USW Computational Basics – Lecture Notes

Georg Jäger and Manfred Füllsack

Chapter 1 –Programming Basics

What is programming?

Programming in general is the activity of **giving instructions to a computer**. Typing an arithmetic instruction into a calculator can therefore be considered programming. You type in the calculation you would like to perform and the computer does the arithmetic work. Simple pocket calculators, however, know only a few commands, mostly only basic arithmetic. More complex ones know, for example, trigonometric functions or the possibility to store and recall numbers. If you want to perform even more complex calculations, you have to rely on "real" computers.

In contrast to the pocket calculator with its keys for each basic arithmetic operation, the computer must now be told that, for example, the root of a number is to be calculated, by using a **programming language**. This could be, for example, the command `sqrt(x)`, which - depending on the so-called syntax of a language - is available in one form or another in every programming language. This has the advantage that the knowledge of one programming language usually ensures that **other programming languages are also easily learned**. In addition, programming skills increase the general problem-solving competence, the ability to think abstractly and to express oneself precisely in a scientific context.

The programming language Python

The lecture - and thus this script - refers to the programming language Python. This has several reasons: Python (https://www.python.org) is a universal, versatile programming language, which

1. is **easy to learn** due to its clear and concise syntax,
2. has become a scientific **standard** in many disciplines
3. is **open** in all its basics and therefore free to use,
4. includes an extensive **standard library** and numerous freely accessible special modules
5. provides an easy to install and comfortable browser-based **programming environment**
6. is constantly being developed further by a **very large community**.

Anaconda

Anaconda (https://www.anaconda.com/) is a so called Python distribution that combines a variety of tools necessary for programming with Python. Anaconda is free (i.e. free of charge), very easy to install, and available for Windows, Mac-OS and Linux. Anaconda contains more than 150 included modules (packages) and basically everything (and much more) that is necessary for programming in the field of environmental systems sciences.

Above all, Anaconda already contains a very comfortable input and output environment (i.e. a development environment) called Jupyter Notebook, which simply runs in the browser you are used to using for your daily Internet surfing (Google-Chrome, Firefox, Internet Explorer ...). That means, after installing Anaconda, no additional or unfamiliar programs are necessary to start working with Python immediately. In the following the installation process for Anaconda with Python 3.7. on a Windows computer is described.

Another Windows installation description can be found here:
https://docs.continuum.io/anaconda/install/windows

For Mac OS installation, see here: https://docs.continuum.io/anaconda/install/mac-os

For Linux see here: https://docs.continuum.io/anaconda/install/linux

Anaconda Installation for Python 3.8

- Open the link https://www.anaconda.com/products/individual#Downloads in your browser.

- If you have an adequate computer, click on the button "64-bit graphical installer" to get Python 3.8 version. Otherwise choose the 32-bit version. This will start the download of the installation file. This may take a few minutes. Make sure that the path to the folder where you save your downloads and the folder name itself does not contain any special characters or spaces. If this is the case, please copy the downloaded file into an appropriate folder before you execute it.
- Double click on the .exe file you downloaded.
- Click the "Run" button in the automatically reopened window.
- Follow the other dialog windows.
- In the License Agreement window, click on the "I Agree" button if you agree to the license terms.
- Follow the dialog windows again. We recommend that you accept the suggested information. In the window *Choose Install Location* you should again make sure that the path to the folder in which you want to install Anaconda, and also the folder name itself does not contain special characters or spaces. With a click on the button "Browse..." you can change the automatically suggested folder and select your desired location. Click "Next >" again to continue with the installation.
- Click on the "Install" button and then "Finish" to complete the installation.

After successful installation, you should see a menu item Anaconda in your Windows start menu, which in turn contains a menu - sub-item Jupyter Notebook.

Jupyter Notebook

The Jupyter Notebook is a very comfortable input and output environment for the programming language Python, i.e. a development environment that runs in the browser already installed on your computer. All examples in this script were written with the Jupyter Notebook and are also displayed in it.

To start the Jupyter Notebook, click on the menu item Jupyter Notebook in the Windows Start menu under Anaconda.

A window opens, which is first black and then fills with white text. This window must not be closed as long as you want to work with the Jupyter Notebook. This is a display for the so-called kernel, the process in which the actual calculations are performed. In addition, your browser, or a new tab in your browser will open.

To open a new Jupyter Notbook, click on the "New" drop-down menu on the right side of this browser tab and select "Python 3".

Chapter 2 – Introduction to Python – The frog pond

Using Jupyter-Notebooks

Jupyter notebooks consist of cells. There are three types of cells:

- Text cells, like this one, in which you can write and format texts
- In-cells, where you can write Python code
- Out-cells, which output the result of the In-cell above them

To execute an in-cell, simply click into the desired cell and press the Shift and Enter keys simultaneously. An out-cell is automatically produced, which contains the result of the in-cell.

Even without any programming knowledge you can already use Python, even if only as a calculator:

In [1]:

```
7 * 7 - 7
```

Out[1]:

```
42
```

In [2]:

```
((12 + 144 + 20 + 3 * 4**0.5) / 7) + 5 * 11
# Exponents are specified as x**y in Python. Thus 4**0.5 means 4 to the pow
# er of 0.5.
# The text behind a rhombus (#) is ignored by the computer and is intended
# as information for users.
```

Out[2]:

```
81.0
```

But Python can do much more than a calculator. In a cell you can write not only single calculations, but whole programs or complete simulations. Let us try to create a very simple simulation to learn the basic commands of Python. Suppose we want to simulate the population development in a frog pond. In the very first approximation we assume that 3 new frogs settle in the pond every year.

Define variables

To store numbers (like the number of frogs in a pond) a so-called variable must be created. This is so to speak the name under which the number is stored. The assignment of a variable name to a number is done in Python with the = sign. It is important that the name is to the left of the = sign and the number to be stored there is to the right. So we start our pond simulation by setting the number of frogs (= *froschanzahl*) to 0.

In [3]:

```
froschanzahl = 0
```

Changing variable values

To change a variable, you can simply overwrite the current value with the new value. This is also done with the = sign.

In [4]:

```
# Year 1:
froschanzahl = 3
```

Retrieving variables

To display variables again use the `print` command. This command writes the current value of the queried variable to the output line. If we write `print(froschanzahl)`, the result should be 3.

In [5]:

```
print(froschanzahl)
```

```
3
```

This is very helpful for our further simulation, because we can use the current number of frogs to calculate the new one. According to our assumption, the number of frogs is 3 more each year than before, mathematically expressed as the `froschanzahl + 3`. Thus, our frog simulation looks like this:

In [6]:

```
froschanzahl = 0
#Year 1
froschanzahl = froschanzahl + 3
#Year 2
froschanzahl = froschanzahl + 3
#Year 3
froschanzahl = froschanzahl + 3
#Year 4
froschanzahl = froschanzahl + 3
#Year 5
froschanzahl = froschanzahl + 3
# To output a result we use the command "print".
print(froschanzahl)
```

```
15
```

In this version, our frog simulation is not only very unspectacular, but also quite inconvenient. But it is a good starting point for a better simulation. As a first step it would be nice if we could set the number of years to be simulated and not always have to simulate exactly 5 years. For this we need an important concept: the For-loop.

For-loops

For-loops are used to execute a command several times in a row, in our case adding frogs. This is what our program would look like with a For-loop:

In [7]:

```
froschanzahl = 0
for it in range(5):
    froschanzahl = froschanzahl + 3
print(froschanzahl)
```

```
15
```

This is much more compact, but therefore also a little more complicated. The first line is exactly the same, we set the `froschanzahl` to 0 and the next command `for` introduces the command for the For-loop. You could read this line as: Make the following command for each integer `it`

that is less than 5, i.e. 5 times in total (for 0,1,2,3,4). After the colon comes the command to be executed in the next line. At the end we let the result be printed again.

Indenting in Python

How does Python know now that we don't want the last command (the `print`) to be executed 5 times? This is done by indenting (tab key). Commands that are directly below each other belong together for Python. This automatically makes Python code easy to read and you don't need brackets. If we really want to have the output after each year, and not at the end, we can indent the print command with the tab key. That way it belongs to the For-loop:

In [8]:

```
froschanzahl = 0
for it in range(5):
    froschanzahl = froschanzahl + 3
    print(froschanzahl)
```

```
3
6
9
12
15
```

Whether commands are inside or outside a For-loop makes a big difference and is a common source of error. If, for example, you would also include the line `froschanzahl = 0` in the For-loop, the number of frogs would be set to 0 each time the loop is passed:

In [9]:

```
for it in range(5):
    froschanzahl = 0
    froschanzahl = froschanzahl + 3
    print(froschanzahl)
```

```
3
3
3
3
3
```

Nevertheless, using a For-loop offers a huge advantage: we only need to change a single number to change the number of years simulated. For example, let's simulate 10 years:

In [10]:

```
froschanzahl = 0
for it in range(10):
    froschanzahl = froschanzahl + 3
print(froschanzahl)
```

```
30
```

A nice style of programming is to put the numbers you want to change more often at the beginning of the program. We define a new variable, so that we don't have to change the number in the middle of the code, but at the beginning. With a comment, future users will know what exactly this variable does.

In [11]:

```
simulationszeit = 10
```

```
# simulationszeit: Simulation Time - The time that the frog pond is
# simulated in years

froschanzahl = 0
for it in range(simulationszeit):
    froschanzahl = froschanzahl + 3
print(froschanzahl)
```

```
30
```

We can now use this program to simulate very long time ranges:

In [12]:

```
simulationszeit = 1000
# simulationszeit: Simulation Time - The time that the frog pond is
# simulated in years

froschanzahl = 0
for it in range(simulationszeit):
    froschanzahl = froschanzahl + 3
print(froschanzahl)
```

```
3000
```

In this simulation, the final result is less exciting than the temporal development. It would be interesting to output the number of frogs at any time. The easiest way is to move the print command into the For-loop:

In [13]:

```
simulationszeit = 30
# simulationszeit: Simulation Time - The time that the frog pond is
# simulated in years

froschanzahl = 0
for it in range(simulationszeit):
    froschanzahl = froschanzahl + 3
    print(froschanzahl, end=', '), # the , end=' ' in the print command writ
# es the results in one line instead of one below the other
```

```
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 66 69 72 75 78
81 84 87 90
```

This way you can see how the frog population is developing, since we have output each value individually. The disadvantage of this variant is that we only save one number to memory. At the end of the simulation we see many numbers, but only the last one is saved (and therefore available for further work). A nicer solution is to use a list.

