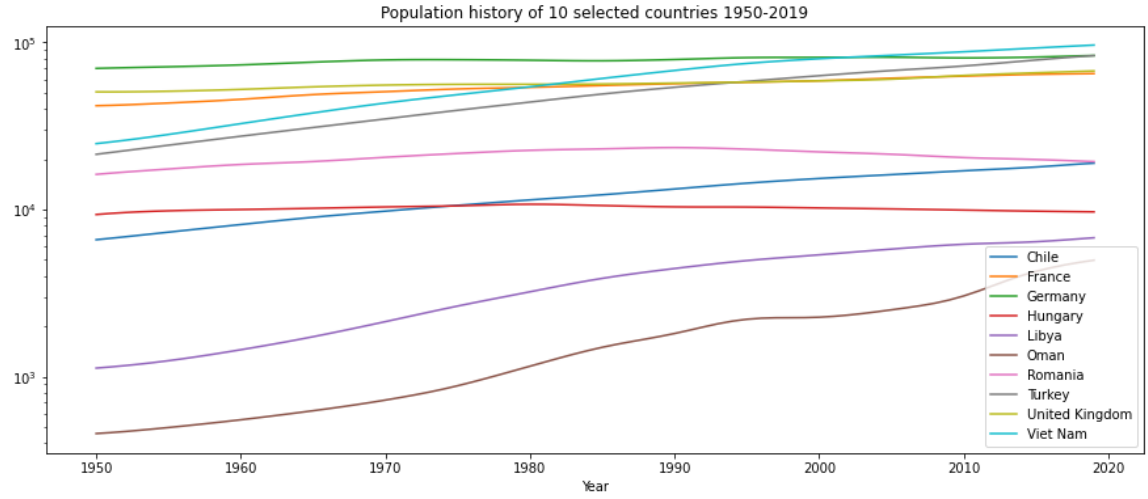
```
('Chile',      Country  Year   PopMale  PopFemale   PopTotal  PopDen
sity
3150   Chile  1950  3335.670   3262.848   6598.518       8.875
3151   Chile  1951  3398.318   3331.262   6729.580       9.051
3152   Chile  1952  3465.497   3404.217   6869.714       9.239
3153   Chile  1953  3535.877   3480.588   7016.465       9.437
3154   Chile  1954  3608.433   3559.476   7167.909       9.640
...      ...   ...       ...        ...        ...         ...
3215   Chile  2015  8844.800   9124.556  17969.356      24.168
3216   Chile  2016  8965.258   9243.814  18209.072      24.490
3217   Chile  2017  9097.252   9373.183  18470.435      24.841
3218   Chile  2018  9228.416   9500.750  18729.166      25.189
3219   Chile  2019  9341.774   9610.261  18952.035      25.489

[70 rows x 6 columns])
```

Question: what happens if we group by the year column?

Based on the grouped *DataFrame*, we select the `PopTotal` *Series* and create a line plot. First the `Year` column is set as an index to be used for the X axis.

In [38]:

```
selected_history.set_index('Year', inplace=True, drop=False)
selected_history.groupby('Country')['PopTotal'].plot(
    kind='line', logy=True,
    figsize=[15, 6], legend=True,
    title='Population history of 10 selected countries 1950-2019')
plt.show()
```



Aggregate functions

Aggregate functions (`min` , `max` , `mean` , `median` , `sum` , etc.) transforms a group of values to a single value. By calling on aggregate function on a grouped *DataFrame*, the aggregated value of each group is calculated.

Let's calculate the largest population for each country they ever had between 1950 and 2019.

In [39]:

```
population_history.groupby('Country').max()
```

Out[39]:

Country	Year	PopMale	PopFemale	PopTotal	PopDensity
Afghanistan	2019	19529.727	18512.030	38041.757	58.269
Albania	2019	1682.757	1611.474	3286.070	119.930
Algeria	2019	21749.666	21303.388	43053.054	18.076
American Samoa	2019	NaN	NaN	59.684	298.420
Andean Community	2019	55331.532	56405.132	111736.664	30.027
...	...	...	...	...	...
Wallis and Futuna Islands	2019	NaN	NaN	15.098	107.843
Western Sahara	2019	304.755	277.703	582.458	2.190
Yemen	2019	14692.284	14469.638	29161.922	55.234
Zambia	2019	8843.214	9017.820	17861.034	24.026
Zimbabwe	2019	6983.353	7662.120	14645.473	37.858

239 rows × 5 columns

Sort the result by the `PopTotal` and only display the `PopTotal`:

In [40]:

```
largest_pop = population_history.groupby('Country').max().sort_values(by = 'PopTotal')['PopTotal']
display(largest_pop)
```

```
Country
Holy See                              0.909
Tokelau                               1.953
Falkland Islands (Malvinas)           3.372
Niue                                  5.242
Saint Pierre and Miquelon             6.435
                                    ...
Pakistan                         216565.317
Indonesia                        270625.567
United States of America         329064.917
India                           1366417.756
China                           1433783.692
Name: PopTotal, Length: 239, dtype: float64
```

Summary exercises on plotting

Exercise 1

Task: Use the *World Countries dataset* defined in the `countries` variable. That dataset contained the *region* for each country. Compute for each region how many countries belong to them. Visualize the results in a pie a chart.

Hint: use groupping.

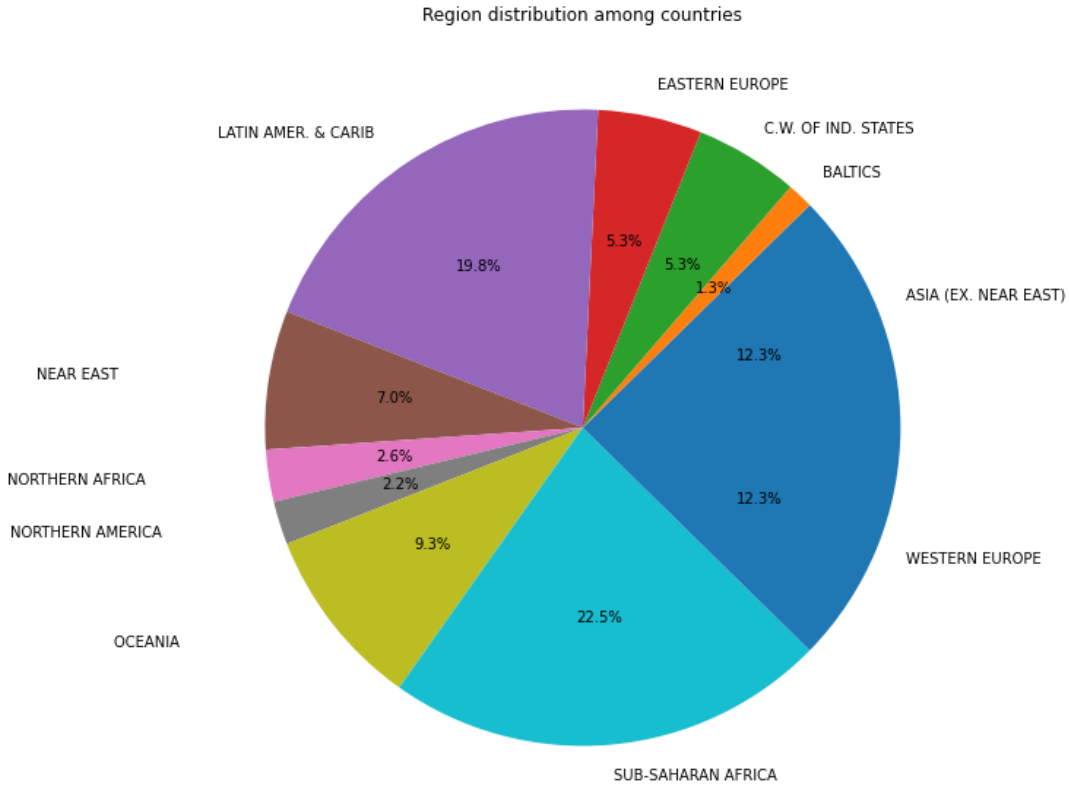
In [41]:

```
countries_by_region = countries.groupby('region').count()['country']
display(countries_by_region)
```

```
region
ASIA (EX. NEAR EAST)         28
BALTICS                       3
C.W. OF IND. STATES          12
EASTERN EUROPE               12
LATIN AMER. & CARIB          45
NEAR EAST                    16
NORTHERN AFRICA               6
NORTHERN AMERICA              5
OCEANIA                      21
SUB-SAHARAN AFRICA           51
WESTERN EUROPE               28
Name: country, dtype: int64
```

In [42]:

```
countries_by_region.plot(kind='pie', figsize=[10,10], autopct='%.1f%%', label=""
,
                         title="Region distribution among countries")
plt.show()
```



Exercise 2

Task: Calculate the accumulated population of the world for each year between 1950 and 2019 based on the *Population History dataset* stored in the `population_history` variable.
Create a bar diagram visualizing how the aggregated population changed over the years.

In [43]:

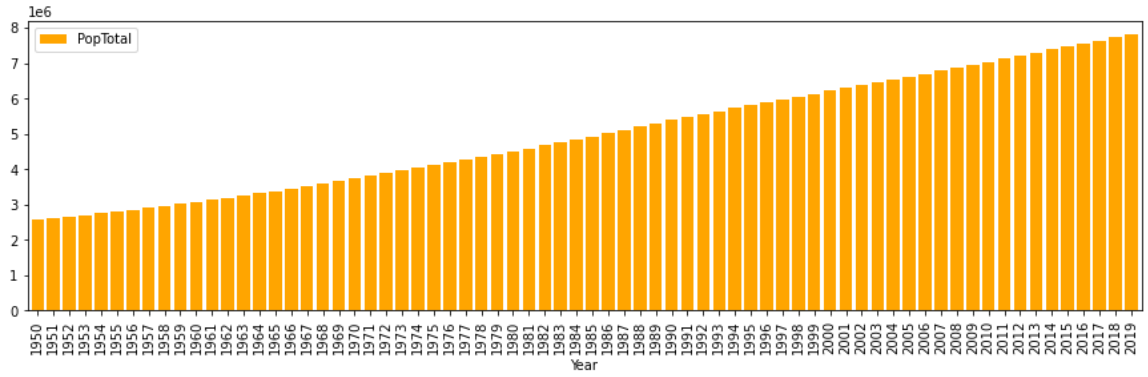
```
aggregated_by_year = population_history.groupby('Year').sum()
display(aggregated_by_year)
```

Year	PopMale	PopFemale	PopTotal	PopDensity
1950	1278875.631	1282748.044	2562089.503	42672.546
1951	1303179.841	1306685.514	2610335.875	42728.190
1952	1327130.022	1330216.610	2657822.039	42972.585
1953	1351073.239	1353698.802	2705252.614	43345.374
1954	1375294.431	1377424.036	2753204.644	43831.847
...	...	...	...	...
2015	3764759.824	3703476.149	7469342.451	106178.776
2016	3807875.525	3745887.267	7554873.938	107422.436
2017	3850938.612	3788132.946	7640187.980	108633.385
2018	3893745.012	3830090.171	7724957.236	109803.197
2019	3936030.563	3871616.311	7808774.650	110916.555

70 rows × 4 columns

In [44]:

```
aggregated_by_year.plot(kind='bar', y='PopTotal', figsize=[15, 4], width=0.8, co
lor='orange')
plt.show()
```



Chapter 11: Spatial data management - vector formats

Shapely

Shapely (https://shapely.readthedocs.io/en/stable/manual.html) is a Python package for manipulation and analysis of planar geometric objects. While Shapely is not concerned with data formats or coordinate systems, it can be readily integrated with packages that are. Indeed, Shapely is a central and essential package to any geometry/geography related work, and many higher abstraction level packages like GeoPandas use Shapely under the hood.

How to install *Shapely*?

We need to install the `shapely` package.

Anaconda - Platform independent

If you have Anaconda installed, open the *Anaconda Prompt* and type in:

```
conda update --all
conda install -c conda-forge shapely
```

Note: updating the currently installed packages to their most recent version can be required to avoid dependency issues. *Note:* we install from the *conda-forge* channel, as it contains more recent versions of these packages compared to the *default* channel of Anaconda.

Python Package Installer (pip) - Linux

If you have standalone Python3 and Jupyter Notebook install on Linux, open a command prompt / terminal and type in:

```
pip3 install shapely
```

Python Package Installer (pip) - Windows

The installation of these packages is much more complicated with *pip* on Windows, because several library binaries must be installed separately or compiled from source. (E.g. the *shapely* package highly depends on the *GEOS* library.) An easier approach is to install these packages from Python binary wheel files (https://www.lfd.uci.edu/~gohlke/pythonlibs/).

Due to its complexity these options are only recommended for advanced Python users; and instead it is **strongly advised to use Anaconda on Windows**.

How to use shapely?

We can either import the complete *shapely module* or just some parts of it which will be used, e.g.:

```
from shapely import geometry
```

Now we can simply refer to e.g. the `shapely.geometry.Point` type simply as `geometry.Point` .

Basic usage of shapely

The fundamental types of geometric objects implemented by Shapely are points, line string, polygons and their collections.

Creation of Shapely objects

Elementary planar geometries can be created from scratch.

Let's define a new point with the coordinate `(5,6)` .

In [1]:

```
from shapely import geometry

point = geometry.Point(5,6)
print(point)
```

```
POINT (5 6)
```

Line strings can be defined through the list of their vertices.

In [2]:

```
line = geometry.LineString([(6,6), (7,7), (8,9)])
print(line)
```

```
LINESTRING (6 6, 7 7, 8 9)
```

Polygons are closed line strings (optionally with holes). The line string is automatically closed. The coordinates can be given with either tuples or lists.

In [3]:

```
rectangle1 = geometry.Polygon([[0,0], [10,0], [10,10], [0,10]])
rectangle2 = geometry.Polygon([(-4,-4), (4,-4), (4,4), (-4,4)])
print(rectangle1)
print(rectangle2)
```

```
POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0))
POLYGON ((-4 -4, 4 -4, 4 4, -4 4, -4 -4))
```

A holed polygon can be defined as an exterior *shell* with a list of inner shells as the holes.

In [4]:

```
rectangle3 = geometry.Polygon([(0,0), (10,0), (10,10), (0,10)],
                              [[(2,2), (2,3), (3,3), (3,2)],
                               [(5,5), (5,7), (7,7), (7,5)]])
print(rectangle3)
```

```
POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0), (2 2, 2 3, 3 3, 3 2, 2 2),
(5 5, 5 7, 7 7, 7 5, 5 5))
```

Manage Shapely objects

For points and line strings the sequence of coordinates can be accessed through the `coords` property, while the separate list of X and Y coordinates can be accessed through the `xy` property of the Shapely objects.

In [5]:

```
print("Coords: {0}".format(list(point.coords)))
x, y = point.xy
print("X coords: {0}".format(x))
print("Y coords: {0}".format(y))
```

```
Coords: [(5.0, 6.0)]
X coords: array('d', [5.0])
Y coords: array('d', [6.0])
```

In [6]:

```
print("Coords: {0}".format(list(line.coords)))
x, y = line.xy
print("X coords: {0}".format(x))
print("Y coords: {0}".format(y))
```

```
Coords: [(6.0, 6.0), (7.0, 7.0), (8.0, 9.0)]
X coords: array('d', [6.0, 7.0, 8.0])
Y coords: array('d', [6.0, 7.0, 9.0])
```

For polygons, the outer shell can be accessed as the `exterior`. *Note: the exterior is also available for points and line strings.*

In [7]:

```
print("Coords: {0}".format(list(rectangle1.exterior.coords)))
x, y = rectangle1.exterior.xy
print("X coords: {0}".format(x))
print("Y coords: {0}".format(y))
```

```
Coords: [(0.0, 0.0), (10.0, 0.0), (10.0, 10.0), (0.0, 10.0), (0.0,
0.0)]
X coords: array('d', [0.0, 10.0, 10.0, 0.0, 0.0])
Y coords: array('d', [0.0, 0.0, 10.0, 10.0, 0.0])
```

The holes of a polygon can be accessed through the `interiors` list of the object.

In [8]:

```
print("Coords: {0}".format(list(rectangle3.exterior.coords)))
x, y = rectangle3.exterior.xy
print("X coords: {0}".format(x))
print("Y coords: {0}".format(y))

print("Holes:")
for hole in rectangle2.interiors:
    print(hole)
```

```
Coords: [(0.0, 0.0), (10.0, 0.0), (10.0, 10.0), (0.0, 10.0), (0.0,
0.0)]
X coords: array('d', [0.0, 10.0, 10.0, 0.0, 0.0])
Y coords: array('d', [0.0, 0.0, 10.0, 10.0, 0.0])
Holes:
```

Various geometric properties can be easily computed through Shapely:

In [9]:

```
print('Area of Rectangle1: {0:.2f}'.format(rectangle1.area))
print('Area of Rectangle2: {0:.2f}'.format(rectangle2.area))
print('Area of Rectangle2: {0:.2f}'.format(rectangle3.area))
print('Length of Line: {0:.2f}'.format(line.length))
```

```
Area of Rectangle1: 100.00
Area of Rectangle2: 64.00
Area of Rectangle2: 95.00
Length of Line: 3.65
```

In [10]:

```
print(point.distance(rectangle2))
print(line.distance(rectangle2))
```

```
2.23606797749979
2.8284271247461903
```

In [11]:

```
print('Rectangle1 contains Point: {0}'.format(rectangle1.contains(point)))
print('Rectangle2 contains Point: {0}'.format(rectangle2.contains(point)))
print('Rectangle1 contains Rectangle2: {0}'.format(rectangle1.contains(rectangle
2)))
print('Rectangle1 intersects Rectangle2: {0}'.format(rectangle1.intersects(recta
ngle2)))
```

```
Rectangle1 contains Point: True
Rectangle2 contains Point: False
Rectangle1 contains Rectangle2: False
Rectangle1 intersects Rectangle2: True
```

Read WKT strings into Shapely objects

Geometries can also be loaded using the *well-known text* (WKT) (https://en.wikipedia.org/wiki/Well-known_text_representation_of_geometry) format.

Well-known text (WKT) is a text markup language for representing vector geometry objects. It is a human-readable, but verbose format. A binary equivalent, known as well-known binary (WKB) is used to transfer and store the same information in a more compact form convenient for computer processing but that is not human-readable.

WKT and WKB are understood by many applications and software libraries, including Shapely.

In [12]:

```
from shapely import wkt

rectangle2 = wkt.loads('POLYGON ((-4 -4, 4 -4, 4 4, -4 4, -4 -4))')
print(rectangle2)
```

```
POLYGON ((-4 -4, 4 -4, 4 4, -4 4, -4 -4))
```

Note: Shapely also displays geometries as a WKT as the default string representation.

GeoPandas

GeoPandas (https://geopandas.org/) is an open source project to make working with geospatial data in Python easier. GeoPandas extends the datatypes used by *pandas* to allow spatial operations on geometric types.

Install *geopandas*

The following Python packages are required to be installed:

- geopandas
- descartes (for visualization)
- mapclassify (for classification of data)
- rtree (spatial indexing of data)

Anaconda - Platform independent

If you have Anaconda installed, open the *Anaconda Prompt* and type in:

```
conda update --all
conda install -c conda-forge geopandas descartes mapclassify rtree
```

Note: updating the currently installed packages to their most recent version can be required to avoid dependency issues.
Note: we install from the *conda-forge* channel, as it contains more recent versions of these packages compared to the *default* channel of Anaconda.

Python Package Installer (pip) - Linux

If you have standalone Python3 and Jupyter Notebook install on Linux, open a command prompt / terminal and type in:

```
pip3 install geopandas descartes mapclassify rtree
```

For the *rtree* Python package you must also install the *libspatialindex-dev* system package, which will require administrative priviliges:

```
sudo apt-get install libspatialindex-dev
```

Python Package Installer (pip) - Windows

The installation of these packages is much more complicated with *pip* on Windows, because several library binaries must be installed separately or compiled from source. (E.g. the *geopandas* package highly depends on the *GDAL* library.)
An easier approach is to install these packages from Python binary wheel files (https://www.lfd.uci.edu/~gohlke/pythonlibs/).

Due to its complexity these options are only recommended for advanced Python users and it is **strongly advised to use Anaconda on Windows**.

How to use *geopandas*?

The geopandas package is also a module which you can simply import. It is usually aliased with the `gpd` abbreviation.

```
import geopandas as gpd
```

Read spatial data

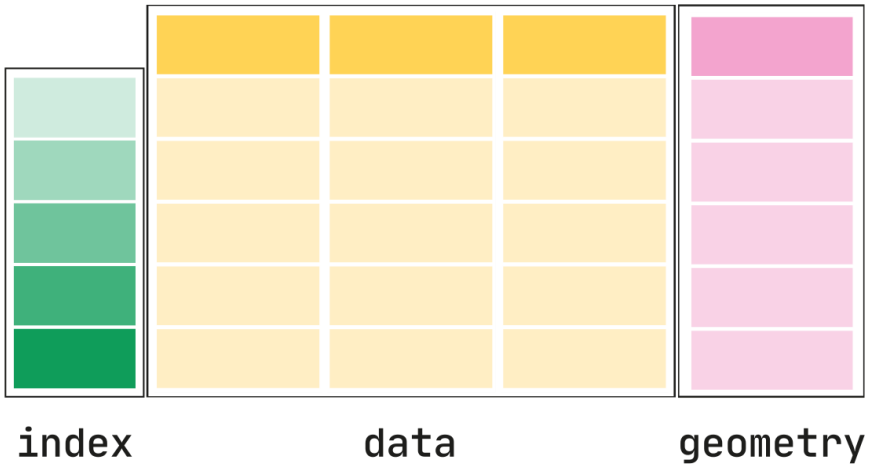
Geopandas can read many vector-based spatial data format including Shapefiles, GeoJSON files and much more. Only the `read_file()` function has to be called. The result is a geopandas dataframe, a *GeoDataFrame*.

Read the `data/ne_10m_admin_0_countries.shp` shapefile located in the `data` folder. This dataset contains both scalar and spatial data of the countries all over the world.
Source: Natural Earth (https://www.naturalearthdata.com/downloads/10m-cultural-vectors/)

In [13]:

```
import geopandas as gpd
import matplotlib.pyplot as plt
%matplotlib inline

countries_gdf = gpd.read_file('../data/ne_10m_admin_0_countries.shp')
display(countries_gdf)
```

GeoPandas uses Shapely to represent geometries. Observe the `geometry` column (the last one), which contains the Shapely geometry objects of the row, displayed as a string (in WKT format) in the table.



Basic usage of *GeoDataFrames*

Since this *GeoDataFrame* has quite a number of columns, some of them are hidden by the display. Let's list all the columns:

In [14]:

```
print(countries_gdf.columns)
```

```
Index(['featurecla', 'scalerank', 'LABELRANK', 'SOVEREIGNT', 'SOV_A
3',
       'ADM0_DIF', 'LEVEL', 'TYPE', 'ADMIN', 'ADM0_A3', 'GEOU_DIF',
'GEOUNIT',
       'GU_A3', 'SU_DIF', 'SUBUNIT', 'SU_A3', 'BRK_DIFF', 'NAME', 'N
AME_LONG',
       'BRK_A3', 'BRK_NAME', 'BRK_GROUP', 'ABBREV', 'POSTAL', 'FORMA
L_EN',
       'FORMAL_FR', 'NAME_CIAWF', 'NOTE_ADM0', 'NOTE_BRK', 'NAME_SOR
T',
       'NAME_ALT', 'MAPCOLOR7', 'MAPCOLOR8', 'MAPCOLOR9', 'MAPCOLOR1
3',
       'POP_EST', 'POP_RANK', 'GDP_MD_EST', 'POP_YEAR', 'LASTCENSU
S',
       'GDP_YEAR', 'ECONOMY', 'INCOME_GRP', 'WIKIPEDIA', 'FIPS_10_',
'ISO_A2',
       'ISO_A3', 'ISO_A3_EH', 'ISO_N3', 'UN_A3', 'WB_A2', 'WB_A3',
'WOE_ID',
       'WOE_ID_EH', 'WOE_NOTE', 'ADM0_A3_IS', 'ADM0_A3_US', 'ADM0_A3
_UN',
       'ADM0_A3_WB', 'CONTINENT', 'REGION_UN', 'SUBREGION', 'REGION_
WB',
       'NAME_LEN', 'LONG_LEN', 'ABBREV_LEN', 'TINY', 'HOMEPART', 'MI
N_ZOOM',
       'MIN_LABEL', 'MAX_LABEL', 'NE_ID', 'WIKIDATAID', 'NAME_AR',
'NAME_BN',
       'NAME_DE', 'NAME_EN', 'NAME_ES', 'NAME_FR', 'NAME_EL', 'NAME_
HI',
       'NAME_HU', 'NAME_ID', 'NAME_IT', 'NAME_JA', 'NAME_KO', 'NAME_
NL',
       'NAME_PL', 'NAME_PT', 'NAME_RU', 'NAME_SV', 'NAME_TR', 'NAME_
VI',
       'NAME_ZH', 'geometry'],
      dtype='object')
```

With a lot of columns it can be useful to select only a few columns to make the displayed results more human-readable. This can be done by in a similar way when selecting a single *Series* from a *DataFrame*, but now we shall define a list of *Series* to select.
Remark: this makes a copy of the dataframe.

In [15]:

```
countries_gdf = countries_gdf[['NAME', 'POP_EST', 'POP_YEAR', 'GDP_MD_EST', 'GDP
_YEAR', 'REGION_UN', 'geometry']]
display(countries_gdf)
```

	NAME	POP_EST	POP_YEAR	GDP_MD_EST	GDP_YEAR	REGION_UN	ge
0	Indonesia	260580739	2017	3028000.0	2016	Asia	MULTIPO (((11 4 117.7(
1	Malaysia	31381992	2017	863000.0	2016	Asia	MULTIPO (((11 4 117.6(
2	Chile	17789267	2017	436100.0	2016	Americas	MULTIPO (((-6(-17 -69.(
3	Bolivia	11138234	2017	78350.0	2016	Americas	PO ((-6(-17 -6(
4	Peru	31036656	2017	410400.0	2016	Americas	MULTIPO (((-6(-17 -69.(
...	...	...	...	...	...	...	
250	Macao	601969	2017	63220.0	2016	Asia	MULTIPO (((11(22 113.5
251	Ashmore and Cartier Is.	0	2017	0.0	2016	Oceania	PO ((12(-12 12(
252	Bajo Nuevo Bank	0	2017	0.0	2016	Americas	PO ((-7(15 -7(1
253	Serranilla Bank	0	0	0.0	0	Americas	PO ((-7(15 -7(1
254	Scarborough Reef	0	2012	0.0	2016	Asia	PO ((11 15 11 1

255 rows × 7 columns

Geopandas extends the capabilties of the pandas library, which means we can use all what we have learned with pandas.

Let's sort the *GeoDataFrame* by the name of the countries:

In [16]:

```
display(countries_gdf.sort_values(by='NAME'))
```

Filter the dataframe to contain only the European countries:

In [17]:

```
condition = countries_gdf['REGION_UN'] == 'Europe'
europe_gdf = countries_gdf[condition]
display(europe_gdf)
```

Sort the European countries by their population in a descending order:

In [18]:

```
display(europe_gdf.sort_values(by = 'POP_EST', ascending = False))
```

Spatial data management in *GeoDataFrames*

We can fetch the CRS (*coordinate reference system*) of the `geometry` column in the *GeoDataFrame*:

In [19]:

```
print(countries_gdf.crs)
```

```
epsg:4326
```

In [20]:

```
display(countries_gdf.crs)
```

```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

As we can observe the spatial data are in *WGS 84 (EPSG:4326)*. Since that is a geographic CRS, it would be unsuitable to calculate the area of the countries.

The geometries can be transformed on-the-fly to a different CRS with GeoPandas. Let's select a projected CRS, *Mercator (EPSG:3857)*.

In [21]:

```
countries_mercator = countries_gdf.to_crs('epsg:3857')
```

Now the area of each geometry can be calculated in km^2 units:

In [22]:

```
countries_mercator['AREA'] = countries_mercator.area / 10**6
display(countries_mercator)
```

Use the `round()` function to limit the number decimal digits, hence we can get rid of the scientific notation:

In [23]:

```
countries_mercator['AREA'] = countries_mercator['AREA'].round(2)
display(countries_mercator)
```

Since the Mercator projection applies is *azimuthal* (meaning the angles are correct), but not *equal-area*, areas inflate with distance from the equator such that the polar regions are grossly exaggerated. Therefore there are great territorial distortion int calculated values, e.g. for Hungary the area is more than twice of the real value.

In [24]:

```
display(countries_mercator[countries_mercator['NAME'] == 'Hungary'])
```

	NAME	POP_EST	POP_YEAR	GDP_MD_EST	GDP_YEAR	REGION_UN	geometı
75	Hungary	9850845	2017	267600.0	2016	Europe	POLYGO ((2546722.84 6097998.60 2544893.518

Let's use the *Mollweide (ESRI:54009)* equal-area projection instead to calculate the proper area of the countries.

In [25]:

```
countries_mollweide = countries_gdf.to_crs('esri:54009')
countries_mollweide['AREA'] = countries_mollweide.area / 10**6
countries_mollweide['AREA'] = countries_mollweide['AREA'].round(2)
display(countries_mollweide[countries_mollweide['NAME'] == 'Hungary'])
```

	NAME	POP_EST	POP_YEAR	GDP_MD_EST	GDP_YEAR	REGION_UN	geometı
75	Hungary	9850845	2017	267600.0	2016	Europe	POLYGO ((1785929.91 5656787.62 1784927.965

Remark: EPSG (*European Petroleum Survey Group*) and ESRI (American company *Environmental Systems Research Institute*) are two authorities providing well-known identifiers (WKID) for CRS.
However these numbers don't overlap for avoiding confusion.

Task: When working with local spatial data for Hungary often the *Uniform National Projection* named *EOV* (abbreviation of *Egységes Országos Vetület*) is utilizied. It is an *azimuthal* projected CRS, and while not *equal-area*, only applies a minimal distortion on the region of Hungary.
Calculate the area of Hungary in EOV!

In [26]:

```
countries_eov = countries_gdf.to_crs('EPSG:23700') # EOV is EPSG:23700
countries_eov.set_index('NAME', drop=False, inplace=True)
countries_eov['AREA'] = countries_eov.area / 10**6
display(countries_eov.loc['Hungary'])
```

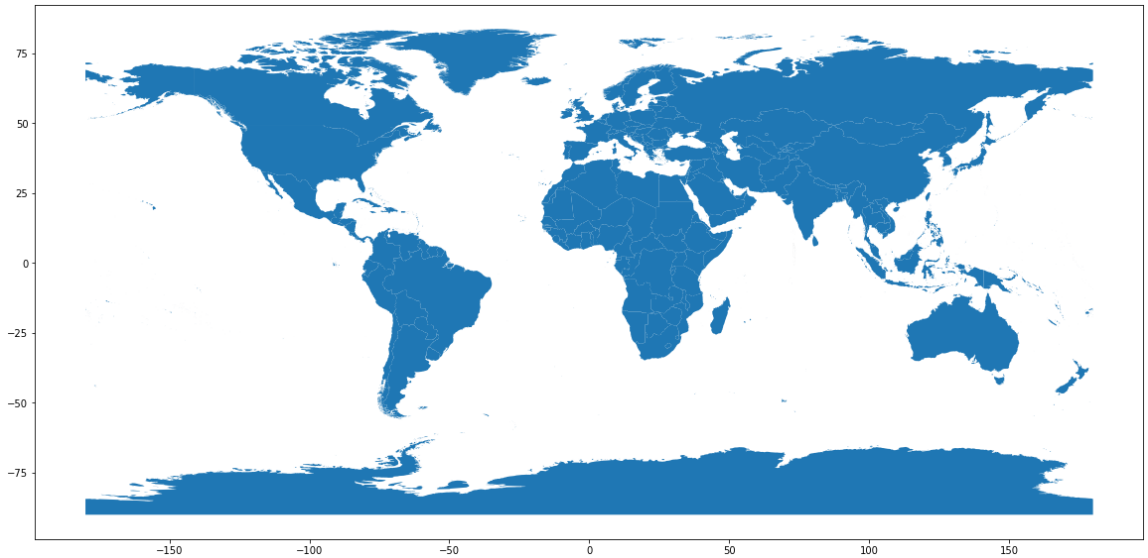
```
NAME                                                     Hungary
POP_EST                                                  9850845
POP_YEAR                                                    2017
GDP_MD_EST                                              267600.0
GDP_YEAR                                                    2016
REGION_UN                                                 Europe
geometry      POLYGON ((936017.8110286188 296237.7384604304,...
AREA                                                93200.728333
Name: Hungary, dtype: object
```

Map making

Geopandas provides a high-level interface to the *matplotlib* library for making maps. Mapping shapes is as easy as using the `plot()` method on a *GeoDataFrame* (or *GeoSeries*).

In [27]:

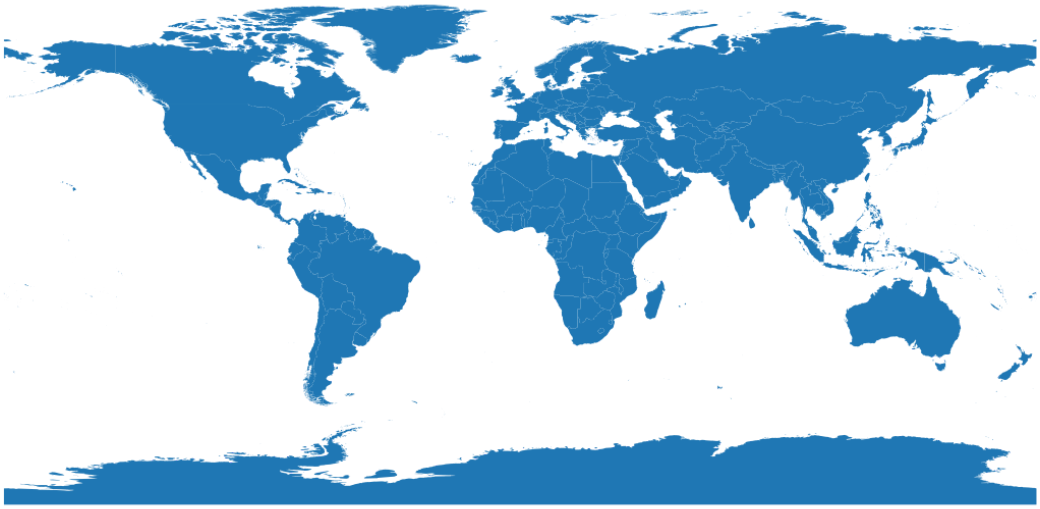
```
countries_gdf.plot(figsize=[20,10])
plt.show()
```



The `plot()` function call on a *GeoDataFrame* (or a regular pandas *DataFrame*) will return an axis configuration object, which we can use to further customize our plot (map in this case). E.g. we can hide the axes with the `set_axis_off()` function:

In [28]:

```
ax = countries_gdf.plot(figsize=[20,10])
ax.set_axis_off()
plt.show()
```

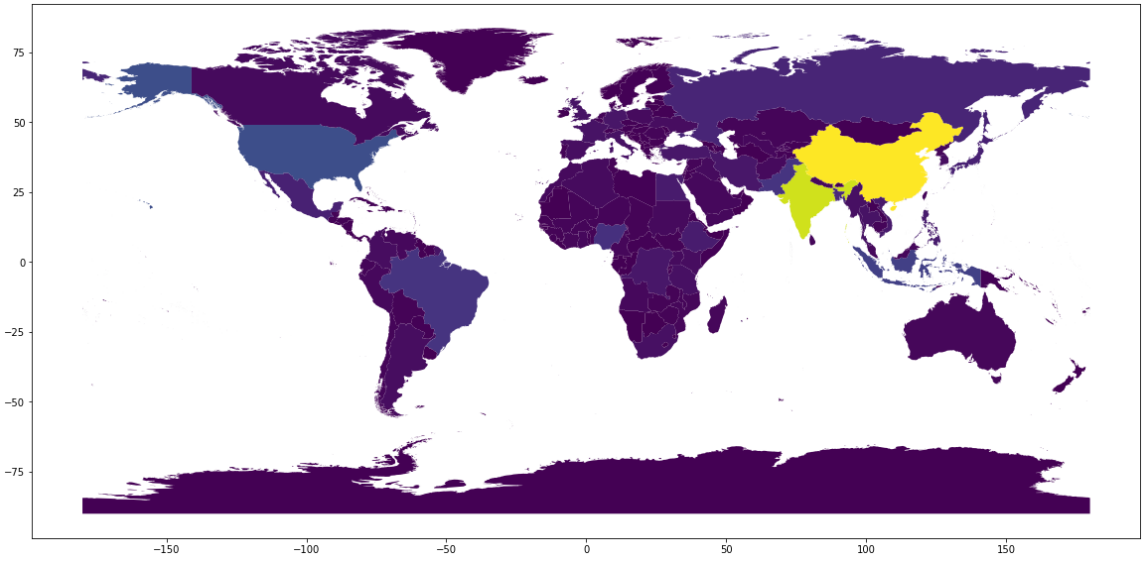


Choropleth maps

Geopandas makes it easy to create so called *choropleth maps* (maps where the color of each shape is based on the value of an associated variable). Simply use the `plot()` method with the `column` argument set to the column whose values you want used to assign colors.

In [29]:

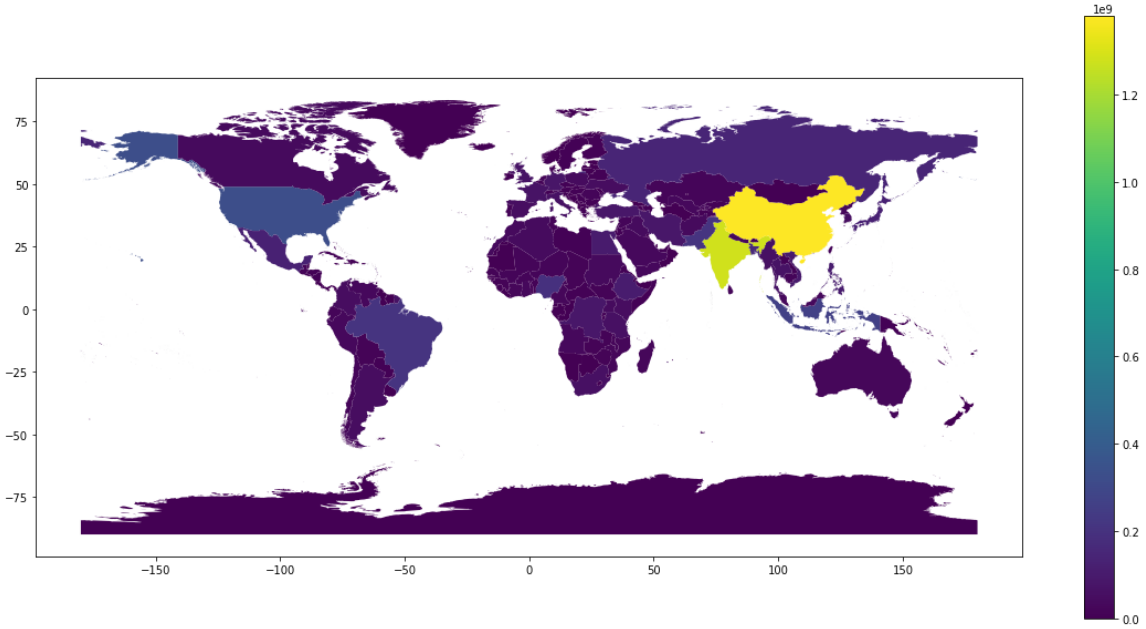
```
countries_gdf.plot(column='POP_EST', figsize=[20,10])
plt.show()
```



Add a legend to the map.

In [30]:

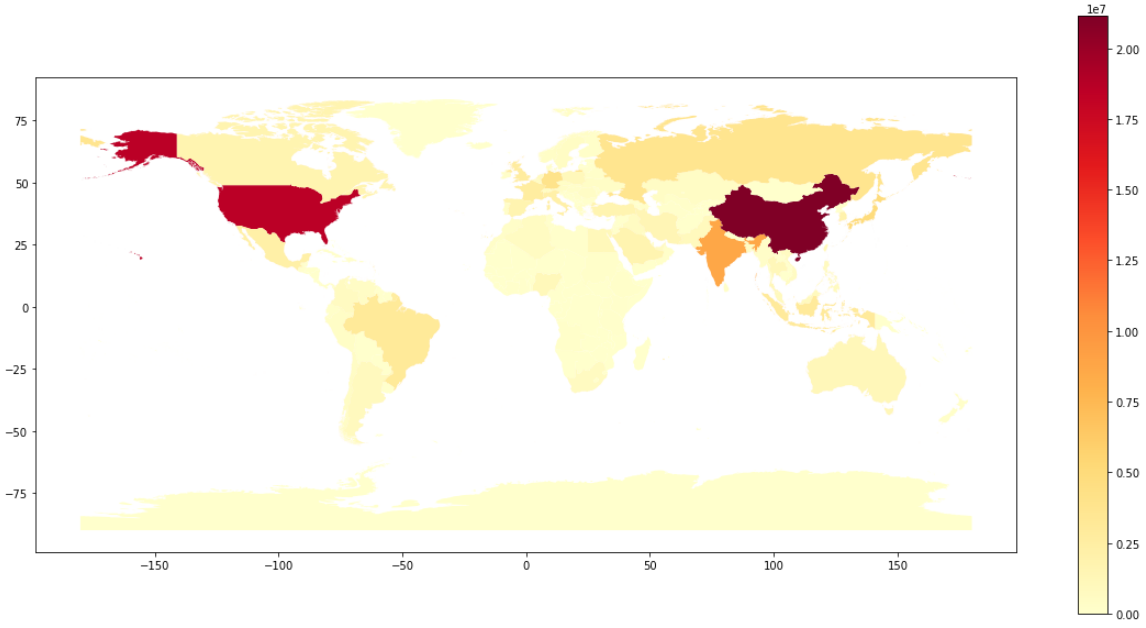
```
countries_gdf.plot(column='POP_EST', legend=True, figsize=[20,10])
plt.show()
```



We can choose from various available color maps. A complete list can be found on the [matplotlib website (https://matplotlib.org/tutorials/colors/colormaps.html)](https://matplotlib.org/tutorials/colors/colormaps.html).

In [31]:

```
countries_gdf.plot(column='GDP_MD_EST', legend=True, cmap='YlOrRd', figsize=[20,
10])
plt.show()
```

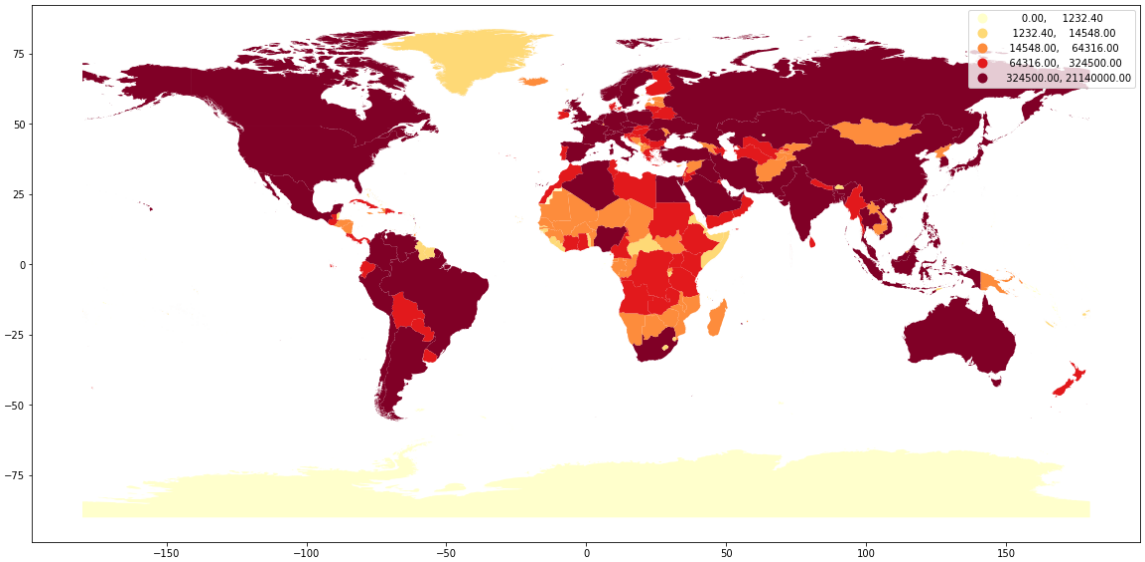


The way color maps are scaled can also be manipulated with the `scheme` option (the *mapclassify* Python library must be installed).

A full list of schemes are available on the project's GitHub page (https://github.com/pysal/mapclassify) and some examples of result on the package's website (https://pysal.org/mapclassify/index.html).

In [32]:

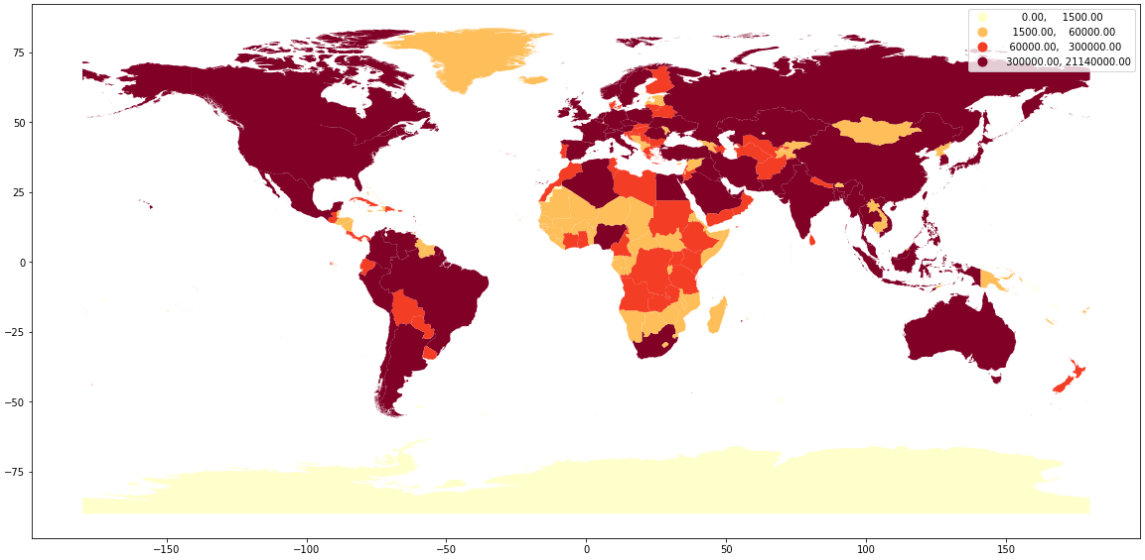
```
countries_gdf.plot(column='GDP_MD_EST', legend=True, cmap='YlOrRd', figsize=[20,
10], scheme='quantiles')
plt.show()
```



With the `user_defined` scheme, a custom classification can be defined.

In [33]:

```
countries_gdf.plot(column='GDP_MD_EST', legend=True, cmap='YlOrRd', figsize=[20,
10],
                   scheme='user_defined', classification_kwds={'bins':[1500, 600
00, 300000]})
plt.show()
```



Multiple layers

We can easily combine the data of multiple *GeoDataFrames* and even visualize them as multiple layers with geopandas.

Open and read a second data source defined in the `data/World_Cities.shp` shapefile, containing scalar and spatial data about major cities all around the world.
Source: ArcGIS (https://hub.arcgis.com/datasets/6996f03a1b364dbab4008d99380370ed_0)

In [34]:

```
cities_gdf = gpd.read_file('../data/World_Cities.shp')
display(cities_gdf)
```

	FID	ObjectID	CITY_NAME	GMI_ADMIN	ADMIN_NAME	FIPS_CNTRY	CNTRY_NAM
0	1001	1500	Koszalin	POL-KSZ	Koszalin	PL	Polan
1	1002	1200	Erzurum	TUR-ERR	Erzurum	TU	Turke
2	1003	1000	Jendouba	TUN-JND	Jundubah	TS	Tunis
3	1004	1501	Szczecin	POL-SZC	Szczecin	PL	Polan
4	1005	1600	Rimnicu Vilcea	ROM-VIL	Vilcea	RO	Roman
...	...	...	...	...	...	...	
2535	996	395	St. Anns Bay	JAM-SAN	Saint Ann	JM	Jamaic
2536	997	396	Port Maria	JAM-SMA	Saint Mary	JM	Jamaic
2537	998	397	Port Antonio	JAM-PRT	Portland	JM	Jamaic
2538	999	398	Spanish Town	JAM-SCT	Saint Catherine	JM	Jamaic
2539	1000	399	May Pen	JAM-CLR	Clarendon	JM	Jamaic

2540 rows × 14 columns

Reduce the number of columns, by selecting only the most important ones:

In [35]:

```
cities_gdf = cities_gdf[['CITY_NAME', 'CNTRY_NAME', 'STATUS', 'POP', 'geometry'
]]
display(cities_gdf)
```

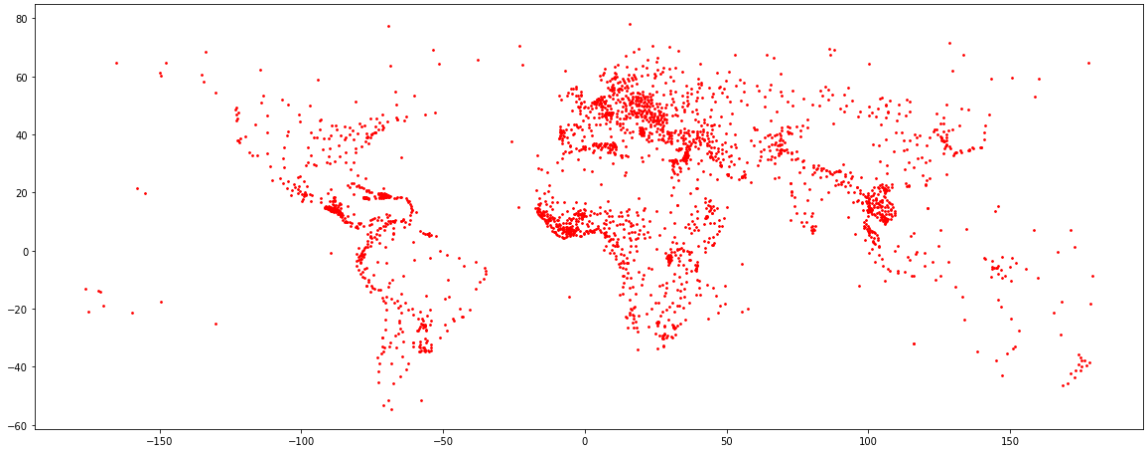
	CITY_NAME	CNTRY_NAME	STATUS	POP	geometry
0	Koszalin	Poland	Provincial capital	107450	POINT (16.18500 54.18600)
1	Erzurum	Turkey	Provincial capital	420691	POINT (41.29200 39.90400)
2	Jendouba	Tunisia	Provincial capital	51408	POINT (8.75000 36.50000)
3	Szczecin	Poland	Provincial capital	407811	POINT (14.53100 53.43800)
4	Rimnicu Vilcea	Romania	Provincial capital	107558	POINT (24.38300 45.11000)
...	...	...	...	...	...
2535	St. Anns Bay	Jamaica	Provincial capital	-999	POINT (-77.19952 18.43264)
2536	Port Maria	Jamaica	Provincial capital	7906	POINT (-76.90000 18.37700)
2537	Port Antonio	Jamaica	Provincial capital	-999	POINT (-76.38000 18.15900)
2538	Spanish Town	Jamaica	Provincial capital	145018	POINT (-76.95200 17.99500)
2539	May Pen	Jamaica	Provincial capital	44755	POINT (-77.24300 17.96900)

2540 rows × 5 columns

Plot the cities:

In [36]:

```
cities_gdf.plot(color='red', markersize=3, figsize=[20,10])
plt.show()
```



Verify whether both datasets use the same coordinate reference system:

In [37]:

```
print(cities_gdf.crs)
print(countries_gdf.crs)
```

```
epsg:4326
epsg:4326
```

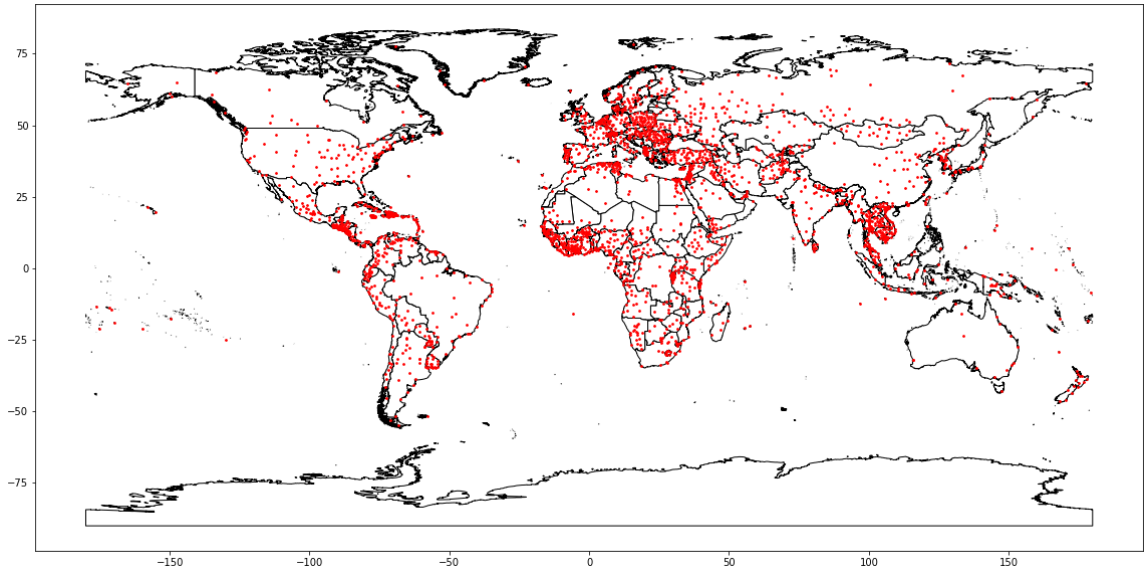
Would be they different, geopandas would also be capable to transform one of the dataframes to the other CRS:

```
cities_gdf = cities_gdf.to_crs(countries_gdf.crs)
```

Create a combined visualization of multiple layers, by simply calling the `plot()` method on all *GeoDataFrames*, but drawing them on the same axis object.

In [38]:

```
base = countries_gdf.plot(color='white', edgecolor='black', figsize=[20, 10])
cities_gdf.plot(ax=base, color='red', markersize=3)
plt.show()
```



Basemaps *(optional)*

Contextily (https://contextily.readthedocs.io/en/latest/) is a Python package to retrieve tile maps from the internet. It can add those tiles as basemap to matplotlib figures.

How to install *contextily*?

```
conda install -c conda-forge contextily
```

How to install *contextily*?

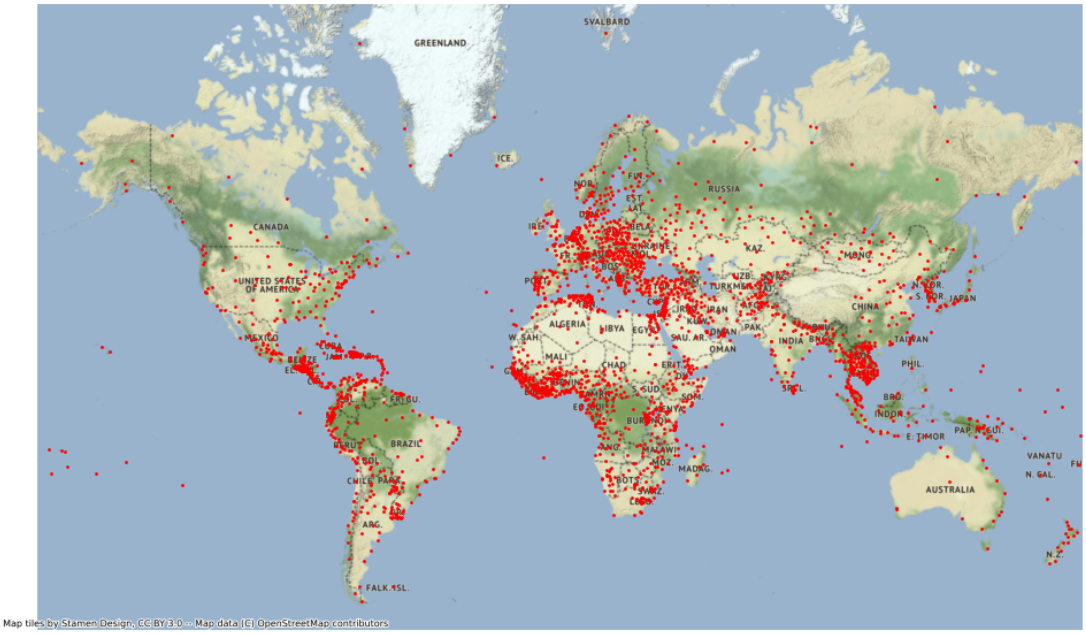
The contextily package is also a module which you can simply import. It is usually aliased with the `ctx` abbreviation.

```
import contextily as ctx
```

The basemap tiles are in the Web Mercator (EPSG:3857) projection. To use them, we must convert our dataset to this CRS first.

In [39]:

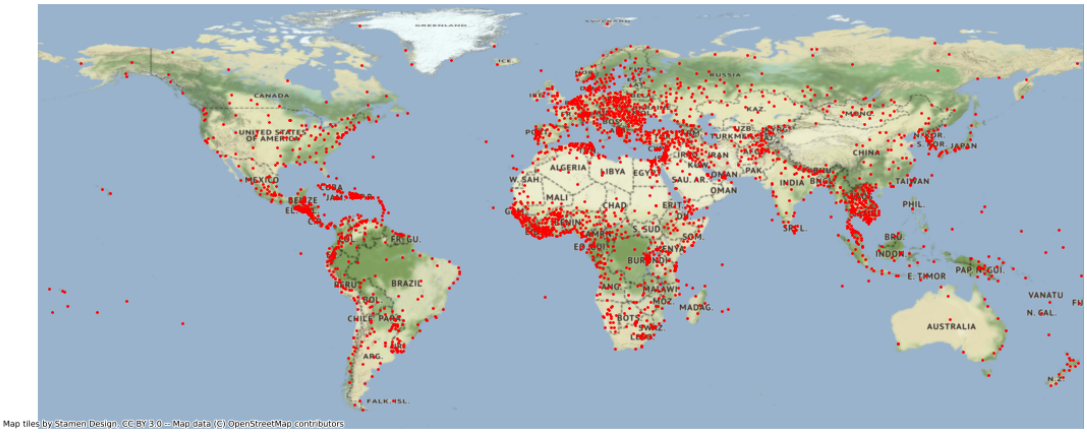
```
import contextily as ctx

# Convert dataset to Web Mercator (EPSG:3857)
cities_mercator = cities_gdf.to_crs('epsg:3857')

ax = cities_mercator.plot(figsize=[20, 10], color='red', markersize=3)
ctx.add_basemap(ax)
ax.set_axis_off()
plt.show()
```



The same journey can be travelled in the opposite direction by leaving your data untouched and warping the tiles coming from the web.

In [40]:

```
import contextily as ctx

ax = cities_gdf.plot(figsize=[20, 10], color='red', markersize=3)
ctx.add_basemap(ax, crs=cities_gdf.crs)
ax.set_axis_off()
plt.show()
```



Note: it is also possible to convert both dataset and the basemap tiles into a different, third CRS.

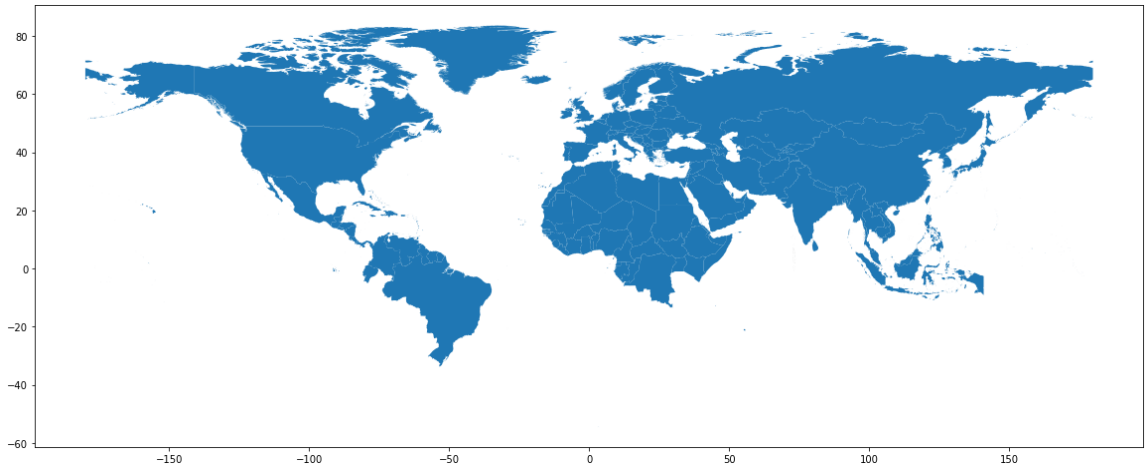
Clipping operation

Geopandas offers a coordinate indexer (`cx`), which can be used to select only the records which geometry overlaps with the selected region.

Let's select and plot the countries in the northern hemisphere.

In [41]:

```
northern_gdf = countries_gdf.cx[:, 0:]
northern_gdf.plot(figsize=[20, 10])
plt.show()
```



Note: with this approach countries overlapping both the northern and southern hemispheres are not clipped.

We can perform real clipping with the `clip()` function of geopandas. As a showcase let's clip the countries and country parts inside the bounding box of Europe; defined with the following polygon (given in WKT format):
`POLYGON ((-10 35, 40 35, 40 70, -10, 70, -10, 35))`.

Geopandas uses the Shapely library in the background to represent and manipulate vector data. Therefore, first we define a regular pandas *DataFrame* named `europe_df` , where the *Coordinates* column will contain a polygon defined with *Shapely*.

In [42]:

```
import pandas as pd
from shapely.geometry import Polygon

europe_df = pd.DataFrame({
    'Name': ['Europe'],
    'Coordinates': [Polygon([(-10, 35), (40, 35), (40, 70), (-10, 70), (-10, 35
)])]
    # the polygon is defined as a closed line
})
display(europe_df)
```

	Name	Coordinates
0	Europe	POLYGON ((-10 35, 40 35, 40 70, -10 70, -10 35))

Now our *GeoDataFrame* can be constructed from the *DataFrame* stored in `europe_df` , by defining which *Series* (column) contains the geometries and the CRS. (Use the CRS of the countries dataset.)

In [43]:

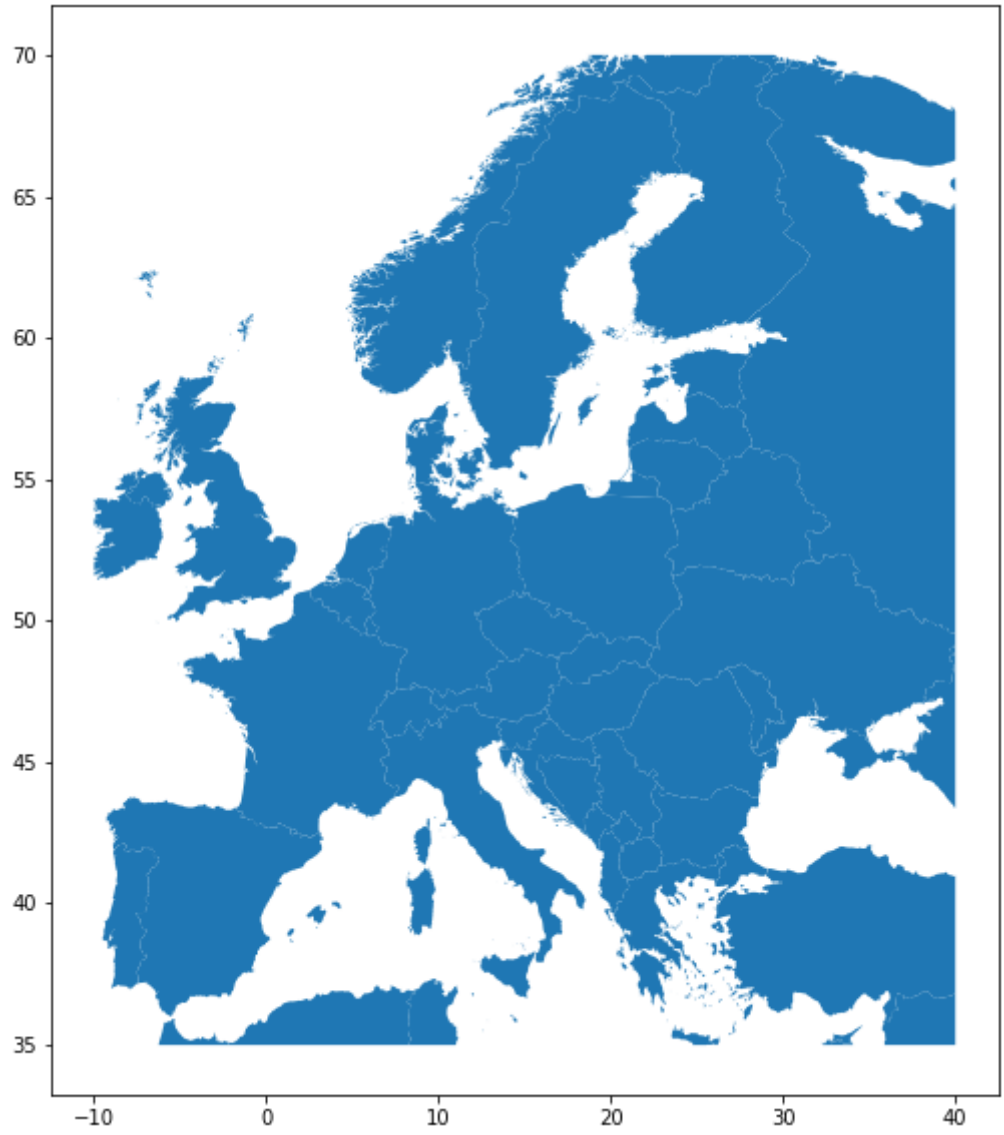
```
europe_gdf = gpd.GeoDataFrame(europe_df, geometry='Coordinates', crs=countries_g
df.crs)
display(europe_gdf)
```

	Name	Coordinates
0	Europe	POLYGON ((-10.00000 35.00000, 40.00000 35.0000...

Finally, we can perform the clipping operation between the *GeoDataFrames*:

In [44]:

```
clipped_gdf = gpd.clip(countries_gdf, europe_gdf)
clipped_gdf.plot(figsize=[10, 10])
plt.show()
```



Attribute join

In an attribute join, a *GeoDataFrame* (or a *GeoSeries*) is combined with a regular pandas *DataFrame* or *Series* based on a common variable. (This is analogous to normal merging or joining in pandas.)

Let's read the *European countries datatset* from the `data/countries_europe.csv` file, which we used in Chapter 9 (09_tabular.pdf). The dataset contains the country name, capital city name, area (in km^2), population (in millions) and the region data for 43 European countries respectively.
Data source: EuroStat (https://ec.europa.eu/eurostat/)

In [45]:

```
countries_europe = pd.read_csv('../data/countries_europe.csv', delimiter = ';')
display(countries_europe)
```

	Country	Capital	Area (km2)	Population (millions)	Region
0	Albania	Tirana	28748	3.20	Southern
1	Andorra	Andorra la Vella	468	0.07	Western
2	Austria	Vienna	83857	7.60	Western
3	Belgium	Brussels	30519	10.00	Western
4	Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern
...	...	...	...	...	...
38	Sweden	Stockholm	449964	8.50	Northern
39	Serbia	Belgrade	66577	7.20	Southern
40	Slovakia	Bratislava	49035	5.30	Central
41	Slovenia	Ljubljana	20250	2.00	Southern
42	Ukraine	Kiev	603700	51.80	Eastern

The *attribute join* can be performed with the `merge()` method, defining the columns used for merging. (Or alternatively `left_index` and `right_index`.)

In [46]:

```
countries_merged = countries_gdf.merge(countries_europe, left_on='NAME', right_o
n='Country')
display(countries_merged)
```

Spatial join

The spatial join (`sjoin()`) function of *geopandas* performs a spatial intersection check between the records of one or two *GeoDataFrames*. (The *rtree* package must be installed for spatial indexing support.)

Let's match the countries and cities based on their spatial location:

In [47]:

```
display(gpd.sjoin(countries_gdf, cities_gdf))
```

Limit the number of columns displayed to get an output easier to interpret:

In [48]:

```
display(gpd.sjoin(countries_gdf, cities_gdf)[['NAME', 'CITY_NAME']])
```

	NAME	CITY_NAME
0	Indonesia	Jayapura
0	Indonesia	Kupang
0	Indonesia	Denpasar
0	Indonesia	Mataram
0	Indonesia	Yogyakarta
...	...	...
246	Bahrain	Sitrah
246	Bahrain	Ar Rifa
246	Bahrain	Jidd Hafs
246	Bahrain	Manama
250	Macao	Macau

2487 rows × 2 columns

Select the cities inside Hungary for a quick verification of the results:

In [49]:

```
condition = countries_gdf['NAME'] == 'Hungary'
hungary_gdf = countries_gdf[condition]
display(gpd.sjoin(hungary_gdf, cities_gdf)[['NAME', 'CITY_NAME']])
```

	NAME	CITY_NAME
75	Hungary	Pecs
75	Hungary	Szeged
75	Hungary	Szekszard
75	Hungary	Kaposvar
75	Hungary	Bekescsaba
75	Hungary	Zalaegerszeg
75	Hungary	Kecskemet
75	Hungary	Veszprem
75	Hungary	Szolnok
75	Hungary	Szekesfehervar
75	Hungary	Szombathely
75	Hungary	Budapest
75	Hungary	Debrecen
75	Hungary	Tatabanya
75	Hungary	Gyor
75	Hungary	Eger
75	Hungary	Nyiregyhaza
75	Hungary	Salgotarjan
75	Hungary	Miskolc

Perform a spatial intersection check between the dataframe containing only Hungary (`hungary_gdf`) and the dataframe containing all countries (`countries_gdf`). The result shall be the neighbouring countries of Hungary.

In [50]:

```
display(gpd.sjoin(hungary_gdf, countries_gdf)[['NAME_left', 'NAME_right']])
```

	NAME_left	NAME_right
75	Hungary	Romania
75	Hungary	Ukraine
75	Hungary	Serbia
75	Hungary	Croatia
75	Hungary	Slovenia
75	Hungary	Hungary
75	Hungary	Austria
75	Hungary	Slovakia

Remark: the `NAME` column was renamed to `NAME_left` and `NAME_right` automatically, since column names must be unique.

Writing spatial data

GeoDataFrames can be easily persisted with the `to_file()` function. As when reading files, various file formats are supported again.

In [51]:

```
clipped_gdf.to_file('11_clipped.shp')
#clipped_gdf.to_file('11_clipped2.geojson', driver='GeoJSON')
```

Summary exercises on vector data management

Beside the `countries_gdf` *GeoDataFrame*, read the `data/ne_10m_rivers_lake_centerlines.shp` shapefile located in the `data` folder. This dataset contains both scalar and spatial data of the larger rivers and lakes around the world.
Source: Natural Earth (https://www.naturalearthdata.com/downloads/10m-physical-vectors/)

In [52]:

```
rivers_gdf = gpd.read_file('../data/ne_10m_rivers_lake_centerlines.shp')
display(rivers_gdf)
```

	dissolve	scalerank	featurecla	name	name_alt	rivernum	note	min_zoom	r
0	0River	1.0	River	Irrawaddy Delta	None	0	None	2.0	I
1	1001Lake Centerline	9.0	Lake Centerline	Tonle Sap	None	1001	None	7.1	
2	1001River	9.0	River	Tonle Sap	None	1001	None	7.1	
3	1002Lake Centerline	9.0	Lake Centerline	Sheksna	None	1002	None	7.1	
4	1002River	9.0	River	Sheksna	None	1002	None	7.1	
...	...	...	...	...	...	...	...	...	
1449	2050Lake Centerline	10.0	Lake Centerline	Tekapo	None	2050	None	7.2	
1450	2049Lake Centerline	10.0	Lake Centerline	Ohau	None	2049	None	7.2	
1451	219River	6.0	River	Po	None	219	Version 4 edit	5.0	
1452	178River	5.0	River	Loire	None	178	Changed in 4.0	4.7	
1453	303Drau	7.0	River	Drau	Drava	303	None	6.0	

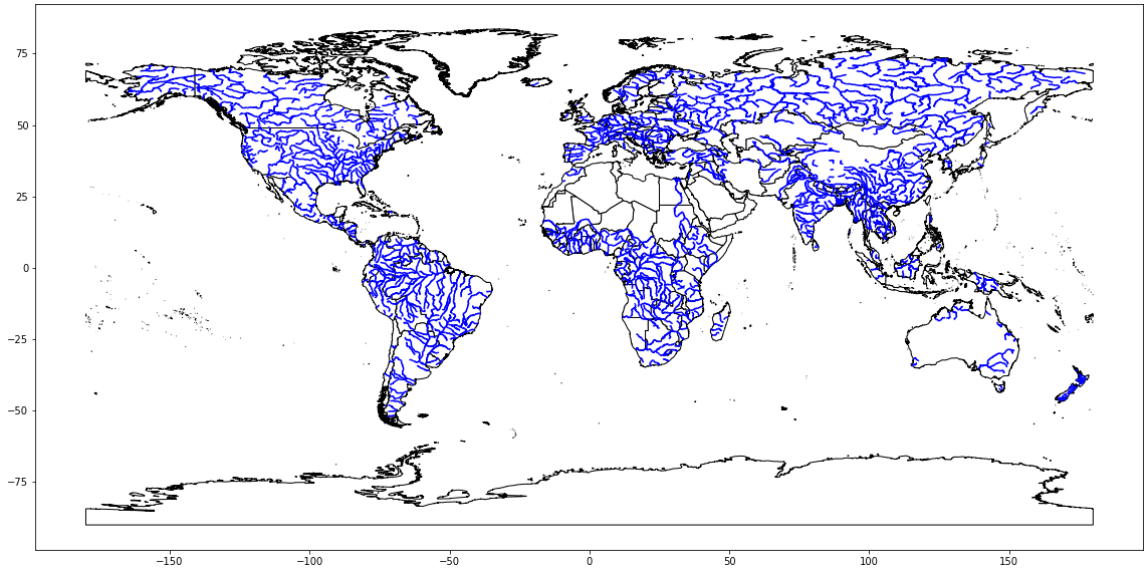
1454 rows × 35 columns

Exercise 1

Visualize the country boundaries and the river/lake layers on the same map. (Rivers and lakes shall be blue.)

In [53]:

```
base = countries_gdf.plot(color='white', edgecolor='black', figsize=[20, 10])
rivers_gdf.plot(ax=base, color='blue')
plt.show()
```

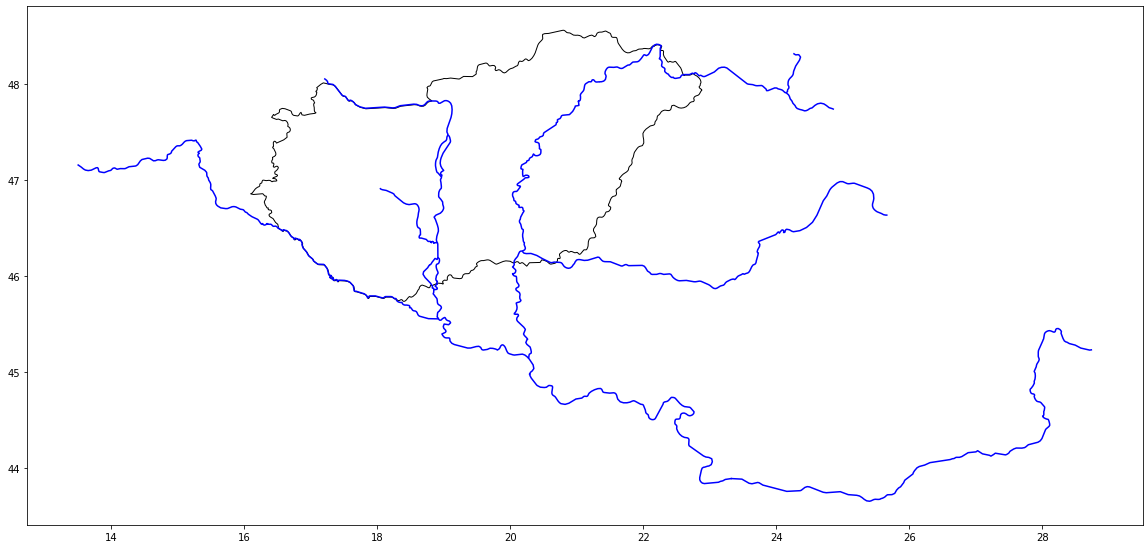


Exercise 2

Visualize only Hungary (on any preferred country) and the rivers flowing through it.

In [54]:

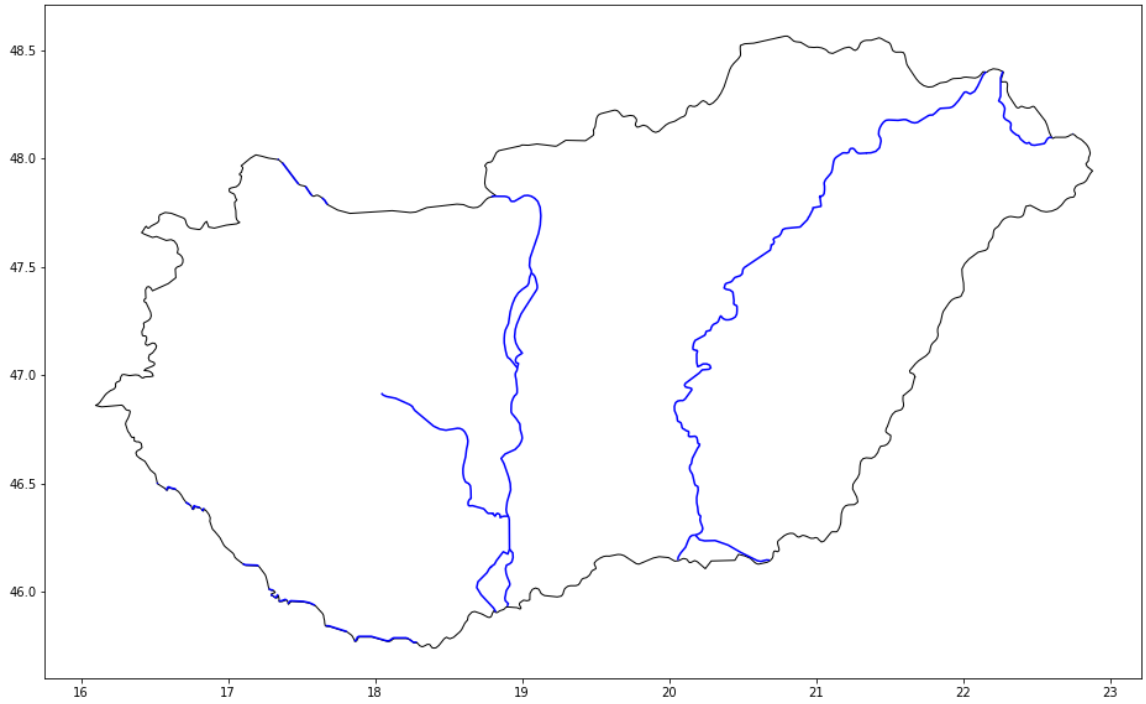
```
hungary_gdf = countries_gdf[countries_gdf['NAME'] == 'Hungary']
hungary_rivers = gpd.sjoin(rivers_gdf, hungary_gdf)

base = hungary_gdf.plot(color='white', edgecolor='black', figsize=[20, 10])
hungary_rivers.plot(ax=base, color='blue')
plt.show()
```



With clipping to country boundaries:

In [55]:

```
hungary_rivers = gpd.clip(hungary_rivers, hungary_gdf)

base = hungary_gdf.plot(color='white', edgecolor='black', figsize=[20, 10])
hungary_rivers.plot(ax=base, color='blue')
plt.show()
```



Exercise 3

Determine for the river *Danube* (or any major river) that which countries it flows through.

Hint: the river might consist of multiple line segments in the river dataset, but you can filter all of them by e.g. the `name_en` *field.*

In [56]:

```
danube_gdf = rivers_gdf[rivers_gdf['name_en'] == 'Danube']
display(danube_gdf)

danube_countries = gpd.sjoin(danube_gdf, countries_gdf)
display(danube_countries[['NAME']])
```

	dissolve	scalerank	featurecla	name	name_alt	rivernum	note	min_zoom	name_e
389	25River	2.0	River	Danube	None	25	None	2.1	Danub
566	38River	2.0	River	Donau	Danube	38	None	2.1	Danub

2 rows × 35 columns

	NAME
389	Bulgaria
389	Romania
389	Ukraine
389	Serbia
389	Croatia
389	Hungary
389	Slovakia
566	Slovakia
566	Austria
566	Germany

Process the Shapely objects in a GeoDataFrame

The `data/hungary_admin_8.shp` shapefile contains the city level administrative boundaries of Hungary.
Data source: *OpenStreetMap (https://data2.openstreetmap.hu/hatarok/)*

Load the dataset into a GeoDataFrame, set the `NAME` column as index and convert it to the EOV coordinate reference system.

In [57]:

```
import geopandas as gpd

cities_admin = gpd.read_file('../data/hungary_admin_8.shp')
print("Initial CRS: {0}".format(cities_admin.crs))

cities_admin.set_index('NAME', inplace=True)
cities_admin.to_crs('epsg:23700', inplace=True) # EOV
print("Converted CRS: {0}".format(cities_admin.crs))

display(cities_admin)
```

```
Initial CRS: epsg:3857
Converted CRS: epsg:23700
```

NAME	ADMIN_LEVE	geometry
Murakeresztúr	8	POLYGON ((480939.034 114618.287, 480958.625 11...
Tótszerdahely	8	POLYGON ((473882.976 118474.207, 474009.733 11...
Molnári	8	POLYGON ((477975.454 117130.010, 478008.561 11...
Semjénháza	8	POLYGON ((480316.699 120040.067, 480358.723 12...
Felsőszölnök	8	POLYGON ((426404.582 173622.019, 426532.558 17...
...	...	...
Milota	8	POLYGON ((924402.526 310431.812, 924402.529 31...
Tiszabecs	8	POLYGON ((927957.308 311701.481, 928042.039 31...
Garbolc	8	POLYGON ((933756.183 296547.014, 933798.205 29...
Magosliget	8	POLYGON ((931597.406 308102.976, 931640.122 30...
Beregdaróc	8	POLYGON ((904890.259 323808.371, 905059.726 32...

3174 rows × 2 columns

Process all the rows in the GeoDataFrame and display only the counties with an area larger than 200 km^2:

In [58]:

```
for name, row in cities_admin.iterrows():
    geom = row['geometry']
    if geom.area / 1e6 >= 200:
        print('{0}, Area: {1:.1f} km2, Centroid: {2}'.format(name, geom.area / 1
e6, geom.centroid))
```

```
Kiskunhalas, Area: 227.6 km2, Centroid: POINT (682217.376155288 1208
89.7788700489)
Kecskemét, Area: 321.2 km2, Centroid: POINT (698210.8659972923 17416
4.1829904111)
Budapest, Area: 526.1 km2, Centroid: POINT (654536.6170633805 23778
9.4103324989)
Szeged, Area: 281.1 km2, Centroid: POINT (734457.1375727949 101202.7
389595481)
Makó, Area: 229.2 km2, Centroid: POINT (764616.0567523418 105196.522
2338889)
...
Hajdúböszörmény, Area: 370.7 km2, Centroid: POINT (830010.1918890307
265122.9472381996)
Debrecen, Area: 461.5 km2, Centroid: POINT (847107.7868464794 24624
5.2987919257)
Miskolc, Area: 236.6 km2, Centroid: POINT (773057.0858664612 306776.
8422291108)
Hajdúnánás, Area: 259.6 km2, Centroid: POINT (824141.1866937836 2813
10.831795416)
Nyíregyháza, Area: 274.5 km2, Centroid: POINT (848770.9471192813 291
701.487003606)
```

The EOV coordinates `(653812, 239106)` are inside the territory of Budapest. Check whether really on this administrative unit contains this location.

In [59]:

```
pos_budapest = geometry.Point(653812, 239106)
for name, row in cities_admin.iterrows():
    geom = row['geometry']

    if geom.contains(pos_budapest):
        print(name)
```

```
Budapest
```

Chapter 12: Spatial data management - raster formats

Rasterio (https://rasterio.readthedocs.io/en/latest/) is a highly useful module for raster processing which you can use for reading and writing several raster formats in Python.

How to install *rasterio*?

We need to install the `rasterio` package.

Anaconda - Platform independent

If you have Anaconda installed, open the *Anaconda Prompt* and type in:

```
conda install -c conda-forge rasterio
```

Python Package Installer (pip) - Linux

If you have standalone Python3 and Jupyter Notebook install on Linux, open a command prompt / terminal and type in:

```
pip3 install rasterio
```

Python Package Installer (pip) - Windows

The installation of *rasterio* is much more complicated with *pip* on Windows, because it depends on the *GDAL* library, for which the binaries must be installed separately or compiled from source. An easier approach is to install these packages from Python binary wheel files (https://www.lfd.uci.edu/~gohlke/pythonlibs/).

Due to its complexity these options are only recommended for advanced Python users and it is **strongly advised to use Anaconda on Windows**.

How to use *rasterio*?

The rasterio package is also a module which you can simply import.

```
import rasterio
```

Opening a dataset

The `open()` function takes a path string or path-like object and returns an opened dataset object. The path may point to a file of any supported raster format.

The `data/LC08_L1TP_188027_20200420_20200508_01_T1_Szekesfehervar.tif` file is a segment of a Landsat 8 satellite image of Székesfehérvár city, Lake Velence and their surroundings, acquired on 2020 April 20.

In [1]:

```
import rasterio
szfv_2020 = rasterio.open('../data/LC08_L1TP_188027_20200420_20200508_01_T1_Szek
esfehervar.tif')
```

Dataset objects have some attributes regarding the opened file:

In [2]:

```
print(szfv_2020.name)
print(szfv_2020.mode) # by default the file is opened in read mode
print(szfv_2020.closed) # will be True after closed() called
```

```
../data/LC08_L1TP_188027_20200420_20200508_01_T1_Szekesfehervar.tif
r
False
```

Properties of the raster data stored in the example GeoTIFF can be accessed through attributes of the opened dataset object.

In [3]:

```
print(szfv_2020.count) # band count
print(szfv_2020.width) # dimensions
print(szfv_2020.height)
```

```
11
1057
645
```

Dataset georeferencing

A GIS raster dataset is different from an ordinary image; its elements (or “pixels”) are mapped to regions on the earth’s surface. All pixels of a dataset is contained within a spatial bounding box.

In [4]:

```
print(szfv_2020.bounds)
```

```
BoundingBox(left=296745.0, bottom=5221185.0, right=328455.0, top=524
0535.0)
```

Our example covers the world from 296745 meters to 328455 meters left to right, and 5221185 meters to 5240535 meters bottom to top. Therefore, it covers a region 31.71 kilometers wide by 19.35 kilometers high.

The value of `bounds` attribute is derived from a more fundamental attribute: the dataset’s geospatial transform.

In [5]:

```
print(szfv_2020.transform)
```

```
| 30.00, 0.00, 296745.00|
| 0.00,-30.00, 5240535.00|
| 0.00, 0.00, 1.00|
```

A dataset's `transform` is an affine transformation matrix that maps pixel locations in *(row, col)* coordinates to *(x, y)* spatial positions. The product of this matrix and `(0, 0)`, the row and column coordinates of the upper left corner of the dataset, is the spatial position of the upper left corner.

In [6]:

```
print(szfv_2020.transform * (0, 0))
```

```
(296745.0, 5240535.0)
```

The position of the lower right corner is obtained similarly.

In [7]:

```
print(szfv_2020.transform * (szfv_2020.width, szfv_2020.height))
```

```
(328455.0, 5221185.0)
```

But what do these numbers mean? 296745 meters from where? These coordinate values are relative to the origin of the dataset's coordinate reference system (CRS).

In [8]:

```
print(szfv_2020.crs)
```

```
EPSG:32634
```

All metadata for the whole raster dataset can be displayed easily if desired:

In [9]:

```
print(szfv_2020.meta)
```

```
{'driver': 'GTiff', 'dtype': 'uint16', 'nodata': None, 'width': 105
7, 'height': 645, 'count': 11, 'crs': CRS.from_epsg(32634), 'transfo
rm': Affine(30.0, 0.0, 296745.0,
       0.0, -30.0, 5240535.0)}
```

Reading raster data

Data from a raster band can be accessed by the band's index number. Following the GDAL (https://gdal.org/) convention (on which library Rasterio depends on), bands are indexed from 1.

In [10]:

```
print(szfv_2020.indexes)
```

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
```

Landsat 8 satellite images contain 11 bands, in the following order:

Band Number	Description	Wavelength	Resolution
Band 1	Coastal / Aerosol	0.433 to 0.453 µm	30 meter
Band 2	Visible blue	0.450 to 0.515 µm	30 meter
Band 3	Visible green	0.525 to 0.600 µm	30 meter
Band 4	Visible red	0.630 to 0.680 µm	30 meter
Band 5	Near-infrared	0.845 to 0.885 µm	30 meter
Band 6	Short wavelength infrared	1.56 to 1.66 µm	30 meter
Band 7	Short wavelength infrared	2.10 to 2.30 µm	60 meter
Band 8	Panchromatic	0.50 to 0.68 µm	15 meter
Band 9	Cirrus	1.36 to 1.39 µm	30 meter
Band 10	Long wavelength infrared	10.3 to 11.3 µm	100 meter
Band 11	Long wavelength infrared	11.5 to 12.5 µm	100 meter

We can read the bands of a dataset with the `read()` method:

In [11]:

```
red = szfv_2020.read(4)
green = szfv_2020.read(3)
blue = szfv_2020.read(2)
```

Bands are simply 2D mathematical matrices stored as multi-dimensional *NumPy* arrays. *NumPy* (https://numpy.org/) is a first-rate library for numerical programming. It is widely used in academia, finance and also in the industry.

Not only *Rasterio*, but the already introduced *Pandas* library (see Chapter 9 (09_tabular.pdf)) is also built on top of *NumPy*, providing high-performance, easy-to-use data structures and data analysis tools, making data manipulation and visualization more convenient.

In [12]:

```
print(type(red))
print(red)
```

```
<class 'numpy.ndarray'>
[[10341 11341 11207 ...  9396 10034  9787]
 [10870  9897  8611 ...  9519  9783 10904]
 [ 9462  8245  7742 ...  9874  9893 10182]
 ...
 [ 8764  9336  9138 ...  9509  9379  9034]
 [ 7363  8361  9568 ... 10178 10898  8784]
 [ 7153  7760  9294 ...  9827 10491  8794]]
```

For a *NumPy* array we can easily get the range and the mean of the values:

In [13]:

```
print(red.min())
print(red.max())
print(red.mean())
```

```
6304
55987
8493.439567886295
```

Values from the array can be addressed by their row, column index.