Lists in Python

Lists consist of several serial values. To create a list, it is best to start with an empty list and then add new elements again and again. Lists also get a unique name like varibales. Empty lists are created with `name=[]` and new entries are added with the `append` command. For our example:

In [14]:

```
simulationszeit = 30
```

```
# simulationszeit: Simulation Time - The time that the frog pond is
# simulated in years

froschanzahl = 0
froschanzahl_liste = [] #empy list is created
froschanzahl_liste.append(froschanzahl) #first list entry (0 frogs) is
# appended

for it in range(simulationszeit):
    froschanzahl = froschanzahl + 3
    # The current number is appended to the list at the end of each loop
    # pass
    froschanzahl_liste.append(froschanzahl)
print(froschanzahl_liste)
```

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57
, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90]
```

Now you can roughly see what is happening, but you cannot imagine much. A graphic representation would be nice. Luckily this is very easy in Python.

Graphics in Python

The command to create plots is `plot`. This command is not standard in every Python program, but usually has to be imported from a so-called package. Packages are collections of commands.

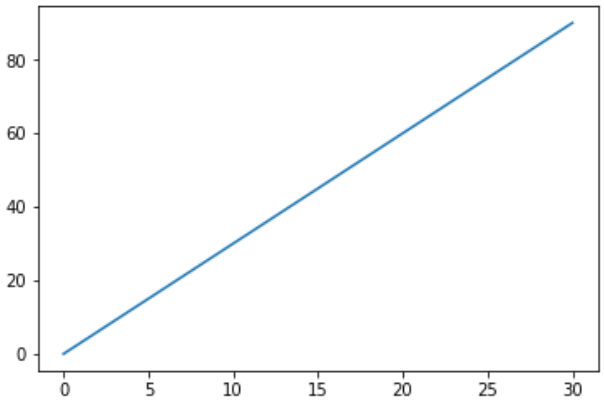
The `plot` command itself works very easy: You simply write the list of numbers you want to display in brackets.

In [2]:

```
import matplotlib.pyplot as plt #we import the package matplotlib.pyplot
# and give it the shortcut plt
# the next line causes the graphics to be displayed directly in the cell
%matplotlib inline

simulationszeit = 30
#simulationszeit: The time that the frog pond is simulated in years

froschanzahl = 0
froschanzahl_liste = [] #empy list is created
froschanzahl_liste.append(froschanzahl) # first list entry (0 frogs) is
# appended

for it in range(simulationszeit):
    froschanzahl = froschanzahl + 3
    # The current number is appended to the list at the end of each loop
    # pass
    froschanzahl_liste.append(froschanzahl)

plt.plot(froschanzahl_liste)
```

Out[2]:

```
[<matplotlib.lines.Line2D at 0x1ef1f2a71d0>]
```



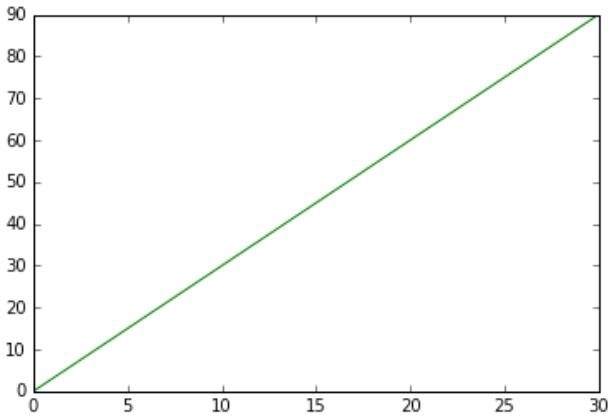
A graphic is much more vivid than a list of numbers. Of course there is still room for improvement. With frogs, for example, it would make sense to display the line in green. Such additional options can easily be added after a comma in the plot command:

In [16]:

```
plt.plot(froschanzahl_liste, color = 'green')
```

Out[16]:

```
[<matplotlib.lines.Line2D at 0xb0f9828>]
```



Note that you have to write the color itself under quotes, otherwise the program would search for a variable called green.

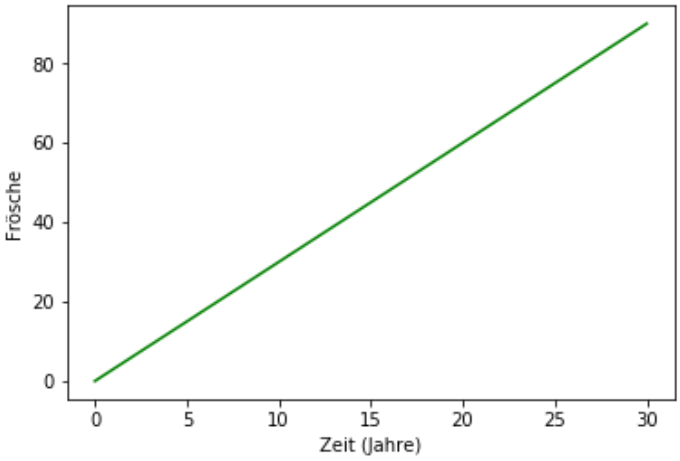
Another possible improvement would be to label the axes:

In [3]:

```
plt.plot(froschanzahl_liste, color = 'green')
plt.xlabel(Time (Years)')
plt.ylabel('Frogs')
```

Out[3]:

```
<matplotlib.text.Text at 0x1ef1f2d64e0>
```



Summary

Variables

Using variables we can assign values to a name. With

```
varname = 42
```

we assign the value 42 to the variable `varname` and overwrite the value currently stored there, if something is already stored there.

With

```
varname = varname + 1
```

we increase the value of `varname` by 1.

The current value of `varname` can be displayed by typing

```
print(varname)
```

For-Loops

With For-loops it is possible to execute the same or similar commands often one after the other. For example, to execute a command 100 times, we use

```
for it in range(100):

    command
```

Note the indentation of the command, which indicates that the command is inside the loop.

Lists

In lists several values can be stored under one name. Empty lists are created with

```
listname = []
```

A new entry (e.g. `newelement`) can be added to the list with

```
listname.append(newelement)
```

Graphics

To use the plot command within a Jupyter notebook, we use the lines

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

```
%matplotlib inline
```

at the beginning of the program. After that we can start with the command

```
plt.plot(listname)
```

graphically display the list with the name `listname`. Additional options like color can be passed after a comma:

```
plt.plot(listname, color='green')
```

Chapter 3 – Population dynamics

In this unit we would like to take a closer look at different possibilities of population development. With the example of the frog pond, we have already learned a very simple form of population development, so-called:

Linear Growth

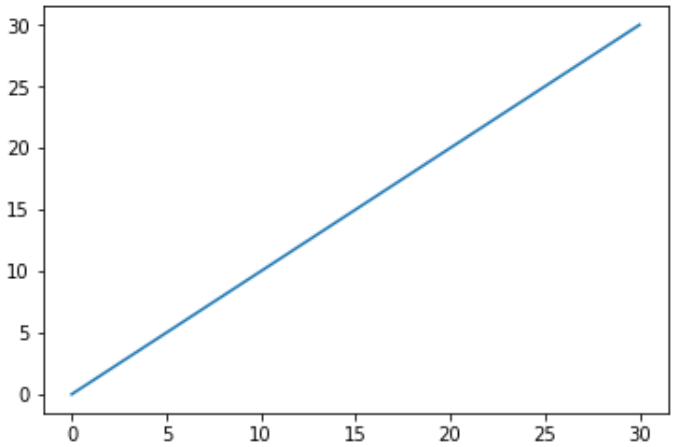
As another example, we will now look at the reproduction of rabbits (= *kaninchen*).

In [2]:

```
import matplotlib.pyplot as plt   # we import the package matplotlib.pyplot
# and give it the shortcut plt
# the next line causes the graphics to be displayed directly in the cell
%matplotlib inline

simulationszeit = 30
#simulationszeit: The time in years that the rabbits are simulated

kaninchenanzahl = 0 # = number of rabbits
kaninchenanzahl_liste = [] # empty list is created
kaninchenanzahl_liste.append(kaninchenanzahl) # first entry (0 rabbits) is
# appended

for it in range(simulationszeit):
    kaninchenanzahl = kaninchenanzahl + 1
    # at the end of each loop pass the current number is appended to the
    # list
    kaninchenanzahl_liste.append(kaninchenanzahl)

plt.plot(kaninchenanzahl_liste)
```

Out[2]:

```
[<matplotlib.lines.Line2D at 0x246c4dacba8>]
```



This form of growth was a good approximation for a frog pond, if we assume that the frogs simply migrate there. However, our rabbit population is increasing by reproduction. That means, the more rabbits there are, the faster new ones are added.

This leads to so-called **exponential growth.**

Exponential growth

We speak of exponential growth when the growth of a quantity is proportional to the quantity itself. Expressed in Python:

```
kaninchen = kaninchen + x * kaninchen
```

where x is a **growth parameter** that we can choose at will. If, for example, we want the number of rabbits to double in each time step, we can choose 1 as growth parameter:

```
kaninchen = kaninchen + 1 * kaninchen
```

which you could also write as `kaninchen = 2 * kaninchen`. If we want each time step to have a growth of 10% we choose x as 0.1:

```
kaninchen = kaninchen + 0.1 * kaninchen
```

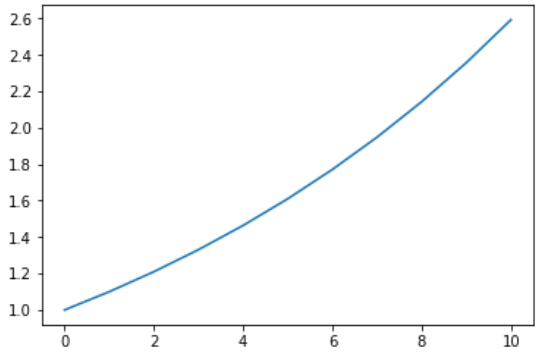
Let's do a simulation:

In [1]:

```
import matplotlib.pyplot as plt
%matplotlib inline

simulationszeit = 10
#simulationszeit: The time in years that the rabbits are simulated

kaninchenanzahl = 1 # = number of rabbits
kaninchenanzahl_liste = [] # empty list is created
kaninchenanzahl_liste.append(kaninchenanzahl) # first entry (1 rabbit) is
appended

for it in range(simulationszeit):
    kaninchenanzahl = kaninchenanzahl + kaninchenanzahl * 0.1 # new number
    # of rabbits is calculated
    # at the end of each loop pass the current number is appended to the
    # list
    kaninchenanzahl_liste.append(kaninchenanzahl)

plt.plot(kaninchenanzahl_liste)
```

Out[1]:

```
[<matplotlib.lines.Line2D at 0x23810428a58>]
```



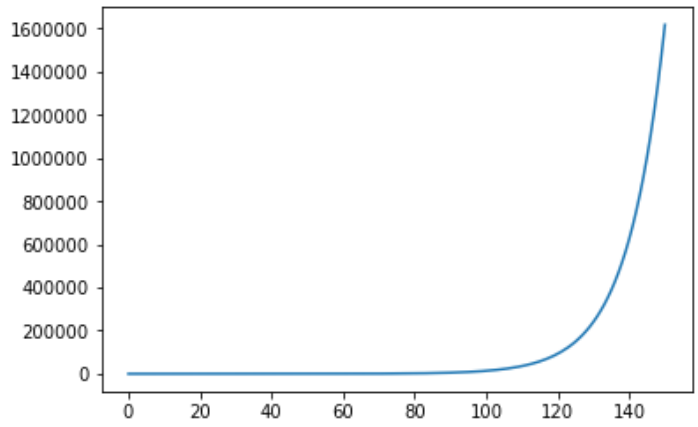
Exponential growth is a pretty good approximation to the real behaviour of a population. However, a problem arises when we consider very long periods of time. Let us set the simulation time to 150 and see what happens:

In [4]:

```
import matplotlib.pyplot as plt
%matplotlib inline

simulationszeit = 150
# simulationszeit: The time in years that the rabbits are simulated

kaninchenanzahl = 1 # = number of rabbits
kaninchenanzahl_liste = [] # empty list is created
kaninchenanzahl_liste.append(kaninchenanzahl) # first entry (1 rabbit) is
# appended

for it in range(simulationszeit):
    kaninchenanzahl = kaninchenanzahl + kaninchenanzahl * 0.1 # new number
    # of rabbits is calculated
    # at the end of each loop pass the current number is appended to the
    # list
    kaninchenanzahl_liste.append(kaninchenanzahl)

plt.plot(kaninchenanzahl_liste)
```

Out[4]:

```
[<matplotlib.lines.Line2D at 0x23810648c50>]
```



The growth becomes faster and faster and the rabbit population explodes. This is unrealistic behavior, because every ecosystem has a certain **capacity limit**, i.e. a maximum number of individuals that can live in the system (due to food or shelter). We should build this into our program as well. Let us assume that there can never be more than 1000 rabbits in our system. If this number is exceeded, no new rabbits should be added. To build this into the program we need a new structure, the so called **if query**.

If query

`If queries` are "if-then-questions", with which we can attach commands to conditions. They always have the following structure:

```
if CONDITION:

    COMMAND
```

`COMMAND` means any instruction to Python, for example changing a variable, a print command or something similar. An `If query` can also contain several commands.

A `COMMAND` can be anything that can be either `true` or `false`. Often an equation is used to define a condition. For example, you check if it is `true` that a variable `x` is equal to a variable `y`. Even more often inequalities are used. So, you check if it is `true` that a variable `x` is larger (or smaller) than a variable `y`.

In the following example we will look at an `If query` in a For loop. The For-loop increases the number of rabbits continuously by 1, the `If query` executes a print command, but only if the number of rabbits is greater than 100.

In [5]:

```
kaninchen = 0 # Variable definition, i.e. assigning the value 0 to the
# variable rabbits


for it in range(105):
    kaninchen = kaninchen + 1
    if kaninchen == 100:  # Comparing the value of the variable rabbits
    # with the value 100
        print("There are now EXACTLY 100 rabbits!")
```

```
There are now EXACTLY 100 rabbits!
```

This example demonstrates the important difference between the single = sign and the double = sign again: The single = is used to assign a value to a variable (`kaninchen`). The double = character is used for comparison.

It is also possible to put several commands under one `If query`. Similar to the For-loop, indentation indicates whether a command is part of the `If query` or not:

In [6]:

```
kaninchen = 95
for it in range(10):
    kaninchen = kaninchen + 1
    if kaninchen > 100:
        print("There are more than 100 rabbits now!")
        print("The exact number of rabbits is ")
        print(kaninchen)
```

```
There are more than 100 rabbits now!
The exact number of rabbits is
101
There are more than 100 rabbits now!
The exact number of rabbits is
102
There are more than 100 rabbits now!
The exact number of rabbits is
103
There are more than 100 rabbits now!
The exact number of rabbits is
104
There are more than 100 rabbits now!
The exact number of rabbits is
105
```

If we don't indent the last two commands, they are no longer part of the `If-Query` and will be executed in any case, no matter how many rabbits there are:

In [8]:

```
kaninchen = 95
for it in range(10):
    kaninchen = kaninchen + 1
    if kaninchen > 100:
        print("There are more than 100 rabbits now!")
    print("The exact number of rabbits is ")
    print(kaninchen)
```

```
The exact number of rabbits is
96
The exact number of rabbits is
97
The exact number of rabbits is
98
The exact number of rabbits is
99
The exact number of rabbits is
100
There are more than 100 rabbits now!
The exact number of rabbits is
101
There are more than 100 rabbits now!
The exact number of rabbits is
102
There are more than 100 rabbits now!
The exact number of rabbits is
103
There are more than 100 rabbits now!
The exact number of rabbits is
104
There are more than 100 rabbits now!
```

```
The exact number of rabbits is
105
```

If we move the two commands to the outside again, they will also be outside the For-loop and will therefore only be executed once, after the For-loop is complete:

In [10]:

```
kaninchen = 95
for it in range(10):
    kaninchen = kaninchen + 1
    if kaninchen > 100:
        print("There are more than 100 rabbits now!")
    print("The exact number of rabbits is ")
print(kaninchen)
```

```
There are more than 100 rabbits now!
There are more than 100 rabbits now!
There are more than 100 rabbits now!
There are more than 100 rabbits now!
There are more than 100 rabbits now!
The exact number of rabbits is
105
```

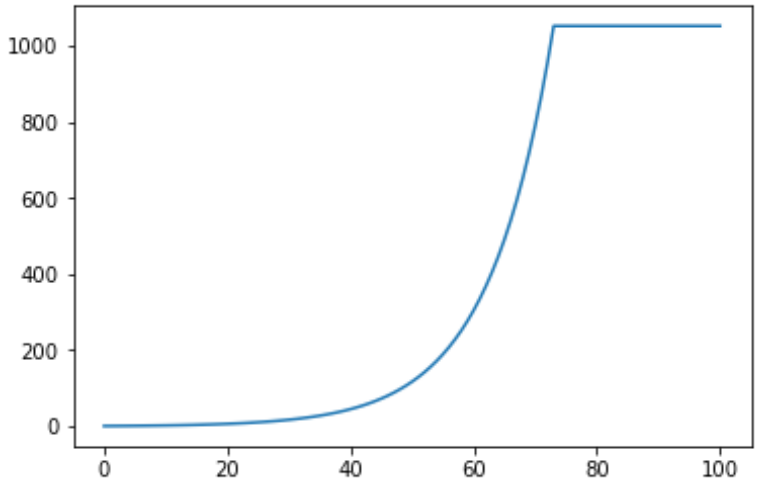
Now that we understand if queries we can implement the limit of 1000 individuals in our program: Only if there are less than 1000 rabbits, the population will grow:

In [15]:

```
import matplotlib.pyplot as plt
%matplotlib inline

simulationszeit = 100
#simulationszeit: The time in years that the rabbits are simulated

kaninchenanzahl = 1 # = number of rabbits
kaninchenanzahl_liste = [] # empty list is created
kaninchenanzahl_liste.append(kaninchenanzahl) # first entry (1 rabbit) is
# appended

for it in range(simulationszeit):
    kaninchenanzahl = kaninchenanzahl + kaninchenanzahl * 0.1 # new number
    # of rabbits is calculated
    # at the end of each loop pass the current number is appended to the
    # list
    kaninchenanzahl_liste.append(kaninchenanzahl)

plt.plot(kaninchenanzahl_liste)
```

Out[15]:

```
[<matplotlib.lines.Line2D at 0x2381099eb00>]
```



Besides exponential growth there are other methods to describe the growth of a population. One can also start from the idea that the current population depends on the population of the last time step on the one hand, but also on the population of the second last time step on the other hand.

This leads to the so-called

Fibonacci sequence

In this model the number of rabbits in the current year is calculated from the number of rabbits from the previous year plus the number of rabbits from the year before last. To program something like this, we first have to learn more about lists in Python:

Lists and loops for advanced users

So far we know how to create empty lists and add elements to lists. But of course you can also query specific list entries. For example, if you want to know the seventh element of the list `kaninchenanzahl_liste` , you can retrieve it with `kaninchenanzahl_liste[6]`. Why not 7? In programming languages it is common to store the first element of a list always with the number 0, in this case as `kaninchenanzahl_liste[0]`. Although this is confusing at first, it is a quite reasonable convention for later applications.

In [5]:

```
kaninchenanzahl_liste[0]
```

Out[5]:

```
0
```

We have already learned about `for loops`, but we have so far ignored one important property. The number *it*, which counts how many times a command has been executed, can be used inside a command:

In [3]:

```
for it in range(5):
    print(it)
```

```
0
1
2
3
4
```

If we now combine these two skills, we have found a way to access previous list entries within a loop. This is exactly what we need for our Fibonacci sequence.

Another important property of for loops is that *it* does not necessarily have to start at 0. For example, if we want to simulate only from year 2 on, we can let the loop start at year two:

In [4]:

```
for it in range(2,5):
    print(it)
```

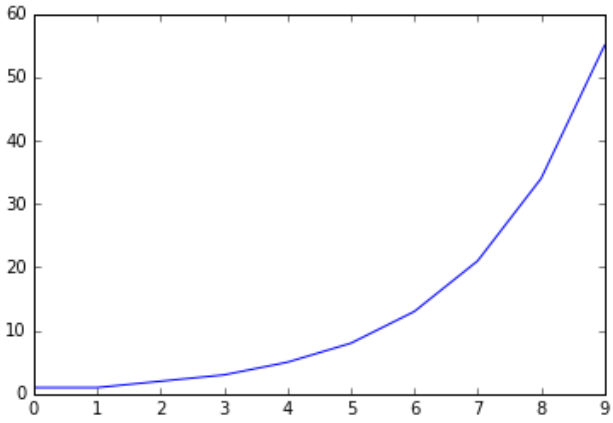
```
2
3
4
```

In [5]:

```
import matplotlib.pyplot as plt
%matplotlib inline

simulationszeit = 10
#simulationszeit: The time in years that the rabbits are simulated

kaninchenanzahl = 1 # = number of rabbits
kaninchenanzahl_liste = [] # empty list is created
kaninchenanzahl_liste.append(kaninchenanzahl) # first entry (1 rabbit) is
# appended

kaninchenanzahl_liste.append(kaninchenanzahl)#second entry (1 rabbit) is
# appended
#we need two rabbits so they can reproduce

for it in range(2, simulationszeit):
    letztesjahr = kaninchenanzahl_liste[it - 1] # number of rabbits last
    # year (= letztesjahr)
    vorletztesjahr = kaninchenanzahl_liste[it - 2] # number of rabbits of
    # the year before last year (= vorletztesjahr)
    kaninchenanzahl = letztesjahr + vorletztesjahr # calculation of the
    # current number of rabbits
    # at the end of each loop pass, the current number is appended to the
    # list
    kaninchenanzahl_liste.append(kaninchenanzahl)

plt.plot(kaninchenanzahl_liste)
```

Out[5]:

```
[<matplotlib.lines.Line2D at 0xb05ff28>]
```



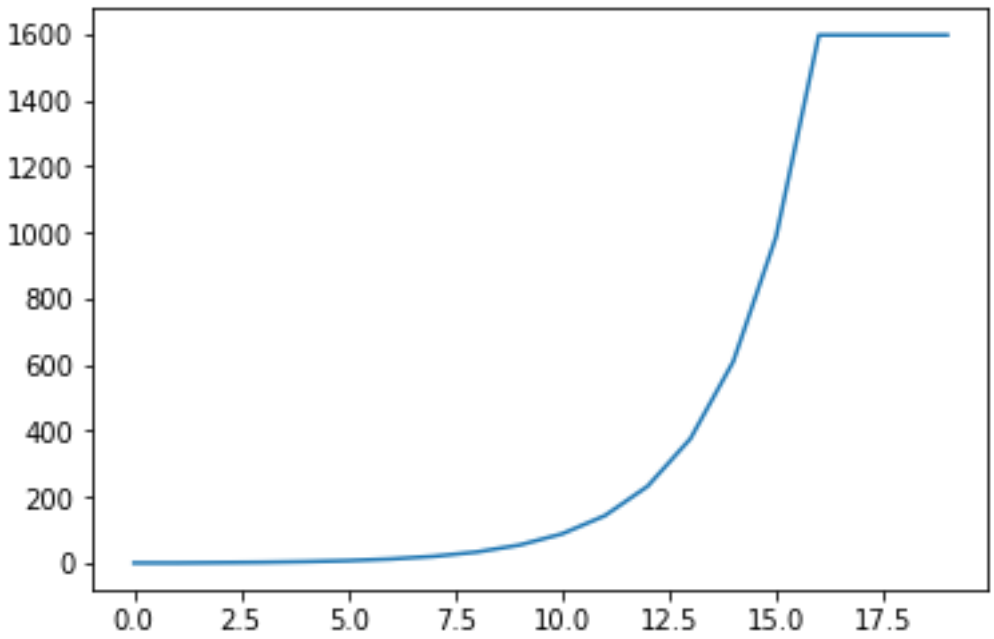
Of course, we can also build in a limitation of growth through an If-query, just as with exponential growth:

In [2]:

```
import matplotlib.pyplot as plt
%matplotlib inline

simulationszeit = 20
#simulationszeit: The time in years that the rabbits are simulated

kaninchenanzahl = 1 # = number of rabbits
kaninchenanzahl_liste = [] # empty list is created
kaninchenanzahl_liste.append(kaninchenanzahl) # first entry (1 rabbit) is
# appended

kaninchenanzahl_liste.append(kaninchenanzahl)#second entry (1 rabbit) is
# appended
#we need two rabbits so they can reproduce

for it in range(2, simulationszeit):
    if kaninchenanzahl < 1000: #ONLY if there are less than 1000 rabbits
        letztesjahr = kaninchenanzahl_liste[it - 1] # number of rabbits
        # last year (= letztesjahr) is calculated
        vorletztesjahr = kaninchenanzahl_liste[it - 2] # number of rabbits
        # the year before last year (= vorletztesjahr) is calculated
        kaninchenanzahl = letztesjahr + vorletztesjahr # calculation of the
        # current number of rabbits
        # at the end of each loop pass, the current number is appended to
        # the list
    kaninchenanzahl_liste.append(kaninchenanzahl)

plt.plot(kaninchenanzahl_liste)
```

Out[2]:

```
[<matplotlib.lines.Line2D at 0x2ab3a336828>]
```



But what if we now want to zoom into the interesting part of the growth (for example the kink). To do this, it will be necessary to select not only individual elements of the list, but entire pieces. This is how it works:
But we can also select several elements at once. To do this, we write in square brackets the first element we want to have, then a colon, and then the first element we don't want to have anymore. The element with the index 10 and the element with the index 11 we get with

In [6]:

```
print(kaninchenanzahl_liste[14:19])
```

```
[610, 987, 1597, 1597, 1597]
```

If no number is written before the colon, the element starts with index zero. If no number is written after the colon, everything is selected up to the last element.

In [10]:

```
print(kaninchenanzahl_liste[:3])
print(kaninchenanzahl_liste[17:])
```

```
[1, 1, 2]
[1597, 1597, 1597]
```

Python also accepts negative numbers as index. The index -1 denotes the last element of a list, the index -2 the second last and so on. This is especially useful if you want to select e.g. the last 5 entries of a list:

In [11]:

```
print(kaninchenanzahl_liste[-5:])
```

```
[987, 1597, 1597, 1597, 1597]
```

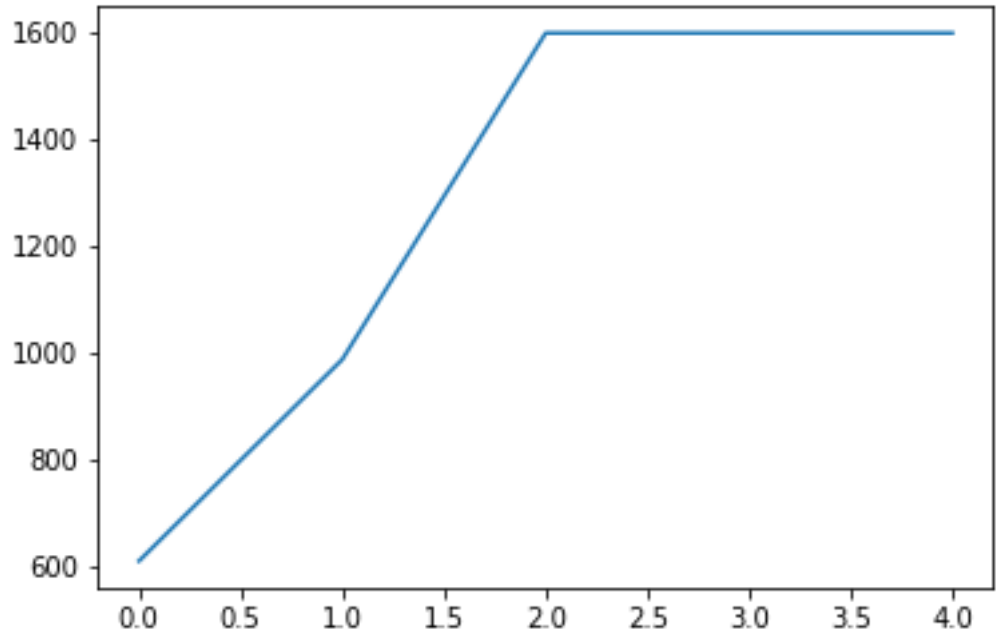
Of course, selecting list items does not only work in combination with the print command, but also with all other commands, for example plot. So we can now zoom in on the relevant part of our graph:

In [12]:

```
# lists are plotted
plt.plot(kaninchenanzahl_liste[14:19])
```

Out[12]:

```
[<matplotlib.lines.Line2D at 0x2ab3a44e2b0>]
```



Summary

Linear growth

With linear growth, a constant number is added to a population in each time step:

```
rabbits = rabbits + x
```

where x indicates how many individuals are added, i.e. how fast the population grows.

Exponential growth

With exponential growth, no constant value is added to the population, but a value that depends directly on the current population. The exact dependence of growth and current population is given by the so-called **growth parameter**:

```
rabbits = rabbits + rabbits * growthparameter
```

If-Queries

`If Queries` can be used to make statements and commands execute only under certain conditions. They have the structure

```
    if CONDITION:

        COMMAND
```

The conditions are mostly inequalities (`rabbits >= 100`) or equations (`rabbits== 100`), which can be either **true** or **false**. It is important to use the double =-sign for a comparison.

For loops for advanced users

The run variable (`it` in the example) can be used inside the loop. So the loop returns

```
for it in range(3):

    print(it)
```

the output

```
0

1

2
```

Lists for advanced users

To use or output the individual elements of a list, use square brackets:

```
listname[2]
```

returns for example the third(!) element. Attention: The first element in the list always has index 0, the second element has index 1, and so on.

Selecting parts of a list

If we want to select several elements of a list, we can use

```
listname[a:b]
```

where `a` is the first element we want to have and `b` is the first element that should no longer be part of the selection. If we leave a empty, the element with index 0 is selected, if we leave b empty, the selection is continued until the last element. The index -1 always means the last element of a list, -2 the second last and so on.

Chapter 4 - Analyzing pollutant logs

In this chapter we will further develop our knowledge of For-loops and If-queries to evaluate the pollution statistics of a company. The pollutant values are stored in a list. In a first step we shall simply issue a warning if the value (=Werte) exceeds the limit of 100 units. An If-query is sufficient for this.

Since we want to have a program that works for lists of any length, we cannot write fixed 10 at the length of the For-loop, but let us calculate the length of the list with `len`.

In [12]:

```
werte = [89,96,125,88,110,112,99,84,50,130]

for it in range(len(werte)): # Loop has the length of the list
    if werte[it] > 100:
        print("Too high pollution levels during the day",it,"!")
```

```
Too high pollutant levels on day 2 !
Too high pollutant levels on day 4 !
Too high pollutant levels on day 5 !
Too high pollutant levels on day 9 !
```

But if our program is only supposed to give an alarm when the value is exceeded on two days in a row, we need another if query. In addition, we have to exclude day 0 because there is no day -1.

In [13]:

```
werte = [89,96,125,88,110,112,99,84,50,130]

for it in range(1,len(werte)): # Loop has the length of the list
    if werte[it] > 100:
        if werte[it-1] > 100:
            print("Too high pollution levels during the day",it,"!")
```

```
Too high pollutant levels on day 5 !
```

You can imagine that such a structure becomes confusing relatively quickly, especially if you have many conditions. Therefore, you can write such if-queries also shortened: You can connect the single conditions with a logical **and** and a logical **or**:

In [14]:

```
werte = [89,96,125,88,110,112,99,84,50,130]

for it in range(1,len(werte)): # Loop has the length of the list
    if werte[it] > 100 and werte[it-1] > 100:
        print("Too high pollution levels during the day",it,"!")
```

```
Too high pollutant levels on day 5 !
```

This allows you to create more complicated links, which can be connected using round brackets. So we could for example require that an additional warning is written if the value is greater than 120. But if the value of the day before is less than 50, there is no alarm. That would look like this:

In [15]:

```
werte = [89,96,125,88,110,112,99,84,50,130]

for it in range(1,len(werte)): # Loop has the length of the list
```

```
    if (werte[it] > 100 and werte[it-1] > 100) or (werte[it] > 120 and werte[it-1] > 50):
        print("Too high pollution levels during the day",it,"!")
```

```
Too high pollutant levels on day 2 !
Too high pollutant levels on day 5 !
```

Next, we want to graphically display the days that have triggered an alarm. For this purpose, we create a list in which we write the number 1 each time the alarm was triggered and 0 if it was not triggered. The first part is simple, we can write a 1 in the list each time directly after the print command. But how do we identify the days without alarm? We could build a similar if-query again and just "flip it over". But is it enough to simply flip the greater and lesser characters? What if we hit exactly the limit? And how does it work with the **or**? In any case, such a solution would be very complicated and error-prone. That's why there is a better solution: Every if-query can also be equipped with an **else,** a command to be done if the if-query is not fulfilled. From the indentation this **else** is on the same level as the **if** to which it belongs:

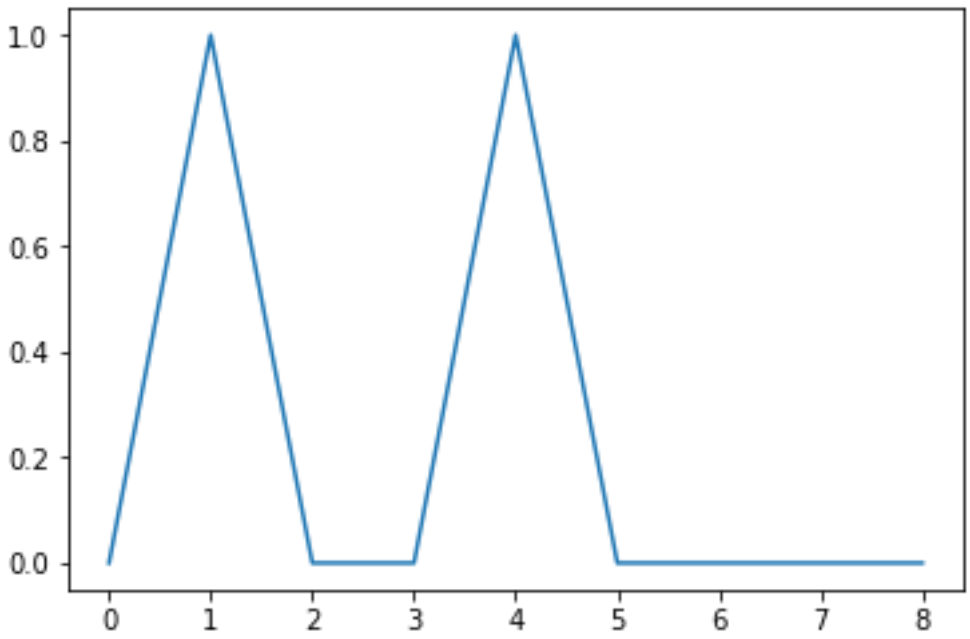
In [16]:

```
import matplotlib.pyplot as plt
%matplotlib inline

werte = [89,96,125,88,110,112,99,84,50,130]
protokoll=[]

for it in range(1,len(werte)): # Loop has the length of the list
    if (werte[it] > 100 and werte[it-1] > 100) or (werte[it] > 120 and werte[it-1] > 50):
        print("Too high pollution levels during the day",it,"!")
        protokoll.append(1)
    else:
        protokoll.append(0)

plt.plot(protokoll)
```

```
Too high pollutant levels on day 2 !
Too high pollutant levels on day 5 !
```

Out[16]:

```
[<matplotlib.lines.Line2D at 0x1835005f2e8>]
```



However, we have assumed a perfect data situation. Real data is rarely complete. It can be assumed that on some days no measurement was made. Such days would then simply appear as 0 in our list, and not only would they not trigger an alarm themselves, but they would also have effects on the day after:

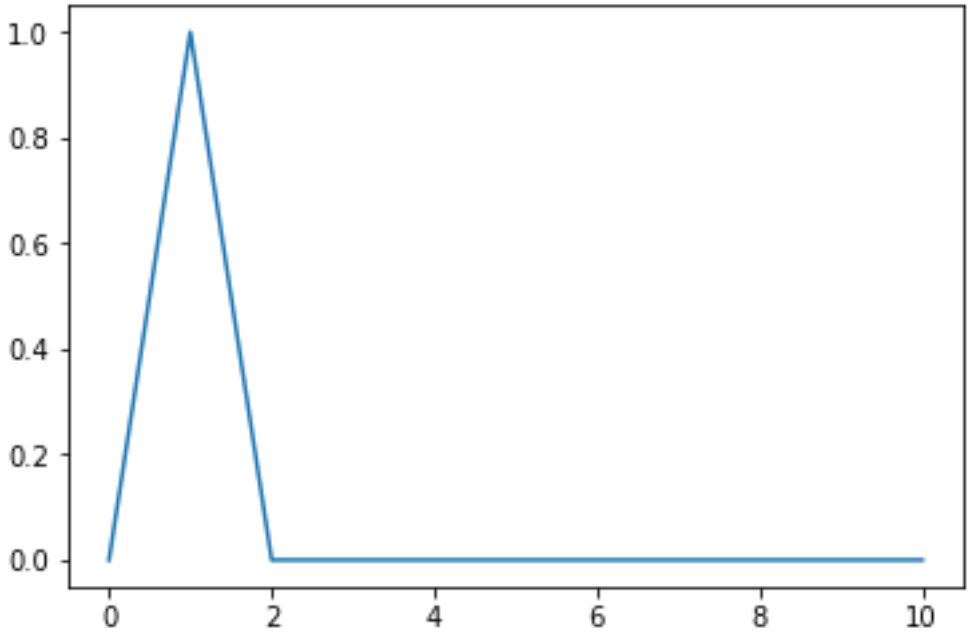
In [33]:

```
import matplotlib.pyplot as plt
%matplotlib inline

werte = [89,96,125,88,110,0,112,99,84,50,0,130]
protokoll=[]

for it in range(1,len(werte)): # Loop has the length of the list
    if (werte[it] > 100 and werte[it-1] > 100) or (werte[it] > 120 and werte[it-1] > 50):
        print("Too high pollution levels during the day" ,it, "!")
        protokoll.append(1)
    else:
        protokoll.append(0)

plt.plot(protokoll)
```

```
Too high pollutant levels on day 2 !
```

Out[33]:

```
[<matplotlib.lines.Line2D at 0x2ab3a52cac8>]
```



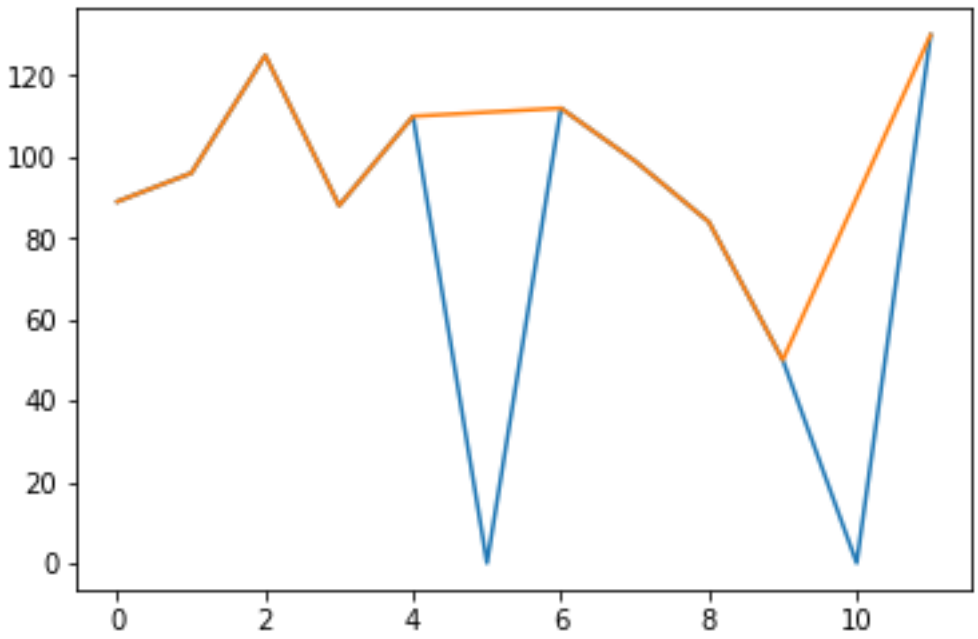
What could we do in such a case? One possibility is to interpolate, i.e. to try to estimate the missing data from the existing data. The easiest way would be to assume that the missing pollutant values are always exactly the average (=`mittelwert`) of the value from the day before and the value from the day after. Let's clean up our list this way and plot the resulting lists:

In [37]:

```
import matplotlib.pyplot as plt
%matplotlib inline

werte = [89,96,125,88,110,0,112,99,84,50,0,130]
werte_neu=[]

for it in range(len(werte)):
    if werte[it] == 0:
        mittelwert = (werte[it-1] + werte[it+1]) / 2
        werte_neu.append(mittelwert)
    else:
        werte_neu.append(werte[it])

plt.plot(werte)
plt.plot(werte_neu)
```

Out[37]:

```
[<matplotlib.lines.Line2D at 0x2ab3a571128>]
```



There are of course further possibilities to interpolate. For example you could consider derivations or include more values. But for the time being we stick to this method and use it for our evaluation:

In [17]:

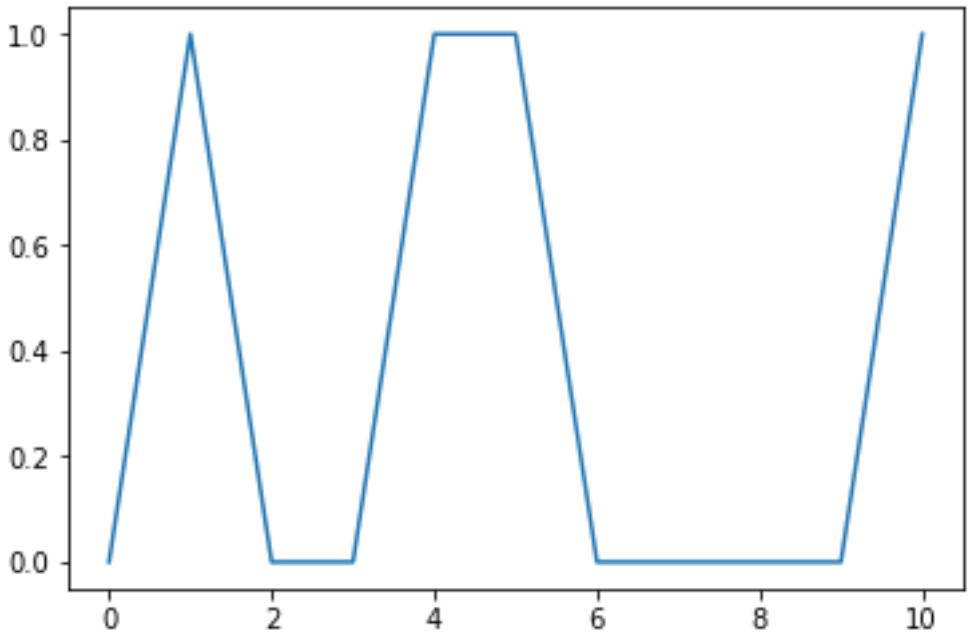
```
import matplotlib.pyplot as plt
%matplotlib inline

werte = [89,96,125,88,110,0,112,99,84,50,0,130]
werte_neu=[]

for it in range(len(werte)):
    if werte[it] == 0:
        mittelwert = (werte[it-1] + werte[it+1]) / 2
        werte_neu.append(mittelwert)
    else:
        werte_neu.append(werte[it])

werte = werte_neu
protokoll=[]

for it in range(1,len(werte)): # Loop has the length of the list
    if (werte[it] > 100 and werte[it-1] > 100) or (werte[it] > 120 and werte[it-1] > 50):
        print("Too high pollution levels during the day",it,"!")
        protokoll.append(1)
    else:
        protokoll.append(0)

plt.plot(protokoll)
```

```
Too high pollutant levels on day 2 !
Too high pollutant levels on day 5 !
Too high pollutant levels on day 6 !
```

```
Too high pollutant levels on day 11 !
```

Out[17]:

```
[<matplotlib.lines.Line2D at 0x18350082a20>]
```



With the help of this interpolation we can now also work with data sets, which were filled with 0s. But real data sets are usually even more difficult to handle. For example, we may come across data that is not even a number, but has a completely different data type, for example character strings. This would be even worse for our program, because it does not lead to wrong results, but to an error message and thus a crash, as we can see here:

In [18]:

```
import matplotlib.pyplot as plt
%matplotlib inline

werte = [89,96," no measurement",125,88,110,0,112,"44",99,84,50,0,130,[99,88],125,"🏠",120]
werte_neu=[]

for it in range(len(werte)):
    if werte[it] == 0:
        mittelwert = (werte[it-1] + werte[it+1]) / 2
        werte_neu.append(mittelwert)
    else:
        werte_neu.append(werte[it])

werte = werte_neu
protokoll=[]

for it in range(1,len(werte)): # Loop has the length of the list
    if (werte[it] > 100 and werte[it-1] > 100) or (werte[it] > 120 and werte[it-1] > 50):
        print("Too high pollution levels during the day",it, "!")
        protokoll.append(1)
    else:
        protokoll.append(0)
```

```
plt.plot(protokoll)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-55dec0f13f96> in <module>()
     16
     17 for it in range(1,len(werte)): # Loop has the length of the list
---> 18     if (werte[it] > 100 and werte[it-1] > 100) or (werte[it] > 120 and werte[it-1] > 5
0):
     19         print("Too high pollutant levels during the day",it, "!")
     20         protokoll.append(1)

TypeError: '>' not supported between instances of 'str' and 'int'
```

The problem here is that you can't compare a string with a number and an error message is displayed. So we would have to "pre-clean up" the data again before cleaning up. Unfortunately, we can't exclude the possibility of further error messages. Nevertheless, we would like to have a program that does not crash, even if an error occurs. Here is a good way out: We try to perform our arithmetic operation, but if an error occurs somewhere, we have to replace the value from the database with the interpolated value.

For this there are **try** and **except** in Python. First the part of the code under **try** is tried to be executed. But if this leads to an error message, the code under **except** is executed. In our case it would look like this:

In [19]:

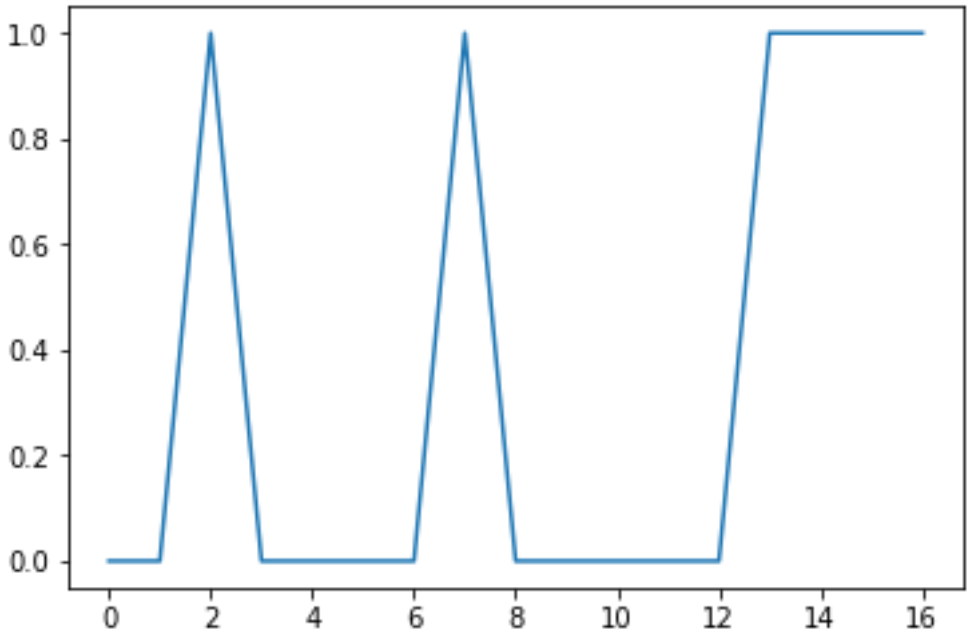
```
import matplotlib.pyplot as plt
%matplotlib inline

werte = [89,96,"no measurement",125,88,110,0,112,"44",99,84,50,0,130,[99,88],125,"🏠",120]
protokoll=[]

for it in range(1,len(werte)): # Loop has the length of the list
    try: # This code block is tried
        if (werte[it] > 100 and werte[it-1] > 100) or (werte[it] > 120 and werte[it-1] > 50):
            print("Too high pollutant levels during the day" ,it, "!")
            protokoll.append(1)
        else:
            protokoll.append(0)
    except: # If the upper block causes an error, this block is executed        werte[it] = (we
rte[it-1] + werte[it + 1]) / 2
        if (werte[it] > 100 and werte[it-1] > 100) or (werte[it] > 120 and werte[it-1] > 50):
            print("Too high pollutant levels during the day" ,it, "!")
            protokoll.append(1)
        else:
            protokoll.append(0)

plt.plot(protokoll)
```

```
Too high pollutant levels on day 3 !
Too high pollutant levels on day 8 !
Too high pollutant levels on day 14 !
```

```
Too high pollutant levels on day 15 !
Too high pollutant levels on day 16 !
Too high pollutant levels on day 17 !
```

Out[19]:

```
[<matplotlib.lines.Line2D at 0x183500fe8d0>]
```



This way we save ourselves the trouble of pre-cleaning the list. But be careful, the 0-values do not cause an error here and thus are not interpolated. We could filter out the 0-values separately before, but we can also use the same **try-except** structure we already have. But for this it is necessary to define an own error message. This works in Python with **raise Exception()**. This command terminates the program and returns an error message. But inside our **try-except** it only causes us to jump into the except block.

So let's build in our own "This number must not be 0" error:

In [20]:

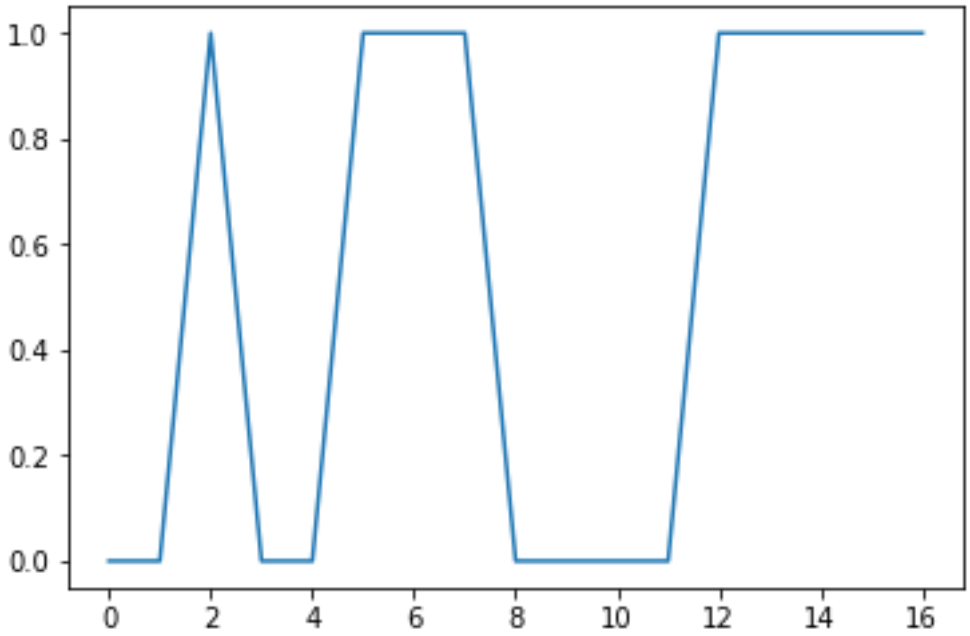
```
import matplotlib.pyplot as plt
%matplotlib inline

werte = [89,96," no measurement ",125,88,110,0,112,"44",99,84,50,0,130,[99,88],125,"🐔",120]
protokoll=[]

for it in range(1,len(werte)): # Loop has the length of the list
    try: # This code block is tried
        if werte[it] == 0:
            raise Exception("This number must not be 0!")
        if (werte[it] > 100 and werte[it-1] > 100) or (werte[it] > 120 and werte[it-1] > 50):
            print("Too high pollutant levels during the day" ,it, "!")
            protokoll.append(1)
        else:
            protokoll.append(0)
    except: # If the upper block causes an error, this block is executed        werte[it] = (w
erte[it-1] + werte[it + 1]) / 2
        if (werte[it] > 100 and werte[it-1] > 100) or (werte[it] > 120 and werte[it-1] > 50):
```