In [14]:

```
print(red[500, 500]) # random position
```

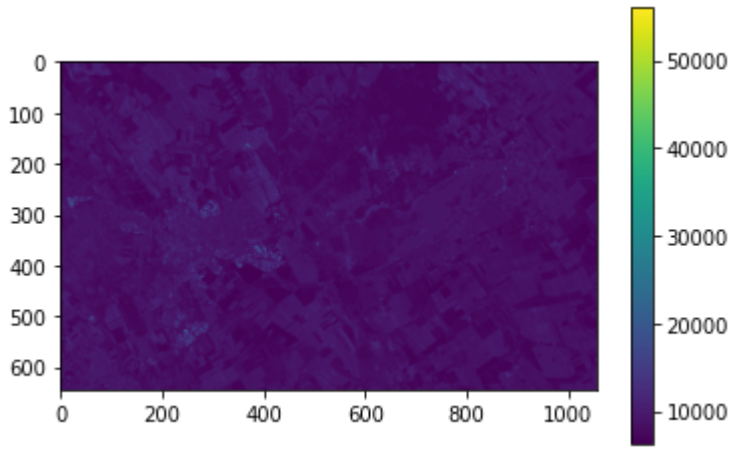
```
10245
```

Plotting

Since *Rasterio* reads raster data into mathematical matrices (*numpy arrays*), plotting a single band as two-dimensional data can be accomplished directly with *matplotlib*, as it also strongly depends on *NumPy*. For detailed information on Numpy, see Appendix 2 (AX02_math.pdf).

In [15]:

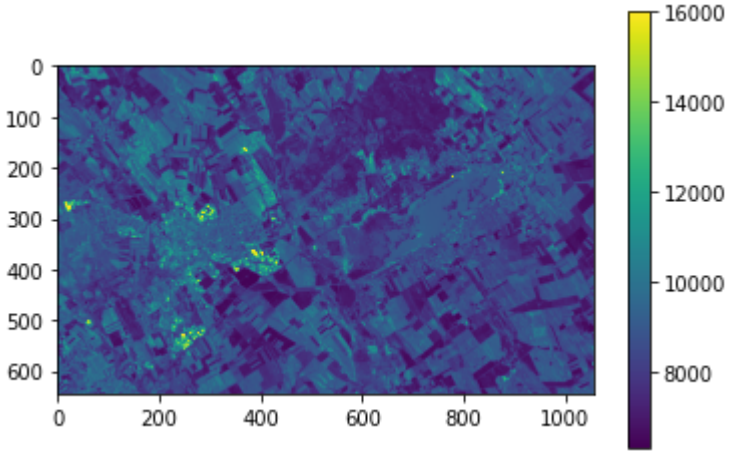
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.imshow(red)
plt.colorbar()
plt.show()
```



In our case the data is not evenly distributed in the range $[0, 65535]$, most values are below 16000. The maximum and minimum value for visualization can be overridden with the `vmax` and `vmin` parameters.

In [16]:

```
plt.imshow(red, vmax=16000)
plt.colorbar()
plt.show()
```



Instead of using a static value (16000), calculate the 99.9% percentile of each bands to remove only the few outliers (0.1%) from visualization.

In [17]:

```
import numpy as np

red_max = np.percentile(red, 99.9)
blue_max = np.percentile(blue, 99.9)
green_max = np.percentile(green, 99.9)
print(red_max)
print(blue_max)
print(green_max)
```

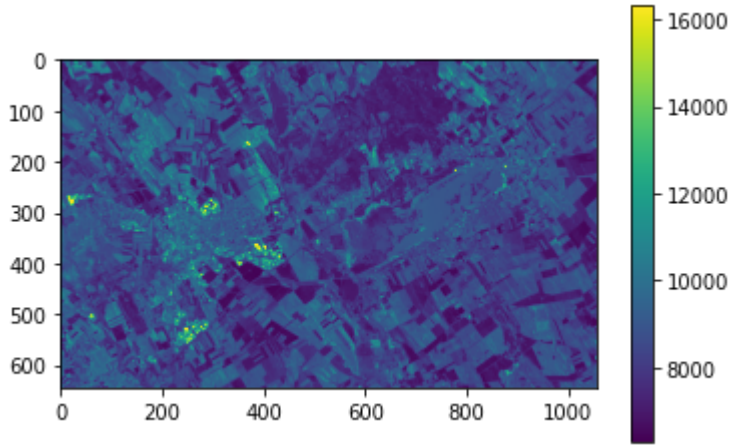
```
16310.472000000067
14844.236000000034
15248.0
```

Remark: here we use the numpy package directly to calculate the 99.9% percentile. NumPy is a module which can be imported as usual and is aliased with the `np` abbreviation in most cases.

The `vmax` parameter can be defined as a dynamic value for the now:

In [18]:

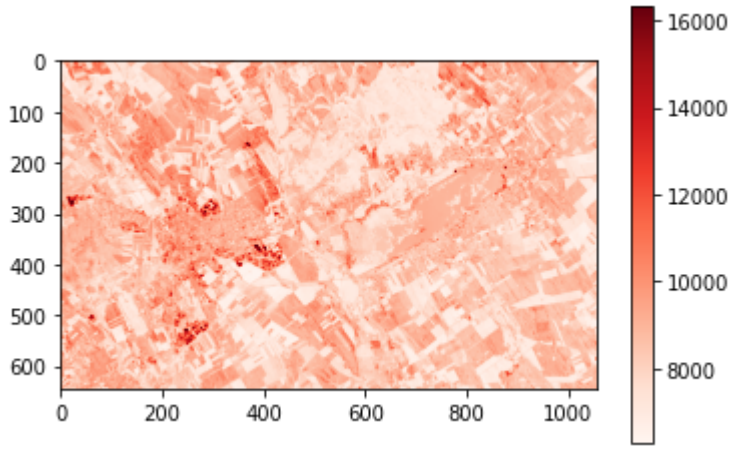
```
plt.imshow(red, vmax=red_max)
plt.colorbar()
plt.show()
```



Color maps can also be used with the `cmap` parameter (*see Chapter 11 (11_spatial_vector.pdf) for more details*).

In [19]:

```
plt.imshow(red, vmax=red_max, cmap='Reds')
plt.colorbar()
plt.show()
```



Histogram

Create a histogram of the visible bands of the Landsat satellite image.

First, create a pandas *DataFrame* from the 3 bands. The *DataFrame* shall contain 3 *Series*: one for each band. The *DataFrame* shall contain as many rows as many pixels are in the image. To achieve this we *flatten* the 2D matrices into 1D vectors. (For *NumPy* both of them are arrays, regardless of their dimensions.)

In [20]:

```
print("2D array:")
print(red)
print("Type: {0}, Size: {1}".format(type(red), red.size))
print()
print("1D array:")
red_vector = red.flatten()
print(red_vector)
print("Type: {0}, Size: {1}".format(type(red), red.size))
```

```
2D array:
[[10341 11341 11207 ...  9396 10034  9787]
 [10870  9897  8611 ...  9519  9783 10904]
 [ 9462  8245  7742 ...  9874  9893 10182]
 ...
 [ 8764  9336  9138 ...  9509  9379  9034]
 [ 7363  8361  9568 ... 10178 10898  8784]
 [ 7153  7760  9294 ...  9827 10491  8794]]
Type: <class 'numpy.ndarray'>, Size: 681765

1D array:
[10341 11341 11207 ...  9827 10491  8794]
Type: <class 'numpy.ndarray'>, Size: 681765
```

In [21]:

```
import pandas as pd

szfv_df = pd.DataFrame({
    'red': red.flatten(),
    'blue': blue.flatten(),
    'green': green.flatten()
})
display(szfv_df)
display(szfv_df.iloc[100000]) # random row
```

	red	blue	green
0	10341	10086	10072
1	11341	10607	10728
2	11207	10433	10688
3	8566	9440	9005
4	7705	8955	8736
...	...	...	...
681760	8517	9096	8383
681761	8338	9026	8247
681762	9827	9842	9359
681763	10491	10163	9789
681764	8794	9314	8599

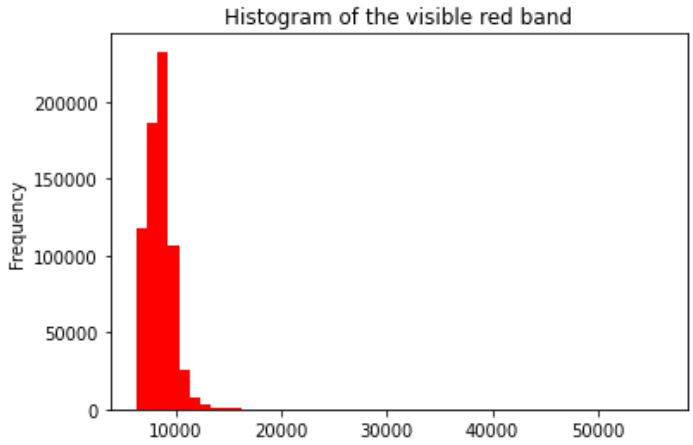
681765 rows × 3 columns

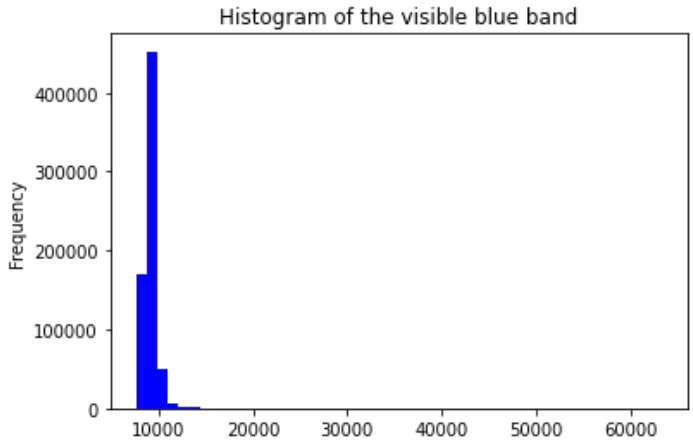
```
red      7363
blue     8577
green    8090
Name: 100000, dtype: uint16
```

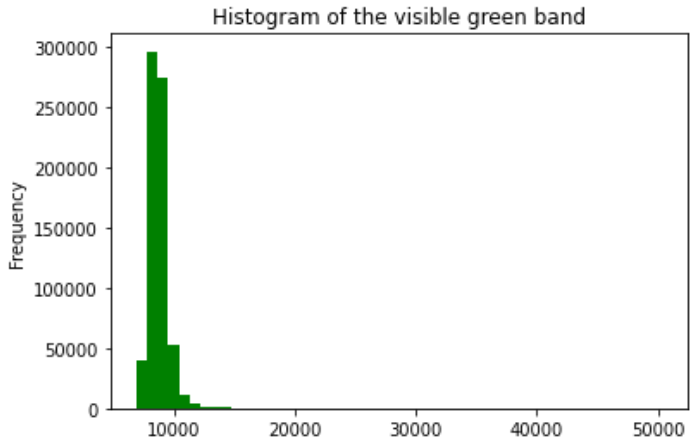
Now we can create the histograms for the *Series* (or for the *DataFrame* to display it on a single plot), as we have learned it in Chapter 10 (10_plotting.pdf).

In [22]:

```
szfv_df['red'].plot(kind='hist', bins=50, color='red', title='Histogram of the v
isible red band')
plt.show()
szfv_df['blue'].plot(kind='hist', bins=50, color='blue', title='Histogram of the
 visible blue band')
plt.show()
szfv_df['green'].plot(kind='hist', bins=50, color='green', title='Histogram of t
he visible green band')
plt.show()
```





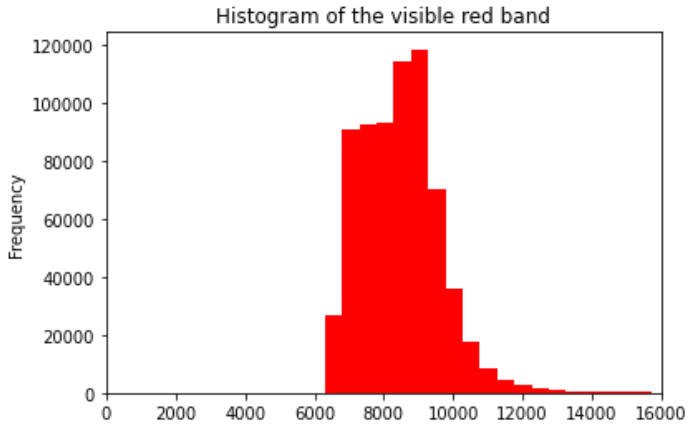


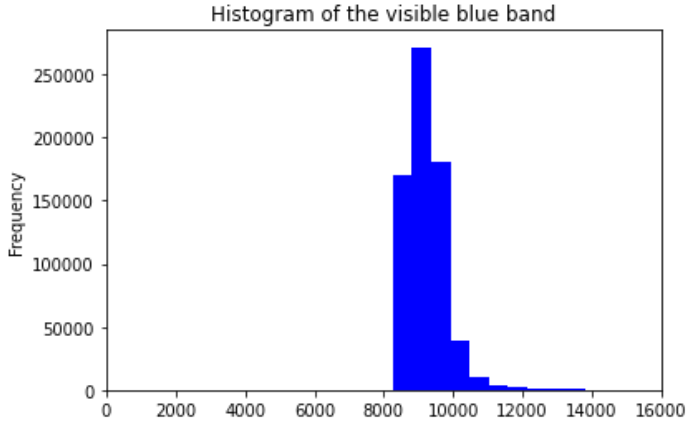
This visually verifies our previous conclusion that most values for the visible colour bands are below 16000.

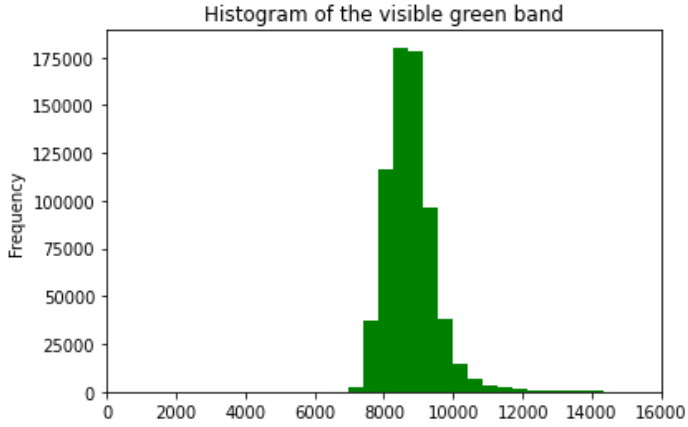
Get the histogram of the "interesting" part:

In [23]:

```
szfv_df['red'].plot(kind='hist', bins=100, xlim=(0, 16000), color='red', title=
'Histogram of the visible red band')
plt.show()
szfv_df['blue'].plot(kind='hist', bins=100, xlim=(0, 16000), color='blue', title
='Histogram of the visible blue band')
plt.show()
szfv_df['green'].plot(kind='hist', bins=100, xlim=(0, 16000), color='green', tit
le='Histogram of the visible green band')
plt.show()
```







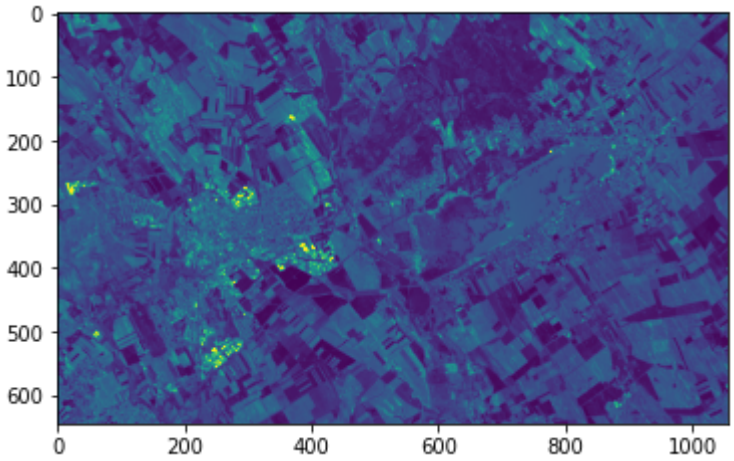
Multi-band plotting

Rasterio also provides `rasterio.plot.show()` to perform common tasks such as displaying multi-band images as RGB and labeling the axes with proper geo-referenced extents.

It can be used for a single band:

In [24]:

```
from rasterio.plot import show

show(red, vmax=red_max)
plt.show()
```



For multiple bands to visualize in a true-color image, the values must be in the range of $[0, 255]$ or in the float range of $[0, 1]$.

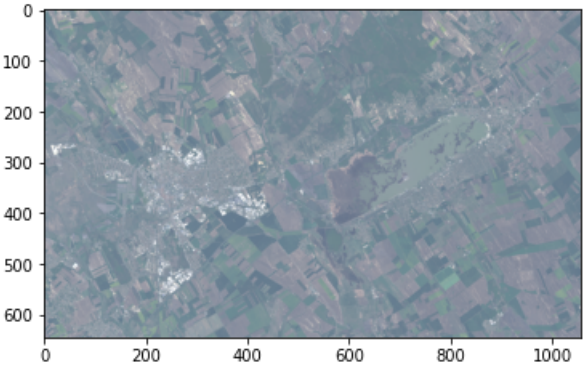
In [25]:

```
# astype('f4') is a numpy function to convert to float (4 byte)
redf = red.astype('f4') / red_max
bluef = blue.astype('f4') / blue_max
greenf = green.astype('f4') / green_max
rgb = [redf, greenf, bluef]
```

In [26]:

```
show(rgb)
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).
```



Increase the figure size:

In [27]:

```
plt.figure(figsize=[10,10])
show(rgb)
plt.show()
```

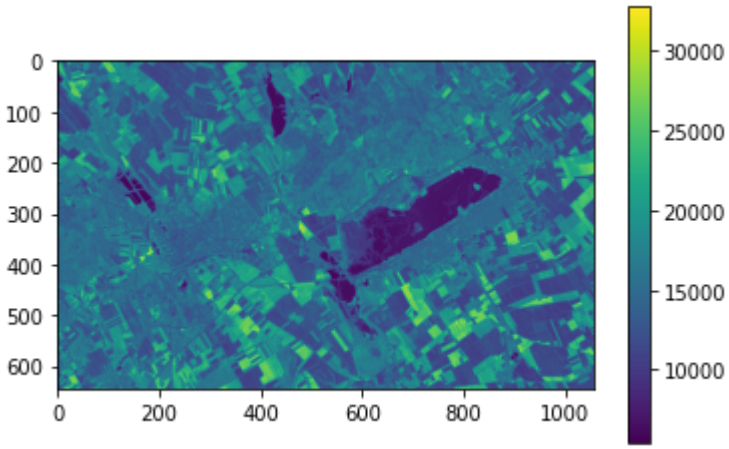
```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).
```

Example computation: NDVI

The Normalized Difference Vegetation Index (https://en.wikipedia.org/wiki/Normalized_difference_vegetation_index) is a simple indicator that can be used to assess whether the target includes healthy vegetation. This calculation uses two bands of a multispectral image dataset, the Red and Near-Infrared (NIR) bands.

In [28]:

```
nir = szfv_2020.read(5)
plt.imshow(nir, vmax=2**15)
plt.colorbar()
plt.show()
```



In [29]:

```
nir_max = np.percentile(nir, 99.9)
print(nir_max)
nirf = nir.astype('f4') / nir_max
```

```
28102.0
```

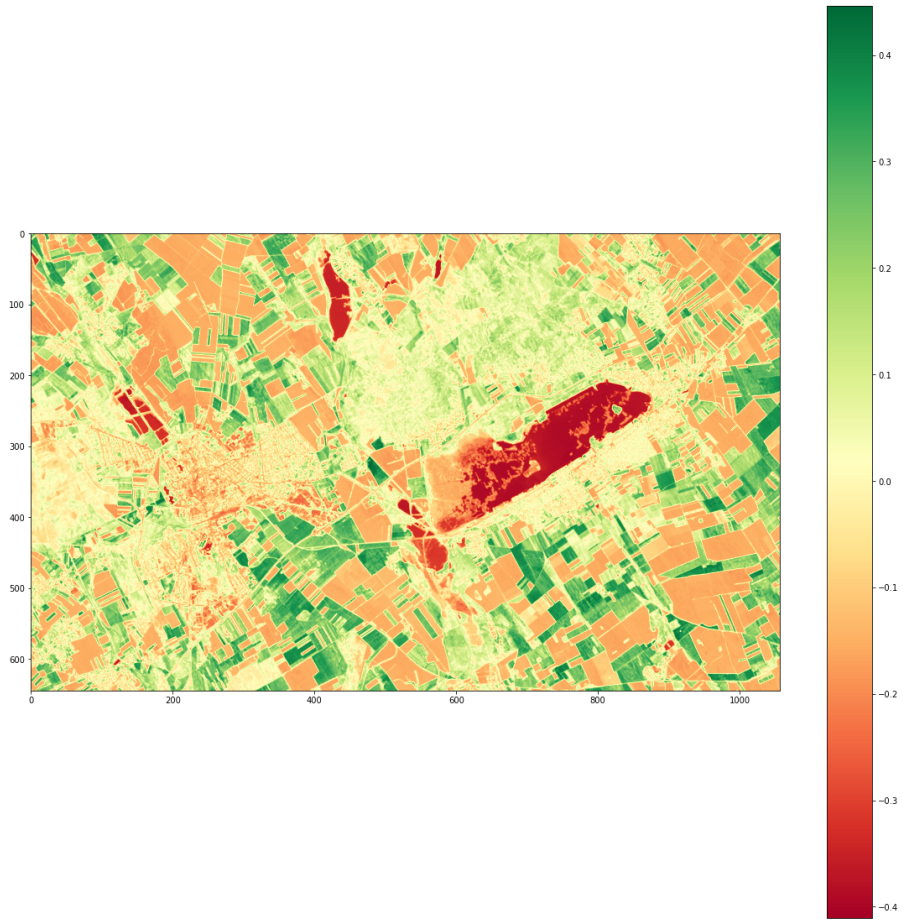
The value of *NDVI* can be calculated with a simple mathematical formula:

$$NDVI = \frac{NIR - Red}{NIR + Red}$$

With Rasterio we can perform the computation on the bands themselves, which will apply the computation to each pixel-pairs.

In [30]:

```
def calc_ndvi(nir, red):
    ndvi = (nir - red) / (nir + red)
    return ndvi

ndvi = calc_ndvi(nirf, redf)
plt.figure(figsize=[20, 20])
plt.imshow(ndvi, cmap='RdYlGn')
plt.colorbar()
plt.show()
```



The value range of an NDVI is -1 to 1. Negative values of NDVI (values approaching -1) correspond to water. Values close to zero (-0.1 to 0.1) generally correspond to barren areas of rock, sand, or snow. Low, positive values represent shrub and grassland (approximately 0.2 to 0.4), while high values indicate temperate and tropical rainforests (values approaching 1).

Summary exercise on raster data management

Exercise 1: NDVI change tracking

The `data/LC08_L1TP_188027_20180501_20180516_01_T1_Szekesfehervar.tif` file is a Landsat 8 satellite image from the same territory as the previous image, but acquired on 2018 May 1, so ca. 2 years earlier.

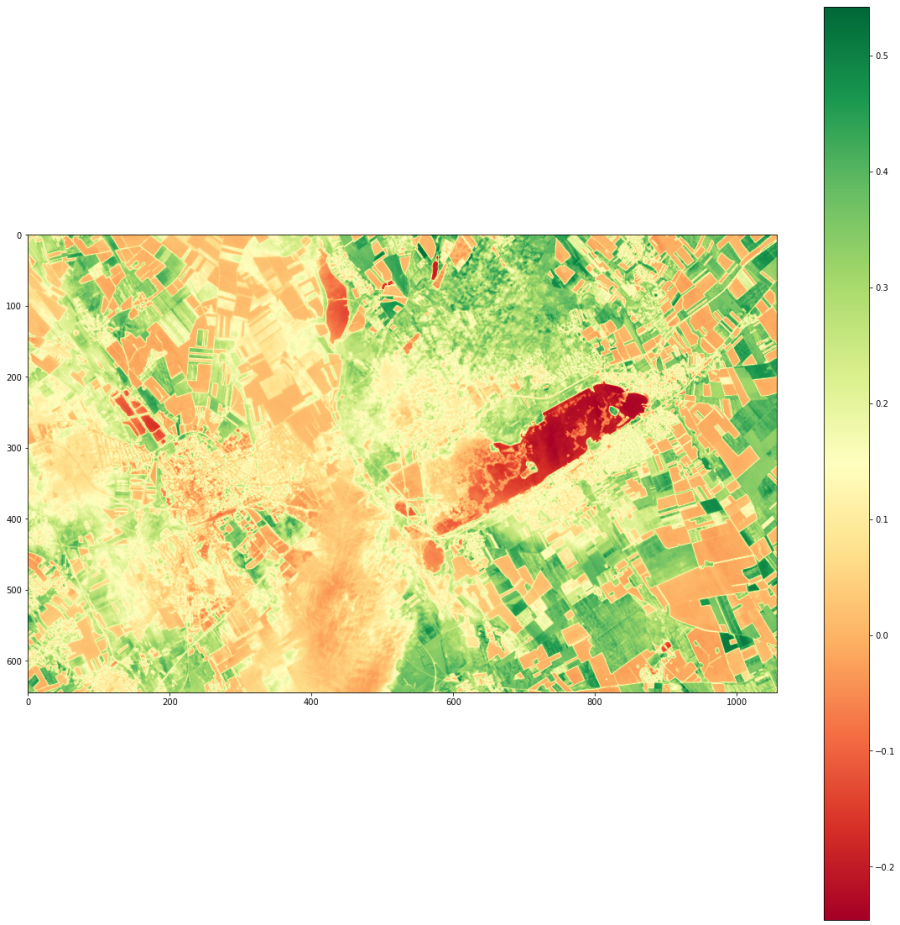
In [31]:

```
szfv_2018 = rasterio.open('../data/LC08_L1TP_188027_20180501_20180516_01_T1_Szekesfehervar.tif')
```

Task 1: Calculate the NDVI for the 2018 Landsat satellite image.

In [32]:

```
import numpy as np
red2 = szfv_2018.read(4)
red2_max = np.percentile(red2, 99.9)
redf2 = red2.astype('f4') / red2_max

nir2 = szfv_2018.read(5)
nir2_max = np.percentile(nir2, 99.9)
nirf2 = nir2.astype('f4') / nir2_max

ndvi2 = calc_ndvi(nirf2, redf2)
plt.figure(figsize=[20, 20])
plt.imshow(ndvi2, cmap='RdYlGn')
plt.colorbar()
plt.show()
```



Task 2: Compute the NDVI difference of the time interval and visaulize it.

Display the metadata of the 2 satellite images to compare them.

In [33]:

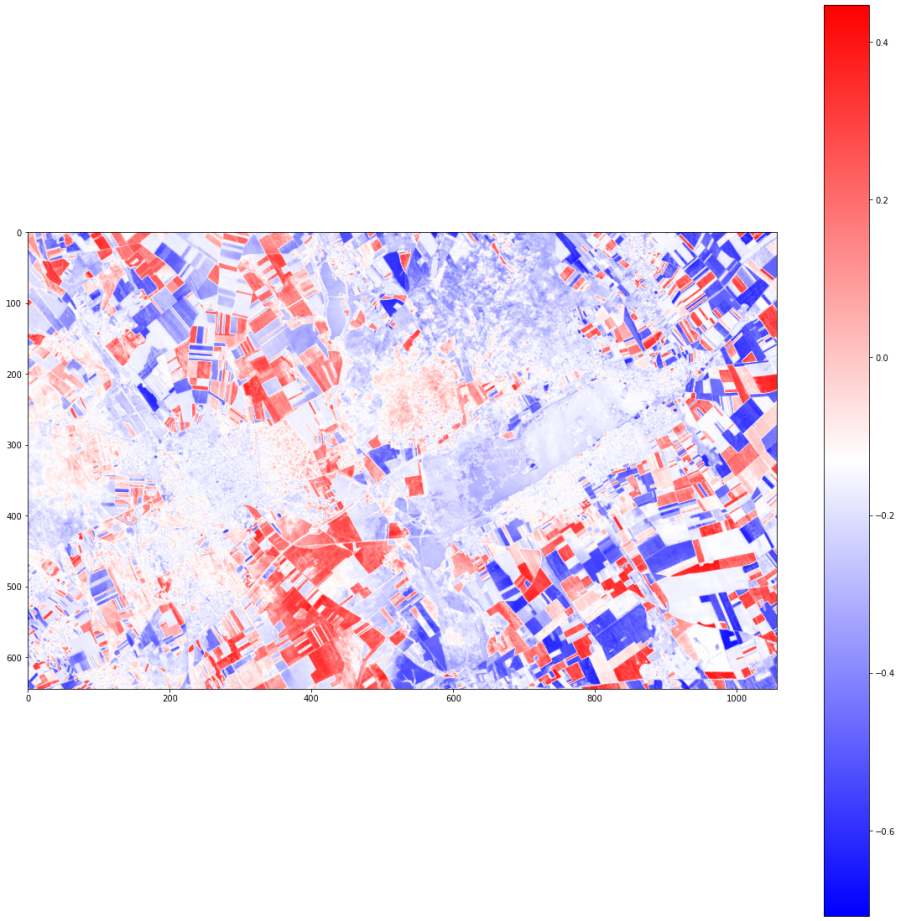
```
print(szfv_2020.meta)
print(szfv_2018.meta)
```

```
{'driver': 'GTiff', 'dtype': 'uint16', 'nodata': None, 'width': 105
7, 'height': 645, 'count': 11, 'crs': CRS.from_epsg(32634), 'transfo
rm': Affine(30.0, 0.0, 296745.0,
       0.0, -30.0, 5240535.0)}
{'driver': 'GTiff', 'dtype': 'uint16', 'nodata': None, 'width': 105
7, 'height': 645, 'count': 11, 'crs': CRS.from_epsg(32634), 'transfo
rm': Affine(30.0, 0.0, 296745.0,
       0.0, -30.0, 5240535.0)}
```

Compute and visualize the NDVI difference:

In [34]:

```
ndvi_diff = ndvi - ndvi2
plt.figure(figsize=[20, 20])
plt.imshow(ndvi_diff, cmap='bwr')
plt.colorbar()
plt.show()
```



Exercise 2: Processing larger images

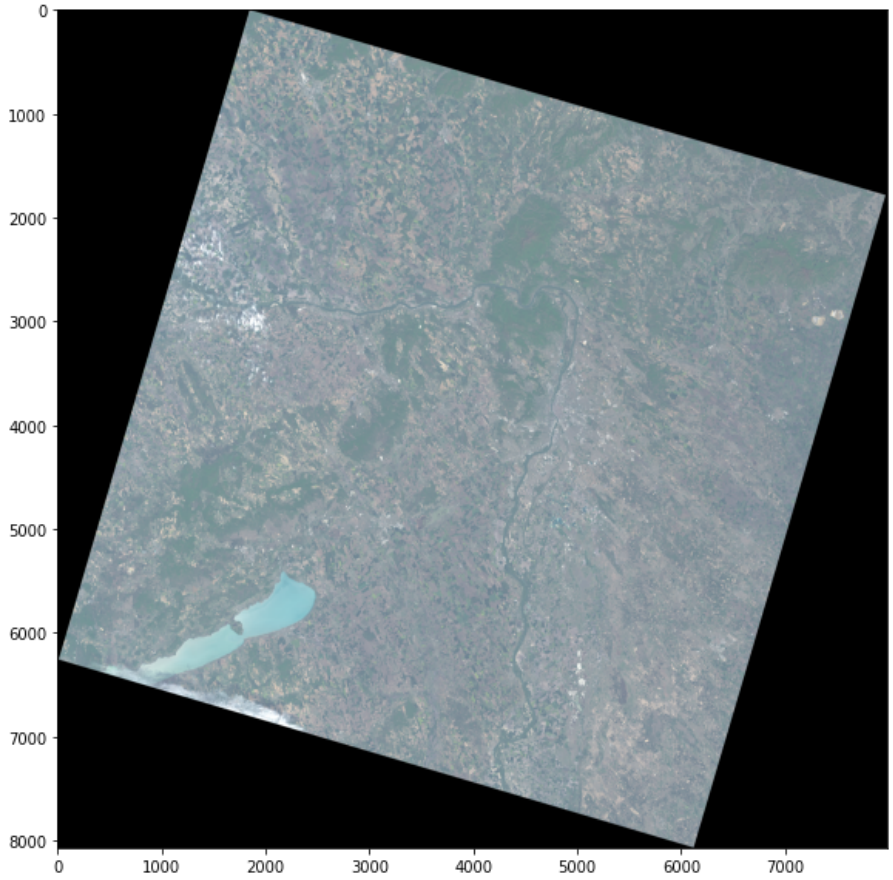
The `LC08_L1TP_188027_20200420_20200508_01_T1` file is a complete Landsat 8 satellite image tile, containing Budapest and parts of Western-Hungary, acquired on 2020 April 20.
Download: https://gis.inf.elte.hu/files/public/landsat-budapest-2020 (https://gis.inf.elte.hu/files/public/landsat-budapest-2020) (1.4 GB)

Task 1: create and RGB visualization for the complete satellite image.

In [35]:

```
bp_2020 = rasterio.open('LC08_L1TP_188027_20200420_20200508_01_T1.tif')

bp_red = bp_2020.read(4)
bp_green = bp_2020.read(3)
bp_blue = bp_2020.read(2)

bp_red_max = np.percentile(bp_red, 99.9)
bp_blue_max = np.percentile(bp_blue, 99.9)
bp_green_max = np.percentile(bp_green, 99.9)

bp_redf = bp_red.astype('f4') / bp_red_max
bp_bluef = bp_blue.astype('f4') / bp_blue_max
bp_greenf = bp_green.astype('f4') / bp_green_max
bp_rgb = [bp_redf, bp_greenf, bp_bluef]

plt.figure(figsize=[10,10])
show(bp_rgb)
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).
```

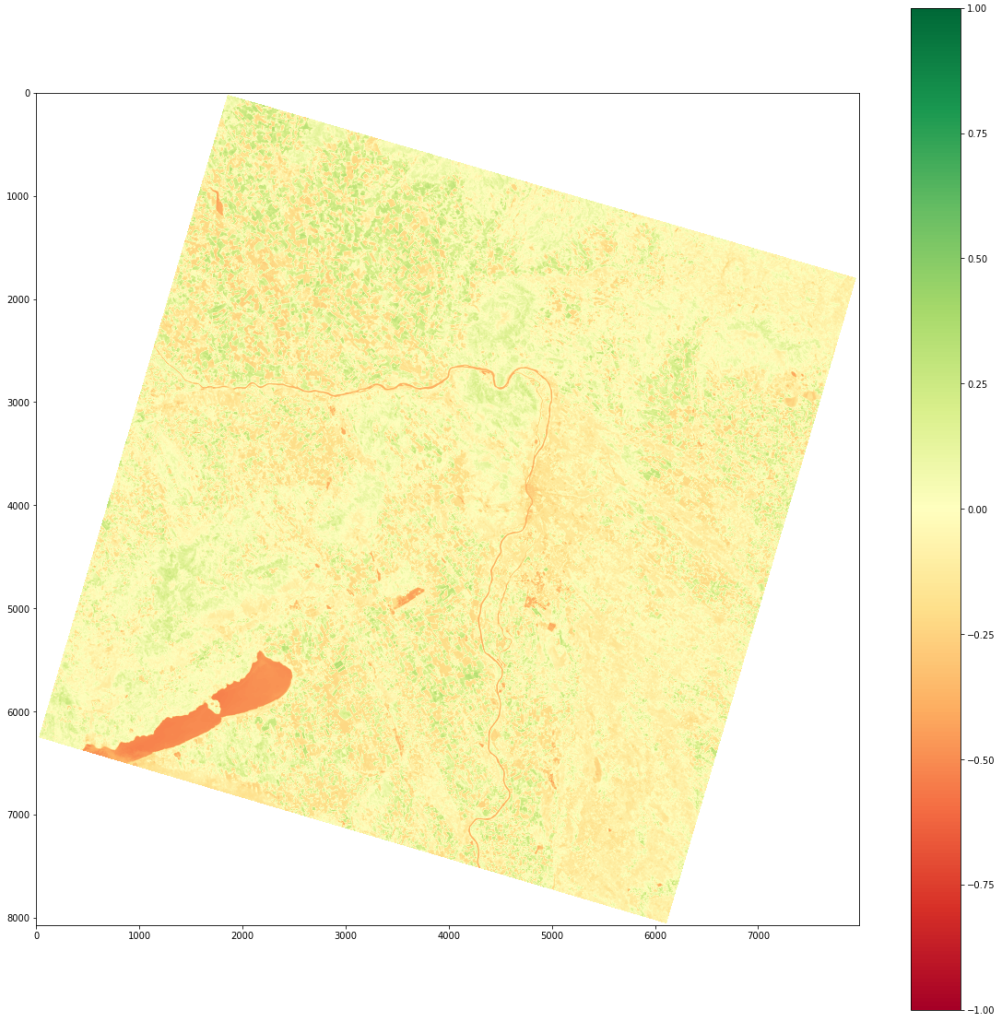


Task 2: calculate the NDVI for the complete satellite image.

In [36]:

```
bp_nir = bp_2020.read(5)
bp_nir_max = np.percentile(bp_nir, 99.99)
bp_nirf = bp_nir.astype('f4') / bp_nir_max

bp_ndvi = calc_ndvi(bp_nirf, bp_redf)
plt.figure(figsize=[20, 20])
plt.imshow(bp_ndvi, cmap='RdYlGn')
plt.colorbar()
plt.show()
```

```
<ipython-input-30-43374d61fc32>:2: RuntimeWarning: invalid value enc
ountered in true_divide
  ndvi = (nir - red) / (nir + red)
```



Chapter 13: Graph construction and management in Python

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. It is usually imported with the *nx* abbreviation.

How to install networkx?

We need to install the networkx package.
Note: if you have installed geopandas, you most likely also installed networkx already, as one of its dependency.

Anaconda - Platform independent

If you have Anaconda installed, open the *Anaconda Prompt* and type in:

```
conda install -c conda-forge networkx
```

Python Package Installer (pip) - Linux

If you have standalone Python3 and Jupyter Notebook install on Linux, open a command prompt / terminal and type in:

```
pip3 install networkx
```

How to use networkx?

The netwrokx package is a module which you can simply import. It is usually aliased with the `nx` abbreviation:

```
import networkx as nx
```

Graph creation

NetworkX supports 4 type of graphs:

- undirected, simple graphs: `Graph`
- directed simple graphs: `DiGraph`
- undirected graph with parallel edges: `MultiGraph`
- directed graph with parallel edges: `MultiDiGraph`

Creation of a new, empty graph is straightforward:

In [1]:

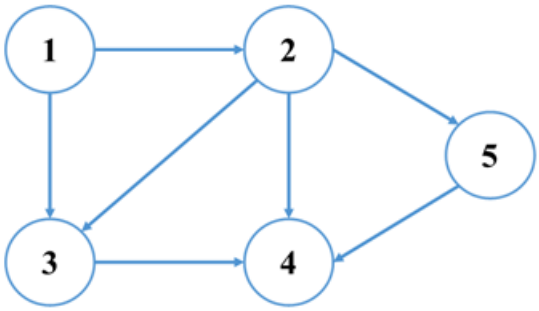
```
import networkx as nx
graph = nx.Graph() # undirected, simple graph
```

Representation

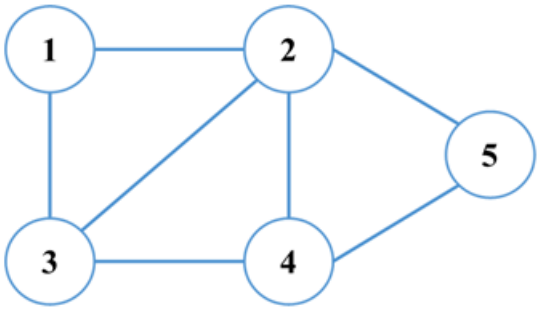
To represent the graphs, two data structures as very common practices are well-known. One has a purely arithmetic representation (*adjacency matrix*), and the other has a mixed arithmetic and chain representation (*edge list* or *neighborhood list*).

Adjacency matrix representation

In graph theory and computer science, an adjacency matrix is a square matrix. Its elements indicate whether pairs of vertices are adjacent in the graph or not.



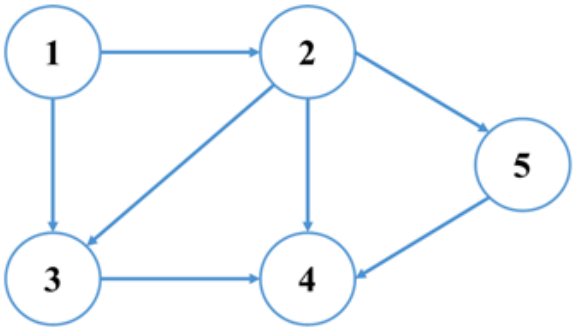
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

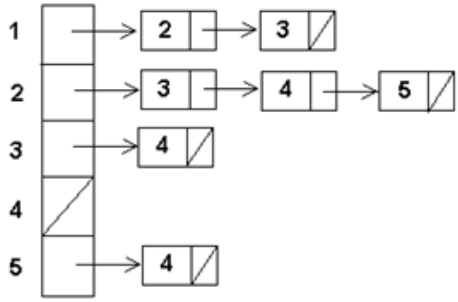


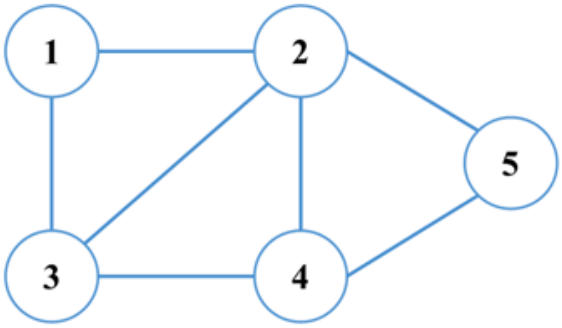
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

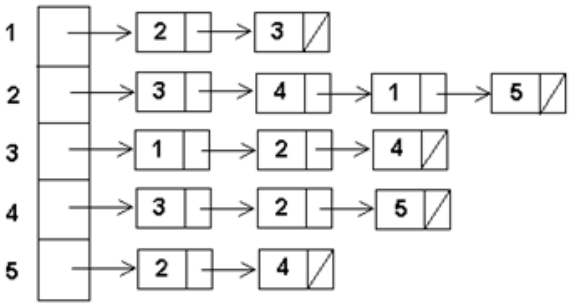
Edge list representation

The edge list is a data structure used to represent a graph as a list of its edges for each vertices. The internal data structures of *NetworkX* is based on the *adjacency list* representation and implemented using Python dictionary data structures.









Building a graph from scratch

We can add nodes and edges to a graph:

In [2]:

```
graph.add_node(1)
graph.add_node(2)
graph.add_node(3)
graph.add_node(4)
graph.add_node(5)
graph.add_node(6)
graph.add_node(7)
graph.add_node(8)
```

In [3]:

```
graph.add_edge(1, 2)
graph.add_edge(1, 3)
graph.add_edge(1, 4)
graph.add_edge(2, 3)
graph.add_edge(2, 5)
graph.add_edge(2, 6)
graph.add_edge(3, 6)
graph.add_edge(4, 5)
graph.add_edge(4, 7)
```

Adding an edge to a non-existing node will also create that particular node:

In [4]:

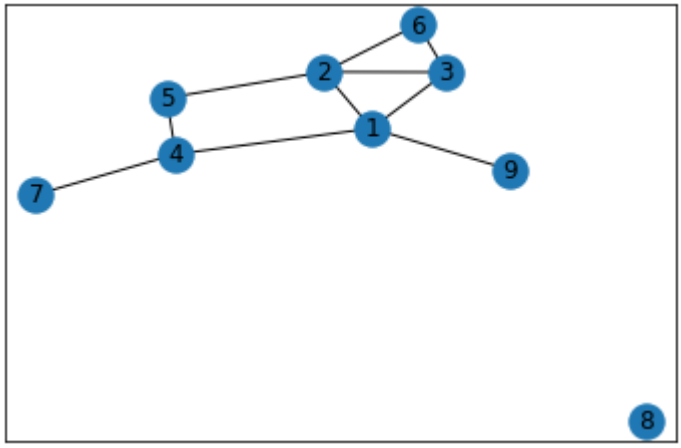
```
graph.add_edge(1, 9)
```

Graph visualization with Matplotlib

NetworkX has **tight integration** with *matplotlib*, therefore visualization of a graph can be done easily.

In [5]:

```
import matplotlib.pyplot as plt

# Special Jupyter Notebook command, so the plots by matplotlib will be displayed
 inside the Jupyter Notebook
%matplotlib inline

nx.draw_networkx(graph)
plt.show()
```



Building a graph from a *pandas* DataFrame

Let's use the following basic dataset of airroutes flight data:

1. From city
2. To city
3. Distance

The dataset is given in the `flights.csv` file in the `data` folder. The used delimiter is the semicolon (`;`) character.

Parse the CSV file into a *pandas* DataFrame:

In [6]:

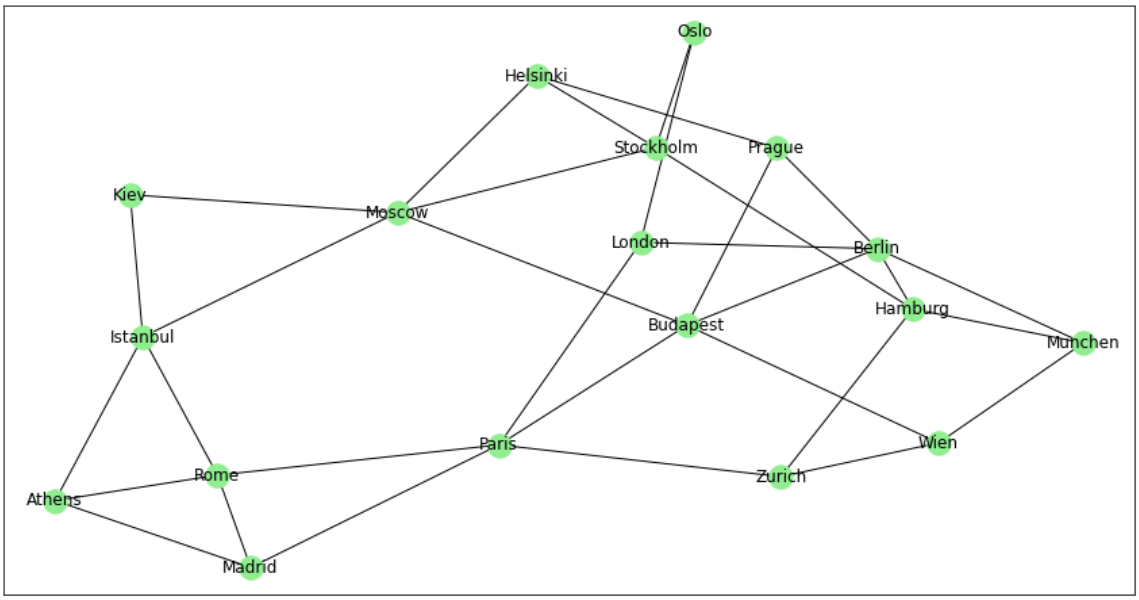
```
import pandas as pd

flight_table = pd.read_csv('../data/flights.csv', delimiter = ';')
display(flight_table)
```

	From city	To city	Distance
0	London	Paris	342
1	London	Berlin	932
2	London	Oslo	1153
3	Paris	Zurich	488
4	Paris	Budapest	1244
...	...	...	...
27	Athens	Istanbul	562
28	Kiev	Istanbul	1056
29	Istanbul	Moscow	1755
30	Rome	Athens	1051
31	Kiev	Moscow	755

NetworkX has a **from** and **to** conversion for *pandas* DataFrames. Assuming all airroutes are bi-directional, build an undirected graph:

In [7]:

```
flight_graph = nx.from_pandas_edgelist(flight_table, 'From city', 'To city')
plt.figure(figsize=[15,8])
nx.draw_networkx(flight_graph, node_color = 'lightgreen')
plt.show()
```



You can define the type of the graph with the optional `create_using` parameter. Its default value is `Graph`.

```
nx.from_pandas_edgelist(flight_table, 'From city', 'To city', create_usin
g = nx.DiGraph)
```

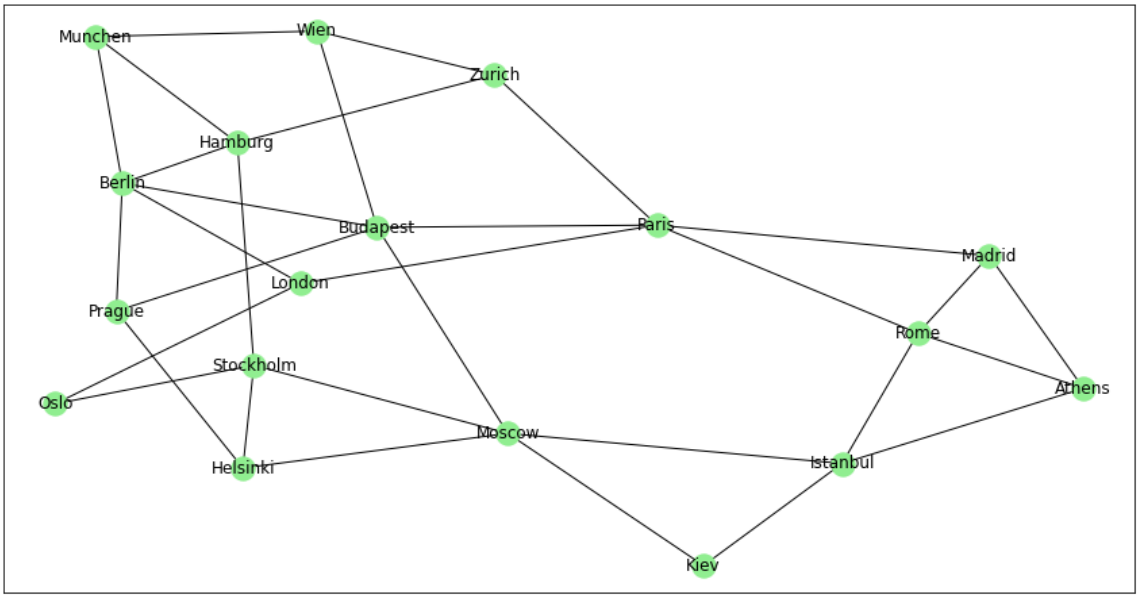
Building a graph from a CSV file *(optional)*

As an alternative solution a CSV file can be processed line-by-line with the built-in **csv** Python package:

In [8]:

```
import csv

flight_graph = nx.Graph()

csv_file = open('../data/flights.csv')
csv_reader = csv.reader(csv_file, delimiter=';')
next(csv_reader, None) # skip header line
for row in csv_reader:
    print('Reading flight {0} <=> {1}, distance: {2}km'.format(row[0], row[1], r
ow[2]))
    flight_graph.add_edge(row[0], row[1])
csv_file.close()

plt.figure(figsize=[15,8])
nx.draw_networkx(flight_graph, node_color = 'lightgreen')
plt.show()
```

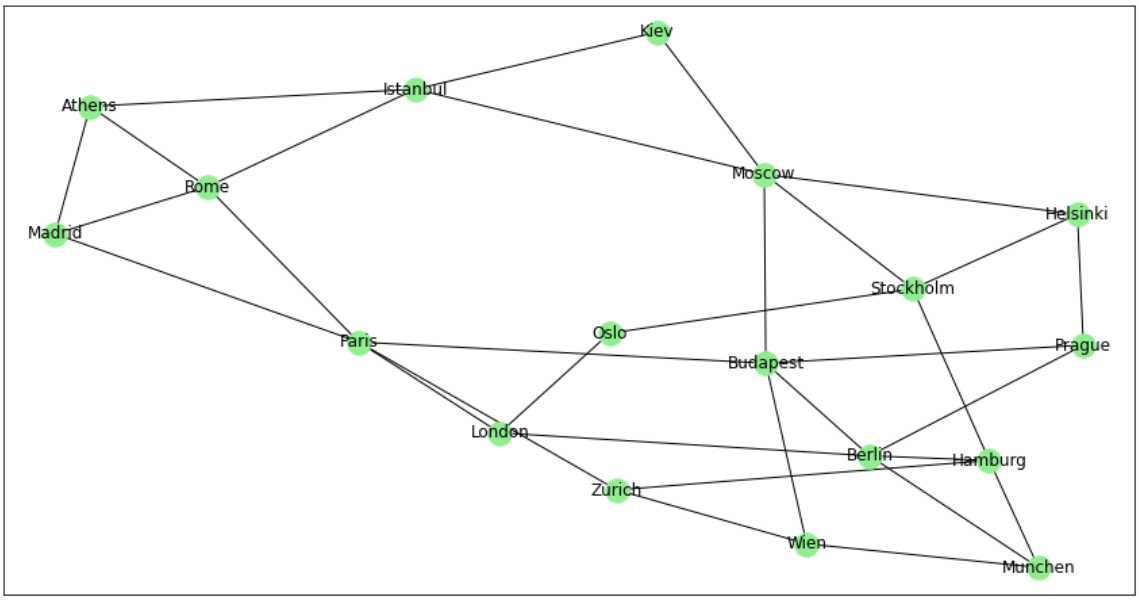
```
Reading flight London <=> Paris, distance: 342km
Reading flight London <=> Berlin, distance: 932km
Reading flight London <=> Oslo, distance: 1153km
Reading flight Paris <=> Zurich, distance: 488km
Reading flight Paris <=> Budapest, distance: 1244km
...
Reading flight Athens <=> Istanbul, distance: 562km
Reading flight Kiev <=> Istanbul, distance: 1056km
Reading flight Istanbul <=> Moscow, distance: 1755km
Reading flight Rome <=> Athens, distance: 1051km
Reading flight Kiev <=> Moscow, distance: 755km
```



Closing an opened file is easy to forget and a common programmer mistake. Use the `with` statement, which will automatically close the file (if it was successfully opened):

In [9]:

```
flight_graph = nx.Graph()

with open('../data/flights.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=';')
    next(csv_reader, None) # skip header line
    for row in csv_reader:
        #print('Reading flight {0} <=> {1}, distance: {2}km'.format(row[0], row
[1], row[2]))
        flight_graph.add_edge(row[0], row[1])

plt.figure(figsize=[15,8])
nx.draw_networkx(flight_graph, node_color = 'lightgreen')
plt.show()
```



Analyzing the graph

Querying the size and degree information

In [10]:

```
print('Number of nodes: {0}'.format(flight_graph.order()))
print('Number of edges:{0}'.format(flight_graph.size()))
print('Degrees of the nodes: {0}'.format(flight_graph.degree()))
```

```
Number of nodes: 18
Number of edges:32
Degrees of the nodes: [('London', 3), ('Paris', 5), ('Berlin', 5),
('Oslo', 2), ('Zurich', 3), ('Budapest', 5), ('Rome', 4), ('Madrid',
3), ('Athens', 3), ('Stockholm', 4), ('Helsinki', 3), ('Moscow', 5),
('Prague', 3), ('Hamburg', 4), ('Munchen', 3), ('Wien', 3), ('Istanb
ul', 4), ('Kiev', 2)]
```

For *directed graphs*, there is also `in_degree` and `out_degree` defined.

Iterate through the nodes

In [11]:

```
for node in flight_graph.nodes:
    print(node)
```

```
London
Paris
Berlin
Oslo
Zurich
Budapest
Rome
Madrid
Athens
Stockholm
Helsinki
Moscow
Prague
Hamburg
Munchen
Wien
Istanbul
Kiev
```

Note: iterating through the graph itself (`flight_graph`) is the same.

Iterate through the edges

In [12]:

```
for from_node, to_node in flight_graph.edges:
    print("{0} <=> {1}".format(from_node, to_node))
```

```
London <=> Paris
London <=> Berlin
London <=> Oslo
Paris <=> Zurich
Paris <=> Budapest
...
Moscow <=> Istanbul
Moscow <=> Kiev
Hamburg <=> Munchen
Munchen <=> Wien
Istanbul <=> Kiev
```

Query the neighbors of a node

In [13]:

```
for neighbor in flight_graph.neighbors('Budapest'):
    print(neighbor)
```

```
Paris
Berlin
Wien
Prague
Moscow
```

Pay attention that it is written as `neighbors` (American English) and *NOT* `neighbours` (British English).

Check node and edge existence

In [14]:

```
if flight_graph.has_node('Budapest'):
    print('The Budapest node exists.')
if flight_graph.has_edge('Budapest', 'Paris'):
    print('The Budapest <=> Paris edge exists.')
```

```
The Budapest node exists.
The Budapest <=> Paris edge exists.
```

Weighted graphs

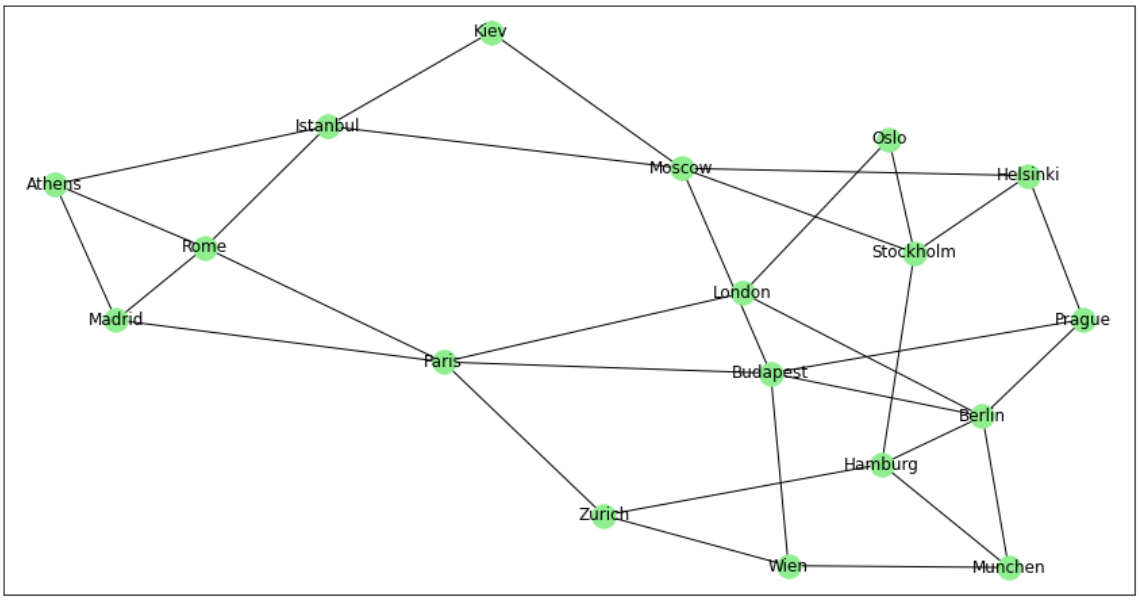
Attributes (metadata) can be assigned to the nodes and edges of a graph.

Building weighted graphs

When creating the graph from a *pandas* DataFrame, the 4th parameter of the `from_pandas_edgelist` function defines which *Series* (columns) of the *DataFrame* shall be added to the edges as attributes. If `True`, all the remaining columns will be added. If `None`, no edge attributes are added to the graph. Its default value is `None`.

In [15]:

```
flight_graph = nx.from_pandas_edgelist(flight_table, 'From city', 'To city', ['D
istance'])
plt.figure(figsize=[15,8])
nx.draw_networkx(flight_graph, node_color = 'lightgreen')
plt.show()
```



Optional: when building a graph "manually", the node and edge attributes can be passed to the `add_node` an `add_edge` methods.

In [16]:

```
flight_graph = nx.Graph()

with open('../data/flights.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=';')
    next(csv_reader, None) # skip header line
    for row in csv_reader:
        print('Reading flight {0} <=> {1}, distance: {2}km'.format(row[0], row[1
], row[2]))
        flight_graph.add_edge(row[0], row[1], dist = row[2])
```

```
Reading flight London <=> Paris, distance: 342km
Reading flight London <=> Berlin, distance: 932km
Reading flight London <=> Oslo, distance: 1153km
Reading flight Paris <=> Zurich, distance: 488km
Reading flight Paris <=> Budapest, distance: 1244km
...
Reading flight Athens <=> Istanbul, distance: 562km
Reading flight Kiev <=> Istanbul, distance: 1056km
Reading flight Istanbul <=> Moscow, distance: 1755km
Reading flight Rome <=> Athens, distance: 1051km
Reading flight Kiev <=> Moscow, distance: 755km
```

Query the edge metadata

The metadata, called the *weight* of an edge can be queried then:

In [17]:

```
print('Metadata for the Budapest <=> Paris edge: {0}'.format(flight_graph['Budap
est']['Paris']))
print('Metadata for all edges from Budapest: {0}'.format(flight_graph['Budapest'
]))
```

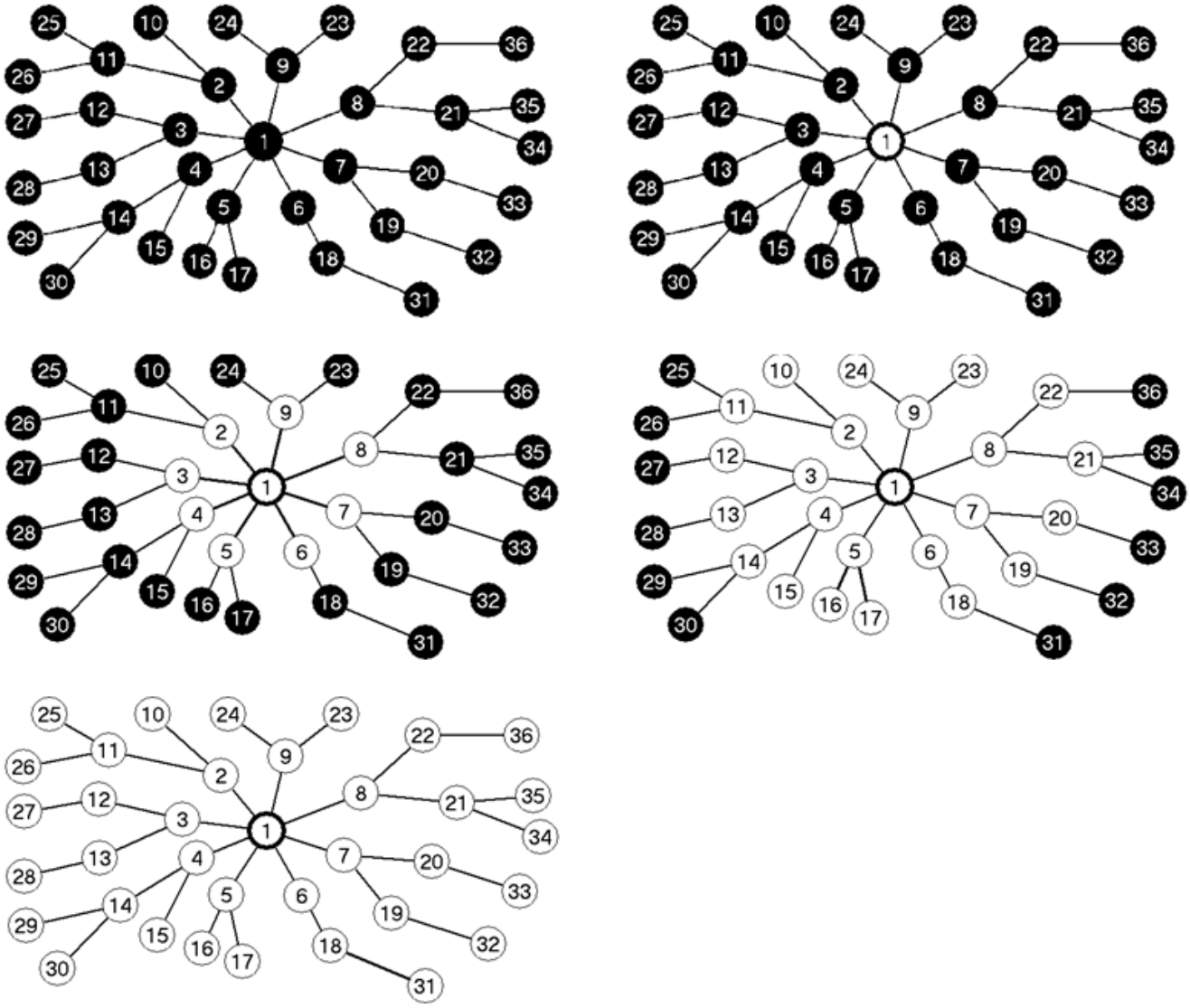
```
Metadata for the Budapest <=> Paris edge: {'dist': '1244'}
Metadata for all edges from Budapest: {'Paris': {'dist': '1244'}, 'B
erlin': {'dist': '688'}, 'Wien': {'dist': '214'}, 'Prague': {'dist':
'444'}, 'Moscow': {'dist': '1569'}}
```

Further readings

- Check out the official NetworkX tutorial (https://networkx.github.io/documentation/stable/tutorial.html).
- Browse the official NetworkX reference (https://networkx.github.io/documentation/stable/reference/index.html).

Breadth-first search

Breadth-first search (*BFS*) is an algorithm for traversing or searching a graph. It starts at some arbitrary node of a graph, and explores all the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level.



The breadth-first search traversal can be implemented with a *queue* data structure (see Chapter 7 (07_collections.pdf#Queues)).

As a showcase, let's request a starting city from the user and a number of maximum flights. Calculate which cities can be reached! Handle the case of a not existing starting city.

In [18]:

```
from collections import deque

start_city = input('Start city: ')
flight_count = int(input('Max number of flights: '))

# Check existence of start city
if flight_graph.has_node(start_city):
    ready_list = []
    process_queue = deque([(start_city, 0)])

    # Process until queue is empty
    while len(process_queue) > 0:
        # Move first item of process queue to ready list
        process_item = process_queue.popleft()
        process_city, process_dist = process_item
        ready_list.append(process_item)

        # NOTE: if process_dist > flight_count, we can halt the algorithm here,
        # all reachable cities are in the ready list
        #if process_dist > flight_count:
        #    break

        # "Expand" the processed node: add its neighbors to the process queue
        for neighbor_city in flight_graph.neighbors(process_city):
            # Only add neighbors which are not already in the ready list or the
 process_queue
            found = (neighbor_city in [city for city, dist in process_queue] or
                     neighbor_city in [city for city, dist in ready_list])

            if not found:
                process_queue.append((neighbor_city, process_dist + 1))

    # Display results
    for city, dist in ready_list:
        if dist <= flight_count:
            print(city)
else:
    print('{0} city is unknown' % start_city)
```

```
Budapest
Paris
Berlin
Wien
Prague
...
Stockholm
Istanbul
Kiev
Oslo
Athens
```

NetworkX contains several traversal algorithms (https://networkx.github.io/documentation/stable/reference/algorithms/traversal.html) out of the box, so we don't need to reimplement them.

In [19]:

```
start_city = input('Start city: ')
flight_count = int(input('Max number of flights: '))

# Check existence of start city
if flight_graph.has_node(start_city):
    reachable_cities = [ start_city ]

    # Do breadth first search
    successors = nx.bfs_successors(flight_graph, start_city, flight_count - 1)
    for item in successors:
        print('{0} -> {1}'.format(item[0], item[1]))
        reachable_cities += item[1]

    print('Reachable cities: {0}'.format(reachable_cities))
else:
    print('{0} city is unknown'.format(start_city))
```

```
Budapest -> ['Paris', 'Berlin', 'Wien', 'Prague', 'Moscow']
Paris -> ['London', 'Zurich', 'Rome', 'Madrid']
Berlin -> ['Hamburg', 'Munchen']
Prague -> ['Helsinki']
Moscow -> ['Stockholm', 'Istanbul', 'Kiev']
Reachable cities: ['Budapest', 'Paris', 'Berlin', 'Wien', 'Prague',
'Moscow', 'London', 'Zurich', 'Rome', 'Madrid', 'Hamburg', 'Munche
n', 'Helsinki', 'Stockholm', 'Istanbul', 'Kiev']
```

Chapter 14: Graph algorithms I. - shortest path

The sample dataset for this lecture is given in the `airports.csv` and `airroutes.csv` files in the `data` folder. (The column separator is the `;` character.)

- The `airports.csv` file contains information about (larger) airports all over the world:
    1. IATA code (International Air Transport Association code, e.g. *BUD* for the Budapest Airport)
    2. ICAO code (International Civil Aviation Organization code, e.g. *LHBP* for the Budapest Airport)
    3. Name
    4. Number of runways
    5. Longest runway length (in foots)
    6. Elevation (in foots)
    7. Country
    8. Country region
    9. City
    10. Latitude
    11. Longitude
- The `airroutes.csv` consists of the direct flight relations between the airports, identifying them with their IATA code. The distance of the airports / length of the flight route is also given (in miles). The flights are directed, if there is a flight between both directions of two airports, then there will be two records in the file, with opposite direction.

Reading the dataset

Read the airport data

First read the airports data into a pandas *DataFrame*.

In [1]:

```
import pandas as pd

airports = pd.read_csv('../data/airports.csv', delimiter = ';')
display(airports)
```

	iata	icao	name	runways	longest	elevation	country	region	city	
0	ATL	KATL	Hartsfield - Jackson Atlanta International Air...	5	12390	1026	US	US-GA	Atlanta	33
1	ANC	PANC	Anchorage Ted Stevens	3	12400	151	US	US-AK	Anchorage	61
2	AUS	KAUS	Austin Bergstrom International Airport	2	12250	542	US	US-TX	Austin	30
3	BNA	KBNA	Nashville International Airport	4	11030	599	US	US-TN	Nashville	36
4	BOS	KBOS	Boston Logan	6	10083	19	US	US-MA	Boston	42
...	...	...	...	...	...	...	...	...	...	
3459	LNL	ZLLN	Cheng Xian Airport	1	9186	3707	CN	CN-62	Longnan	33
3460	XAI	ZHXY	Xinyang Minggang Airport	1	8858	4528	CN	CN-41	Xinyang	32
3461	YYA	ZGYY	Sanhe Airport	1	8530	230	CN	CN-43	Yueyang	29
3462	BQJ	UEBB	Batagay Airport	2	6562	699	RU	RU-SA	Batagay	67
3463	DPT	UEBD	Deputatskij Airport	1	7021	920	RU	RU-SA	Deputatskij	69

3464 rows × 11 columns

Note: the length of the longest runway and the elevation is given in foots.

Lets set the column `iata` as the index column, so each row of data will be accessible later by indexing the airports with their IATA code.

In [2]:

```
airports.set_index('iata', inplace=True)
display(airports)
```

iata	icao	name	runways	longest	elevation	country	region	city	la
ATL	KATL	Hartsfield - Jackson Atlanta International Air...	5	12390	1026	US	US-GA	Atlanta	33.63670
ANC	PANC	Anchorage Ted Stevens	3	12400	151	US	US-AK	Anchorage	61.17440
AUS	KAUS	Austin Bergstrom International Airport	2	12250	542	US	US-TX	Austin	30.19450
BNA	KBNA	Nashville International Airport	4	11030	599	US	US-TN	Nashville	36.12450
BOS	KBOS	Boston Logan	6	10083	19	US	US-MA	Boston	42.36430
...	...	...	...	...	...	...	...	...	.
LNL	ZLLN	Cheng Xian Airport	1	9186	3707	CN	CN-62	Longnan	33.78972
XAI	ZHXY	Xinyang Minggang Airport	1	8858	4528	CN	CN-41	Xinyang	32.54055
YYA	ZGYY	Sanhe Airport	1	8530	230	CN	CN-43	Yueyang	29.31250
BQJ	UEBB	Batagay Airport	2	6562	699	RU	RU-SA	Batagay	67.64777
DPT	UEBD	Deputatskij Airport	1	7021	920	RU	RU-SA	Deputatskij	69.39250

3464 rows × 10 columns

Reminder: the `set_index` function can be configured to modify the index in place or return a new *Dataframe* with the `inplace` parameter (defaults to `False`). It can also be configured to drop or keep the index column with the `drop` parameter (defaults to `True`).

The information of the Budapest Airport can now be accessed both by numerical and associative (string) indexing:

In [3]:

```
print('The Budapest airport by the numerical index:')
print(airports.iloc[111])
print()
print('The Budapest airport by the associative index:')
print(airports.loc['BUD'])
```

```
The Budapest airport by the numerical index:
icao                                                 LHBP
name         Budapest Ferenc Liszt International Airport
runways                                                 2
longest                                             12162
elevation                                             495
country                                                HU
region                                              HU-PE
city                                             Budapest
lat                                             47.436901
lon                                               19.2556
Name: BUD, dtype: object

The Budapest airport by the associative index:
icao                                                 LHBP
name         Budapest Ferenc Liszt International Airport
runways                                                 2
longest                                             12162
elevation                                             495
country                                                HU
region                                              HU-PE
city                                             Budapest
lat                                             47.436901
lon                                               19.2556
Name: BUD, dtype: object
```

The number of runways the Budapest Airport can be fetched (or modified) now 4 possible ways:

In [4]:

```
print(airports.iloc[111]['runways'])
print(airports.loc['BUD']['runways'])
print(airports['runways'][111])
print(airports['runways']['BUD'])
```

```
2
2
2
2
```

Read the airroutes data

In [5]:

```
airroutes = pd.read_csv('../data/airroutes.csv', delimiter = ';')
display(airroutes)
```

	from	to	distance
0	ATL	AUS	811
1	ATL	BNA	214
2	ATL	BOS	945
3	ATL	BWI	576
4	ATL	DCA	546
...	...	...	...
50225	NRR	CPX	23
50226	LNL	PKX	708
50227	XAI	PKX	498
50228	YYA	PKX	726
50229	BQJ	DPT	178

50230 rows × 3 columns

Note: the distance is given in miles.

Build a graph

NetworkX has an integrated conversion for *pandas* DataFrames which can be used.
Lets create a directed graph (`networkx.DiGraph`) from the flights. The edges shall be weighted with the distance of the routes.

In [6]:

```
import networkx as nx

flight_graph = nx.from_pandas_edgelist(airroutes, 'from', 'to', ['distance'], cr
eate_using = nx.DiGraph)

print('Metadata for the BUD -> JFK edge: {0}'.format(flight_graph['BUD']['JFK'
]))
```

```
Metadata for the BUD -> JFK edge: {'distance': 4356}
```

Reminder: The 4th parameter defines which *Series* (columns) of the *DataFrame* shall be added to the edges as attributes. If `True` , all of the remaining columns will be added. If `None` , no edge attributes are added to the graph. Its default value is `None` .

Calculating the shortest path

NetworkX supports various [shortest path algorithms (https://networkx.org/documentation/stable/reference/algorithms/shortest_paths.html)]:

- *Dijkstra* and *Bellman-Ford* algorithm to compute shortest path between source and all other reachable nodes;
- *Floyd-Warshall* algorithm to find the shortest path between all node pairs.

Beside the algorithm-specific functions, NetworkX also provides a uniform interface to calculate the shortest paths from a starting point to a target (or to all):

```
nx.shortest_path(graph, source, target, weight, method)
```

The default algorithm is *Dijsktra*.

Example

Calculate the path between 2 user given airports with the minimal number of transfers.

In [7]:

```
from_airport = input("From airport: ")
to_airport = input("To airport: ")

if flight_graph.has_node(from_airport) and flight_graph.has_node(to_airport):
    route = nx.shortest_path(flight_graph, from_airport, to_airport)
    print("Route: {0}".format(route))

    length = 0
    for i in range(1, len(route)):
        length += flight_graph[route[i-1]][route[i]]['distance']
    print("Length: {0} mi".format(length))
else:
    print("Source or destination airport was not found.")
```

```
Route: ['BUD', 'JFK', 'CAN', 'PKX', 'YYA']
Length: 14194 mi
```

Calculate the shortest path by distance between 2 user given airports.

In [8]:

```
from_airport = input("From airport: ")
to_airport = input("To airport: ")

if flight_graph.has_node(from_airport) and flight_graph.has_node(to_airport):
    route = nx.dijkstra_path(flight_graph, from_airport, to_airport, 'distance')
    length = nx.dijkstra_path_length(flight_graph, from_airport, to_airport, 'di
stance')
    print("Route: {0} ({1} mi)".format(route, length))
else:
    print("Source or destination airport was not found.")
```

```
Route: ['BUD', 'SVO', 'HET', 'PKX', 'YYA'] (5339 mi)
```

Calculate the shortest between 2 user given airports by distance, but with the following additional conditions:

- airports with no longer runway than 8000 feets cannot be used;
- airports with only 1 runway has a 50% penalty of the distance.

In [9]:

```
def custom_distance(from_node, to_node, edge_attr):
    if airports.loc[to_node]['longest'] < 8000:
        return None
    if airports.loc[to_node]['runways'] == 1:
        return edge_attr['distance'] * 1.5
    return edge_attr['distance']

from_airport = input("From airport: ")
to_airport = input("To airport: ")

if flight_graph.has_node(from_airport) and flight_graph.has_node(to_airport):
    route = nx.dijkstra_path(flight_graph, from_airport, to_airport, custom_dist
ance)
    length = nx.dijkstra_path_length(flight_graph, from_airport, to_airport, cus
tom_distance)
    print("Route: {0} ({1} mi)".format(route, length))
else:
    print("Source airport was not found.")
```

```
Route: ['BUD', 'SVO', 'HET', 'PKX', 'YYA'] (5702.0 mi)
```

Calculate which airports can be reached from a starting, user given airport within a reach of also user given distance (in miles). Also compute the shorthest path by distance to each of them.

In [12]:

```
from_airport = input("From airport: ")
max_distance = int(input("Max distance: "))

if flight_graph.has_node(from_airport):
    lengths, routes = nx.single_source_dijkstra(flight_graph, from_airport, None
, max_distance, 'distance')
    for to_airport in routes.keys():
        print("{0} -> {1}: {2} ({3} mi)".format(from_airport, to_airport, routes
[to_airport], lengths[to_airport]))
else:
    print("Source airport was not found.")
```

```
JFK -> JFK: ['JFK'] (0 mi)
JFK -> BOS: ['JFK', 'BOS'] (186 mi)
JFK -> BWI: ['JFK', 'BWI'] (184 mi)
JFK -> DCA: ['JFK', 'DCA'] (213 mi)
JFK -> IAD: ['JFK', 'IAD'] (227 mi)
JFK -> PHL: ['JFK', 'PHL'] (93 mi)
JFK -> PWM: ['JFK', 'PWM'] (273 mi)
JFK -> ROC: ['JFK', 'ROC'] (263 mi)
JFK -> ORF: ['JFK', 'ORF'] (290 mi)
JFK -> BUF: ['JFK', 'BUF'] (300 mi)
JFK -> RIC: ['JFK', 'RIC'] (288 mi)
JFK -> SYR: ['JFK', 'SYR'] (208 mi)
JFK -> BTV: ['JFK', 'BTV'] (266 mi)
JFK -> ORH: ['JFK', 'ORH'] (149 mi)
JFK -> ACK: ['JFK', 'ACK'] (198 mi)
JFK -> HYA: ['JFK', 'HYA'] (195 mi)
JFK -> LGA: ['JFK', 'PHL', 'LGA'] (187 mi)
JFK -> HPN: ['JFK', 'PHL', 'HPN'] (207 mi)
JFK -> EWR: ['JFK', 'PHL', 'EWR'] (172 mi)
JFK -> MDT: ['JFK', 'PHL', 'MDT'] (176 mi)
JFK -> BDL: ['JFK', 'PHL', 'EWR', 'BDL'] (287 mi)
JFK -> ISP: ['JFK', 'PHL', 'ISP'] (222 mi)
JFK -> SWF: ['JFK', 'PHL', 'SWF'] (220 mi)
JFK -> ABE: ['JFK', 'PHL', 'ABE'] (148 mi)
JFK -> AVP: ['JFK', 'PHL', 'AVP'] (197 mi)
JFK -> PHF: ['JFK', 'PHL', 'PHF'] (294 mi)
JFK -> ELM: ['JFK', 'PHL', 'ELM'] (273 mi)
JFK -> SCE: ['JFK', 'PHL', 'SCE'] (246 mi)
JFK -> BGM: ['JFK', 'PHL', 'BGM'] (259 mi)
JFK -> ITH: ['JFK', 'PHL', 'ITH'] (285 mi)
JFK -> HVN: ['JFK', 'PHL', 'HVN'] (249 mi)
JFK -> IPT: ['JFK', 'PHL', 'IPT'] (222 mi)
JFK -> SBY: ['JFK', 'PHL', 'SBY'] (200 mi)
JFK -> LEB: ['JFK', 'BOS', 'LEB'] (295 mi)
JFK -> MVY: ['JFK', 'ACK', 'MVY'] (228 mi)
JFK -> PVC: ['JFK', 'BOS', 'PVC'] (231 mi)
JFK -> EWB: ['JFK', 'ACK', 'EWB'] (253 mi)
JFK -> HGR: ['JFK', 'IAD', 'HGR'] (282 mi)
```

Calculate which cities can be reached from a starting, user given city within a reach of also user given distance (in miles).

In [13]:

```
from_city = input("From city: ")
max_distance = int(input("Max distance: "))

from_airports = airports[airports['city'] == from_city].index
result = []
for from_airport in from_airports:
    routes = nx.single_source_dijkstra_path(flight_graph, from_airport, max_dist
ance, 'distance')
    to_airports = routes.keys()
    to_cities = [airports.loc[ap]['city'] for ap in to_airports]
    result += to_cities

result_unique = set(result) # remove duplicates
print(sorted(result_unique)) # sort the printed result
```

```
['Allentown', 'Baltimore', 'Binghamton', 'Boston', 'Buffalo', 'Burli
ngton', 'Elmira/Corning', 'Hagerstown', 'Harrisburg', 'Hartford', 'H
yannis', 'Islip', 'Ithaca', 'Lebanon', 'Manchester', "Martha's Viney
ard", 'Nantucket', 'New Bedford', 'New Haven', 'New York', 'Newark',
'Newburgh', 'Newport News', 'Norfolk', 'Philadelphia', 'Portland',
'Provincetown', 'Richmond', 'Rochester', 'Salisbury', 'State Colleg
e', 'Syracuse', 'Washington D.C.', 'White Plains', 'Wilkes-Barre/Scr
anton', 'Williamsport', 'Worcester']
```

Chapter 15: Graph algorithms II. - minimum spanning tree

Read the `hungary_cities.shp` shapefile located in the `data` folder. This dataset contains both scalar and spatial data of the Hungarian cities:

1. City Id
2. City Name
3. County Name
4. Status (town, city, independent city, national capital, capital district)
5. KSH code (unique statistical code for the city)

Source: *ELTE FI, Institute of Cartography and Geoinformatics*

In [1]:

```
import geopandas as gpd

cities = gpd.read_file('../data/hungary_cities.shp')
display(cities)
```

	Id	County	City	Status	KSH	geometry
0	1	FEJÉR	Aba	town	17376	POINT (610046.800 187639.000)
1	2	BARANYA	Abaliget	town	12548	POINT (577946.100 89280.800)
2	3	HEVES	Abasár	town	24554	POINT (721963.700 273880.300)
3	4	BORSOD-ABAUJ-ZEMPLÉN	Abaújalpár	town	15662	POINT (812129.200 331508.200)
4	5	BORSOD-ABAUJ-ZEMPLÉN	Abaújkér	town	26718	POINT (809795.600 331138.300)
...	...	...	...	...	...	...
3142	3143	GYÕR-MOSON-SOPRON	Zsira	town	04622	POINT (471324.200 237577.200)
3143	3144	CSONGRÁD	Zsombó	town	17765	POINT (721098.100 109690.000)
3144	3145	BORSOD-ABAUJ-ZEMPLÉN	Zsujta	town	11022	POINT (815027.400 353143.100)
3145	3146	SZABOLCS-SZATMÁR-BEREG	Zsurk	town	13037	POINT (884847.700 344952.800)
3146	3147	BORSOD-ABAUJ-ZEMPLÉN	Zubogy	town	19105	POINT (763123.300 338338.600)

3147 rows × 6 columns

The correct encoding of the file should be automatically detected. In case the Hungarian characters are displayed incorrectly, you may specify the encoding manually:

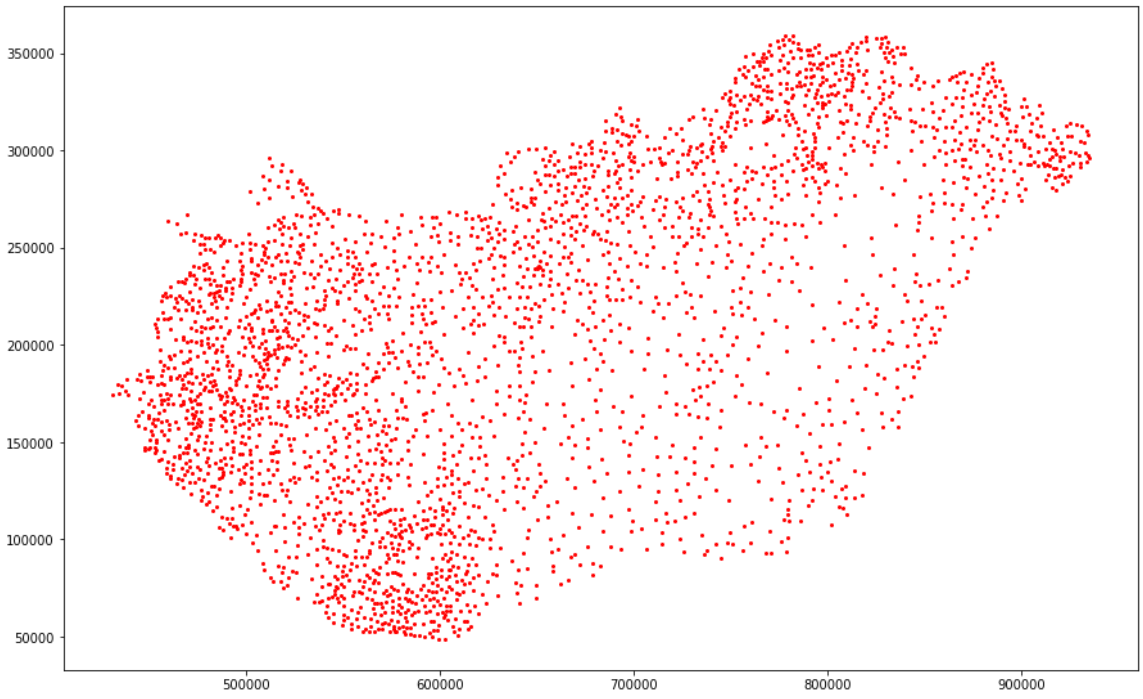
```
cities = gpd.read_file('../data/hungary_cities.shp', encoding='latin1')
```

Visualize the *GeoDataFrame*

Plot the location of all Hungarian cities:

In [2]:

```
import matplotlib.pyplot as plt
%matplotlib inline

cities.plot(figsize=[15,10], color='red', markersize=4)
plt.show()
```

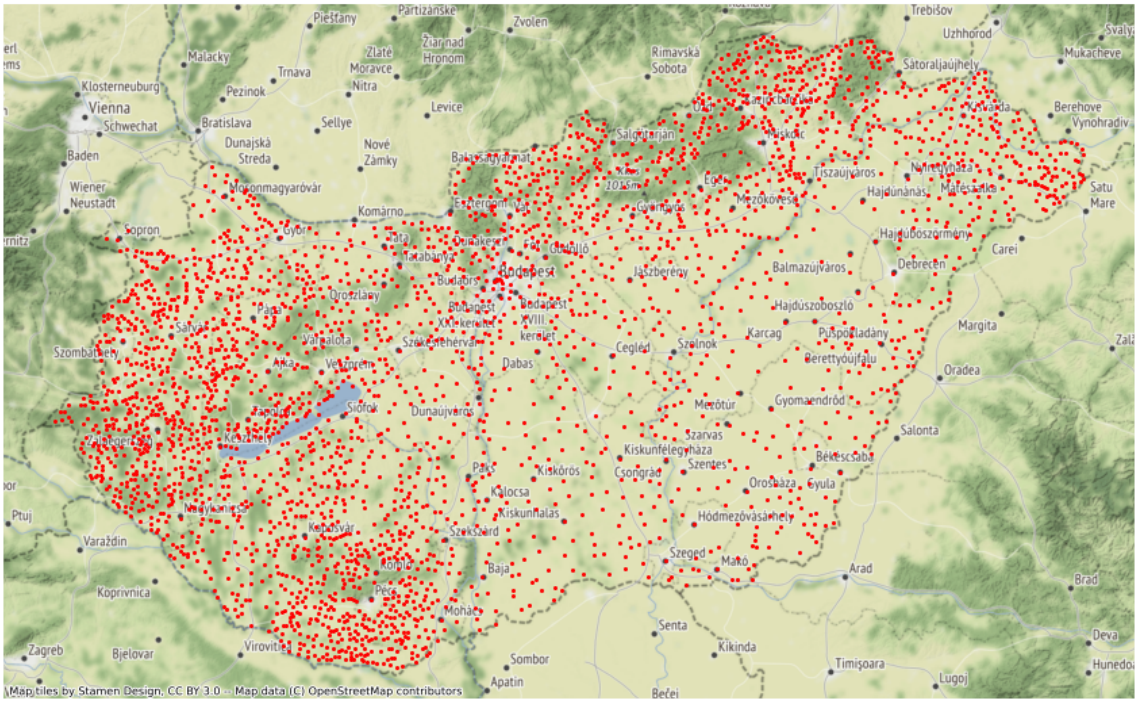


Add a raster base map to it with the *contextily* package:

In [3]:

```
import contextily as ctx

# Display the CRS
print(cities.crs)

# Set the CRS to EOV projection (EPSG:23700) if None
if(cities.crs == None):
    cities.set_crs('epsg:23700', inplace=True)

# Display the CRS
print(cities.crs)

# Transform the GeoDataFrame to Web Mercator projection (EPSG:3857) to display c
orrectly with the base map
ax = cities.to_crs('epsg:3857').plot(figsize=[15,10], color='red', markersize=4)
ax.set_axis_off()
ctx.add_basemap(ax)
plt.show()
```

```
None
epsg:23700
```



Create a minimum spanning tree

NetworkX supports both the *Prim* and the *Kruskal* algorithm for building a minimum / maximum spanning tree (https://networkx.org/documentation/stable/reference/algorithms/tree.html#module-networkx.algorithms.tree.mst), with a uniform interface. The default is *Kruskal*.

```
nx.minimum_spanning_tree(graph, weight, algorithm)
```

Example

Step 1: Create an undirected graph with the towns as the nodes.

In [4]:

```
import networkx as nx

# Create empty, undirected graph
graph = nx.Graph()

for index, row in cities.iterrows():
     graph.add_node(row['City'],
        county = row['County'],
        status = row['Status'],
        ksh_code = row['KSH'],
        location = row['geometry']
    )

# Check results
print(graph.nodes['Esztergom'])
```

```
{'county': 'KOMÁROM-ESZTERGOM', 'status': 'city', 'ksh_code': '2513
1', 'location': <shapely.geometry.point.Point object at 0x7f45966344
c0>}
```

Display the location in WKT format:

In [5]:

```
print(graph.nodes['Esztergom']['location'].wkt)
```

```
POINT (627140 272097.8)
```

Fetch the *(X,Y)* coordinates form the location:

In [6]:

```
print(graph.nodes['Esztergom']['location'].x)
print(graph.nodes['Esztergom']['location'].y)
```

```
627140.0
272097.8
```

Calculate the location between 2 cities with the *Pythagoras theorem*:

In [7]:

```
import math

def dist(loc_a, loc_b):
    return math.sqrt(math.pow(loc_a.x - loc_b.x, 2) +
                     math.pow(loc_a.y - loc_b.y, 2))


print(dist(graph.nodes['Esztergom']['location'], graph.nodes['Budapest']['locati
on']))
```

```
39476.19752399673
```

The *Point* type has a built-in `distance()` method to do that:

In [8]:

```
print(graph.nodes['Esztergom']['location'].distance(graph.nodes['Budapest']['loc
ation']))
```

```
39476.19752399673
```

Step 2: Create a complete graph (add all possible edges).

In [9]:

```
import math

for city_from in graph.nodes:
    location_from = graph.nodes[city_from]['location']
    for city_to in graph.nodes:
        location_to = graph.nodes[city_to]['location']
        if city_from < city_to: # we do not need to add all edges twice
            # Add edge to the graph with distance as its cost
            graph.add_edge(city_from, city_to,
                distance = graph.nodes[city_from]['location'].distance(graph.nod
es[city_to]['location']))

# Check results
print(graph['Esztergom']['Debrecen'])
```

```
{'distance': 218626.45554703576}
```

Step 3: Calculate the minimum spanning tree as a new graph.

In [10]:

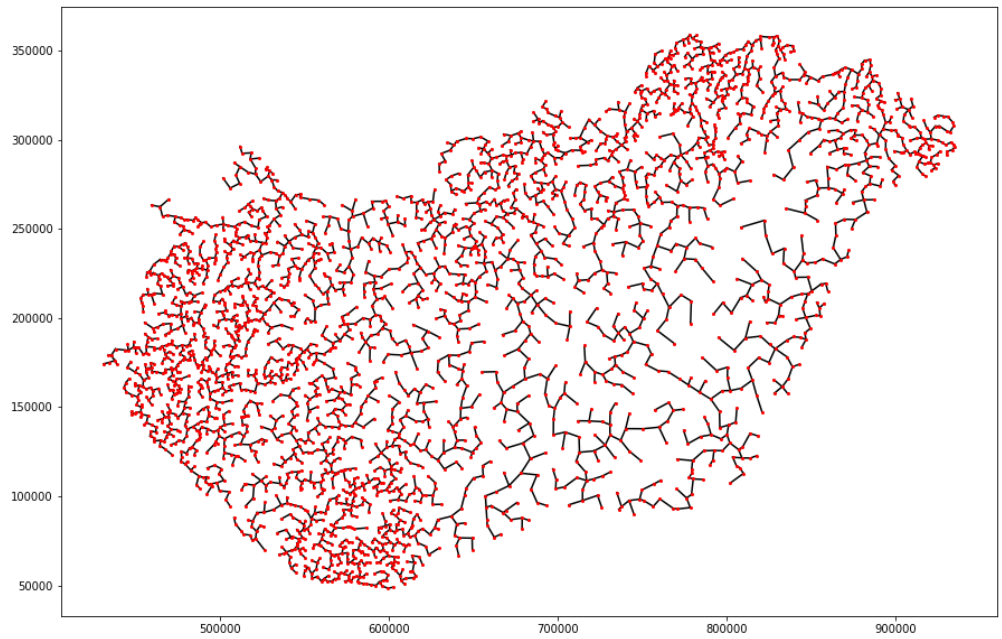
```
print('Number of nodes in original graph: {0}'.format(graph.order()))
print('Number of edges in original graph: {0}'.format(graph.size()))

spanning_tree = nx.minimum_spanning_tree(graph, weight = 'distance')

print('Number of nodes in spanning tree: {0}'.format(spanning_tree.order()))
print('Number of edges in spanning tree: {0}'.format(spanning_tree.size()))
```

```
Number of nodes in original graph: 3147
Number of edges in original graph: 4950231
Number of nodes in spanning tree: 3147
Number of edges in spanning tree: 3146
```

Step 4: Visualize results.

In [11]:

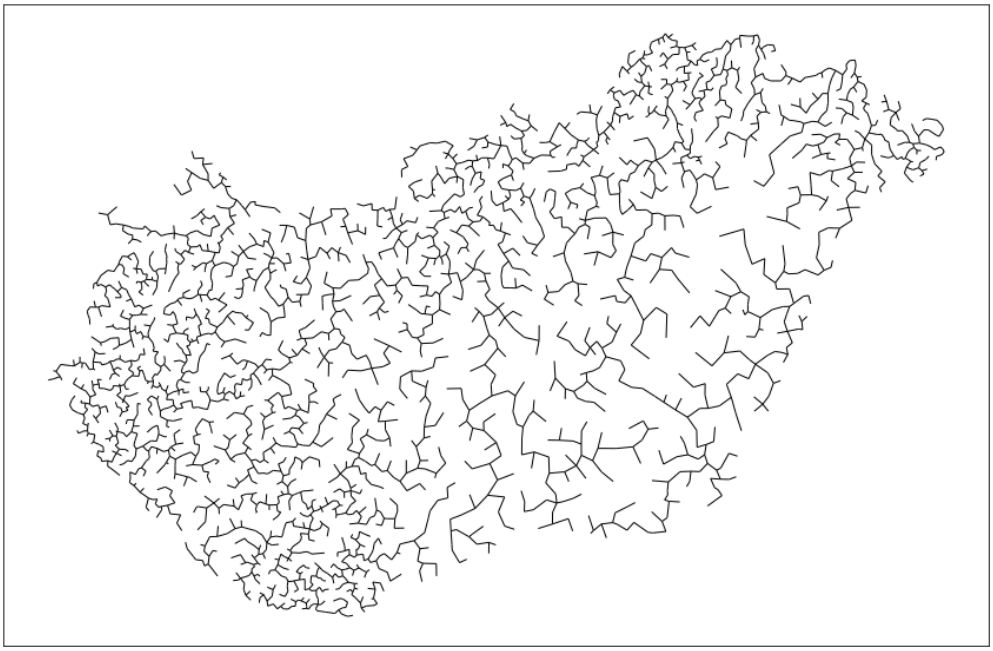
```
# Start new plot figure
plt.figure(figsize=[15,10])

# Plot all edges as black lines in the MST
for edge in spanning_tree.edges:
    city_from = edge[0]
    city_to   = edge[1]

    location_from = spanning_tree.nodes[city_from]['location']
    location_to   = spanning_tree.nodes[city_to]['location']
    plt.plot([location_from.x, location_to.x], [location_from.y, location_to.y],
 color='black')

# Plot all cities as red dots
for city in spanning_tree.nodes:
    location = spanning_tree.nodes[city]['location']
    plt.plot(location.x, location.y, color='red', marker='o', markersize=2)

# Display plot
plt.show()
```



Alternative approach: use NetworkX to draw the plot.

In [12]:

```
# Add all city coordinates a tuples to the nodes of the graph.
for node in spanning_tree.nodes:
    spanning_tree.nodes[node]['coords'] = spanning_tree.nodes[node]['location'].
coords[0]

# Visualize the spanning tree, using the positions in the coords field.
plt.figure(figsize=[15,10])
nx.draw_networkx(spanning_tree, nx.get_node_attributes(spanning_tree, 'coords'),
 with_labels=False, node_size=0)
plt.show()
```



Chapter 16: Spatial indexing

Package installation

This chapter covers spatial indexing with *KD-trees*, *Quadtrees* and *R-trees*. The package requirement for these spatial indexes are the `scipy.spatial`, `pyqtree` and `rtree` modules respectively.

Anaconda

If you have Anaconda installed, the `scipy` package was installed together with, you only need to install `pyqtree` and `rtree`. Open the *Anaconda Prompt* and type in:

```
conda install -c conda-forge pyqtree rtree
```

Python Package Installer (pip)

If you have standalone Python3 and Jupyter Notebook install, open a command prompt / terminal and type in:

```
pip3 install scipy pyqtree rtree
```

You most likely have already installed `rtree`, as it was an optional dependency for `geopandas` in Chapter 11 (11_spatial_vector.pdf).

Process the dataset

Read the `hungary_cities.shp` shapefile located in the `data` folder. This dataset contains both scalar and spatial data of the Hungarian cities, and should be familiar from Chapter 15 (15_graph_spanning_tree.pdf).

In [1]:

```
import geopandas as gpd

cities = gpd.read_file('../data/hungary_cities.shp')
display(cities)
```

	Id	County	City	Status	KSH	geometry
0	1	FEJÉR	Aba	town	17376	POINT (610046.800 187639.000)
1	2	BARANYA	Abaliget	town	12548	POINT (577946.100 89280.800)
2	3	HEVES	Abasár	town	24554	POINT (721963.700 273880.300)
3	4	BORSOD-ABAUJ-ZEMPLÉN	Abaújalpár	town	15662	POINT (812129.200 331508.200)
4	5	BORSOD-ABAUJ-ZEMPLÉN	Abaújkér	town	26718	POINT (809795.600 331138.300)
...	...	...	...	...	...	...
3142	3143	GYÕR-MOSON-SOPRON	Zsira	town	04622	POINT (471324.200 237577.200)
3143	3144	CSONGRÁD	Zsombó	town	17765	POINT (721098.100 109690.000)
3144	3145	BORSOD-ABAUJ-ZEMPLÉN	Zsujta	town	11022	POINT (815027.400 353143.100)
3145	3146	SZABOLCS-SZATMÁR-BEREG	Zsurk	town	13037	POINT (884847.700 344952.800)
3146	3147	BORSOD-ABAUJ-ZEMPLÉN	Zubogy	town	19105	POINT (763123.300 338338.600)

3147 rows × 6 columns

Minimal bounding box

Calculate the minimal bounding box for all the points! (We will use it later.)

In [2]:

```
def get_x(point):
    return point.x

def get_y(point):
    return point.y

# Calculating the minimal bounding box
min_x = min(cities['geometry'], key = get_x).x # or cities.geometry
max_x = max(cities['geometry'], key = get_x).x
min_y = min(cities['geometry'], key = get_y).y
max_y = max(cities['geometry'], key = get_y).y

print("Bounding box: ({0:.1f}, {1:.1f}) - ({2:.1f}, {3:.1f})".format(min_x, min_
y, max_x, max_y))
```

```
Bounding box: (431339.2, 48431.5) - (934944.4, 359044.9)
```

Lambda functions *(optional)*

Python lambdas are little, anonymous functions, subject to a more restrictive but more concise syntax than regular Python functions.

Lambda functions can have any number of arguments but only one expression. The evaluated expression is the return value of the function.

A lambda function in python has the following syntax:

```
lambda arguments: expression
```

Lambda functions can be used wherever function objects are required.

In [3]:

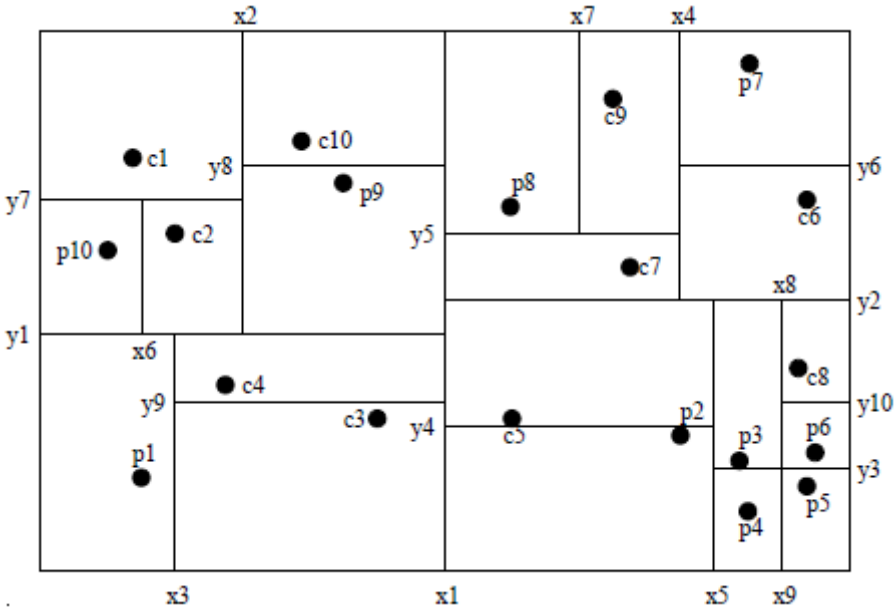
```
# Calculating the minimal bounding box
min_x = min(cities['geometry'], key = lambda p: p.x).x
max_x = max(cities['geometry'], key = lambda p: p.x).x
min_y = min(cities['geometry'], key = lambda p: p.y).y
max_y = max(cities['geometry'], key = lambda p: p.y).y

print("Bounding box: ({0:.1f}, {1:.1f}) - ({2:.1f}, {3:.1f})".format(min_x, min_
y, max_x, max_y))
```

```
Bounding box: (431339.2, 48431.5) - (934944.4, 359044.9)
```

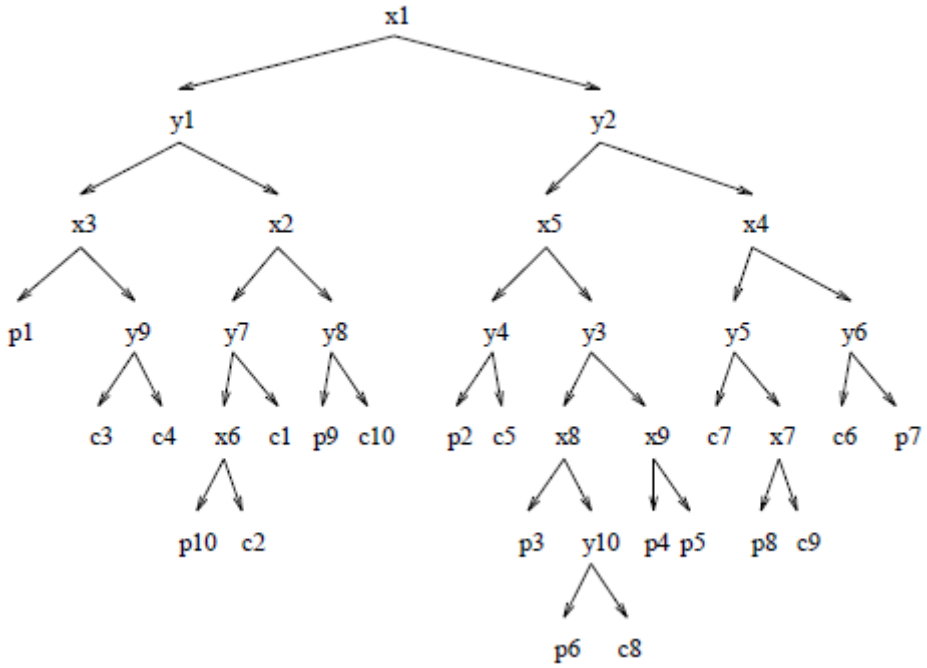
KdTree

A [kdTree (https://en.wikipedia.org/wiki/K-d_tree)](https://en.wikipedia.org/wiki/K-d_tree) (short for *k-dimensional tree*) is a space-partitioning data structure for organizing points in a k-dimensional space. KdTrees are especially useful for searches involving a multidimensional search key, e.g. nearest neighbor searches and range searches.

Example KdTree:



Representation:



Select a random city and create a point which we will query later.

In [4]:

```
import random
random.seed(42) # for reproducibility

idx = random.randint(0, len(cities) - 1)
city = cities.iloc[idx]
print(city)
```

```
Id                              2620
County                          PEST
City                      Szigethalom
Status                           town
KSH                             13277
geometry    POINT (646998.8 219076.5)
Name: 2619, dtype: object
```

Create the *query point*, by slightly distorting the location of the selected city.

In [5]:

```
from shapely.geometry import Point

city_point = city.geometry
query_point = Point(city_point.x + 1, city_point.y + 2)

print("City location: {0}".format(city_point))
print("Query location: {0}".format(query_point))
```

```
City location: POINT (646998.8 219076.5)
Query location: POINT (646999.8 219078.5)
```

Construct the KD-Tree

The `scipy` module can construct KD-Tree from a list of points, where each point is represented by a 2 element list or tuple.

In [6]:

```
points = [(p.x, p.y) for p in cities['geometry']]
print(points[:10])
```

```
[(610046.8, 187639.0), (577946.1, 89280.8), (721963.7, 273880.3), (8
12129.2, 331508.2), (809795.6, 331138.3), (791113.0, 341953.5), (808
664.6, 328230.8), (792853.4, 338292.6), (817486.0, 356056.1), (76721
4.3, 237868.5)]
```

Now the *KD-Tree* can be constructed.

In [7]:

```
import scipy.spatial
kdtree = scipy.spatial.KDTree(points)
```

Pointwise query

Query the closest neighbor to the query point.

In [8]:

```
print("City location: {0}".format(city_point))
print("Query location: {0}".format(query_point))

dist, idx = kdtree.query(query_point)

print("Closest neighbor: distance = {0:.4f}, index = {1}, point = {2}".format(di
st, idx, points[idx]))
print("Closest neighbor city: {0}".format(cities.iloc[idx]['City']))
```

```
City location: POINT (646998.8 219076.5)
Query location: POINT (646999.8 219078.5)
Closest neighbor: distance = 2.2361, index = 2619, point = (646998.
8, 219076.5)
Closest neighbor city: Szigethalom
```

Query the 3 closest neighbors to the query point.

In [9]:

```
distances, indices = kdtree.query(query_point, k = 3)

print("Query location: {0}".format(query_point))
print("3 closest neighbors:")
for i in range(len(indices)):
    idx = indices[i]
    dist = distances[i]
    print("{0}. neighbor: distance = {1:.4f}, index = {2}, point = {3}, city =
{4}".format(i+1, dist, idx, points[idx], cities.iloc[idx]['City']))
```

```
Query location: POINT (646999.8 219078.5)
3 closest neighbors:
1. neighbor: distance = 2.2361, index = 2619, point = (646998.8, 219
076.5), city = Szigethalom
2. neighbor: distance = 3087.9825, index = 2864, point = (643968.9,
219669.5), city = Tököl
3. neighbor: distance = 3250.9858, index = 2622, point = (649095.2,
221564.1), city = Szigetszentmiklós
```

Query the 50 closest neighbors to the query point within 10km.

In [10]:

```
distances, indices = kdtree.query(query_point, k = 50, distance_upper_bound = 10
000)
print("Distance list: %s" % distances)
print("Index list: %s" % indices)
```

```
Distance list: [2.23606798e+00 3.08798248e+03 3.25098578e+03 4.37588
711e+03
 5.82197378e+03 5.82989477e+03 6.07146723e+03 6.09635922e+03
 6.88039701e+03 7.86611138e+03 8.30077594e+03 8.36841190e+03
 9.92969764e+03            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf]
Index list: [2619 2864 2622 2678 1619  971  660 2618 2547  646  733
586   95 3147
 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 31
47
 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 31
47
 3147 3147 3147 3147 3147 3147 3147 3147]
```

Most likely will only find less than 50 neighbors in a 10km range, but the index list has still 50 elements. For the invalid elements the `indices[i]` is not a valid index, but instead equals to `len(cities)`. So with a simple check we can detect the end of the valid results.

In [11]:

```
valid_indices = [idx for idx in indices if idx < len(cities)]
print(valid_indices)
```

```
[2619, 2864, 2622, 2678, 1619, 971, 660, 2618, 2547, 646, 733, 586,
95]
```

In [12]:

```
print("50 closest neighbors within 10km:")
for i in range(len(valid_indices)):
    idx = valid_indices[i]
    dist = distances[i]
    print("{0}. neighbor: distance = {1:.1f}, index = {2}, location = {3}, city
 = {4}".format(i+1, dist, idx, points[idx], cities.iloc[idx]['City']))
```

```
50 closest neighbors within 10km:
1. neighbor: distance = 2.2, index = 2619, location = (646998.8, 219
076.5), city = Szigethalom
2. neighbor: distance = 3088.0, index = 2864, location = (643968.9,
219669.5), city = Tököl
3. neighbor: distance = 3251.0, index = 2622, location = (649095.2,
221564.1), city = Szigetszentmiklós
4. neighbor: distance = 4375.9, index = 2678, location = (651262.9,
220065.6), city = Taksony
5. neighbor: distance = 5822.0, index = 1619, location = (646007.0,
213341.8), city = Majosháza
6. neighbor: distance = 5829.9, index = 971, location = (644860.0, 2
24501.5), city = Halásztelek
7. neighbor: distance = 6071.5, index = 660, location = (651533.1, 2
15039.7), city = Dunavarsány
8. neighbor: distance = 6096.4, index = 2618, location = (643902.1,
213827.8), city = Szigetcsép
9. neighbor: distance = 6880.4, index = 2547, location = (640127.5,
218744.8), city = Százhalombatta
10. neighbor: distance = 7866.1, index = 646, location = (653626.9,
223316.1), city = Dunaharaszti
11. neighbor: distance = 8300.8, index = 733, location = (640833.1,
224635.0), city = Érd
12. neighbor: distance = 8368.4, index = 586, location = (651301.4,
211900.3), city = Délegyháza
13. neighbor: distance = 9929.7, index = 95, location = (647222.5, 2
09151.3), city = Áporka
```

Exercise

Task 1: Implement a linear search for the closest point instead of using a *KD-Tree*!

In [13]:

```
def find_closest(points, query):
    min_dist = None
    min_point = None
    for point in points:
        dist = point.distance(query)
        if min_dist is None or dist < min_dist:
            min_dist = dist
            min_point = point
    return min_point

print("City location: {0}".format(city_point))
print("Query location: {0}".format(query_point))
closest_point = find_closest(cities['geometry'], query_point)
print("Closest location: {0}".format(closest_point))
```

```
City location: POINT (646998.8 219076.5)
Query location: POINT (646999.8 219078.5)
Closest location: POINT (646998.8 219076.5)
```

Task 2: Compare the execution time of the linear search and the spatial index query (logarithmic asymptotic complexity) approach!

Hint: import the `time` module to record the timestamp before and after the execution of the desired algorithm:

```
start = time.time()
# ... measured code ...
end = time.time()
print("Execution time: {0:.6f}s".format(end-start))
```

In [14]:

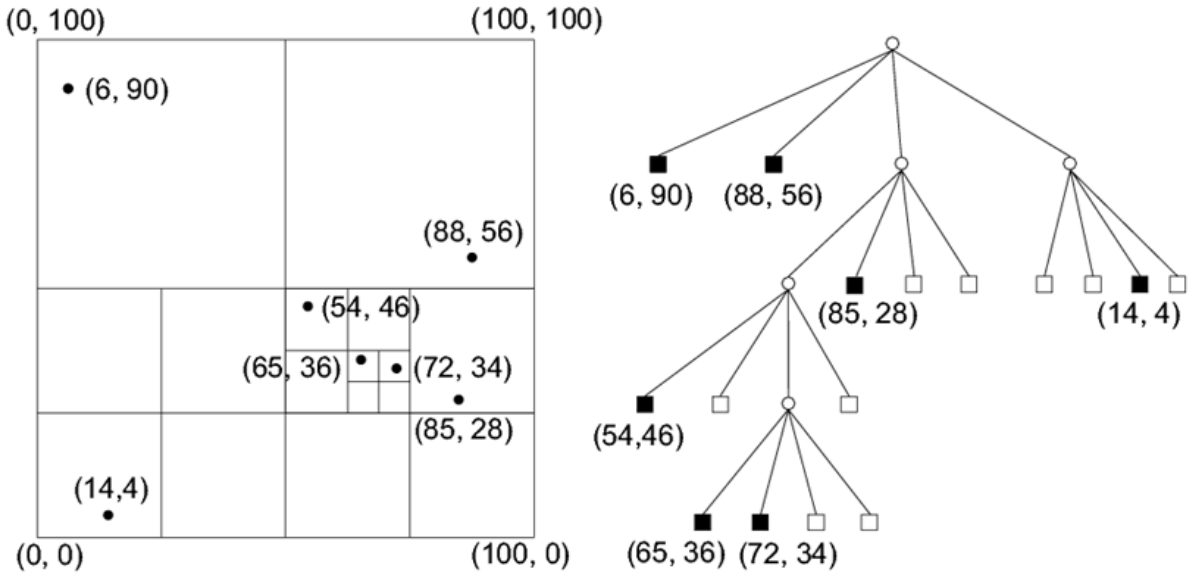
```
import time

start = time.time()
find_closest(cities['geometry'], query_point)
end = time.time()
print("Linear search execution time: {0:.6f}s".format(end-start))

start = time.time()
kdtree.query(query_point)
end = time.time()
print("KD-tree search execution time: {0:.6f}s".format(end-start))
```

```
Linear search execution time: 0.018702s
KD-tree search execution time: 0.000258s
```

Quadtree

A quadtree (https://en.wikipedia.org/wiki/Quadtree) is a tree data structure in which each internal node has exactly four children. The 3 dimensional analog of quadtree is the octree (https://en.wikipedia.org/wiki/Octree).

Quadtree example:



Create a 10x10km query area around a point.

In [15]:

```
query_area_size = 10000
query_area = (
    query_point.x - query_area_size/2,
    query_point.y - query_area_size/2,
    query_point.x + query_area_size/2,
    query_point.y + query_area_size/2
)
print("Query area: {0}, side length = {1:.1f} km".format(query_area, query_area_
size / 1000))
```

```
Query area: (641999.8, 214078.5, 651999.8, 224078.5), side length =
10.0 km
```

Construct the Quad-tree

In [16]:

```
import pyqtree

quadtree = pyqtree.Index(bbox=(min_x, min_y, max_x, max_y))
for i in range(len(points)):
    obj = { "id": i, "point": points[i] }
    quadtree.insert(obj, points[i]) # object, bbox
```

Note: for a polygon, the first argument should be the indexed object (e.g. the polygon itself), and the second argument should be the bounding box of the polygon.

Areawise query

In [17]:

```
matches = quadtree.intersect(query_area)
print("Matches: {0}".format(matches))
```

```
Matches: [{'id': 660, 'point': (651533.1, 215039.7)}, {'id': 2619,
'point': (646998.8, 219076.5)}, {'id': 2622, 'point': (649095.2, 221
564.1)}, {'id': 2678, 'point': (651262.9, 220065.6)}, {'id': 2864,
'point': (643968.9, 219669.5)}]
```

In [18]:

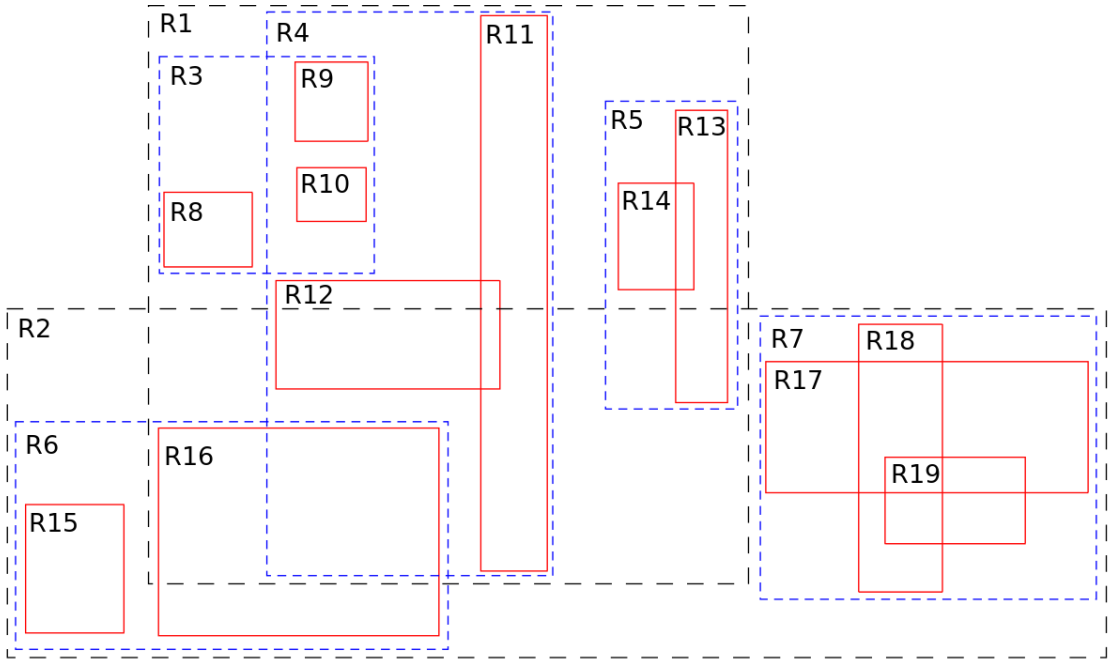
```
for obj in matches:
    print("Index: {0}, Location: {1}, City: {2}".format(obj['id'], obj['point'],
 cities.iloc[obj['id']]['City']))
```

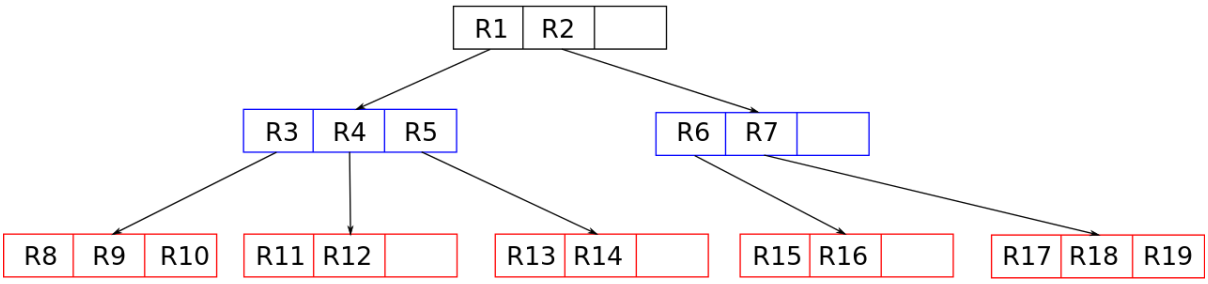
```
Index: 660, Location: (651533.1, 215039.7), City: Dunavarsány
Index: 2619, Location: (646998.8, 219076.5), City: Szigethalom
Index: 2622, Location: (649095.2, 221564.1), City: Szigetszentmiklós
Index: 2678, Location: (651262.9, 220065.6), City: Taksony
Index: 2864, Location: (643968.9, 219669.5), City: Tököl
```

R-Tree

Inspired by the B-tree (https://en.wikipedia.org/wiki/B-tree) for scalara data, he key idea of the R-tree (https://en.wikipedia.org/wiki/R-tree) index structure is to group nearby objects and represent them with their minimum bounding rectangle in the next higher level of the tree, The *"R"* in R-tree stands for rectangle.

R-tree for 2 dimensional data:





We will use the same `query_area` for demonstration, as before with the *Quadtree*.

Construct the R-Tree

In [19]:

```
from rtree import index as rtree_index

rtree = rtree_index.Index()
for i in range(len(points)):
    rtree.insert(i, points[i]) # index, bbox
```

Areawise query

In [20]:

```
matches = rtree.intersection(query_area)
print("Matches: {0}".format(list(matches)))
```

```
Matches: [2622, 2678, 2619, 2864, 660]
```

In [21]:

```
matches = rtree.intersection(query_area)
for idx in matches:
    city = cities.iloc[idx]
    print("Index: {0}, Location: {1}, City: {2}".format(idx, city['geometry'], c
ity['City']))
```

```
Index: 2622, Location: POINT (649095.2 221564.1), City: Szigetszentm
iklós
Index: 2678, Location: POINT (651262.9 220065.6), City: Taksony
Index: 2619, Location: POINT (646998.8 219076.5), City: Szigethalom
Index: 2864, Location: POINT (643968.9 219669.5), City: Tököl
Index: 660, Location: POINT (651533.1 215039.7), City: Dunavarsány
```

GeoPandas integration

If the `rtree` module is installed, the `geopandas` module utilizes an *R-tree* in the background to spatially index the spatial objects in a *GeoDataFrame*.

This spatial index can be accessed directly as the `sindex` property of the *GeoDataFrame*:

In [22]:

```
print(cities.sindex)
matches = cities.sindex.intersection(query_area)
print("Matches: {0}".format(list(matches)))
```

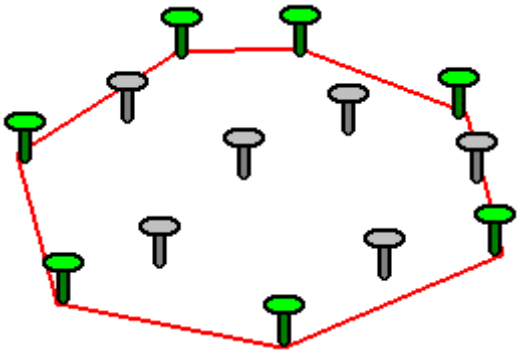
```
rtree.index.Index(bounds=[431339.156, 48431.5, 934944.4, 359044.9],
size=3147)
Matches: [660, 2619, 2864, 2678, 2622]
```

The *R-Tree* spatial index is also used by the `sjoin()` and `clip()` function of *geopandas*.

Chapter 17: Geometric algorithms - Convex Hull

The convex hull of a set of points, is the smallest convex polygon for which each point in the set is either on the boundary of the polygon or in its interior.

We can visualize what the convex hull looks like by imagining that the points are nails sticking out of the plane. Take an elastic rubber band, stretch it around the nails and let it go. It will snap around the nails and assume a shape that minimizes its length. The area enclosed by the rubber band is called the convex hull of the points. This leads to an alternative definition of the convex hull of a finite set of points in the plane: it is the unique convex polygon whose vertices are points from and which contains all points.



Jarvis's march

Jarvis's march computes the convex hull of a set Q of points by a technique also known as the *gift wrapping algorithm*. The algorithm was named after R. A. Jarvis, who published it in 1973.

The algorithm simulates wrapping a piece of paper around the set of points. We start by taping the end of the paper to the lowest point in the set, that is, the point with the lowest Y-coordinate, picking the leftmost such point in case of a tie. We know that this point must be a vertex of the convex hull. We pull the paper to the right to make it wrapping "tight" and then we pull it higher until it touches a point. This point must also be a vertex of the convex hull. Keeping the paper "tight", we continue in this way around the set of vertices until we come back to our original starting point.

The algoirhtm has an $O(n * h)$ asymptotic complexity, where n is the number of points and h is the number of points on the convex hull.

Orientation

Given line *(A, B)* and point *M*, check whether M is left or right from the line, more precisely whether *A -> B -> M* is a closckwise or counter-clockwise turn?

$$det := (Bx - Ax) * (My - Ay) - (By - Ay) * (Mx - Ax)$$

- if $det > 0$: counter-clockwise
- if $det < 0$: clockwise
- if $det = 0$: collinear

Graham's scan

Graham's scan solves the convex-hull problem by maintaining a stack of candidate points. It pushes each point of the input set onto the stack one time, and it eventually pops from the stack each point that is not a vertex of the convex hull. When the algorithm terminates, the stack contains exactly the vertices of the convex hull, in counter-clockwise order of their appearance on the boundary. The algorithm is named after Ronald Graham, who published the original version in 1972.

The algorithm consists of 3 steps:

1. Find the point with the lowest Y-coordinate, picking the leftmost such point in case of a tie. Call this point P.
2. The set of points must be sorted in increasing order of the angle they and the point P make with the X-axis. (Sorting algorithm were discussed in Chapter 6 (06_sorting.pdf).)
3. Initiate an empty stack. Then consider each points in the sorted list in sequence iteratively. For each point, it is first determined whether traveling from the two points immediately preceding this point constitutes making a left turn or a right turn in orientation.
   - If a left turn, push the point onto the stack.
   - If a right turn, the second-to-last point is not part of the convex hull (lies inside it), and is therefore removed from the stack. The same determination is then made for the set of the latest point, and the two points that immediately precede the point found to have been inside the hull, and is repeated until a left turn set is encountered, at which point the algorithm moves on to the next point in the set of points in the sorted list.

The algorithm has an $O(n * log(n)$ asymptotic complexity. Thus, this algorithm is not *output-sensitive* (compare to Jarvis's march).

Quickhull

The *Quickhull* method uses the *divide and conquer* approach similar to that of Quicksort (06_sorting.pdf#Quicksort), from which its name derives. The original algorithm was described by Scott Greenfield in 1990. The algorithm was later extended to work in n-dimensional space.

The algorithm contains the following steps:

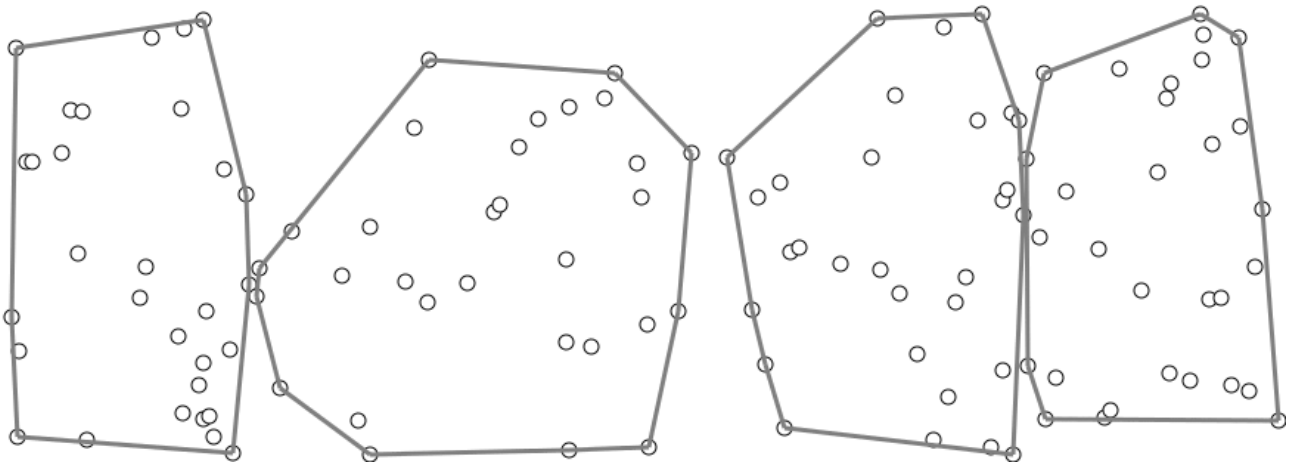
1. Find the points with minimum and maximum X-coordinates, as these will always be part of the convex hull. If case of a tie, pick the ones with minimum/maximum Y-coordinates correspondingly.
2. Use the line formed by these two points to divide the set in two subsets of points, which will be processed recursively (*"divide and conquer"*).
3. For both sides, determine the point with the maximum distance from the line. This point forms a triangle with those of the line. The points lying inside of that triangle cannot be part of the convex hull and can therefore be ignored in the next steps.
4. Repeat the previous step on the two lines formed by the triangle.
5. Continue the recursion until no more points are left. In the end, all points selected constitute the convex hull.

The asymptotic complexity of the algorithm is $O(n * log(r))$, where r is the number of processed points.

Chan's algorithm

Chan's algorithm is an optimal *output-sensitive algorithm* to compute the convex hull. It was named after Timothy M. Chan, who published the algorithm in 1996.

The algorithm combines Graham's scan (or other algorithm with $O(n * log(n))$ complexity) with Jarvis's march ($O(n * h)$), in order to obtain an optimal $O(n * log(h))$ complexity, where n is the number of points and h is the number of vertices of the output (the convex hull).



QuickHull with *Shapely*

Read the `hungary_cities.shp` shapefile located in the `data` folder. This dataset contains both scalar and spatial data of the Hungarian cities, and should be familiar from Chapter 15 (15_graph_spanning_tree.pdf).

In [1]:

```
import geopandas as gpd
from scipy.spatial import ConvexHull

cities_gdf = gpd.read_file('../data/hungary_cities.shp')
display(cities_gdf)
```

	Id	County	City	Status	KSH	geometry
0	1	FEJÉR	Aba	town	17376	POINT (610046.800 187639.000)
1	2	BARANYA	Abaliget	town	12548	POINT (577946.100 89280.800)
2	3	HEVES	Abasár	town	24554	POINT (721963.700 273880.300)
3	4	BORSOD-ABAUJ-ZEMPLÉN	Abaújalpár	town	15662	POINT (812129.200 331508.200)
4	5	BORSOD-ABAUJ-ZEMPLÉN	Abaújkér	town	26718	POINT (809795.600 331138.300)
...	...	...	...	...	...	...
3142	3143	GYÕR-MOSON-SOPRON	Zsira	town	04622	POINT (471324.200 237577.200)
3143	3144	CSONGRÁD	Zsombó	town	17765	POINT (721098.100 109690.000)
3144	3145	BORSOD-ABAUJ-ZEMPLÉN	Zsujta	town	11022	POINT (815027.400 353143.100)
3145	3146	SZABOLCS-SZATMÁR-BEREG	Zsurk	town	13037	POINT (884847.700 344952.800)
3146	3147	BORSOD-ABAUJ-ZEMPLÉN	Zubogy	town	19105	POINT (763123.300 338338.600)

3147 rows × 6 columns

Shapely can compute the convex hull of any geometry through the `convex_hull` attribute.

The `geometry` column of the GeoDataFrame contains Shapely points (see Chapter 11 (11_spatial_vector.pdf)), but we need to create a *MultiPoint* of all cities to calculate their aggregated concex hull.

In [2]:

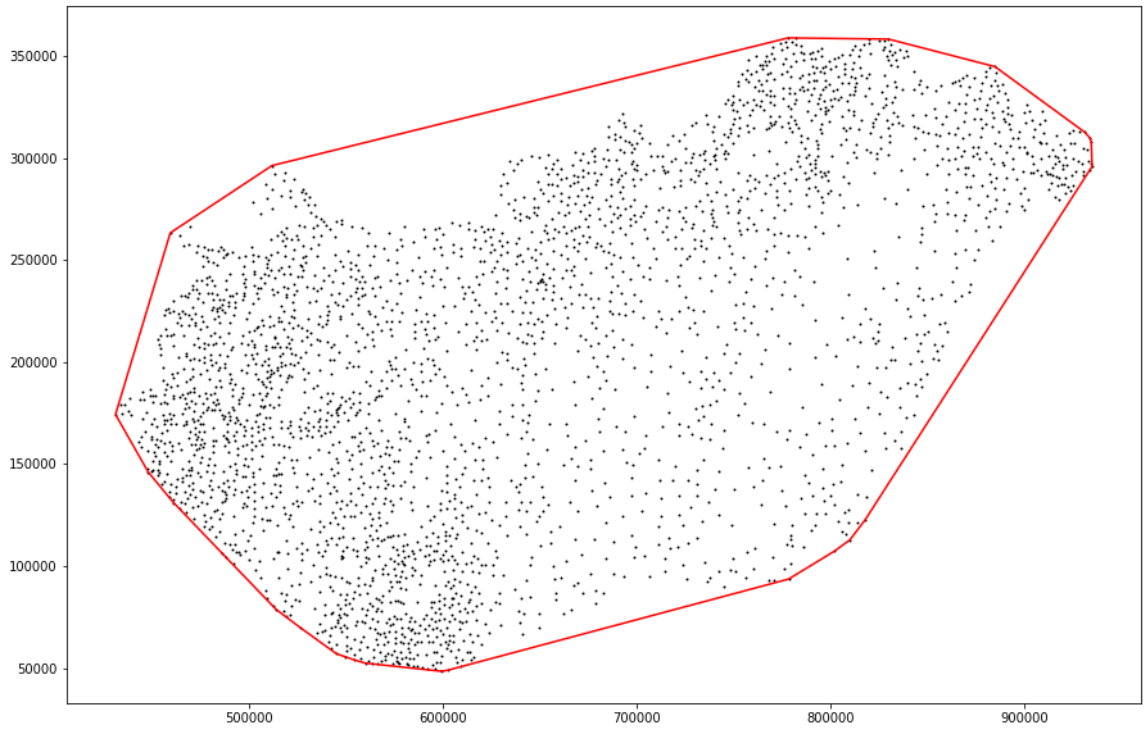
```
from shapely import geometry

multipoint = geometry.MultiPoint(cities_gdf.geometry)
hull = multipoint.convex_hull
print(hull)
```

```
POLYGON ((599595.6 48431.5, 560116.5 52448.8, 554768.8 53926, 54516
1.9 57103.6, 526519.8 69931.67999999999, 514301.4 78648.89999999999,
487965.5 104677.9, 461349.736 131087.715, 458840.1 133907.4, 448204.
1 146147.2, 431339.156 174384.3, 459598.524 263436.1, 512056.1 29642
9.8, 778249.4 359044.9, 830043.1 358431.7, 884847.7 344952.8, 93100
5.4 312885.6, 933862.3 309972.7, 934499.7 307908.4, 934944.4 296041.
6, 818003.3 122843.7, 810002.9 112845.9, 801928.9 107279.7, 778496.3
93680, 602643.7 48962.9, 599595.6 48431.5))
```

Plot figure:

In [3]:

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure(figsize=[15, 10])

# Add all points to plot
for point in cities_gdf.geometry:
    plt.plot(point.x, point.y, color='black', marker='o', markersize=1)

# Fetch the list of X and Y coordinates of the convex hull
line_x, line_y = hull.exterior.xy
# Plot linestring
plt.plot(line_x, line_y, color='red')

# Display plot
plt.show()
```



Quickhull with *SciPy* (*optional*)

As an alternative approach, we can use *SciPy* to cimpute the convex hull.

SciPy (pronounced "Sigh Pie") is library used for scientific computing and technical computing for mathematics, science, and engineering. *SciPy* is built on top of *NumPy*, *Matplotlib* and *Pandas* and are tightly integrated with them. It is one of the most widely used Python package in the scientific community.

How to install *SciPy*?

If you have Anaconda installed, then `scipy` was already installed together with it.

If you have a standalone Python3 and Jupyter Notebook installation, open a command prompt / terminal and type in:

```
pip3 install scipy
```

How to use *SciPy*?

SciPy consists of sub-packages for various scientific areas. For us the `spatial` package is in focus, which contains spatial algorithms, like the *QuickHull* or the *KdTree* (see Chapter 16 (16_spatial_indexing.pdf)).

```
import scipy.spatial
```

Fetch points for cities:

In [4]:

```
points = [(geom.x, geom.y) for geom in cities_gdf.geometry]
print("Number of points: {0}".format(len(points)))
```

```
Number of points: 3147
```

Calculate convex hull:

In [5]:

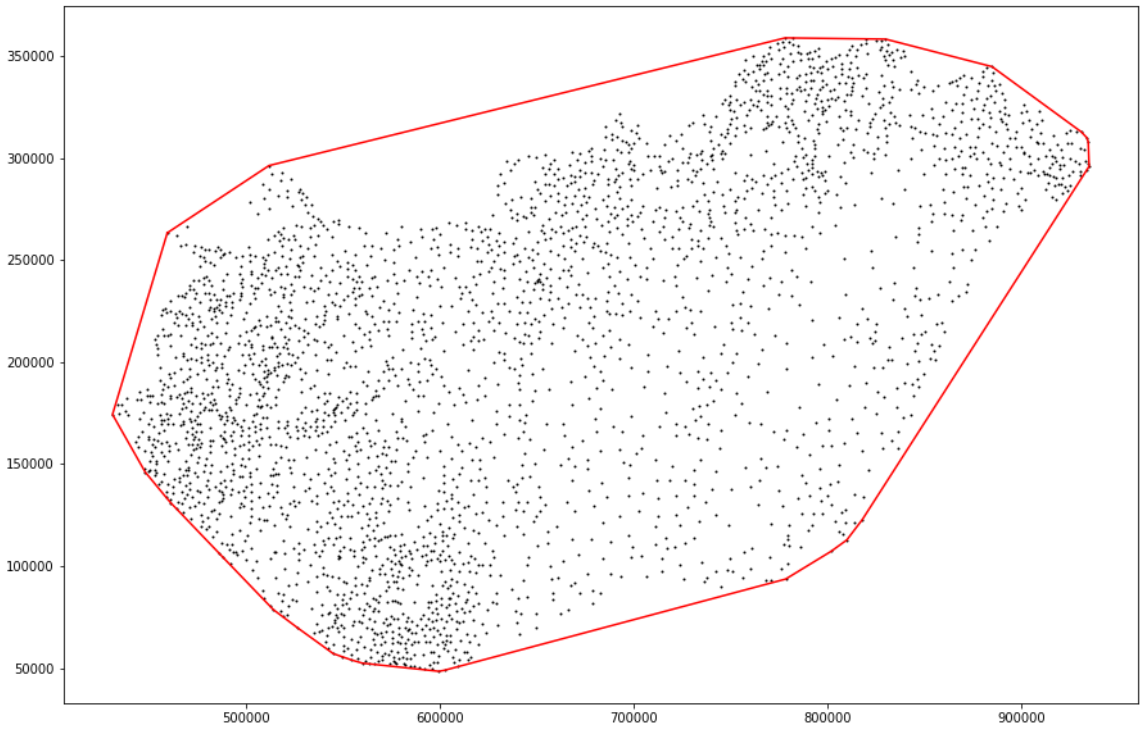
```
hull = ConvexHull(points)
print("Number of vertices on hull: {0}".format(len(hull.vertices)))
print("Hull vertices: {0}".format(hull.vertices))
```

```
Number of vertices on hull: 25
Hull vertices: [  93  783  853 2847 1289 3115  334  198  782  635 31
08 1245  257 1845
  204  601 1575  849 1589 2892 2769 3145 2222 2844 2247]
```

Plot figure:

In [6]:

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure(figsize=[15, 10])

# Add all points to plot
for point in points:
    plt.plot(point[0], point[1], color='black', marker='o', markersize=1)

# Calculate convex hull linestring
line_x = [points[idx][0] for idx in hull.vertices]
line_y = [points[idx][1] for idx in hull.vertices]
# Add first point of hull to the end, so the linestring will be closed.
line_x.append(points[hull.vertices[0]][0])
line_y.append(points[hull.vertices[0]][1])
# Plot linestring
plt.plot(line_x, line_y, color='red')

# Display plot
plt.show()
```



Chapter 18: Clustering and classification

The method of determining the properties of the thematic classes directly from the reference data is called **supervised classification**, because the analyst actually “supervises” how the discriminant functions of the classes are formed by providing the reference data.

Unsupervised classification methods on the other hand group data points (e.g. pixels) together based on their similarities, with no information from the user about which ones belong together. The user selects the independent or predictor variables of interest, and the chosen algorithm does the rest. This doesn’t mean that you don’t need to know what you’re classifying, however. Once a classification is produced, it’s up to the user to interpret it and decide which types of features correspond to which generated classes, or if they even do correspond nicely.

Unsupervised classification is also called **clustering**.

K-Means clustering algorithm

The *K-Means* method is one of the most common unsupervised classification approach.

The algorithm requires an arbitrarily specified initial cluster centres that are represented by the means of the data points assigned to them. As a naïve solution, the user only defines the number of clusters and random data points are selected as their initial centers.

This will generate a very crude set of clusters. The data points are then reassigned to the cluster with the closest center, and the centers are recomputed. The process is repeated as many times as necessary such that there is no further movement of the data points between clusters. In practice, with large data sets, the process is not run to completion and some other stopping rule is used.

Considering the squared distance between each data point and the respective cluster center as the *squared error*, the sum of squared errors (*SSE*) progressively reduces with each iteration. If the *Euclidean distance* is used as a metric, this simply means to accumulate the squared distances for all points and their respective cluster center.

Although no general proof of convergence exists for this algorithm, it can be expected to yield acceptable results when the data exhibit characteristic pockets which are relatively far from each other. In most practical cases the application of this algorithm will require experimenting with various values of inital clusters (the value of k), as well as different choices of starting configurations.

Clustering raster data

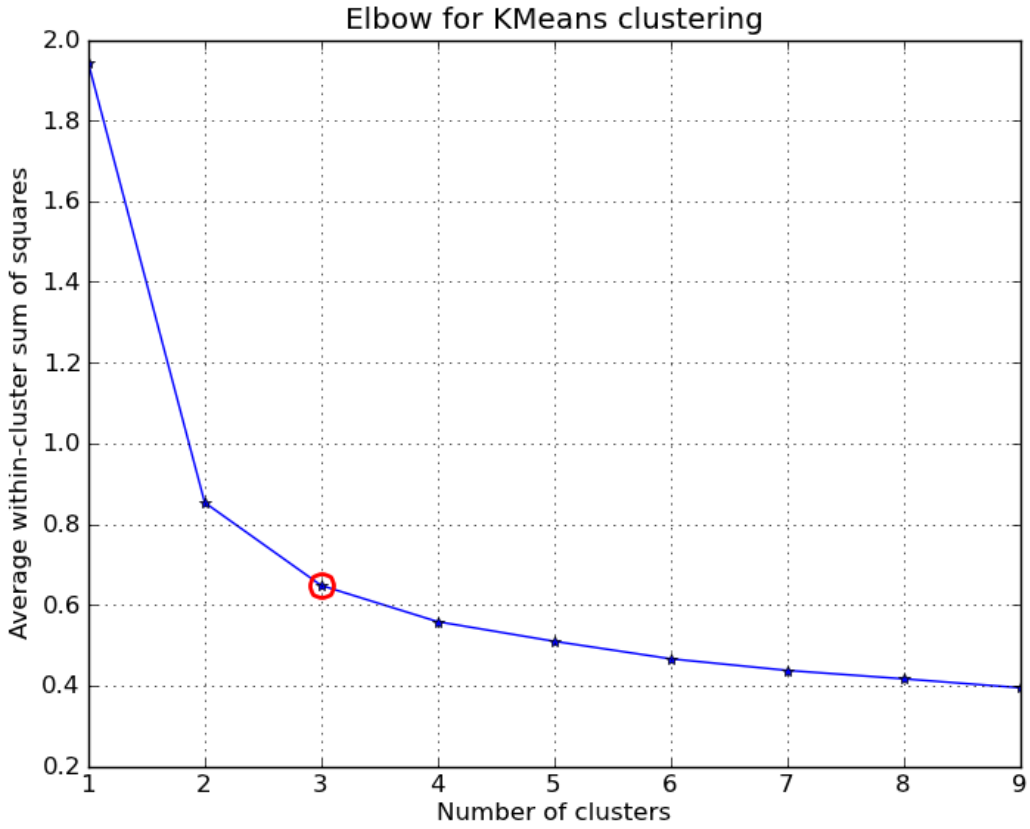
The K-Means method can be use with an arbitrary *distance function*. For raster imagery the distance is computed as if the pixel values were coordinates. For example, if the insensity values of two RGB pixels were $(25, 42, 37)$ and $(31, 40, 32)$, the squared distance would be $(25-31)^2 + (42-40)^2 + (37-32)^2 = 65$ in the 3 dimensional spectral space, no matter where the pixels were in relation to each other spatially.

Elbow method

One of the most challenging tasks in the K-Means clustering algorithm is to choose the right value of the clusters (the value of k). What should be the right value of k and how to choose it?

The *Elbow Method* is one of the most popular methods to determine the optimal value of k. The idea is to run K-Means clustering on the dataset for a range of values of k (e.g. from 1 to 10), and for each value of k calculate the sum of squared errors (*SSE*).

Then, visualize a line chart of the SSE for each value of k. If the line chart looks like an arm, then the "elbow" on the arm is the value of k that is the best. The idea is that we want a small SSE, but that the SSE tends to decrease toward 0 as we increase k. (The SSE is 0 when k is equal to the number of data points in the dataset, because then each data point has its own cluster, and there is no error between it and the center of its cluster.) Hence we select the value of k at the “elbow”, i.e. the point after which the line chart starts decreasing in a linear fashion.



K-Means clustering in Python

Scikit-learn (also known as *sklearn*) is a machine learning library for Python. It features various classification, regression and clustering algorithms including *k-means*.

How to install *scikit-learn*?

If you have Anaconda installed, then `scikit-learn` was already installed together with it.

If you have a standalone Python3 and Jupyter Notebook installation, open a command prompt / terminal and type in:

```
pip3 install scikit-learn
```

Clustering vector data

Read the `hungary_cities.shp` shapefile located in the `data` folder. This dataset contains both scalar and spatial data of the Hungarian cities, and should be familiar from Chapter 15 (15_graph_spanning_tree.pdf).

In [1]:

```
import geopandas as gpd
from sklearn.cluster import KMeans

cities_gdf = gpd.read_file('../data/hungary_cities.shp')
display(cities_gdf)
```

	Id	County	City	Status	KSH	geometry
0	1	FEJÉR	Aba	town	17376	POINT (610046.800 187639.000)
1	2	BARANYA	Abaliget	town	12548	POINT (577946.100 89280.800)
2	3	HEVES	Abasár	town	24554	POINT (721963.700 273880.300)
3	4	BORSOD-ABAUJ-ZEMPLÉN	Abaújalpár	town	15662	POINT (812129.200 331508.200)
4	5	BORSOD-ABAUJ-ZEMPLÉN	Abaújkér	town	26718	POINT (809795.600 331138.300)
...	...	...	...	...	...	...
3142	3143	GYÕR-MOSON-SOPRON	Zsira	town	04622	POINT (471324.200 237577.200)
3143	3144	CSONGRÁD	Zsombó	town	17765	POINT (721098.100 109690.000)
3144	3145	BORSOD-ABAUJ-ZEMPLÉN	Zsujta	town	11022	POINT (815027.400 353143.100)
3145	3146	SZABOLCS-SZATMÁR-BEREG	Zsurk	town	13037	POINT (884847.700 344952.800)
3146	3147	BORSOD-ABAUJ-ZEMPLÉN	Zubogy	town	19105	POINT (763123.300 338338.600)

3147 rows × 6 columns

Fetch points for cities:

In [2]:

```
points = [(geom.x, geom.y) for geom in cities_gdf.geometry]
print("Number of points: {0}".format(len(points)))
```

```
Number of points: 3147
```

Cluster the points using the *K-Means algorithm*:

In [3]:

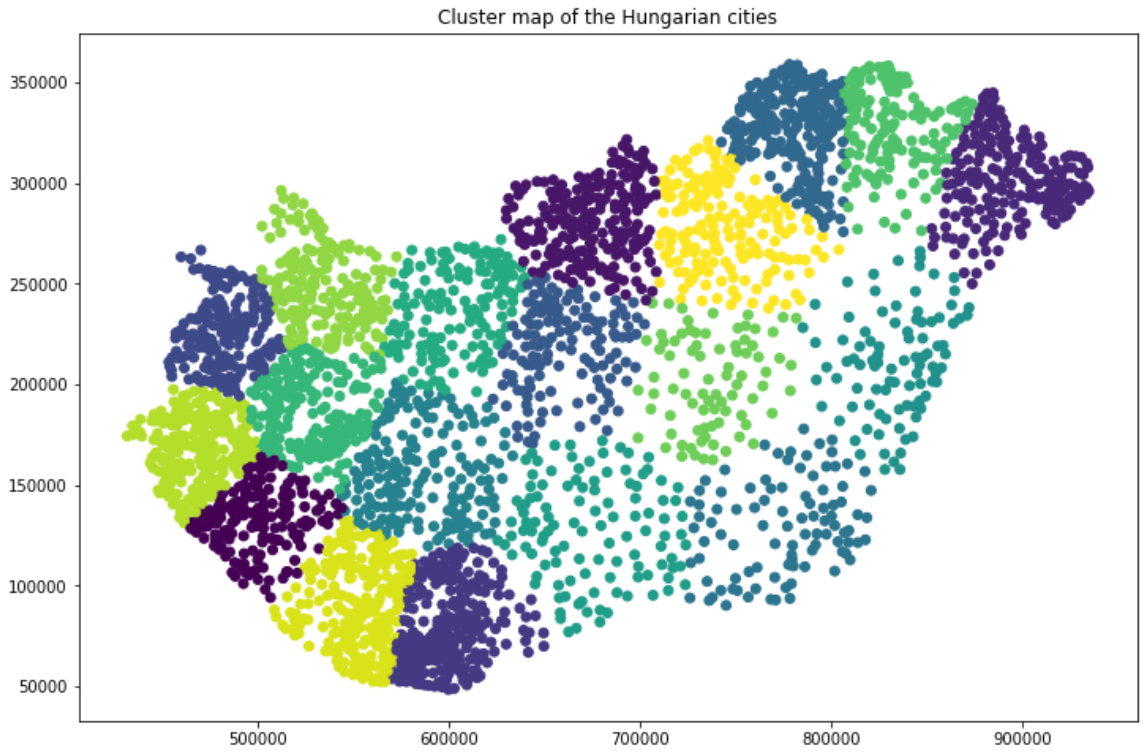
```
pred = KMeans(n_clusters=19).fit_predict(points)
print(pred)
print(len(pred))
```

```
[ 8  3 18 ... 13  2  6]
3147
```

Plot figure:

In [4]:

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure(figsize=(12, 8))

# Fetch list of X and Y coordinates
xs = [point[0] for point in points]
ys = [point[1] for point in points]

# Put the cluster points on the plot
plt.scatter(xs, ys, c=pred)

# Display plot
plt.title("Cluster map of the Hungarian cities")
plt.show()
```



Clustering raster images

Read the dataset

The `data/LC08_L1TP_188027_20200420_20200508_01_T1_Szekesfehervar.tif` file is a segment of a Landsat 8 satellite image of Székesfehérvár city, Lake Velence and their surroundings, acquired on 2020 April 20. It should be familiar from Chapter 12 (12_spatial_raster.pdf).

In [5]:

```
import rasterio
szfv_2020 = rasterio.open('../data/LC08_L1TP_188027_20200420_20200508_01_T1_Szek
esfehervar.tif')
print(szfv_2020.count) # band count
print(szfv_2020.width) # dimensions
print(szfv_2020.height)
```

```
11
1057
645
```

Read the red, green blue and NIR bands:

In [6]:

```
blue = szfv_2020.read(2)
green = szfv_2020.read(3)
red = szfv_2020.read(4)
nir = szfv_2020.read(5)
```

Single-band clustering

Cluster the satellite image based on the near-infrared band.

In [7]:

```
nir_1d = nir.reshape(nir.shape[0] * nir.shape[1], 1)
print(nir_1d.shape)
```

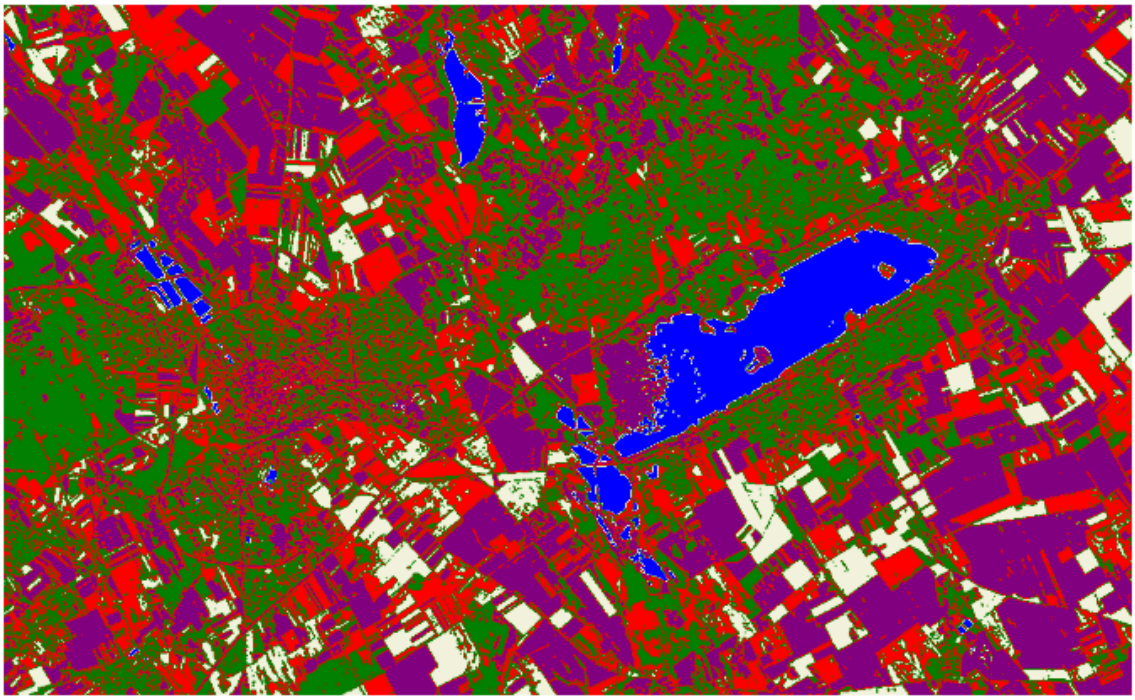
```
(681765, 1)
```

In [20]:

```
pred = KMeans(n_clusters=5).fit_predict(nir_1d)
img_clusters = pred.reshape(nir.shape)
```

In [24]:

```
import matplotlib.colors as mc
cmap = mc.LinearSegmentedColormap.from_list('', ['purple', 'red', 'green', 'beig
e', 'blue'])

plt.figure(figsize=[12,12])
plt.imshow(img_clusters, cmap=cmap)
plt.axis('off')
plt.show()
```



Multi-band clustering

Cluster the satellite image based on the RGBN (red, blue, green NIR) bands.

In [10]:

```
red_1d   = red.reshape(red.shape[0] * red.shape[1], 1)
green_1d = green.reshape(green.shape[0] * green.shape[1], 1)
blue_1d  = blue.reshape(blue.shape[0] * blue.shape[1], 1)

rgbn_1d = [(0, 0, 0, 0)] * (red.shape[0] * red.shape[1])
for i in range(red.shape[0] * red.shape[1]):
    rgbn_1d[i] = (red_1d[i, 0], green_1d[i, 0], blue_1d[i, 0], nir_1d[i, 0])

print(rgbn_1d[10000]) # print random item
```

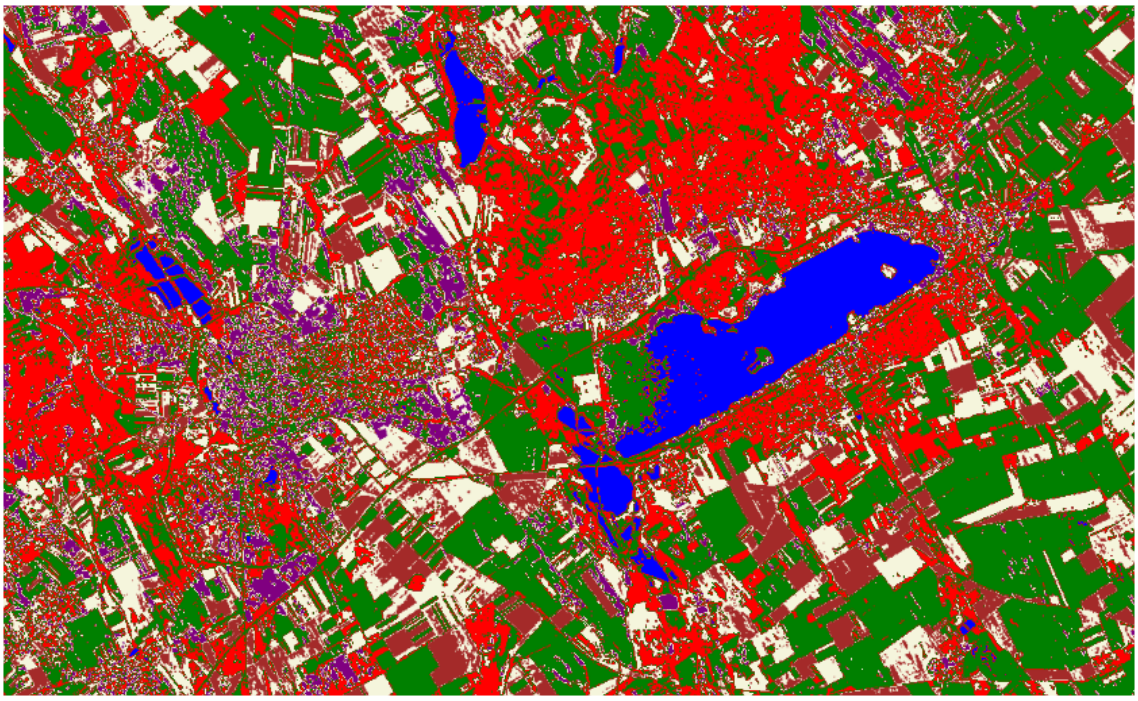
```
(8434, 8678, 9156, 15104)
```

In [28]:

```
pred = KMeans(n_clusters=6).fit_predict(rgbn_1d)
img_clusters = pred.reshape(red.shape)
```

In [32]:

```
cmap = mc.LinearSegmentedColormap.from_list('', ['blue', 'red', 'green', 'brown'
, 'beige', 'purple'])

plt.figure(figsize=[15,15])
plt.imshow(img_clusters, cmap=cmap)
plt.axis('off')
plt.show()
```



Downsampling

The `LC08_L1TP_188027_20200420_20200508_01_T1` file is a complete Landsat 8 satellite image tile, containing Budapest and parts of Western-Hungary, acquired on 2020 April 20.
Download: https://gis.inf.elte.hu/files/public/landsat-budapest-2020 (https://gis.inf.elte.hu/files/public/landsat-budapest-2020) (1.4 GB)

In [13]:

```
import rasterio
bp_2020 = rasterio.open('LC08_L1TP_188027_20200420_20200508_01_T1.tif')
print(bp_2020.count) # band count
print(bp_2020.width) # dimensions
print(bp_2020.height)
```

```
11
7981
8071
```

To speed up processing larger raster files, we may downsample them for the price of reducing the accuracy of the result.

First, define the resampling function:

In [14]:

```
from rasterio.enums import Resampling

def read_resampled_band(dataset, band, resample_factor):
    data = dataset.read(band,
        out_shape=(
            1,
            int(dataset.height * resample_factor),
            int(dataset.width * resample_factor)
        ),
        resampling=Resampling.bilinear
    )
    return data
```

Read the blue, green, read and near-infrared bands into *Numpy* arrays. Resample them to a smaller size to make further processing (clustering especially) faster.

In [15]:

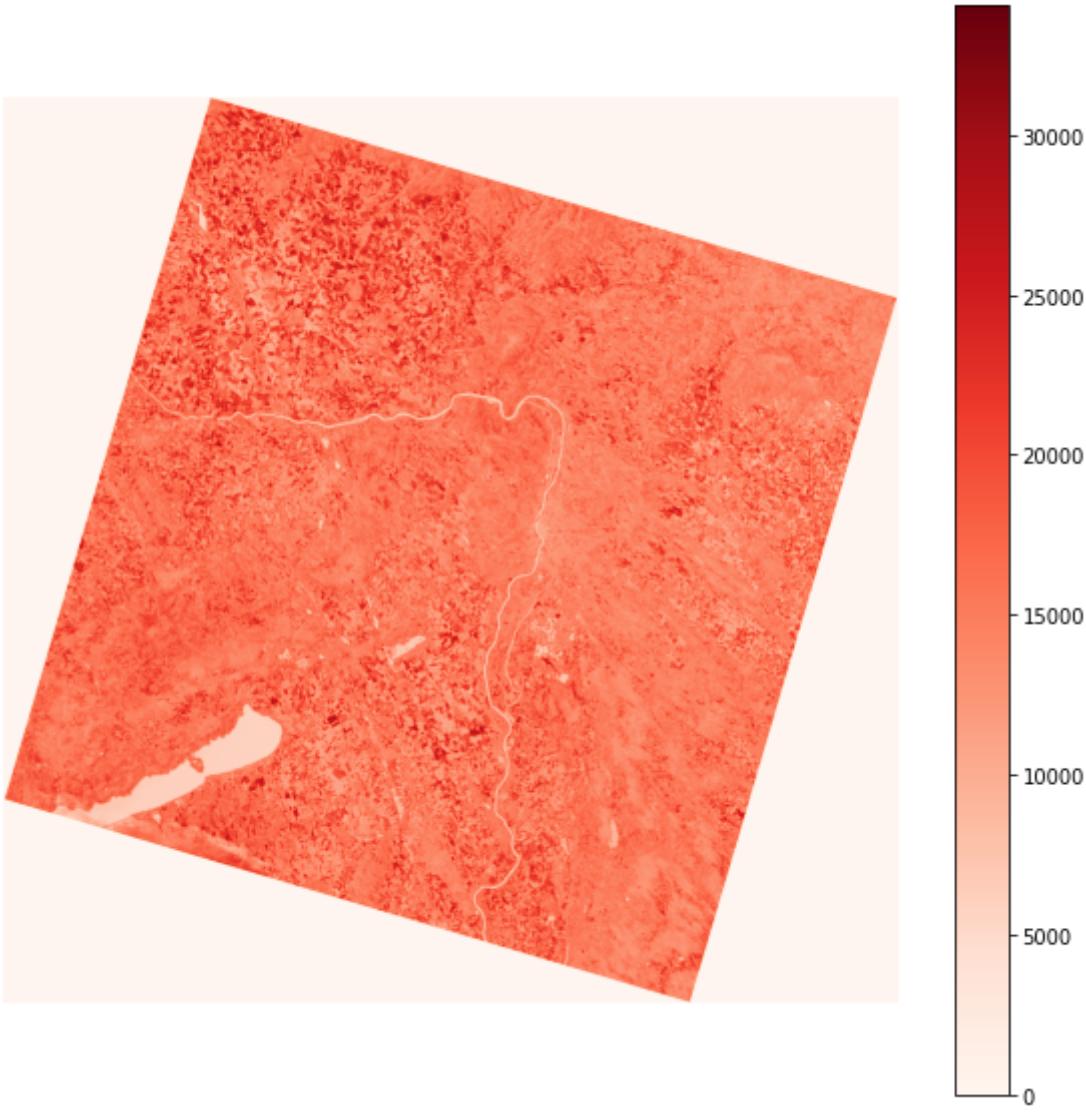
```
bp = {}
bp['blue'] = read_resampled_band(bp_2020, 2, 1/4)
bp['green'] = read_resampled_band(bp_2020, 3, 1/4)
bp['red'] = read_resampled_band(bp_2020, 4, 1/4)
bp['nir'] = read_resampled_band(bp_2020, 5, 1/4)

print(bp['red'].shape)
```

```
(2017, 1995)
```

Display the near-infrared band for verification:

In [16]:

```
plt.figure(figsize=[10,10])
plt.imshow(bp['nir'], cmap='Reds')
plt.axis('off')
plt.colorbar()
plt.show()
```

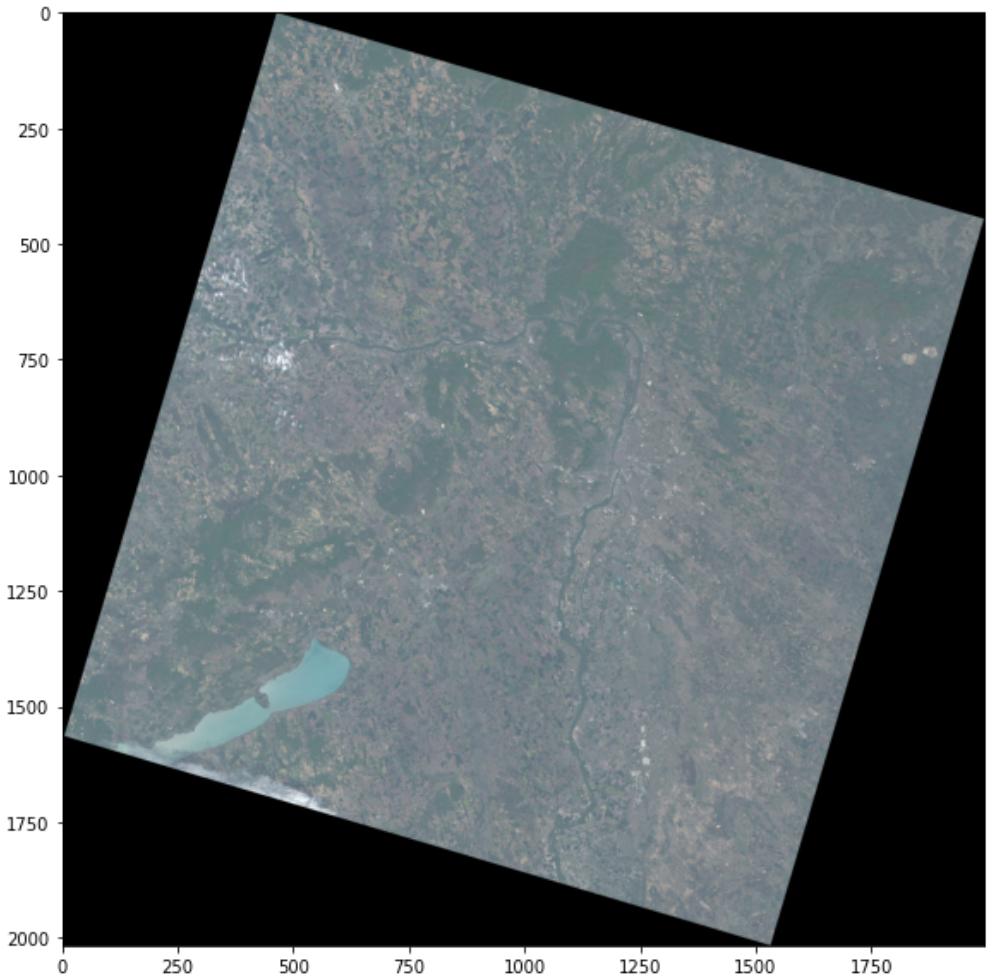


Display the RGB image for verification:

In [17]:

```
from rasterio.plot import show
import numpy as np

bp['red_max'] = np.percentile(bp['red'], 99.99)
bp['blue_max'] = np.percentile(bp['blue'], 99.99)
bp['green_max'] = np.percentile(bp['green'], 99.99)

# astype('f4') is a numpy function to convert to float (4 byte)
bp['redf'] = bp['red'].astype('f4') / bp['red_max']
bp['bluef'] = bp['blue'].astype('f4') / bp['blue_max']
bp['greenf'] = bp['green'].astype('f4') / bp['green_max']
bp['rgb'] = [bp['redf'], bp['greenf'], bp['bluef']]

plt.figure(figsize=[10,10])
show(bp['rgb'])
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).
```



Summary exercise on clustering

Implement a functions which performs single band clustering on a *rasterio* band (*NumPy* array). Execute it on the NIR band of the complete satellite image.

Example on how it shall work:

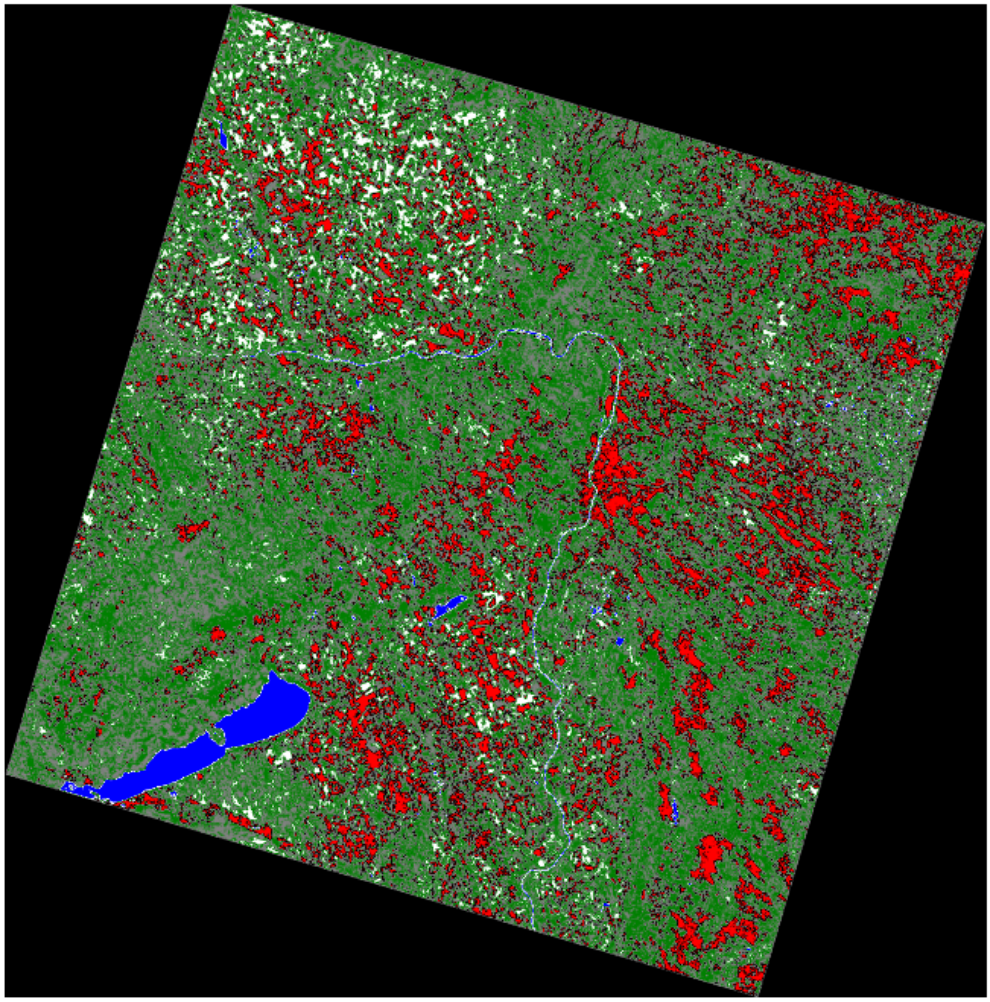
```
single_band_clustering(bp['nir'], ['red', 'black', 'gray', 'green', 'whit
e', 'blue']) # 6 clusters with these colors
```

In [18]:

```
def single_band_clustering(band, clusters=['red', 'black', 'gray', 'green', 'whi
te', 'blue']):
    band_1d = band.reshape(band.shape[0] * band.shape[1], 1)

    pred = KMeans(n_clusters=len(clusters)).fit_predict(band_1d)
    img_clusters = pred.reshape(band.shape)

    cmap = mc.LinearSegmentedColormap.from_list('', clusters)

    plt.figure(figsize=[12,12])
    plt.imshow(img_clusters, cmap=cmap)
    plt.axis('off')
    plt.show()
```

In [19]:

```
single_band_clustering(bp['nir'])
```



Appendix 1: Strings

Advanced string operations

Concatenation: +

For string the + operator is used for concatenation, joining multiple strings together.

In [1]:

```
word1 = 'Hello'
word2 = 'Python'
greet = word1 + ' ' + word2 + '!'
print(greet)
```

```
Hello Python!
```

Multiplication: *

The * operator is used for "multiplying" a string, repeating and concatenating it the given times.

In [2]:

```
greet3times = greet * 3
print(greet3times)
```

```
Hello Python!Hello Python!Hello Python!
```

Length: len()

The len() statement returns the length of the string.

In [3]:

```
print(len(greet))
```

```
13
```

String indexing and slicing: []

A single charcter of a string can be access by indexing it, starting from zero:

In [4]:

```
print(greet[0])
```

```
H
```

Question: what will happen if we index with a negative number?

In [5]:

```
print(greet[-1])
```

```
!
```

Question: what will happen if we with a number larger than the length of the string?

In [6]:

```
print(greet[100])
```

```
---------------------------------------------------------------------------
-------
IndexError                                Traceback (most recent cal
l last)
<ipython-input-6-eb8fbb2c6e43> in <module>
----> 1 print(greet[100])

IndexError: string index out of range
```

We can also create substrings by fetching a slice of a string.
Note that the end index is exclusive, so if the slice is given as `[4:6]`, then the characters with the index 4 and 5 will be sliced.

In [7]:

```
print(greet[0:5])
print(greet[6:7])
```

```
Hello
P
```

The first (start) index can be omitted, by default it will be zero:

In [8]:

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

```
Hello
```

The second (end) index can also be omitted, by default it will be the end of the string:

In [9]:

```
print(greet[6:])
```

```
Python!
```

Question: what happens if we omit both the start and the end index?

In [10]:

```
print(greet[:])
```

```
Hello Python!
```

Question: what happens if we use negative indices?

In [11]:

```
print(greet[-7:])
print(greet[1:-2])
```

```
Python!
ello Pytho
```

Question: what happens if the end index is larger than the length of the string?

In [12]:

```
print(greet[6:100])
```

```
Python!
```

Built-in string functions

A comprehensive list of the built-in functions can be found in the 'string library' (https://docs.python.org/3/library/stdtypes.html#string-methods) reference documentation.

These string functions are *methods*, which means they can be called on a string instance (value or variable) in a form `stringvar.method(parameters)`. They do not modify the original string, but return a new instance.

Lowercase: `lower`

Replace all letters to lowercase.

In [13]:

```
print(greet)
greet_lower=greet.lower()
print(greet_lower)
```

```
Hello Python!
hello python!
```

Uppercase: `upper`

Replace all letters to uppercase.

In [14]:

```
print(greet)
greet_upper=greet.upper()
print(greet_upper)
```

```
Hello Python!
HELLO PYTHON!
```

Capitalization: `capitalize` and `title`

Replace the very first letter or the first letter of each words to uppercase. The rest will be turned to lowecase.

In [15]:

```
print(greet_lower)
greet_capital=greet_lower.capitalize()
print(greet_capital)

greet_title=greet_lower.title()
print(greet_title)
```

```
hello python!
Hello python!
Hello Python!
```

Substring search: `find`

Looks up the first occurance of a character or a substring in a string. The result is the starting index position of the first occurance as an `integer`. Keep in mind that the first index is `0`! The returned value is `-1` if the substring was not found.

In [16]:

```
print(greet)
location = greet.find('Python')
print(location)

print(greet)
location = greet.find('java')
print(location)
```

```
Hello Python!
6
Hello Python!
-1
```

The starting index of the search can also be passed to the function. This way multiple occurances of a substring can be looked up.

In [17]:

```
print(greet3times)
location = greet3times.find('Python')
print(location)

location = greet3times.find('Python', location + 1)
print(location)
```

```
Hello Python!Hello Python!Hello Python!
6
19
```

This function is case-sensitive.
If you would like to search for both lower and uppercase variants, you may convert the string to lowercase first!

In [18]:

```
print(greet)
location = greet.find('python')
print(location)

print(greet.lower())
location = greet.lower().find('python')
print(location)
```

```
Hello Python!
-1
hello python!
6
```

Substring replace: `replace`

Replace **all** occurances of a substring to another substring.

This function is also case-sensitive.

In [19]:

```
greet_alternative = greet3times.replace('Hello', 'Hi')
print(greet_alternative)
```

```
Hi Python!Hi Python!Hi Python!
```

Stripping: `lstrip`, `rstrip`, `strip`

All functions are used to trim unrequired whitespace characters (spaces, tabulators, newlines) from a string.

- `lstrip` - remove whitespace characters from the lefthand side.
- `rstrip` - remove whitespace characters from the righthand side.
- `stri` - remove whitespace characters from both sides.

In [20]:

```
greet_world = '   --== Hello World  ==-- '
print(greet_world.lstrip())
print(greet_world.rstrip())
print(greet_world.strip())
```

```
--== Hello World  ==--
   --== Hello World  ==--
--== Hello World  ==--
```

The characters to remove can also be specified otherwise:

In [21]:

```
print(greet_world.strip(' -='))
```

```
Hello World
```

Prefix and suffix check: `startswith`, `endswith`

These functions verifies whether a string starts or ends with the given substring. The result is a *boolean* value (`True` or `False` .)

This function is also case-sensitive.

In [22]:

```
print(greet.startswith('Hello'))
print(greet.startswith('Hi'))
```

```
True
False
```

Splitting: `split`

Split a string into a list of substring by defining a so-called *separator* or *delimiter* character or string. The *separator* is removed from the string.

In [23]:

```
print(greet3times)
words = greet3times.split('!')
print(words)
```

```
Hello Python!Hello Python!Hello Python!
['Hello Python', 'Hello Python', 'Hello Python', '']
```

Question: why is there an empty string at the end of the result list?

Logical operations on strings

Containment check: `in`

Verify whether a letter or a substring occures *anywhere* inside a string. The result is a *boolean* value (`True` or `False` .)

In [24]:

```
print('p' in greet)
print('P' in greet)
print('Python' in greet)
```

```
False
True
True
```

In [25]:

```
if 'P' in greet:
    print('Contains a letter P!')
```

```
Contains a letter P!
```

Equality check: `==`

Perform a case-sensitive equality check between two strings.

In [26]:

```
if word2 == 'Python':
    print('It was Python.')
else:
    print('It was not Python.')
```

```
It was Python.
```

Summary exercise on strings

Task: request the name, birth year, email address and spoken languages of the user. The spoken languages are requested as a string, separated by commas.

Check whether the following validation rules are matched. If any of the data is invalid, display an error message and request a repeated entry of the data.

- The name must contain at least 2 parts. (There must be a space inside it.)
- The birth year must be a number, between 1900 and 2019.
- The email address must contain a `@` letter and must end with a `elte.hu` domain.

When the data was given successfully, trim any unnccesseary whitespaces and display it in a corrected format:

- The name shall be displayed with each part starting with a capital letter.
- Beside the birth year, calculate the (possible) age of the current user.
- The email address shall be lowercase.
- The spoken languages shall be displayed as a list of languages instead of a single string.

In [27]:

```
import datetime

def valid_name(name):
    name = name.strip()
    return len(name.split(' ')) >= 2

def valid_birthyear(year):
    year = year.strip()
    try:
        year_num = int(year)
        return year_num >= 1900 and year_num <= 2019
    except:
        return False

def valid_email(email):
    email = email.strip()
    return '@' in email and email.endswith('elte.hu')

def format_name(name):
    return name.strip().title()

def format_age(year):
    now = datetime.datetime.now()
    age_max = now.year - int(year)
    age_min = max(age_max - 1, 0)
    if age_max != age_min:
        return str(age_min) + "/" + str(age_max)
    else:
        return str(age_max)

def format_email(email):
    return email.strip().lower()

def format_langs(langs):
    langs = langs.strip().split(',')
    langs = list(map(str.strip, langs))
    return langs

name = input("Name: ")
while not valid_name(name):
    print("Incorrect format for name.")
    name = input("Name: ")

birthyear = input("Birth year: ")
while not valid_birthyear(birthyear):
    print("Incorrect format for birth year.")
    birthyear = input("Birth year: ")

email = input("Email: ")
while not valid_email(email):
    print("Incorrect format for email.")
    email = input("Email: ")

langs = input("Spoken languages: ")

print("Name: %s" % format_name(name))
print("Birth year: %s (age: %s)" % (birthyear, format_age(birthyear)))
print("Email: %s" % format_email(email))
print("Languages: %s" % format_langs(langs))
```

```
Incorrect format for name.

Incorrect format for birth year.

Incorrect format for email.

Incorrect format for email.

Name: John Smith
Birth year: 1985 (age: 35/36)
Email: johnsmith@elte.hu
Languages: ['english', 'german', 'hungarian']
```

Appendix 2: Mathematical operations

NumPy (https://numpy.org/) is a first-rate library for numerical programming. It is widely used in academia, finance and also in the industry.

The *Pandas* library introduced in Chapter 9 (09_tabular.pdf) is also built on top of *NumPy*, providing high-performance, easy-to-use data structures and data analysis tools, making data manipulation and visualization more convinient.

How to install numpy?

If you have Anaconda installed, then numpy was already installed together with it.

If you have a standalone Python3 and Jupyter Notebook installation, open a command prompt / terminal and type in:

```
pip3 install numpy
```

How to use numpy?

The numpy package is a module which you can simply import. It is usually aliased with the `np` abbreviation:

```
import numpy as np
```

NumPy Arrays

The most important structure that NumPy defines is an array data type formally called a `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html) - for *N dimensional array*.

In [1]:

```
import numpy as np

a = np.zeros(3)
a
```

Out[1]:

```
array([0., 0., 0.])
```

In [2]:

```
type(a)
```

Out[2]:

```
numpy.ndarray
```

NumPy arrays are somewhat like native Python lists, except that:

- data must be homogeneous (all elements of the same type);
- these types must be one of the data types (dtypes) provided by NumPy.;

The most important of these dtypes are:

- `float64` : 64 bit floating-point number
- `int64` : 64 bit integer
- `bool` : 8 bit True or False There are also dtypes to represent complex numbers, unsigned integers, etc.

The default dtype for arrays is `float64` :

In [3]:

```
a = np.zeros(3)
type(a[0])
```

Out[3]:

```
numpy.float64
```

If we want to use integers we can specify it:

In [4]:

```
a = np.zeros(3, dtype=int)
type(a[0])
```

Out[4]:

```
numpy.int64
```

Shape and Dimension

Here `b` is a flat array with no dimension - neither row nor column vector.

The dimension is recorded in the `shape` attribute, which is a tuple.

In [5]:

```
b = np.zeros(10)
b.shape
```

Out[5]:

```
(10,)
```

To give it dimension, we can change the `shape` attribute:

In [6]:

```
b.shape = (10, 1)
b
```

Out[6]:

```
array([[0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.]])
```

Make it a 2 by 2 array:

In [7]:

```
b = np.zeros(4)
b.shape = (2, 2)
b
```

Out[7]:

```
array([[0., 0.],
       [0., 0.]])
```

Dimension can also be specified initially when using the `np.zeros()` function.

In [8]:

```
b = np.zeros((2, 2))
b
```

Out[8]:

```
array([[0., 0.],
       [0., 0.]])
```

You can probably guess what `np.ones` creates.

In [9]:

```
b = np.ones(10)
b
```

Out[9]:

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Creating Arrays

We have already discussed `np.zeros()` and `np.ones()` .

Set up a grid of evenly spaced numbers.

In [10]:

```
b = np.linspace(2, 4, 5)
b
```

Out[10]:

```
array([2. , 2.5, 3. , 3.5, 4. ])
```

Create an identity matrix.

In [11]:

```
b = np.identity(3)
b
```

Out[11]:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

NumPy arrays can be created from Python lists, tuples, etc.

In [12]:

```
b = np.array([10, 20])
b
```

Out[12]:

```
array([10, 20])
```

The data type can also be configured, here `float` is equivalent to `np.float64` :

In [13]:

```
b = np.array((10, 20), dtype=float)
b
```

Out[13]:

```
array([10., 20.])
```

Create a 2 dimensional, 2 by 2 array:

In [14]:

```
b = np.array([[1, 2], [3, 4]])
b
```

Out[14]:

```
array([[1, 2],
       [3, 4]])
```

Array indexing

For a flat array, indexing is the same as Python sequences.

In [15]:

```
c = np.linspace(1, 2, 5)
c
```

Out[15]:

```
array([1.  , 1.25, 1.5 , 1.75, 2.  ])
```

In [16]:

```
c[0]
```

Out[16]:

```
1.0
```

In [17]:

```
c[1:3]
```

Out[17]:

```
array([1.25, 1.5 ])
```

In [18]:

```
c[-1]
```

Out[18]:

```
2.0
```

For 2D arrays we use an index position for each dimension.

In [19]:

```
d = np.array([[1, 2], [3, 4]])
d
```

Out[19]:

```
array([[1, 2],
       [3, 4]])
```

In [20]:

```
d[0, 1]
```

Out[20]:

```
2
```

Note that indices are still zero-based, to maintain compatibility with Python sequences.

Columns and rows can be extracted as follows:

In [21]:

```
d[0, :]
```

Out[21]:

```
array([1, 2])
```

In [22]:

```
d[:, 1]
```

Out[22]:

```
array([2, 4])
```

NumPy arrays of integers can also be used to extract elements.

In [23]:

```
indices = np.array((0, 2, 3))
c[indices]
```

Out[23]:

```
array([1.  , 1.5 , 1.75])
```

A NumPy array of boolean values can be used to filter elements at the `True` locations.

In [24]:

```
e = np.array([0, 1, 1, 0, 0], dtype=bool)
e
```

Out[24]:

```
array([False,  True,  True, False, False])
```

In [25]:

```
c[e]
```

Out[25]:

```
array([1.25, 1.5 ])
```

Array Methods

Numpy arrays have useful methods, many of them should be familiar from previous lectures.

In [26]:

```
f = np.array((3, 2, 4, 1))
f
```

Out[26]:

```
array([3, 2, 4, 1])
```

In [27]:

```
f.sort() # Sorts a in place
f
```

Out[27]:

```
array([1, 2, 3, 4])
```

In [28]:

```
f.sum() # Sum
```

Out[28]:

```
10
```

In [29]:

```
f.mean() # Mean
```

Out[29]:

```
2.5
```

In [30]:

```
f.max() # Max
```

Out[30]:

```
4
```

In [31]:

```
f.argmax() # Returns the index of the maximal element
```

Out[31]:

```
3
```

In [32]:

```
f.cumsum() # Cumulative sum of the elements
```

Out[32]:

```
array([ 1,  3,  6, 10])
```

In [33]:

```
f.cumprod() # Cumulative product of the elements
```

Out[33]:

```
array([ 1,  2,  6, 24])
```

In [34]:

```
f.var() # Variance
```

Out[34]:

```
1.25
```

In [35]:

```
f.std() # Standard deviation
```

Out[35]:

```
1.118033988749895
```

In [36]:

```
f.shape = (2, 2)
f
```

Out[36]:

```
array([[1, 2],
       [3, 4]])
```

In [37]:

```
f.transpose() # or simpy f.T
```

Out[37]:

```
array([[1, 3],
       [2, 4]])
```

Many of the methods discussed above have equivalent functions in the NumPy namespace, e.g.:

In [38]:

```
print("Sum: {0}".format(np.sum(f)))
print("Mean: {0:.2f}".format(np.mean(f)))
```

```
Sum: 10
Mean: 2.50
```

Arithmetic Operations

The operators `+`, `-`, `*`, `/` and `**` all act **elementwise** on NumPy arrays.

In [39]:

```
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
a + b
```

Out[39]:

```
array([ 6,  8, 10, 12])
```

In [40]:

```
a * b
```

Out[40]:

```
array([ 5, 12, 21, 32])
```

In [41]:

```
a + 10
```

Out[41]:

```
array([11, 12, 13, 14])
```

In [42]:

```
a * 10
```

Out[42]:

```
array([10, 20, 30, 40])
```

Multi dimensional arrays follow the same general rules.

In [43]:

```
a.shape = (2, 2)
b.shape = (2, 2)
a + b
```

Out[43]:

```
array([[ 6,  8],
       [10, 12]])
```

In [44]:

```
a + 10
```

Out[44]:

```
array([[11, 12],
       [13, 14]])
```

In [45]:

```
a * b
```

Out[45]:

```
array([[ 5, 12],
       [21, 32]])
```

Calculate the *dot product* of two NumPy arrays.

In [46]:

```
np.dot(a, b)
```

Out[46]:

```
array([[19, 22],
       [43, 50]])
```

The `@` operator does the same thing.

In [47]:

```
a @ b
```

Out[47]:

```
array([[19, 22],
       [43, 50]])
```

Calculate the *cross product* of two NumPy arrays.

In [48]:

```
np.cross(a, b)
```

Out[48]:

```
array([-4, -4])
```

Random generation

Generate random numbers of the *standard normal* distribution:

In [49]:

```
g = np.random.randn(3)
g
```

Out[49]:

```
array([-0.04562699, -0.7312714 , -0.37247379])
```

Generate random integers between a lower (inclusive) and a higher (exclusive) bound:

In [50]:

```
g = np.random.randint(0, 100, 5)
g
```

Out[50]:

```
array([51, 57, 89, 13,  5])
```

Mutability and Copying Arrays

NumPy arrays are mutable data types, like Python lists. In other words, their contents can be altered (mutated) in memory after initialization.

To make an independent copy of a NumPy array, the `np.copy()` function can be used.

In [51]:

```
h = g
i = g.copy()
h[0] = 42

print(g)
print(h)
print(i)
```

```
[42 57 89 13  5]
[42 57 89 13  5]
[51 57 89 13  5]
```

Vectorized Functions

The `np.vectorize()` creates a *vectorized* function, which can be performed on a NumPy array in an elementwise manner.

In [52]:

```
# is_even() can be called on an integer number
def is_even(x): return x % 2 == 0

# is_even_vectorized() can be called on an array of integers
is_even_vectorized = np.vectorize(is_even)
is_even_vectorized(g)
```

Out[52]:

```
array([ True, False, False, False, False])
```

The NumPy function `np.where()` provides a vectorized alternative.

In [53]:

```
np.where(g % 2 == 0, 1, 0)
```

Out[53]:

```
array([1, 0, 0, 0, 0])
```

Comparisons

As a rule, comparisons on arrays are done elementwise.

In [54]:

```
z = np.array([2, 3])
y = np.array([2, 3])
z == y
```

Out[54]:

```
array([ True,  True])
```

In [55]:

```
y[0] = 5
z == y
```

Out[55]:

```
array([False,  True])
```

In [56]:

```
z != y
```

Out[56]:

```
array([ True, False])
```

The situation is similar for `>`, `<`, `>=` and `<=`.

We can also do comparisons against scalars:

In [57]:

```
x = np.linspace(0, 10, 5)
x
```

Out[57]:

```
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

In [58]:

```
x > 3
```

Out[58]:

```
array([False, False,  True,  True,  True])
```

This is particularly useful for *conditional extraction*:

In [59]:

```
cond = x > 3
x[cond]
```

Out[59]:

```
array([ 5. ,  7.5, 10. ])
```

Of course we can - and frequently do - perform this in one step:

In [60]:

```
x[x > 3]
```

Out[60]:

```
array([ 5. ,  7.5, 10. ])
```

Linear algebra

In [61]:

```
k = np.array([[1, 2], [3, 4]])
k
```

Out[61]:

```
array([[1, 2],
       [3, 4]])
```

Compute the determinant:

In [62]:

```
np.linalg.det(k)
```

Out[62]:

```
-2.0000000000000004
```

Compute the inverse:

In [63]:

```
np.linalg.inv(k)
```

Out[63]:

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Interpolation

Generate 20 evenly distributed number between 0 and 10 into `x`. Generate the sine function value into `y` for each elements in `x`.

In [64]:

```
x = np.linspace(0, 10, 20)
y = np.sin(x)
print(x)
print(y)
```

```
[ 0.          0.52631579  1.05263158  1.57894737  2.10526316  2.6315
7895
  3.15789474  3.68421053  4.21052632  4.73684211  5.26315789  5.7894
7368
  6.31578947  6.84210526  7.36842105  7.89473684  8.42105263  8.9473
6842
  9.47368421 10.        ]
[ 0.          0.50235115  0.86872962  0.99996678  0.86054034  0.4881
8921
 -0.01630136 -0.5163796  -0.87668803 -0.99970104 -0.85212237 -0.4738
9753
  0.03259839  0.53027082  0.88441346  0.99916962  0.84347795  0.4594
799
 -0.04888676 -0.54402111]
```

Generate 100 evenly distributed number between 0 and 10 into `xvals`. Calculate the interpolated values into `yinterp` for each elements in `xvals`, based on `x` and `y`.

In [65]:

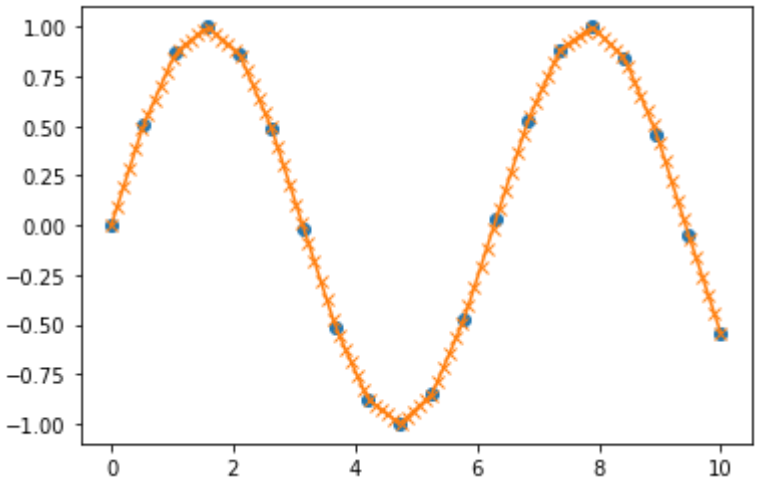
```
xvals = np.linspace(0, 10, 100)
yinterp = np.interp(xvals, x, y)
print(xvals)
print(yinterp)
```

```
[ 0.          0.1010101   0.2020202   0.3030303   0.4040404   0.5050
5051
  0.60606061  0.70707071  0.80808081  0.90909091  1.01010101  1.1111
1111
  ...
  9.09090909  9.19191919  9.29292929  9.39393939  9.49494949  9.5959
596
  9.6969697   9.7979798   9.8989899  10.        ]
[ 0.          0.09641083  0.19282166  0.28923248  0.38564331  0.4820
5414
  0.55786304  0.6281781   0.69849316  0.76880822  0.83912328  0.8833
1152
  ...
  0.32083445  0.22326913  0.12570381  0.0281385  -0.06889218 -0.1639
1797
 -0.25894376 -0.35396954 -0.44899533 -0.54402111]
```

Visualize the results on a plot. *(For plotting, see Chapter 10 (10_plotting.pdf).)*

In [66]:

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot(x, y, 'o')
plt.plot(xvals, yinterp, '-x')
plt.show()
```



Exercise Book 1

Covering the materials of Chapters 1-4.
Topics: control structures, user input, exception handling, random generation, lists, function definition

Task 1: Armstrong numbers

Produce all *Armstrong numbers* smaller than N. The value of N is given by the user. Validate the user input!

A number is an *Armstrong number* (https://en.wikipedia.org/wiki/Narcissistic_number), if it is the sum of its own digits, each raised to the power of the number of digits. For example 153 is an *Armstrong number*, because $1^3 + 5^3 + 3^3 = 153$. The first few *Armstrong numbers* are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, etc.

In [1]:

```
valid_input = False
while not valid_input:
    try:
        N = int(input("N := "))
        valid_input = True
    except:
        print("That is not a number.")

print('Armstrong numbers smaller than %d:' % N)
for number in range(0, N):
    orig_number = number
    digits = len(str(number))
    result = 0
    while number > 0:
        last_digit = number % 10
        number = number // 10
        result += last_digit ** digits
    if result == orig_number:
        print(orig_number)
```

```
Armstrong numbers smaller than 10000:
0
1
2
3
4
5
6
7
8
9
153
370
371
407
1634
8208
9474
```

Task 2: Perfect numbers

Produce the first N *Perfect numbers*. The value of N is given by the user. Validate the user input!

In number theory, a *perfect number* (https://en.wikipedia.org/wiki/Perfect_number) is a positive integer that is equal to the sum of its positive divisors, excluding the number itself. For instance, 6 has divisors 1, 2 and 3 (excluding itself), and $1 + 2 + 3 = 6$, so 6 is a *perfect number*. The first few *perfect numbers* are: 6, 28, 496, etc.

In [2]:

```
valid_input = False
while not valid_input:
    try:
        N = int(input("N := "))
        valid_input = True
    except:
        print("That is not a number.")

print('The first %d perfect numbers:' % N)
found_numbers = 0
number = 1
while found_numbers < N:
    result = 0
    for div in range(1, number):
        if number % div == 0:
            result += div
    if result == number:
        print(number)
        found_numbers += 1
    number += 1
```

```
The first 4 perfect numbers:
6
28
496
8128
```

Task 3: Greatest common divisor

Calculate the *greatest common divisor* of 2 numbers!

Request 2 integer numbers from the user and calculate their greatest common divisor. E.g. for 30 and 105 their greatest common divisor is 15. Do not use the `math.gcd()` built-in function to solve the task.

Hint: use the Euclidean algorithm (https://en.wikipedia.org/wiki/Euclidean_algorithm)

In [3]:

```
def gcd(a, b):
    while a != b:
        if a > b:
            a -= b
        else:
            b -= a
    return a

num1 = int(input('First number: '))
num2 = int(input('Second number: '))

print('The greatest common divisor of %d and %d is %d' %(num1, num2, gcd(num1, n
um2)))
```

```
The greatest common divisor of 30 and 105 is 15
```

Task 4: Rock–paper–scissors

Implement the popular *rock–paper–scissors* (https://en.wikipedia.org/wiki/Rock%E2%80%93paper%E2%80%93scissors) game, where the user can play against the computer! The human player can type in *rock*, *paper* or *scissors*. Handle incorrect input and request the input again if it does not match one of the three previous options. The computer player randomly chooses one of the options. The game finishes when one of the players won. (It continues with another round upon a draw.)

In [4]:

```
import random
options = ['rock', 'paper', 'scissors']
computer = random.choice(options)

# Read user input with validation
def read_user_input():
    user_input = input("Please type in 'rock', 'paper' or 'scissors': ")
    while not user_input in options:
        print("There must have been a typo. Please try again: ")
        user_input = input("Please type in 'rock', 'paper' or 'scissors': ")
    return user_input

user_input = read_user_input()
while user_input == computer:
    print("There is a tie!")
    computer = random.choice(options)
    user_input = read_user_input()

if user_input == "rock":
    if computer == "paper":
        print ("Paper beats rock. The computer won.")
    else:
        print("Rock beats scissors. You won!")
elif user_input == "paper":
    if computer == "rock":
        print("Paper beats rock. You won!")
    else:
        print("Scissors beat paper. The computer won.")
elif user_input == "scissors":
    if computer == "rock":
        print("Rock beats scissors. The computer won.")
    else:
        print ("Scissors beats paper. You won.")
```

```
Rock beats scissors. You won!
```

Task 5: Guess a number

Write a program which can think a of number between 1 and 100, randomly. The task of the user is to guess that number. In each round the user can make a guess, and the program replies whether the guess is correct or it was too small or too large. The game ends when the user succefully guesses the number.

In [5]:

```
import random
number = random.randint(1, 100)

guess = int(input("Guess my number between 1 and 100: "))
while guess != number:
    if guess > number:
        print("Your number is too large. Try again. ")
        guess = int(input("Guess a number: "))
    elif guess < number:
        print("Your number is too small. Try again.")
        guess = int(input("Guess a number: "))
else:
    print("This is the correct number!")
```

```
Your number is too large. Try again.

Your number is too small. Try again.

Your number is too small. Try again.

Your number is too large. Try again.

Your number is too small. Try again.

This is the correct number!
```

Question: if the human player is smart, what is the minimum number of guesses, which is always enough?

Task 6: Separation by parity

Given a *list* of numbers, write a program, which separates the odd and even integers in separate lists. E.g.:

```
Input: [45, 83, 90, 11, 24, 98, 87, 39, 9, 6]
Even numbers: [90, 24, 98, 6]
Odd numbers: [45, 83, 11, 87, 39, 9]
```

Here is a list of 20 random numbers between 1 and 100:

In [6]:

```
import random

numbers = []
for i in range(20):
    numbers.append(random.randint(1, 100))
print(numbers)
```

```
[68, 48, 82, 81, 37, 41, 96, 8, 95, 96, 97, 20, 54, 52, 28, 17, 76,
54, 6, 23]
```

Now write a program which separates them:

In [7]:

```
even_numbers=[]
odd_numbers=[]
for i in numbers:
    if i%2==0:
        even_numbers.append(i)
    else:
        odd_numbers.append(i)
print('The even numbers are: %s' % even_numbers)
print('The odd numbers are: %s' % odd_numbers)
```

```
The even numbers are: [68, 48, 82, 96, 8, 96, 20, 54, 52, 28, 76, 5
4, 6]
The odd numbers are: [81, 37, 41, 95, 97, 17, 23]
```

Task 7: Pyramid

Write the `pyramid(height)` function, which displays a pattern like a pyramid with an asterisk. The height of the pyramid can be defined by the user.

E.g. for $height = 4$, the pyramid would look like:

```
   *
  ***
 *****
*******
```

In [8]:

```
def pyramid(height):
    for i in range(height):
        print(' '*(height-i-1) + '*'*(2*i+1))

h=int(input('Height of the pyramid: '))
pyramid(h)
```

```
     *
    ***
   *****
  *******
 *********
***********
```

Task 8: Collatz sequence

Write a function which produces the *Collatz sequence* and returns it as a list. The function receives the starting value for the sequence. Request the starting value from the user, call the function and display the generated *Collatz sequence*.

The *Collatz sequence* (https://en.wikipedia.org/wiki/Collatz_conjecture) has a starting value and the next item of the sequence is always calculated from the previous one, defined as follows:

- if the number is even, divide it by two;
- if the number is odd, triple it and add one.

More formally: $f(n) = \begin{cases} n/2 & n \equiv 0 \pmod 2 \\ 3n+1 & n \equiv 1 \pmod 2 \end{cases}$

The sequence stops upon reaching 1. For instance, starting with $n = 12$, one gets the sequence 12, 6, 3, 10, 5, 16, 8, 4, 2, 1.

In [9]:

```
def collatz(number):
    sequence=[number]
    while number>1:
        if number%2==0:
            number=number//2
            sequence.append(number)
        else:
            number=3*number+1
            sequence.append(number)
    return sequence

number=int(input('Starting number of Collatz sequence: '))
print(collatz(number))
```

```
[123, 370, 185, 556, 278, 139, 418, 209, 628, 314, 157, 472, 236, 11
8, 59, 178, 89, 268, 134, 67, 202, 101, 304, 152, 76, 38, 19, 58, 2
9, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2,
1]
```

Task 9: Anagram

An anagram is a word or phrase formed by rearranging the letters of a different word or phrase. For example the words *spear* and *pears* are anagrams. Write a function, which decides whether two words are anagrams or not and returns a boolean value accordingly (*True* or *False*).

Hint: pay attention that a single letter may accour multiple times in a word.

In [11]:

```
def is_anagram(word1, word2):
    # If the size does not match, they cannot be anagrams
    if len(word1) != len(word2):
        return False

    # Check for each character in word1 whether it is also in word2
    for ch in word1:
        if not ch in word2:
            return False
    return True

a=input('First word: ')
b=input('Second word: ')
print('Are they anagrams? %s' % is_anagram(a, b))
```

```
Are they anagrams? True
```

Task 10: Circularly identical lists

A list is called *circular* if we consider the first element as next of the last element. *Circularly identical* lists contain the same elements in the same order, given that two lists that can be obtained from each other if one or more of the elements in one of the lists are rotated/displaced from their original index and placed at the beginning. E.g. the lists `[10, 20, 30, 40, 50]` and `[40, 50, 10, 20, 30]` are *circularly identical*, but not with `[50, 40, 10, 20, 30]` .

Write a function which decides whether the given 2 parameter lists are circularly identical or not and returns a boolean value accordingly (*True* or *False*).

In [18]:

```
def is_circularly_identical(listA, listB):
    listA2 = listA + listA
    for i in range(len(listA2)):
        if listA2[i] == listB[0]:
            n = 1
            while (n < len(listB)) and (i + n < len(listA2)) and (listA2[i+n] ==
 listB[n]):
                n += 1
                if n == len(listB):
                    return True

    return False


list1 = [10, 20, 30, 40, 50]
list2 = [40, 50, 10, 20, 30]
list3 = [50, 40, 10, 20, 30]

print('list1 is circularly identical with list2: {0}'.format(is_circularly_ident
ical(list1, list2)))
print('list1 is circularly identical with list3: {0}'.format(is_circularly_ident
ical(list1, list3)))
print('list2 is circularly identical with list3: {0}'.format(is_circularly_ident
ical(list2, list3)))
```

```
list1 is circularly identical with list2: True
list1 is circularly identical with list3: False
list2 is circularly identical with list3: False
```

Exercise Book 2

Covering the materials of Chapters 7-8.
Topics: collection data structures, object oriented programming

In the following 4 lists you will find the country name, capital city name, area (in km^2) and population (in millions) data for 43 European countries respectively.

In [1]:

```
countries = ['Albania', 'Andorra', 'Austria', 'Belgium', 'Bosnia and Herzegovin
a', 'Bulgaria', 'Czech Republic', 'Denmark', 'United Kingdom', 'Estonia', 'Belar
us', 'Finland', 'France', 'Greece', 'Netherlands', 'Croatia', 'Ireland', 'Icelan
d', 'Kosovo', 'Poland', 'Latvia', 'Liechtenstein', 'Lithuania', 'Luxembourg', 'M
acedonia', 'Hungary', 'Malta', 'Moldova', 'Monaco', 'Montenegro', 'Germany', 'No
rway', 'Italy', 'Portugal', 'Romania', 'San Marino', 'Spain', 'Switzerland', 'Sw
eden', 'Serbia', 'Slovakia', 'Slovenia', 'Ukraine']
capitals = ['Tirana', 'Andorra la Vella', 'Vienna', 'Brussels', 'Sarajevo', 'Sof
ia', 'Prague', 'Copenhagen', 'London', 'Tallin', 'Minsk', 'Helsinki', 'Paris',
'Athens', 'Hague', 'Zagreb', 'Dublin', 'Reykjavik', 'Prishtina', 'Warsaw', 'Rig
a', 'Vaduz', 'Vilnius', 'luxembourg', 'Skopje', 'Budapest', 'Valletta', 'Chisina
u', 'Monaco', 'Podgorica', 'Berlin', 'Oslo', 'Rome', 'Lisbon', 'Bucharest', 'San
 Marino', 'Madrid', 'Berne', 'Stockholm', 'Belgrade', 'Bratislava', 'Ljubljana',
 'Kiev']
areas = [28748, 468, 83857, 30519, 51130, 110912, 78864, 43077, 244100, 45100, 2
07600, 338145, 543965, 131957, 33933, 56500, 70283, 103000, 10887, 312683, 63700
, 160, 65200, 2586, 25713, 93036, 316, 33700, 2, 13812, 357042, 323877, 301277,
92389, 237500, 61, 504782, 41293, 449964, 66577, 49035, 20250, 603700]
populations = [3.2, 0.07, 7.6, 10.0, 4.5, 9.0, 10.4, 5.1, 57.2, 1.6, 10.3, 4.9,
56.2, 10.0, 14.8, 4.7, 3.5, 0.3, 2.2, 37.8, 2.6, 0.03, 3.6, 0.4, 2.1, 10.4, 0.3,
 4.4, 0.03, 0.6, 78.6, 4.2, 57.5, 10.5, 23.2, 0.03, 38.8, 6.7, 8.5, 7.2, 5.3, 2.
0, 51.8]
```

Let's display the data stored in all lists:

In [2]:

```
print("Countries:")
print(countries)
print("----------")
print("Capitals:")
print(capitals)
print("----------")
print("Areas (in km2):")
print(areas)
print("----------")
print("Populations (in millions):")
print(populations)
```

```
Countries:
['Albania', 'Andorra', 'Austria', 'Belgium', 'Bosnia and Herzegovin
a', 'Bulgaria', 'Czech Republic', 'Denmark', 'United Kingdom', 'Esto
nia', 'Belarus', 'Finland', 'France', 'Greece', 'Netherlands', 'Croa
tia', 'Ireland', 'Iceland', 'Kosovo', 'Poland', 'Latvia', 'Liechtens
tein', 'Lithuania', 'Luxembourg', 'Macedonia', 'Hungary', 'Malta',
'Moldova', 'Monaco', 'Montenegro', 'Germany', 'Norway', 'Italy', 'Po
rtugal', 'Romania', 'San Marino', 'Spain', 'Switzerland', 'Sweden',
'Serbia', 'Slovakia', 'Slovenia', 'Ukraine']
----------
Capitals:
['Tirana', 'Andorra la Vella', 'Vienna', 'Brussels', 'Sarajevo', 'So
fia', 'Prague', 'Copenhagen', 'London', 'Tallin', 'Minsk', 'Helsink
i', 'Paris', 'Athens', 'Hague', 'Zagreb', 'Dublin', 'Reykjavik', 'Pr
ishtina', 'Warsaw', 'Riga', 'Vaduz', 'Vilnius', 'luxembourg', 'Skopj
e', 'Budapest', 'Valletta', 'Chisinau', 'Monaco', 'Podgorica', 'Berl
in', 'Oslo', 'Rome', 'Lisbon', 'Bucharest', 'San Marino', 'Madrid',
'Berne', 'Stockholm', 'Belgrade', 'Bratislava', 'Ljubljana', 'Kiev']
----------
Areas (in km2):
[28748, 468, 83857, 30519, 51130, 110912, 78864, 43077, 244100, 4510
0, 207600, 338145, 543965, 131957, 33933, 56500, 70283, 103000, 1088
7, 312683, 63700, 160, 65200, 2586, 25713, 93036, 316, 33700, 2, 138
12, 357042, 323877, 301277, 92389, 237500, 61, 504782, 41293, 44996
4, 66577, 49035, 20250, 603700]
----------
Populations (in millions):
[3.2, 0.07, 7.6, 10.0, 4.5, 9.0, 10.4, 5.1, 57.2, 1.6, 10.3, 4.9, 5
6.2, 10.0, 14.8, 4.7, 3.5, 0.3, 2.2, 37.8, 2.6, 0.03, 3.6, 0.4, 2.1,
10.4, 0.3, 4.4, 0.03, 0.6, 78.6, 4.2, 57.5, 10.5, 23.2, 0.03, 38.8,
6.7, 8.5, 7.2, 5.3, 2.0, 51.8]
```

The index position of the elements in the lists ties the information for each country together:

In [3]:

```
for idx in range(len(countries)):
    print("Name: %s, Capital: %s, Area: %d km2, Population: %.2f millions" % (co
untries[idx], capitals[idx], areas[idx], populations[idx]))
```

```
Name: Albania, Capital: Tirana, Area: 28748 km2, Population: 3.20 mi
llions
Name: Andorra, Capital: Andorra la Vella, Area: 468 km2, Population:
0.07 millions
Name: Austria, Capital: Vienna, Area: 83857 km2, Population: 7.60 mi
llions
...
Name: Slovakia, Capital: Bratislava, Area: 49035 km2, Population: 5.
30 millions
Name: Slovenia, Capital: Ljubljana, Area: 20250 km2, Population: 2.0
0 millions
Name: Ukraine, Capital: Kiev, Area: 603700 km2, Population: 51.80 mi
llions
```

Task 1: List of dictionaries

Storing the data in 4 separate lists is not comfortable. Construct a list of dictionaries programatically:

- each item in the list is a dictionary;
- each dictionary contains the relevant information for a single country.

The result should be like the following:

```
[
    {
        'country': 'Albania',
        'capital': 'Tirana',
        'area': 28748,
        'population': 3.2
    },

    ...

    {
        'country': 'Ukraine',
        'capital': 'Kiev',
        'area': 603700,
        'population': 51.8
    }
]
```

In [4]:

```
dataset = []
for idx in range(len(countries)):
    dataset.append({
        'country': countries[idx],
        'capital': capitals[idx],
        'area': areas[idx],
        'population': populations[idx]
    })
print(dataset)
```

```
[{'country': 'Albania', 'capital': 'Tirana', 'area': 28748, 'populat
ion': 3.2}, {'country': 'Andorra', 'capital': 'Andorra la Vella', 'a
rea': 468, 'population': 0.07}, {'country': 'Austria', 'capital': 'V
ienna', 'area': 83857, 'population': 7.6},
...,
{'country': 'Slovakia', 'capital': 'Bratislava', 'area': 49035, 'pop
ulation': 5.3}, {'country': 'Slovenia', 'capital': 'Ljubljana', 'are
a': 20250, 'population': 2.0}, {'country': 'Ukraine', 'capital': 'Ki
ev', 'area': 603700, 'population': 51.8}]
```

Task 2: Population density

Calculate the population density for each country (in people / km^2 unit) and extends each country with this information.

The result should be like the following:

```
[
    {
        'country': 'Albania',
        'capital': 'Tirana',
        'area': 28748,
        'population': 3.2,
        'density': 111.31209127591485
    },

    ...

    {
        'country': 'Ukraine',
        'capital': 'Kiev',
        'area': 603700,
        'population': 51.8,
        'density': 85.80420738777539
    }
]
```

In [5]:

```
for item in dataset:
    item['density'] = item['population'] * 1e6 / item['area']
print(dataset)
```

```
[{'country': 'Albania', 'capital': 'Tirana', 'area': 28748, 'populat
ion': 3.2, 'density': 111.31209127591485}, {'country': 'Andorra', 'c
apital': 'Andorra la Vella', 'area': 468, 'population': 0.07, 'densi
ty': 149.57264957264957}, {'country': 'Austria', 'capital': 'Vienn
a', 'area': 83857, 'population': 7.6, 'density': 90.63047807577185},

...,
{'country': 'Slovakia', 'capital': 'Bratislava', 'area': 49035, 'pop
ulation': 5.3, 'density': 108.08606097685326}, {'country': 'Sloveni
a', 'capital': 'Ljubljana', 'area': 20250, 'population': 2.0, 'densi
ty': 98.76543209876543}, {'country': 'Ukraine', 'capital': 'Kiev',
'area': 603700, 'population': 51.8, 'density': 85.80420738777539}]
```

Task 3: Highest density

Find the country with the highest population density.

In [6]:

```
max_idx = 0

for idx in range(1, len(dataset)):
    if dataset[idx]['density'] > dataset[max_idx]['density']:
        max_idx = idx

print(dataset[max_idx])
```

```
{'country': 'Monaco', 'capital': 'Monaco', 'area': 2, 'population':
0.03, 'density': 15000.0}
```

Task 4: Object oriented approach

Task A): define a class named `Country`, which can store a country's name, capitaly city, area and population. Construct a list of *objects*, where each object is an instance of the `Country` class.

Task B): add a `density()` method to the `Country` class, which calculates the population density for that country dynamically. Find the country with the highest population density.

In [7]:

```
class Country():
    def __init__(self, name, capital, area, population):
        self.name = name
        self.capital = capital
        self.area = area
        self.population = population

    def density(self):
        return self.population * 1e6 / self.area

dataset2 = []
for idx in range(len(countries)):
    dataset2.append(Country(countries[idx], capitals[idx], areas[idx], populatio
ns[idx]))

max_idx = 0
for idx in range(1, len(dataset2)):
    if dataset2[idx].density() > dataset2[max_idx].density():
        max_idx = idx

print(dataset2[max_idx].name)
```

```
Monaco
```

Exercise Book 3

Covering the materials of Chapters 9-10.
Topics: tabular data, plotting and diagram visualization

Open and read the attached `data/airports.csv` file, containing information about (larger) airports all over the world:

1. IATA code (International Air Transport Association code, e.g. *BUD* for the Budapest Airport)
2. ICAO code (International Civil Aviation Organization code, e.g. *LHBP* for the Budapest Airport)
3. Name
4. Number of runways
5. Longest runway length (in foots)
6. Elevation (in foots)
7. Country
8. Country region
9. City
10. Latitude
11. Longitude

The columns in each row are delimited with `;` characters (instead of the default `,`).

In [1]:

```
import pandas as pd

airports = pd.read_csv('../data/airports.csv', delimiter=';')
display(airports)
```

	iata	icao	name	runways	longest	elevation	country	region	city	
0	ATL	KATL	Hartsfield - Jackson Atlanta International Air...	5	12390	1026	US	US-GA	Atlanta	33
1	ANC	PANC	Anchorage Ted Stevens	3	12400	151	US	US-AK	Anchorage	61
2	AUS	KAUS	Austin Bergstrom International Airport	2	12250	542	US	US-TX	Austin	30
3	BNA	KBNA	Nashville International Airport	4	11030	599	US	US-TN	Nashville	36
4	BOS	KBOS	Boston Logan	6	10083	19	US	US-MA	Boston	42
...	...	...	...	...	...	...	...	...	...	
3459	LNL	ZLLN	Cheng Xian Airport	1	9186	3707	CN	CN-62	Longnan	33
3460	XAI	ZHXY	Xinyang Minggang Airport	1	8858	4528	CN	CN-41	Xinyang	32
3461	YYA	ZGYY	Sanhe Airport	1	8530	230	CN	CN-43	Yueyang	29
3462	BQJ	UEBB	Batagay Airport	2	6562	699	RU	RU-SA	Batagay	67
3463	DPT	UEBD	Deputatskij Airport	1	7021	920	RU	RU-SA	Deputatskij	69

3464 rows × 11 columns

Task 1

Write a program that calculates and prints for each country the number of airports in that country. Sort the list by the number of airports.

In [2]:

```
display(airports.groupby('country').count()['iata'].sort_values(ascending=False
))
```

```
country
US    583
CN    217
CA    205
AU    130
RU    126
     ...
GM      1
NR      1
GN      1
GP      1
KW      1
Name: iata, Length: 231, dtype: int64
```

Task 2

Write a program that calculates and prints which city has the highest elevation. If a city has multiple airports, calculate the average (mean) elevation of the airports in that city.

In [3]:

```
display(airports.groupby('city').mean()['elevation'].idxmax())
```

```
'Daocheng'
```

Task 3

Write a program that displays the city names which has at least 5 runways accumulated. Sort the city list by the number of runways decreasing and also display the number of runway in each city.

Note: keep in mind that a city might have multiple airports!

In [4]:

```
airports_city = airports.groupby('city').sum()
display(airports_city[airports_city['runways'] >= 5].sort_values(by='runways', ascending=False)['runways'])
```

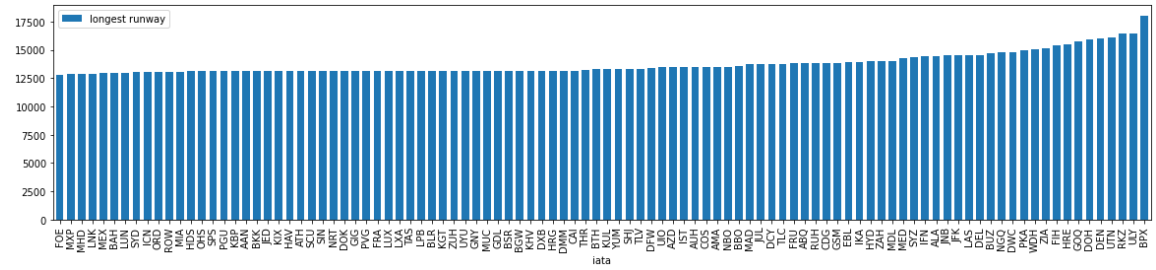
```
city
Chicago            12
Dallas             10
Houston             9
London              9
Denver              9
Hamilton            8
...
Carlsbad            5
Barcelona           5
Atlanta             5
Alexandria          5
Wilmington          5
Name: runways, dtype: int64
```

Task 4

Create a *bar plot*, displaying length of the longest runway for each airport. The aiports shall be sorted by the longest runway length (ascending). Visualize only the top 100 aiports, so the diagram will be readable. Set an appropriate figure size, so all bars and labels are readable.

In [5]:

```
import matplotlib.pyplot as plt
%matplotlib inline

airports.sort_values(by='longest').tail(100).plot(kind='bar', x='iata', y='longest', figsize=[20,4], width=0.7, label='longest runway')
plt.show()
```

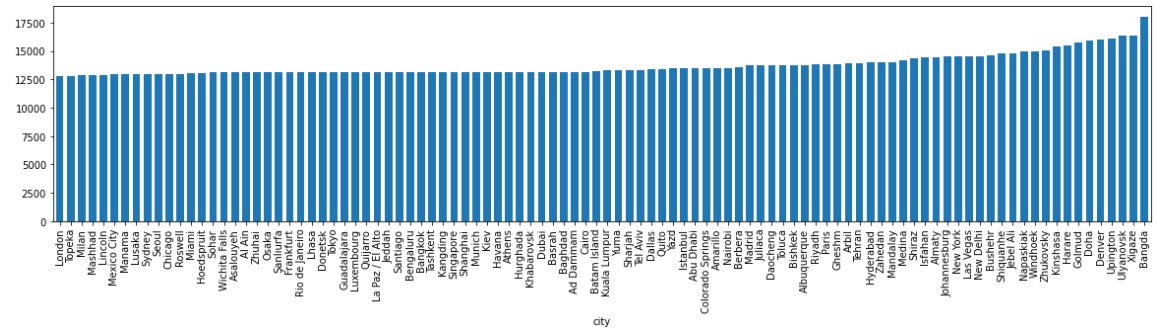


Task 5

Create a *bar plot*, displaying length of the longest runway for each **city**. The cities shall be sorted by the longest runway length (ascending). Visualize only the top 100 cities, so the diagram will be readable. Set an appropriate figure size, so all bars and labels are readable.

In [6]:

```
airports.groupby('city').max()['longest'].sort_values().tail(100).plot(kind='bar', figsize=[20,4], width=0.7, label='longest runway')
plt.show()
```



Exercise Book 4

Covering the materials of Chapter 11.
Topics: vector spatial data management with geopandas

In the attached `data` folder the following attached datasets are given for this assignment:

- `hungary_admin_8.shp` , containing the city level administrative boundaries of Hungary. (Data source: *OpenStreetMap* (https://data2.openstreetmap.hu/hatarok/))
- `hungary_population_2020.csv` , containing the population of Hungarian cities on 2020 January 1. (Data source: *Hungarian Government* (https://www.nyilvantarto.hu/hu/statisztikak))
- `hungary_population_2011.csv` , containing the population of Hungarian cities on 2011 January 1. (Data source: *Hungarian Government* (https://www.nyilvantarto.hu/hu/statisztikak))

Note: in the CSV files the columns are delimited with `;` characters (instead of the default `,`).

Task 1

Write a program that creates a thematic map for Hungary based on the adminstrative boundaries of the cities and their population in 2020.
(Use the *All population* field from the CSV file.)

In [1]:

```
import pandas as pd
import geopandas as gpd

# Read the datasets
cities = gpd.read_file('../data/hungary_admin_8.shp')
cities = cities[['NAME', 'geometry']]
cities.set_index('NAME', inplace=True)

population_2020 = pd.read_csv('../data/hungary_population_2020.csv', delimiter =
 ';')
population_2020.set_index('City', inplace=True)
```

In [2]:

```
# Add the population DataSeries to the cities "manually"
df = cities.copy()
df['All population'] = [None] * len(cities)

# Get the indexes which are present in both DataFrames
indexes = set(cities.index) & set(population_2020.index)
for index in indexes:
    df.loc[index, 'All population'] = population_2020.loc[index]['All populatio
n']

display(df)
```

NAME	geometry	All population
Murakeresztúr	POLYGON ((1875811.200 5837364.810, 1875829.320...	1712
Tótszerdahely	POLYGON ((1865447.010 5842664.860, 1865626.780...	1081
Molnári	POLYGON ((1871422.780 5840886.420, 1871468.690...	689
Semjénháza	POLYGON ((1874690.000 5845206.400, 1874749.090...	566
Felsőszölnök	POLYGON ((1793789.650 5920727.330, 1793969.030...	578
...	...	...
Milota	POLYGON ((2530430.020 6120050.180, 2530441.900...	998
Tiszabecs	POLYGON ((2535824.870 6121698.150, 2535957.370...	1550
Garbolc	POLYGON ((2543379.140 6098625.170, 2543444.730...	150
Magosliget	POLYGON ((2540997.680 6116051.390, 2541064.470...	332
Beregdaróc	POLYGON ((2502185.370 6141427.990, 2502438.910...	1038

3174 rows × 2 columns

In [3]:

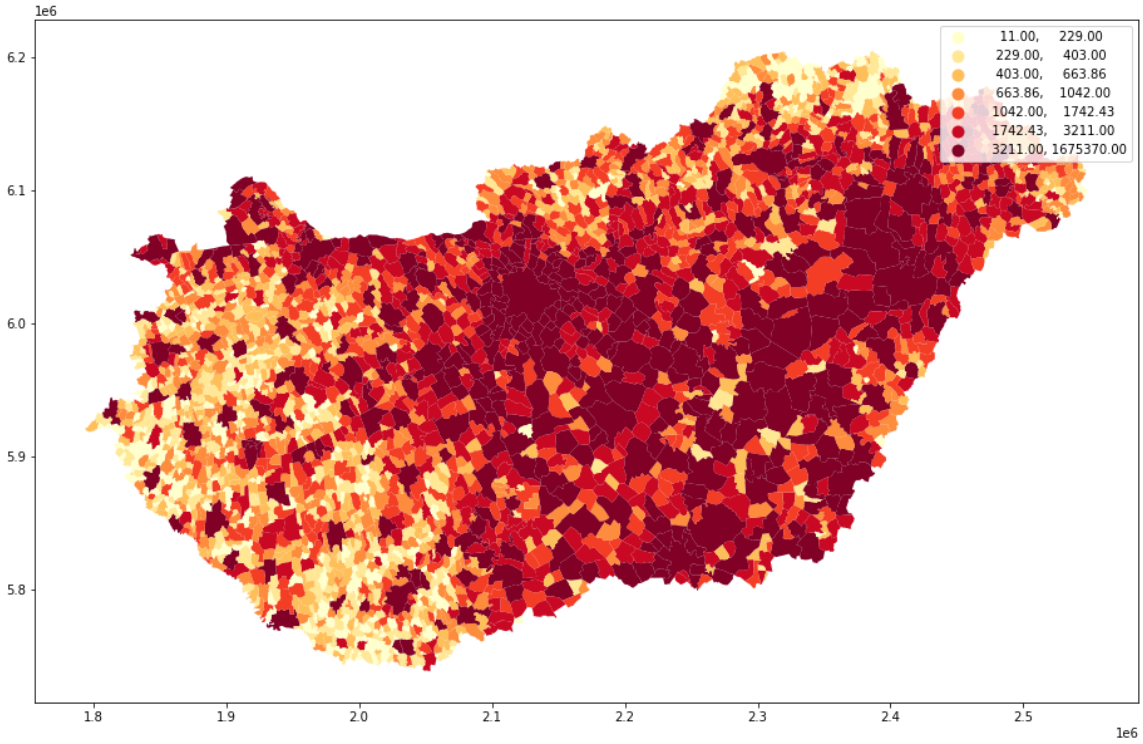
```
# This can be done in an easier and more efficient way with pandas' merge() func
tion
df = cities.merge(population_2020, left_index=True, right_index=True)
display(df)
```

	geometry	County	Male population	Female population	All population
Aba	POLYGON ((2053298.230 5953037.810, 2053422.400...	FEJ	2286	2359	4645
Abaliget	POLYGON ((2010640.140 5804436.120, 2010699.370...	BAR	351	334	685
Abasár	POLYGON ((2223181.920 6072744.320, 2223372.480...	HEV	1168	1297	2465
Abaújalpár	POLYGON ((2362488.130 6156262.020, 2362552.180...	BOR	37	32	69
Abaújkér	POLYGON ((2354110.960 6161271.720, 2354200.650...	BOR	290	314	604
...	...	...	...	...	...
Őrimagyarósd	POLYGON ((1837790.790 5925016.100, 1837832.070...	VAS	117	113	230
Őriszentpéter	POLYGON ((1821634.850 5916224.320, 1821691.900...	VAS	568	589	1157
Őrtilos	POLYGON ((1878544.010 5830691.040, 1878570.100...	SOM	250	230	480
Ősagárd	POLYGON ((2132697.110 6082877.080, 2132823.710...	NOG	149	183	332
Ősi	POLYGON ((2020338.210 5965613.320, 2020523.530...	VES	1048	1027	2075

3174 rows × 5 columns

In [4]:

```
import matplotlib.pyplot as plt
%matplotlib inline

# Create the plot
df.plot(column='All population', figsize=[20,10], legend=True, cmap='YlOrRd', sc
heme='quantiles', k=7)
plt.show()
```



Task 2

Write a program that adds the population data for 2011 and 2020 to the Shapefile as new scalar fields to each city; and save it as a new Shapefile.

In [5]:

```
population_2011 = pd.read_csv('../data/hungary_population_2011.csv', delimiter =
 ';')
population_2011.set_index('City', inplace=True)

df = df.merge(population_2011, left_index=True, right_index=True, suffixes=[' 20
20', ' 2011'])
df.rename(columns={'County 2020':'County'}, inplace=True)
del df['County 2011']
display(df)
```

	geometry	County	Male population 2020	Female population 2020	All population 2020	Male population 2011	Fe popula
Aba	POLYGON ((2053298.230 5953037.810, 2053422.400...	FEJ	2286	2359	4645	2273	
Abaliget	POLYGON ((2010640.140 5804436.120, 2010699.370...	BAR	351	334	685	315	
Abasár	POLYGON ((2223181.920 6072744.320, 2223372.480...	HEV	1168	1297	2465	1191	
Abaújalpár	POLYGON ((2362488.130 6156262.020, 2362552.180...	BOR	37	32	69	46	
Abaújkér	POLYGON ((2354110.960 6161271.720, 2354200.650...	BOR	290	314	604	329	
...	...	...	...	...	...	...	
Őrimagyarósd	POLYGON ((1837790.790 5925016.100, 1837832.070...	VAS	117	113	230	114	
Őriszentpéter	POLYGON ((1821634.850 5916224.320, 1821691.900...	VAS	568	589	1157	571	
Őrtilos	POLYGON ((1878544.010 5830691.040, 1878570.100...	SOM	250	230	480	285	
Ősagárd	POLYGON ((2132697.110 6082877.080, 2132823.710...	NOG	149	183	332	148	
Ősi	POLYGON ((2020338.210 5965613.320, 2020523.530...	VES	1048	1027	2075	1086	

3173 rows × 8 columns

In [6]:

```
# Save it to file
df.to_file('hungary_population.shp')
```

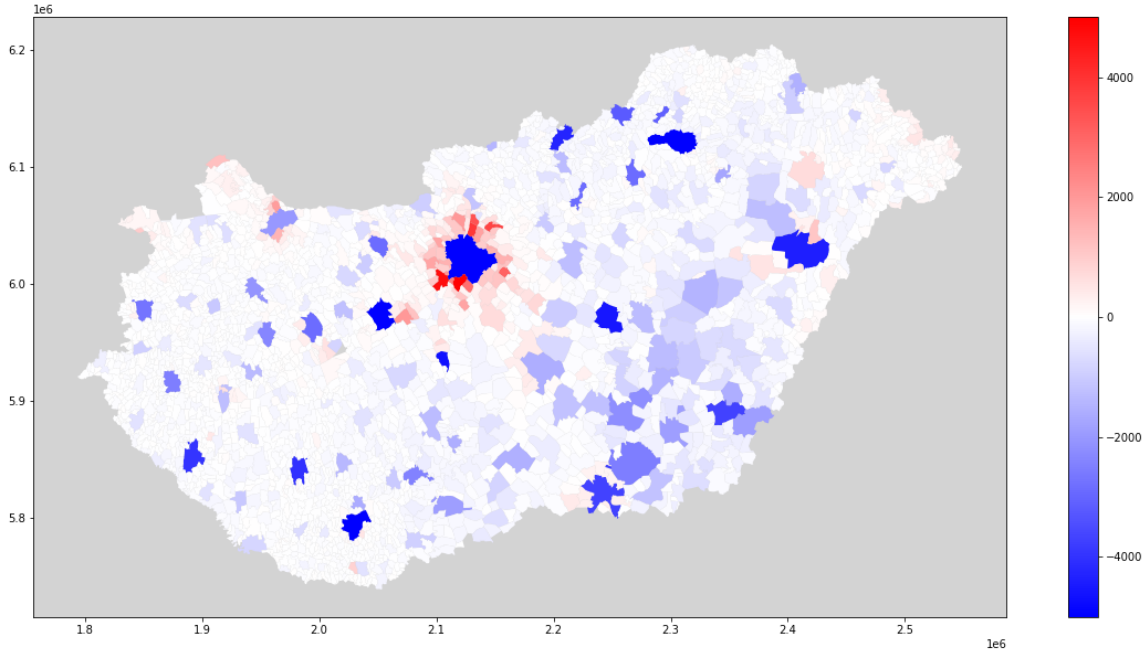
```
<ipython-input-6-ac2d3a3b95d7>:2: UserWarning: Column names longer t
han 10 characters will be truncated when saved to ESRI Shapefile.
  df.to_file('hungary_population.shp')
```

Task 3

Write a program that creates a thematic map for Hungary based on the adminstrative boundaries of the cities and their population change between 2011 and 2020.

In [7]:

```
df['Population difference'] = df['All population 2020'] - df['All population 201
1']

ax = df.plot(column='Population difference', figsize=[20,10], legend=True, cmap=
'bwr', vmin=-5000, vmax=5000)
ax.set_facecolor("lightgray") # background color
plt.show()
```

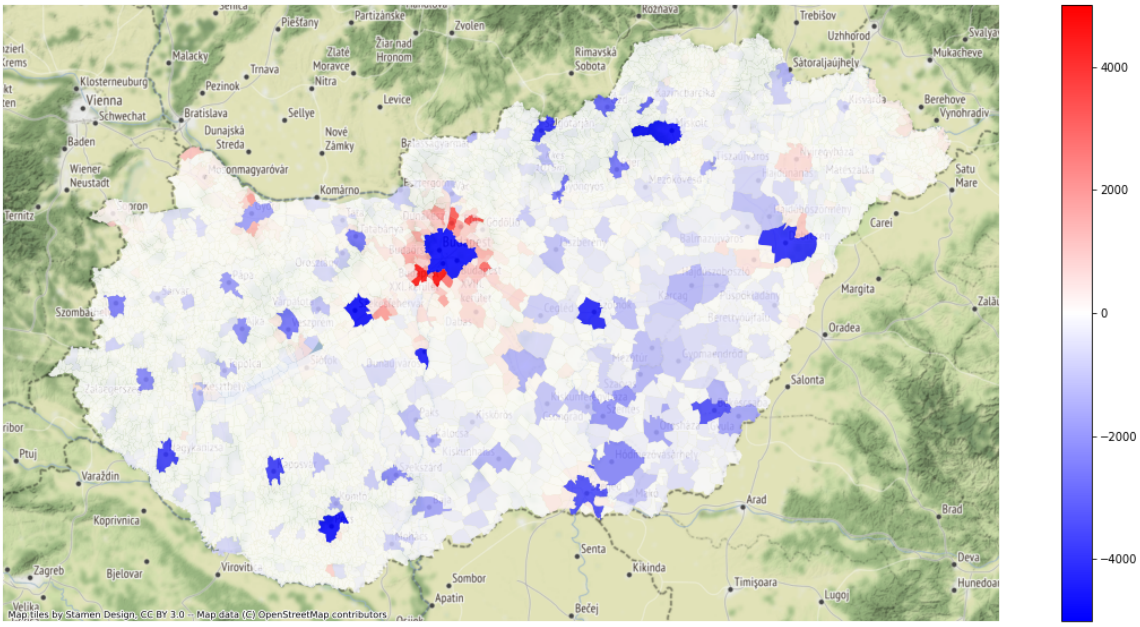


Optional: add a raster basemap with *contextily*.

In [8]:

```
# How to install: conda install -c conda-forge contextily
# How to use: https://contextily.readthedocs.io/en/latest/
import contextily as ctx

# Verify CRS, must be Web Mercator (EPSG:3857) to add a base map with the contex
tily module.
print(df.crs)
if df.crs == 'epsg:3857':
    ax = df.plot(column='Population difference', figsize=[20,10], legend=True, c
map='bwr', vmin=-5000, vmax=5000, alpha=0.85)
    ctx.add_basemap(ax)
    ax.set_axis_off()
    plt.show()
else:
    print('CRS must be EPSG:3857, instead {0} was given'.format(df.crs))
```

```
epsg:3857
```



Task 4

Write a program that creates a thematic map for Hungary based on the adminstrative boundaries of the cities and their population density in 2020.

In [9]:

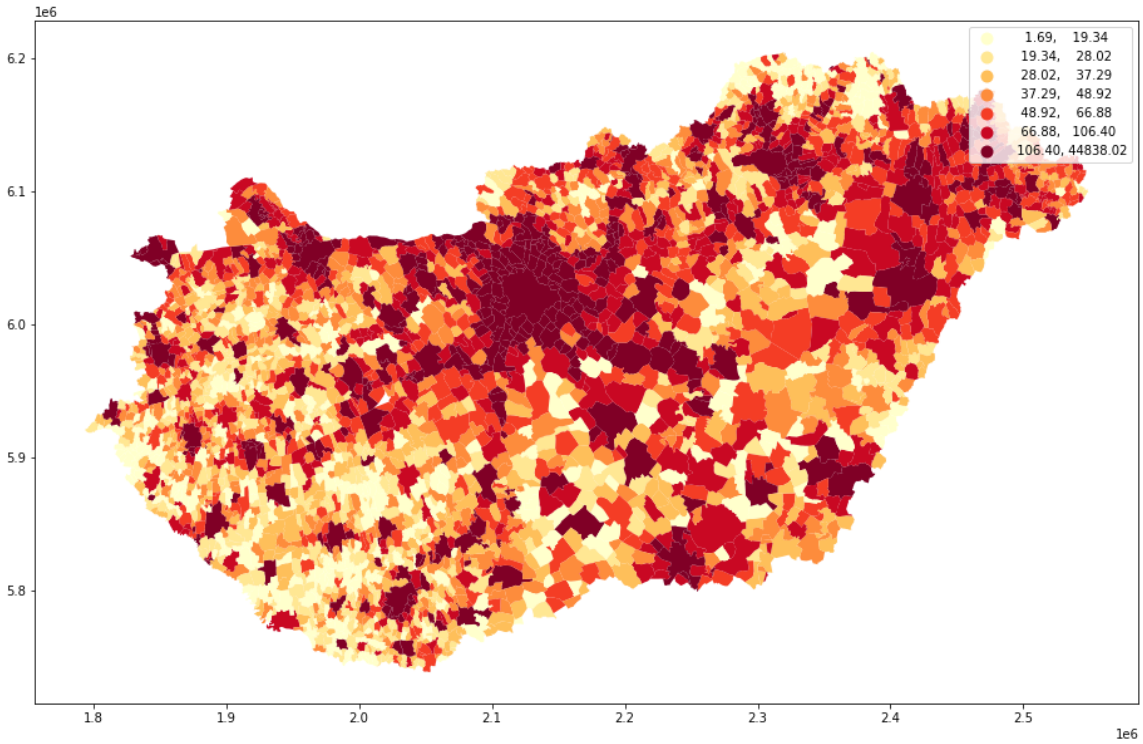
```
df_eov = df.to_crs('EPSG:23700') # EOV is EPSG:23700
df['Area'] = df_eov.area / 10**6
df['Density 2020'] = df['All population 2020'] / df['Area']
display(df)
```

	geometry	County	Male population 2020	Female population 2020	All population 2020	Male population 2011	Fe popula
Aba	POLYGON ((2053298.230 5953037.810, 2053422.400...	FEJ	2286	2359	4645	2273	
Abaliget	POLYGON ((2010640.140 5804436.120, 2010699.370...	BAR	351	334	685	315	
Abasár	POLYGON ((2223181.920 6072744.320, 2223372.480...	HEV	1168	1297	2465	1191	
Abaújalpár	POLYGON ((2362488.130 6156262.020, 2362552.180...	BOR	37	32	69	46	
Abaújkér	POLYGON ((2354110.960 6161271.720, 2354200.650...	BOR	290	314	604	329	
...	...	...	...	...	...	...	
Őrimagyarósd	POLYGON ((1837790.790 5925016.100, 1837832.070...	VAS	117	113	230	114	
Őriszentpéter	POLYGON ((1821634.850 5916224.320, 1821691.900...	VAS	568	589	1157	571	
Őrtilos	POLYGON ((1878544.010 5830691.040, 1878570.100...	SOM	250	230	480	285	
Ősagárd	POLYGON ((2132697.110 6082877.080, 2132823.710...	NOG	149	183	332	148	
Ősi	POLYGON ((2020338.210 5965613.320, 2020523.530...	VES	1048	1027	2075	1086	

3173 rows × 11 columns

In [10]:

```
df.plot(column='Density 2020', figsize=[20,10], legend=True, cmap='YlOrRd', sche
me='quantiles', k=7)
plt.show()
```



Exercise Book 5

Covering the materials of Chapters 12-14.
Topics: graph algorithms and spatial indexing

In the attached `data` folder the following shapefiles are available for this assignment

- `osm_roads_hungary.shp` , containing the road network of Hungary as linestrings. (Data source: *OpenStreetMap* (https://download.geofabrik.de/europe/hungary.html))
- `hungary_cities.shp` , containing the Hungarian cities as points. (Data source: *ELTE FI, Institute of Cartography and Geoinformatics*)

The `osm_roads_hungary.shp` file is in *WGS 84 (EPSG:4326)* coordinate reference system. The projection of the `hungary_cities.shp` file is not defined, but the data is in *EOV (EPSG:23700)*, which you have to set it manually.

Task 1

Read the input data from the Shapefiles.
Display the cities and the road network on a map using the *matplotlib* library. (Roads shall be colored black and cities shall be red dots.)

Among the roads filter only the more significant types. **In all the following tasks you will only have to work with these type of roads.**
More significant type of roads are where the `fclass` column of the *GeoDataFrame* is among the following values:
`motorway` , `primary` , `secondary` , `tertiary` , `motorway_link` , `primary_link` , `secondary_link` , `tertiary_link` .

Hint: reproject the roads to EOV before displaying the GeoDataFrame.

In [1]:

```
import geopandas as gpd

# Read the shapefiles
roads = gpd.read_file('../data/osm_roads_hungary.shp')
cities = gpd.read_file('../data/hungary_cities.shp')

# Set the CRS to EOV projection (EPSG:23700) if None for cities
if cities.crs == None:
    cities.set_crs('epsg:23700', inplace=True)

# Display road types
print(roads['fclass'].unique())
```

```
['residential' 'secondary' 'primary' 'tertiary' 'service' 'unclassif
ied'
 'footway' 'primary_link' 'track' 'pedestrian' 'steps' 'secondary_li
nk'
 'path' 'cycleway' 'motorway_link' 'motorway' 'tertiary_link' 'trunk
_link'
 'living_street' 'track_grade2' 'trunk' 'track_grade4' 'track_grade
1'
 'track_grade5' 'track_grade3' 'bridleway' 'unknown']
```

In [2]:

```
# Filter roads by their type
roads = roads[roads['fclass'].isin(['motorway', 'primary', 'secondary', 'tertiar
y',
                                    'motorway_link', 'primary_link', 'secondary_
link', 'tertiary_link'])]
```

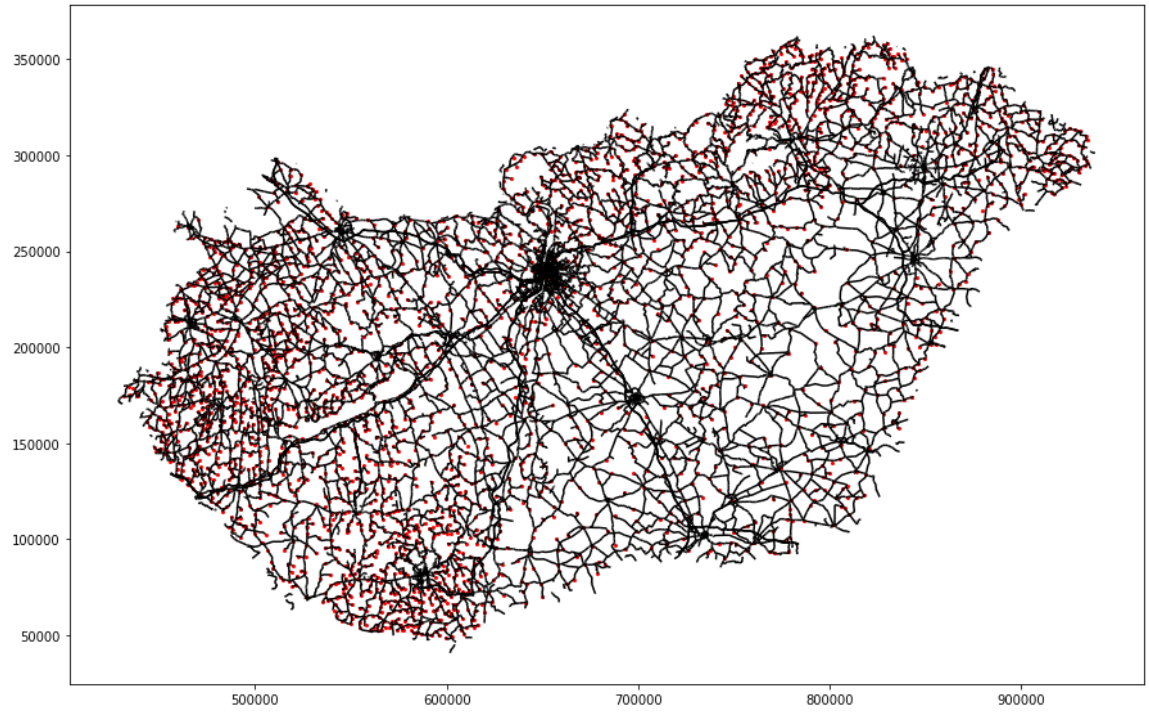
In [3]:

```
print("Cities CRS: {0}, Roads CRS: {1}".format(cities.crs, roads.crs))

# Convert CRS to EOV
if cities.crs != 'epsg:23700':
    cities.to_crs('epsg:23700', inplace=True)

if roads.crs != 'epsg:23700':
    roads.to_crs('epsg:23700', inplace=True)

print("Cities CRS: {0}, Roads CRS: {1}".format(cities.crs, roads.crs))
```

```
Cities CRS: epsg:23700, Roads CRS: epsg:4326
Cities CRS: epsg:23700, Roads CRS: epsg:23700
```

In [4]:

```
import matplotlib.pyplot as plt
%matplotlib inline

# Plot roads and cities
base = roads.plot(figsize=[15,10], color='black')
cities.plot(ax=base, color='red', markersize=4)

# Display plot
plt.show()
```



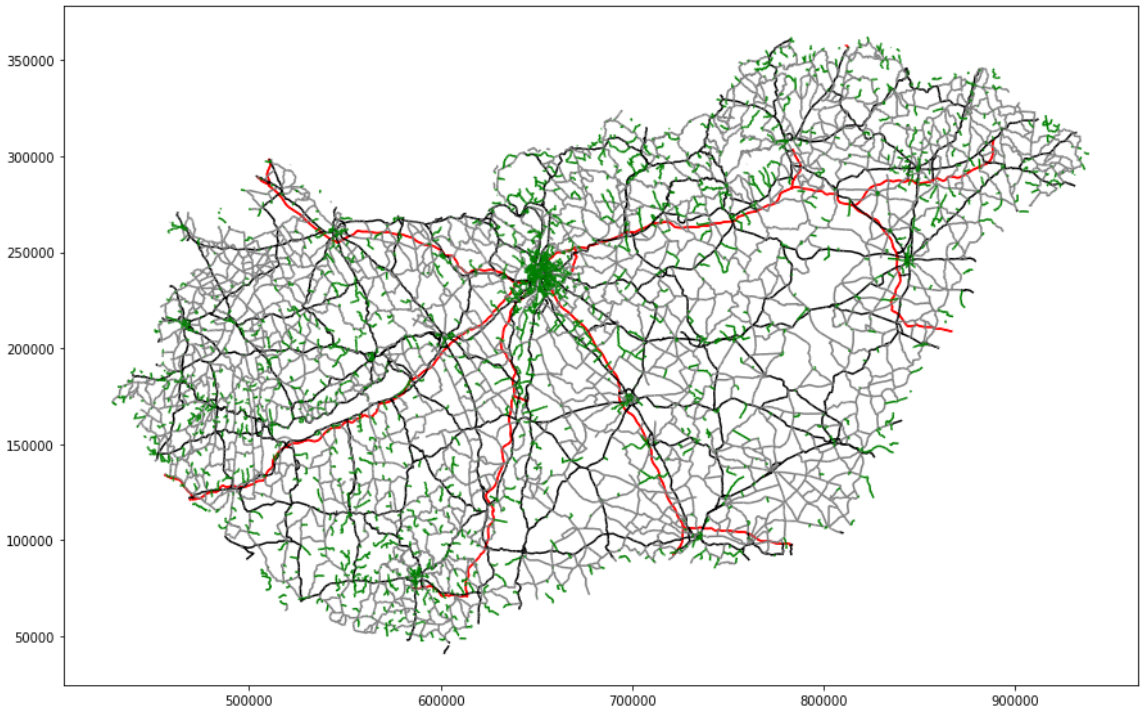
Task 2

Display the road network of Hungary, but with a different coloring based on their class:

- motorways should be red;
- primary roads should be black;
- secondary roads should be gray;
- tertiary roads should be green.

In [5]:

```
base = roads[roads['fclass'].isin(['motorway', 'motorway_link'])].plot(figsize=[
15,10], color='red')
roads[roads['fclass'].isin(['primary', 'primary_link'])].plot(ax=base, color='bl
ack')
roads[roads['fclass'].isin(['secondary', 'seondary_link'])].plot(ax=base, color=
'gray')
roads[roads['fclass'].isin(['tertiary', 'tertiary_link'])].plot(ax=base, color=
'green')

# Display plot
plt.show()
```



Task 3

Build a graph of the road network and compute the shorthest path (based on distance) between Győr and Debrecen.
Display the complete road network (with black color) on a map and overlay the found shorthest path wirh red color.

Pairs of EOV X and Y coordinates of the cities are given below, which shall be precisely on the roads.

- Győr: (261414.51778597923, 544944.4764306903)
- Debrecen: (247370.13702113688, 842839.3118560591)

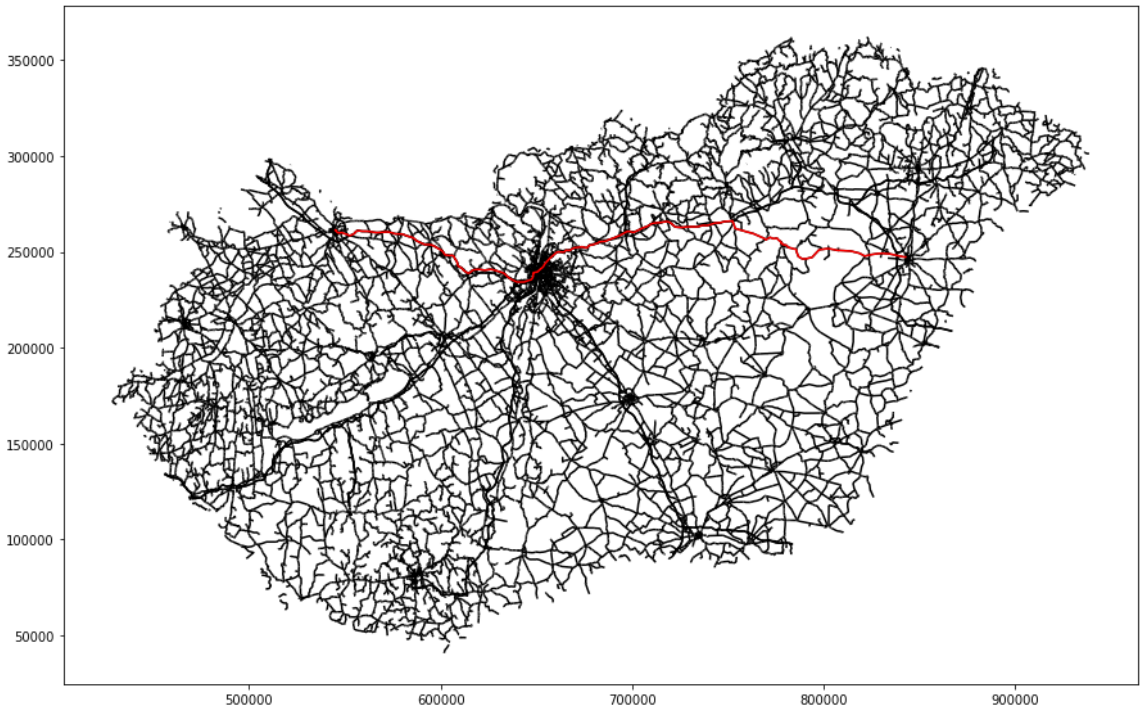
Hint: the coordinates in the Shapefiles are in a switched (Y, X) order.

In [6]:

```
import math
import networkx as nx

# Create empty, undirected graph
graph = nx.Graph()

# Iterate through all linestrings
for idx, row in roads.iterrows():
    line = row['geometry']
    for i in range(1, len(line.coords)):
        p1 = line.coords[i-1]
        p2 = line.coords[i]

        # Since the file is in the EOV CRS, we can calulate the distance in SI
        # based on the Pythagoras theorem
        dist = math.sqrt(pow(p1[0] - p2[0], 2) + pow(p1[1] - p2[1], 2)) / 1000

        # Add it to the graph
        graph.add_edge(p1, p2, distance = dist, road_class = row['fclass'])
```

In [7]:

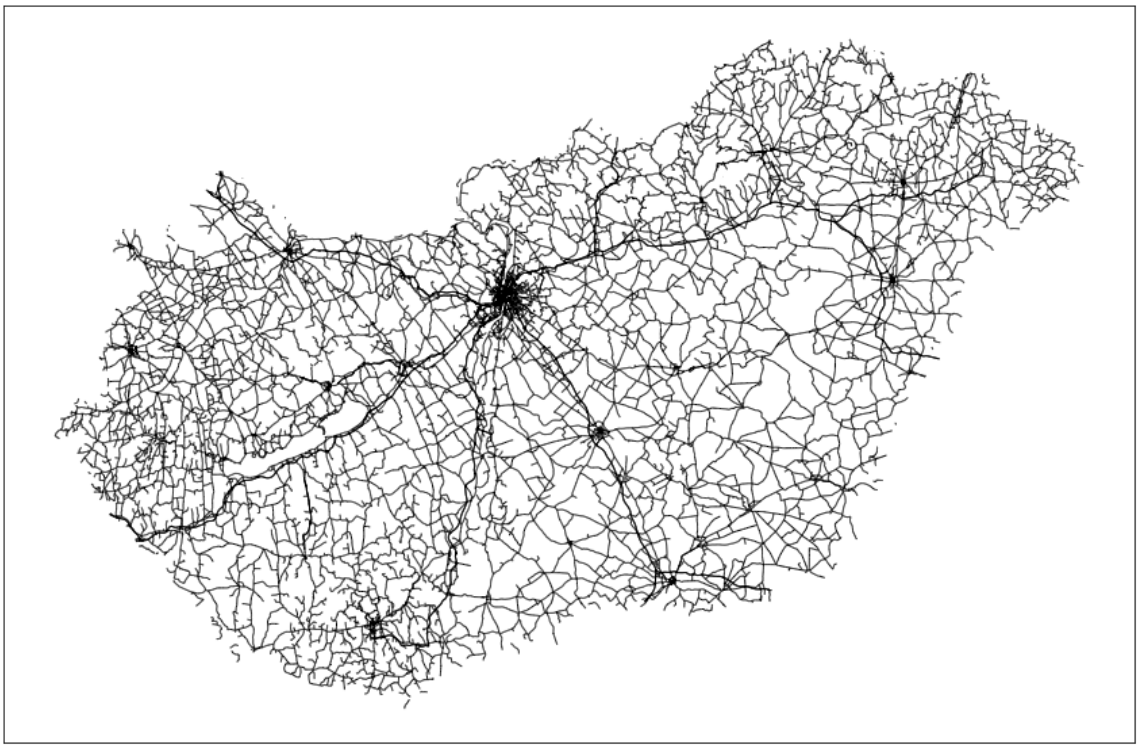
```
# Given positions
pos_gyor = (544944.4764306903, 261414.51778597923)
pos_debrecen = (842839.3118560591, 247370.13702113688)

# Calculate the shortest distance between Berlin and Budapest
path = nx.shortest_path(graph, pos_gyor, pos_debrecen, weight = 'distance')

# Draw roads
roads.plot(figsize=[15,10], color='black')

# Draw path function
def draw_path(path):
    # Get the X an Y positions into separate lists for the shortest path
    xs = [coord[0] for coord in path]
    ys = [coord[1] for coord in path]
    # Add it to the plot
    plt.plot(xs, ys, color='red')

# Draw path
draw_path(path)

# Display plot
plt.show()
```



Alternative approach: draw the graph with *networkx*

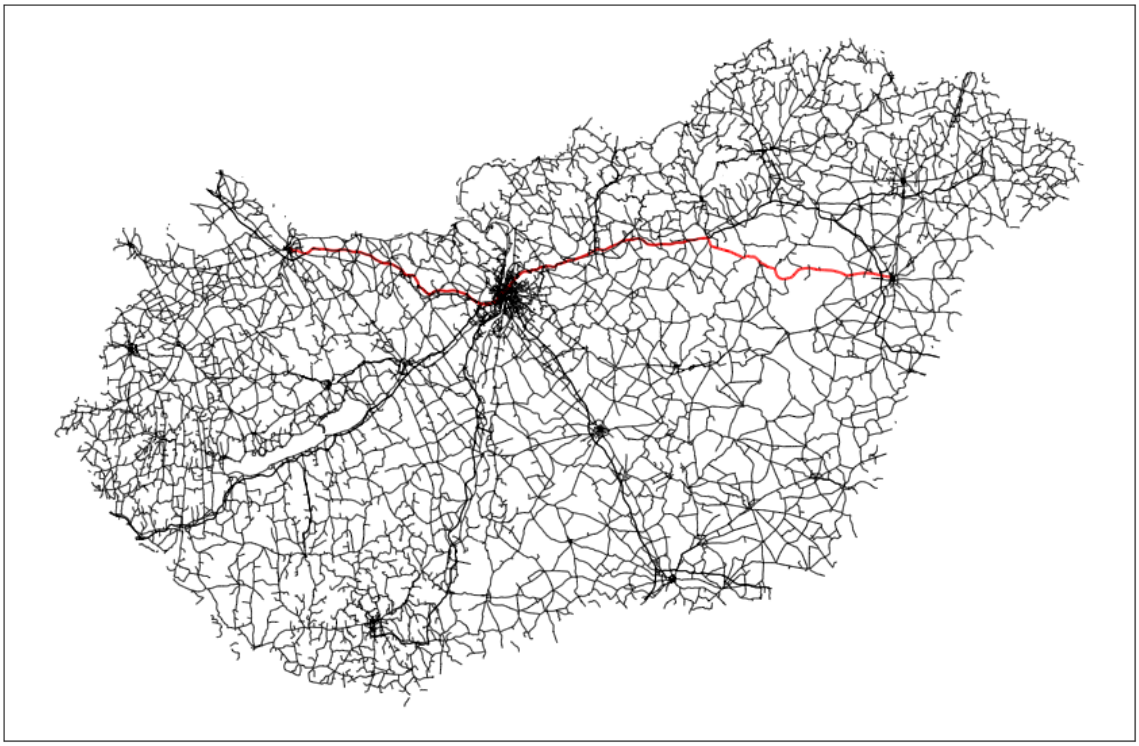
In [8]:

```
# Add the position to the nodes as attributes
for idx, row in roads.iterrows():
    line = row['geometry']
    for i in range(len(line.coords)):
        p = line.coords[i]
        graph.nodes[p]['position'] = p

# Draw the graph
plt.figure(figsize=[15,10])
nx.draw_networkx(graph, nx.get_node_attributes(graph, 'position'), with_labels=F
alse, node_size=0)
plt.show()
```



In [9]:

```
# Add default color and width attributes to all edges
for source, target in graph.edges:
    graph[source][target]['color'] = 'black'
    graph[source][target]['width'] = 1.0

# Modify the color and width attributes for the edges on the shorthest path
for i in range(1, len(path)):
    source = path[i-1]
    target = path[i]
    graph[source][target]['color'] = 'red'
    graph[source][target]['width'] = 2.0

# Compute list of edge colors and widths
edge_color_list = [graph[source][target]['color'] for source, target in graph.ed
ges]
edge_width_list = [graph[source][target]['width'] for source, target in graph.ed
ges]

# Draw the graph
plt.figure(figsize=[15,10])
nx.draw_networkx(graph, nx.get_node_attributes(graph, 'position'), with_labels=F
alse, node_size=0,
                 edge_color = edge_color_list, width = edge_width_list)
plt.show()
```



Task 4

Let the user define the start and target coordinates for the shorthest path search. (Input validation is not required.)
Since the user given coordinates are not necessarily on any of the roads, it is required to find the nearest vertices in the graph (both for the start and the goal).

Test data:

- Győr: (261473, 545052)
- Debrecen: (247367, 842842)

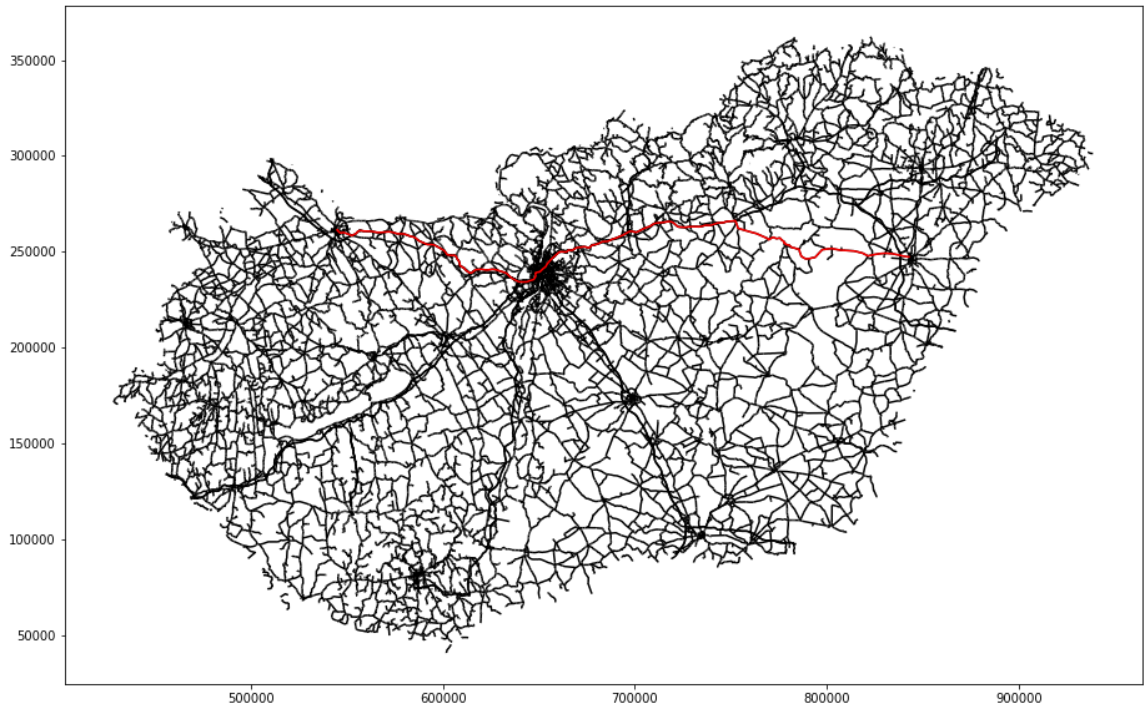
Hint: build a Kd-tree form the vertices of the graph.

In [10]:

```
import scipy.spatial # for Kd-Tree support

# Put all points on road network into list
all_points = []

for line in roads.geometry:
    for coord in line.coords:
        all_points.append(coord)

all_points = list(set(all_points))

# Build a Kd-tree
kdtree = scipy.spatial.KDTree(all_points)
```

In [13]:

```
# Read user input
from_x = float(input('Start X coordinate: '))
from_y = float(input('Start Y coordinate: '))
to_x = float(input('Target X coordinate: '))
to_y = float(input('Target Y coordinate: '))

def closest_point(point):
    dist, idx = kdtree.query(point)
    return all_points[idx]

# Find closest points in road network
from_point = closest_point((from_y, from_x))
to_point = closest_point((to_y, to_x))

# Calculate the shortest distance between the given coordinates
path = nx.shortest_path(graph, from_point, to_point, weight = 'distance')

# Draw roads
roads.plot(figsize=[20, 10], color='black')

# Draw path
draw_path(path)

# Display plot
plt.show()
```



Task 5

Let the user define the name of the start and the target city instead of coordinates.
The Hungarian cities and their location is given in the `hungary_cities.shp` Shapefile. (These locations marks the centorid of the cities are not necessarily on any of the roads.)

Validate the user input whether the given cities exist. Show an error message if not.

In [14]:

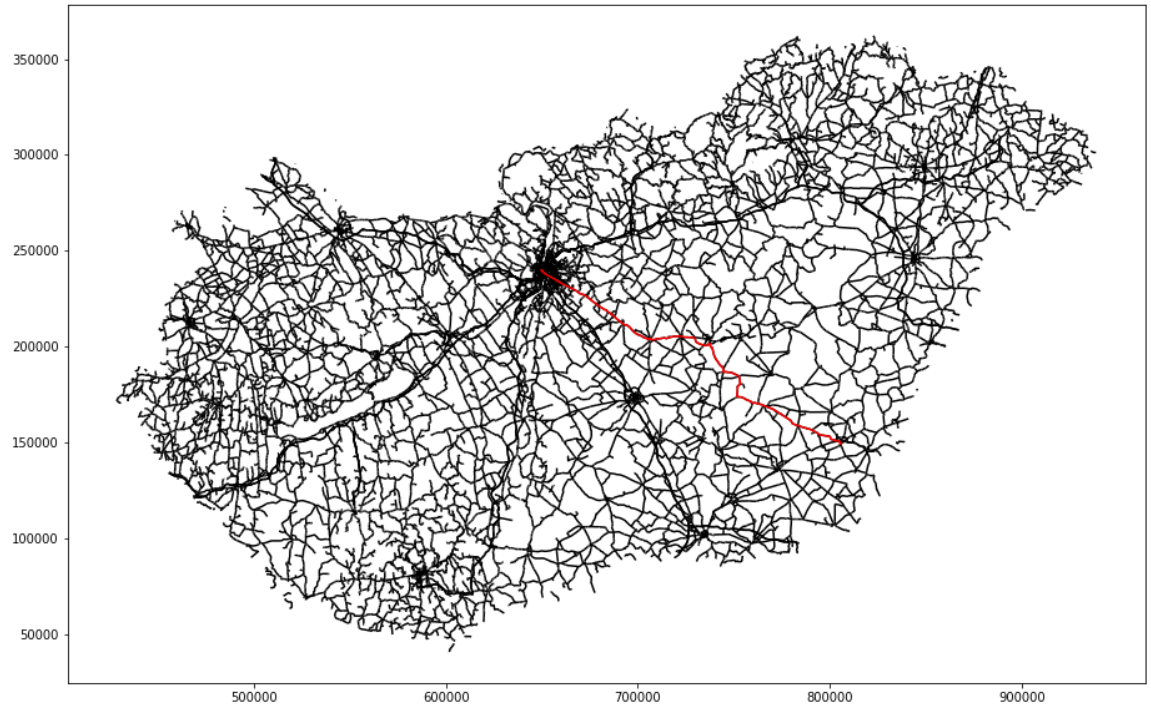
```
# Set index in cities GeoDataFrame
cities.set_index('City', inplace=True)
display(cities)
```

City	Id	County	Status	KSH	geometry
Aba	1	FEJÉR	town	17376	POINT (610046.800 187639.000)
Abaliget	2	BARANYA	town	12548	POINT (577946.100 89280.800)
Abasár	3	HEVES	town	24554	POINT (721963.700 273880.300)
Abaújalpár	4	BORSOD-ABAUJ-ZEMPLÉN	town	15662	POINT (812129.200 331508.200)
Abaújkér	5	BORSOD-ABAUJ-ZEMPLÉN	town	26718	POINT (809795.600 331138.300)
...	...	...	...	...	...
Zsira	3143	GYÕR-MOSON-SOPRON	town	04622	POINT (471324.200 237577.200)
Zsombó	3144	CSONGRÁD	town	17765	POINT (721098.100 109690.000)
Zsujta	3145	BORSOD-ABAUJ-ZEMPLÉN	town	11022	POINT (815027.400 353143.100)
Zsurk	3146	SZABOLCS-SZATMÁR-BEREG	town	13037	POINT (884847.700 344952.800)
Zubogy	3147	BORSOD-ABAUJ-ZEMPLÉN	town	19105	POINT (763123.300 338338.600)

3147 rows × 5 columns

In [15]:

```
# Read user input
from_city = input('Start city: ')
to_city = input('Target city: ')

# Check whether the given cities exist in the dataset
if from_city in cities.index and to_city in cities.index:
    # Find coordinates for cities
    from_x = cities.loc[from_city].geometry.y
    from_y = cities.loc[from_city].geometry.x
    to_x = cities.loc[to_city].geometry.y
    to_y = cities.loc[to_city].geometry.x

    # Find closest points in road network
    from_point = closest_point((from_y, from_x))
    to_point = closest_point((to_y, to_x))

    # Calculate the shortest distance between the given cities
    path = nx.shortest_path(graph, from_point, to_point, weight = 'distance')

    # Draw roads
    roads.plot(figsize=[20, 10], color='black')

    # Draw path
    draw_path(path)

    # Display plot
    plt.show()
else:
    print('Start or target city not found!')
```



Task 6

Instead of calculating the shortest path based on geographical distance between two user given cities, calculate the shorthest path based on the travel time! Let's define the following speed limits for the various classes of roads:

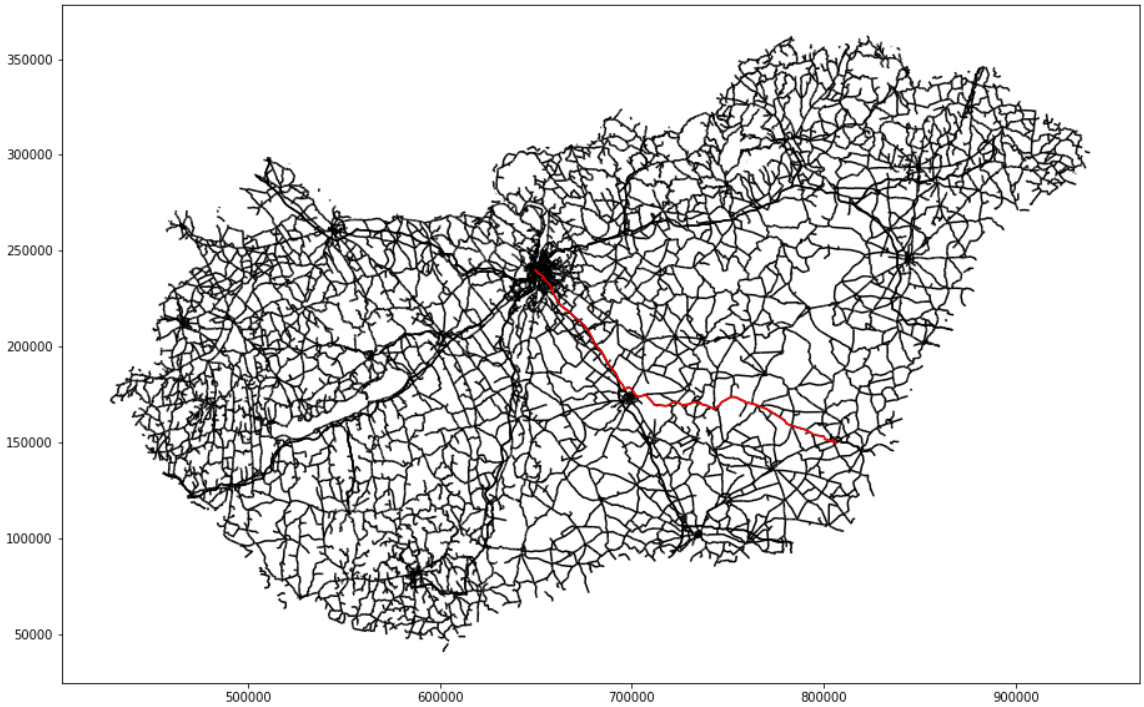
- motorway: 130 km/h
- primary road: 90 km/h
- secondary road: 70 km/h (since their condition is degraded)
- tertiary road: 50 km/h

We assume the that user an drive with the speed limit on all roads.

Test data: the Budapest - Békéscsaba path should be different when the shortest path based on geographical distance between and shortest path based on travel time is calculated.

In [16]:

```
def custom_distance(from_node, to_node, edge_attr):
    speed_limit = 50
    if edge_attr['road_class'] in ['motorway', 'motorway_link']:
        speed_limit = 130
    if edge_attr['road_class'] in ['primary', 'primary_link']:
        speed_limit = 90
    if edge_attr['road_class'] in ['secondary', 'secondary_link']:
        speed_limit = 70
    return edge_attr['distance'] / speed_limit

# Read user input
from_city = input('Start city: ')
to_city = input('Target city: ')

# Check whether the given cities exist in the dataset
if from_city in cities.index and to_city in cities.index:
    # Find coordinates for cities
    from_x = cities.loc[from_city].geometry.y
    from_y = cities.loc[from_city].geometry.x
    to_x = cities.loc[to_city].geometry.y
    to_y = cities.loc[to_city].geometry.x

    # Find closest points in road network
    from_point = closest_point((from_y, from_x))
    to_point = closest_point((to_y, to_x))

    # Calculate the shortest distance  between the given cities
    path = nx.shortest_path(graph, from_point, to_point, weight = custom_distanc
e)

    # Draw roads
    roads.plot(figsize=[20, 10], color='black')

    # Draw path
    draw_path(path)

    # Display plot
    plt.show()
else:
    print('Start or target city not found!')
```


350000
300000
250000
200000
150000
100000
50000
500000
600000
700000
800000
900000