```
            print("Too high pollutant levels during the day" ,it, "!")
            protokoll.append(1)
        else:
            protokoll.append(0)

plt.plot(protokoll)
```

```
Too high pollutant levels on day 3 !
Too high pollutant levels on day 6 !
Too high pollutant levels on day 7 !
Too high pollutant levels on day 8 !
Too high pollutant levels on day 13 !
Too high pollutant levels on day 14 !
Too high pollutant levels on day 15 !
Too high pollutant levels on day 16 !
Too high pollutant levels on day 17 !
```

Out[20]:

```
[<matplotlib.lines.Line2D at 0x1835015cdd8>]
```



Attention: Working with **try** and **except** leads to simple and elegant solutions, but it is relatively dangerous: Even if you make programming errors, they are simply caught by except and you don't see well what happened. Therefore, you should pay special attention when working with try and except. It is important that we take a closer look at how errors in Python actually work:

In [56]:

```
testwerte = [99, 120, 0, 101, "n.a.", 140]

for it in range(len(testwerte)):
    if testwerte[it] > 100:
        print("alert!")
```

```
Alert!
Alert!
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-56-09c2b072c013> in <module>()
      2
      3 for it in range(len(testwerte)):
----> 4     if testwerte[it] > 100:
      5         print("Alert!")
      6


TypeError: '>' not supported between instances of 'str' and 'int'
```

Here an error is triggered when we get to the entry "n.a.". This is a string, so we cannot compare it with the number 100. It is the wrong data type, therefore the error is called "TypeError". Here we can also add our own error type:

In [60]:

```
testwerte = [99, 120, 0, 101, "n.a.", 140]

for it in range(len(testwerte)):
    if testwerte[it] == 0:
        raise Exception("This value must not be 0!")
    if testwerte[it] > 100:
        print("Alert!")
```

```
Alarm!
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-60-cd0f50d30d93> in <module>()
      3 for it in range(len(testwerte)):
      4     if testwerte[it] == 0:
----> 5         raise Exception("This value must not be 0!")
      6     if testwerte[it] > 100:
      7         print("Alert!")


Exception: This value must not be 0!
```

Now we get our individualized error message. Let's now implement **try-except**. For the sake of simplicity we will alternatively just print text that says that an entry was skipped.

In [62]:

```
testwerte = [99, 120, 0, 101, "k.A.", 140]

for it in range(len(testwerte)):
    try:
        if testwerte[it] == 0:
            raise Exception("This value must not be 0!")
        if testwerte[it] > 100:
            print("Alert!")
    except:
        print("An entry was skipped!")
```

```
Alert!
An entry was skipped!
Alert!
```

```
An entry was skipped!
Alert!
```

But this program is quite dangerous now, because we don't see the error messages anymore. So if we make an error while programming, it will be filtered out the same way:

In [66]:

```
testwerte = [99, 120, 0, 101, "n.a.", 140]


for it in range(len(testwerte)):
    try:
        if testwerte[it] == 0:
            raise Exception("This value must not be 0!")
        if testwerte > 100: # ATTENTION: There is now a built-in error:We have removed the [it]
            print("Alert!")
    except:
        print("An entry was skipped!")
```

```
An entry was skipped!
An entry was skipped!
An entry was skipped!
An entry was skipped!
An entry was skipped!
An entry was skipped!
```

To prevent such a thing from happening, or at least reduce the chance of it happening unnoticed, it is recommended to at least output the error messages that occur. The program is not interrupted, but you can still see what happened. So, although we skip the error message, we have some of the advantages of an error message.

For this we have to extend our **try-except** structure a little bit. We want to output the exact error message. To do this, use `except Exception as errormsg` and then use the exact error text stored in the variable `errormsg`.

In [72]:

```
testwerte = [99, 120, 0, 101, "n.a.", 140]

for it in range(len(testwerte)):
    try:
        if testwerte[it] == 0:
            raise Exception("This value must not be 0!")
        if testwerte[it] > 100:
            print("Alert!")
    except Exception as errormsg:
        print("An entry was skipped! The error was:")
        print(errormsg)
```

```
Alert!
An entry was skipped! The error was:
This value must not be 0!
Alert!
An entry was skipped! The error was:
'>' not supported between instances of 'str' and 'int'
```

```
Alert!
```

To separate the actual log from the error log, we can start a separate list for this, which we can only view when needed:

In [25]:

```
testwerte = [99, 120, 0, 101, "n.a.", 140]
fehlerlog=[]
for it in range(len(testwerte)):
    print("day",it)

    try:
        if testwerte[it] == 0:
            raise Exception("This value must not be 0!")
        if testwerte[it] > 100:
            print("alert!")
        else:
            print("no alert!")
    except Exception as errormsg:
        print("error!")
        fehlerlog.append(it)
        fehlerlog.append("An entry was skipped! The error was:")
        fehlerlog.append(errormsg)
    else: # also the except can have an else, which is executed,if the except was not triggered
        fehlerlog.append(it)
        fehlerlog.append("free of errors!")


print("Error log:")
print(fehlerlog)
```

```
day 0
no alert!
day 1
alert!
day 2
error!
day 3
alert!
day 4
error!
day 5
alert!
Error log:
[0, 'free of errors!', 1, 'free of errors!', 2, 'An entry was skipped! The error was:', Except
ion('This value must not be 0!'), 3, 'free of errors!', 4, 'An entry was skipped! The error wa
s:', TypeError("'>' not supported between instances of 'str' and 'int'"), 5, 'free of errors!'
]
```

In this way we can deal with errors relatively well. But Try-except offers much more possibilities: We could distinguish the type of errors and find different solutions depending on the type of error.

In general, you have to be careful when using **try** and **except**: In most cases, an error message is a problem that should be solved by changing the program until no more error messages appear. Nevertheless, there are situations where (potential) error messages are unavoidable, for example when data is read in from outside and you cannot yet know what kind of data it will be. In this case, it is useful to use **try** and **except** to prevent this.

Summary

If-Else queries

The **If-Else query** is an extension of the **If-query**. It extends this structure with additional commands that are to be executed if the condition is **not true**. It has the structure:

```
if CONDITION:

      COMMAND IF TRUE

else:

      COMMAND IF WRONG
```

and / or

With **and** and **or** you can connect several conditions to each other, either in a way, that all conditions must be fulfilled, or in a way, that only one must be fulfilled.

- true **and** true = true
- true **and** wrong = wrong
- true **or** true = true
- true **or** wrong = true
- wrong **or** wrong = wrong

Try and Except

With `try` and `except` it is possible to let a program continue to run despite an error message. If an error occurs in the try-block, the program jumps to the except-block. But to make sure that we notice that an error occurs, it is useful to at least log the error message. To do so, `except Exception as errormsg:` and then you have access to the error message under the variable name `errormsg`.

Chapter 5 - Coupled differential equations: Predator-Prey-Systems

In a previous chapter of this script we have already looked at the development of an animal population. But this was still very simplified, because in most natural cases the growth of an animal population has a dependence on other sizes, for example the population of another animal species. Such dependencies and influences of different dynamics are to a certain extent the normal case in the systems that are interesting for systems science. The concept of systems is based on the assumption that *interacting*, i.e. mutually influencing, dynamics generate something in their interaction that cannot be observed without this interaction, or at least not in this way.

With respect to these interdependent dynamics, one speaks of *coupled* dynamics. And mathematics knows the method of **coupled differential equations** for their analysis. The standard example of such coupled differential equation systems in systems science are **predator-prey systems**, for which there is usually no analytical (purely mathematical) solution. Therefore, in the following we want to simulate a historical example of such a predator-prey system using Python.
Let's start similarly as we have simulated rabbits (=Hasen) in an earlier chapter. Within a For-loop, the population of rabbits grew exponentially. In addition, we also include a second species, the lynx (=Luchs).

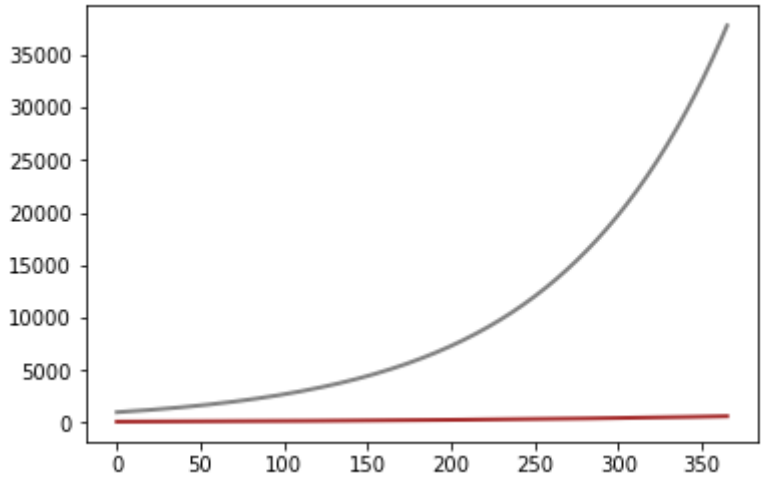
In [1]:

```
# Importing from matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# Starting conditions for rabbits
hasen = 1000           # how many rabbits are present at the beginning
hasen_liste = []       # empty list
hasen_liste.append(hasen) # saves the first entry in the list
hasen_wachstum = 0.01  # Growth rate, how fast the rabbits multiply


# Starting conditions for lynxes
luchse = 100           # how many lynxes are present at the beginning
luchs_liste = []       # empty list
luchs_liste.append(luchse) # saves the first entry in the list
luchs_wachstum = 0.005 # Growth rate, how fast the lynxes reproduce

for it in range(365): # Loop over 365 days
    # Population equations:
    # population = population + growth
    # population = population + growth rate * population
    hasen =  hasen + hasen_wachstum * hasen
    luchse = luchse + luchs_wachstum * luchse

    # per current population is added to the lists
    hasen_liste.append(hasen)
    luchs_liste.append(luchse)

# Lists are plotted (lw indicates the thickness of the lines)
```

```
plt.plot(hasen_liste, color = "gray", lw = 2)
plt.plot(luchs_liste, color = "brown", lw = 2)
```

Out[1]:

```
[<matplotlib.lines.Line2D at 0x26d15b5bc18>]
```



Now we also have to include the term in the equation, which describes that animals die. We want to express this elegantly and need to consider the following information in our equations:

If there are **many rabbits**, the **lynxes can reproduce faster**.

If there are **many lynxes**, **more rabbits die**.

How do we build this in mathematically? Without influencing the animal species among themselves, it would be

$$\Delta \text{ Hasen} = a * \text{Hasen} - b * \text{Hasen}$$
$$\Delta \text{ Luchse} = c * \text{Luchse} - d * \text{Luchse}$$

Now we want to include this additional dependency:

$$\Delta \text{ Hasen} = a * \text{Hasen} - b \sim * \text{Hasen} * \text{Luchse}$$
$$\Delta \text{ Luchse} = c \sim * \text{Luchse} * \text{Hasen} - d * \text{Luchse}$$

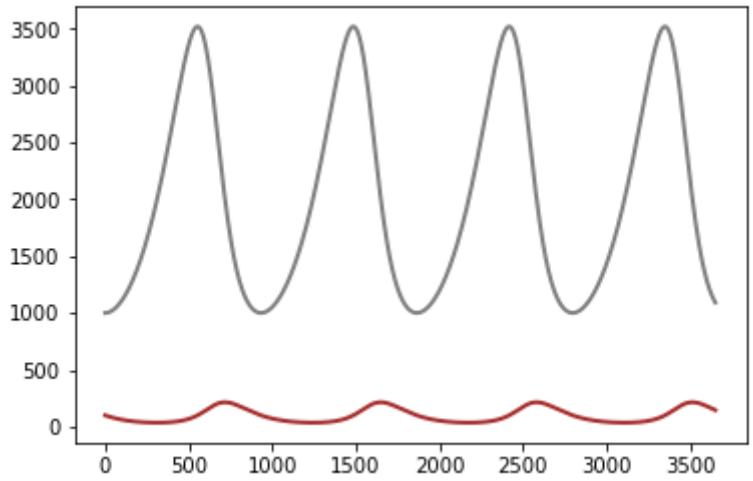
In [36]:

```
# Importing from matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# Starting conditions for rabbits
hasen = 1000           # how many rabbits are present at the beginning
hasen_liste = []       # empty list
hasen_liste.append(hasen) # saves the first entry in the list
hasen_wachstum = 0.005  # Growth rate, how fast the rabbits multiply
hasen_sterben = 0.005 / 100



# Starting conditions for lynxes
luchse = 100           # how many lynxes are present at the beginning
luchs_liste = []       # empty list
luchs_liste.append(luchse) # saves the first entry in the list
```

```
luchs_wachstum = 0.005 / 1000
luchs_sterben = 0.01


for it in range(10 * 365): # Loop over 10 * 365 days
    # Population equations:
    # population = population + growth - death
    hasen =  hasen + hasen_wachstum * hasen - hasen_sterben * hasen * luchse
    luchse = luchse + luchs_wachstum * luchse * hasen - luchs_sterben * luchse

    # per current population is added to the lists
    hasen_liste.append(hasen)
    luchs_liste.append(luchse)

# Lists are plotted (lw indicates the thickness of the lines)
plt.plot(hasen_liste, color = "gray", lw = 2)
plt.plot(luchs_liste, color = "brown", lw = 2)
```

Out[36]:

```
[<matplotlib.lines.Line2D at 0x26d250d65f8>]
```



But we make a small mistake: When we calculate the new rabbit population, we correctly use the rabbit and lynx population from the last time step. Then we overwrite the number of rabbits. So, when we calculate the lynxes in the next line, we are already using the new number of rabbits. Here we should actually use the number from the previous day. So, we have to replace `hasen` with `hasen_liste[it-1]` and `luchse` with `luchs_liste[it-1]`.

In addition, we have to start our loop not at 0 but at 1.

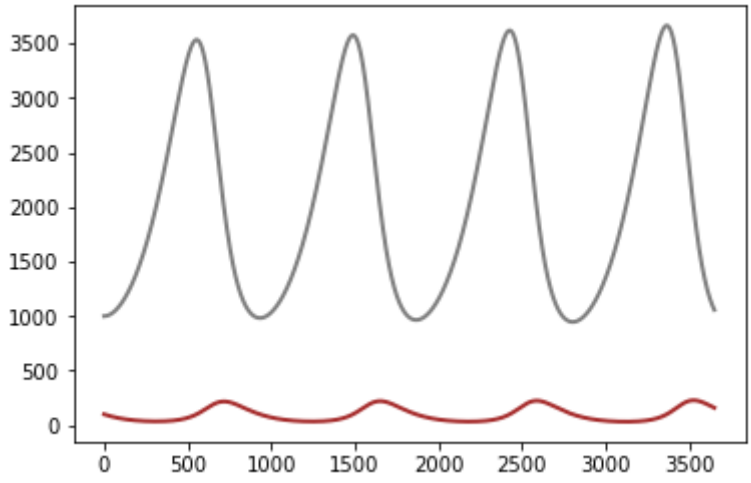
In [40]:

```
# Importing from matplotlib
import matplotlib.pyplot as plt
%matplotlib inline


# Starting conditions for rabbits
hasen = 1000           # how many rabbits are there at the beginning
hasen_liste = []       # empty list
hasen_liste.append(hasen) # saves the first entry in the list
hasen_wachstum = 0.005  # Growth rate, how fast the rabbits multiply
```

```
hasen_sterben = 0.005 / 100



# Starting conditions for lynxes
luchse = 100           # how many lynxes are present at the beginning
luchs_liste = []       # empty list
luchs_liste.append(luchse) #saves the first entry in the list
luchs_wachstum = 0.005 / 1000
luchs_sterben = 0.01



for it in range(1,10 * 365): # oop over 10 * 365 days
    # Population equations:
    # population = population + growth - death
    hasen =  hasen_liste[it-1] + hasen_wachstum * hasen_liste[it-1] - hasen_sterben * hasen_liste[it-1] * luchs_liste[it-1]
    luchse = luchs_liste[it-1] + luchs_wachstum * luchs_liste[it-1] * hasen_liste[it-1] - luchs_sterben * luchs_liste[it-1]

    # per current population is added to the lists
    hasen_liste.append(hasen)
    luchs_liste.append(luchse)

# Lists are plotted (lw indicates the thickness of the lines)
plt.plot(hasen_liste, color = "gray", lw = 2)
plt.plot(luchs_liste, color = "brown", lw = 2)
```

Out[40]:

```
[<matplotlib.lines.Line2D at 0x26d15b5b588>]
```



This calculation is now accurate to the day. If we want more accurate results, we could shorten our time steps, i.e. to hours, minutes or seconds. We could also make infinitely small time steps. The **difference equation** would then become a **differential equation**.

This system of **coupled differential equations** is well known. It is called **Lotka-Volterra system**:

$$\frac{dH}{dt} = \alpha * H - \beta * H * L$$

$$\frac{dL}{dt} = \gamma * L * H - \phi * L$$

But for now we are satisfied with the accuracy of our difference equations. For a Lotka-Volterra system there are 3 rules. In the following we can check whether these rules also apply to our system of difference equations.

Lotka-Volterra rule 1

The predator and prey populations oscillate periodically and at different times. The predator population follows the prey population in time.

To check this, we need to improve our graphical representation a little. Since there are much less lynxes than rabbits, it is difficult to compare the values. It would be better if we don't show the absolute population, but the relative population, i.e. the current population divided by the starting population.

So what we want to do is to divide each entry of the population lists by 1000 or by 100. This is not so easy with lists, but it works with numpy-arrays. So we convert our lists to numpy-arrays, which we can then divide without problems:

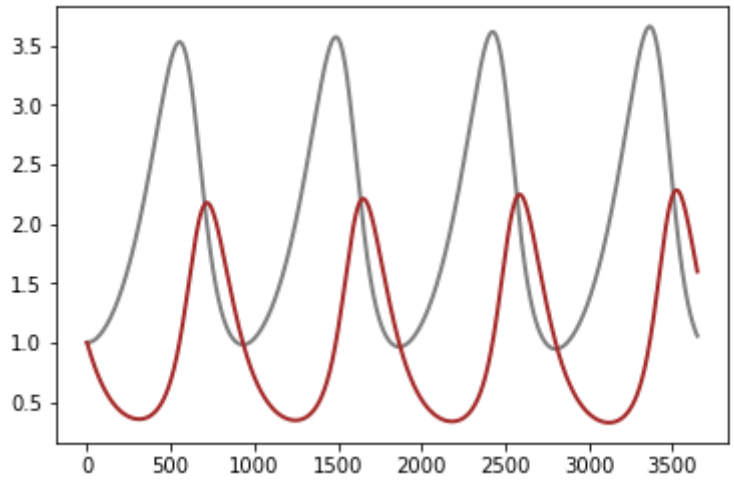
In [44]:

```
# Importing from matplotlib

import matplotlib.pyplot as plt

%matplotlib inline

import numpy as np


# Starting conditions for rabbits

hasen = 1000           # how many rabbits are present at the beginning

hasen_liste = []       # empty list

hasen_liste.append(hasen) # saves the first entry in the list

hasen_wachstum = 0.005  # Growth rate, how fast the rabbits multiply

hasen_sterben = 0.005 / 100



# Starting conditions for lynxes

luchse = 100           # how many lynxes are present at the beginning

luchs_liste = []       # empty list

luchs_liste.append(luchse) # saves the first entry in the list

luchs_wachstum = 0.005 / 1000

luchs_sterben = 0.01



for it in range(1,10 * 365): # Loop over 10 * 365 days

    # Population equations:

    # population = population + growth - death

    hasen = hasen_liste[it-1] + hasen_wachstum * hasen_liste[it-1] - hasen_sterben * hasen_liste[it-1] * luchs_liste[it-1]

    luchse = luchs_liste[it-1] + luchs_wachstum * luchs_liste[it-1] * hasen_liste[it-1] - luchs_sterben * luchs_liste[it-1]


    # per current population is added to the lists

    hasen_liste.append(hasen)
```

```
    luchs_liste.append(luchse)


# Lists are converted to arrays and normalized, i.e. divided by the starting population.
hasen_array = np.array(hasen_liste)
hasen_array = hasen_array / 1000


luchs_array = np.array(luchs_liste)
luchs_array = luchs_array / 100
# Lists are plotted (lw indicates the thickness of the lines)
plt.plot(hasen_array, color = "gray", lw = 2)
plt.plot(luchs_array, color = "brown", lw = 2)
```

Out[44]:

```
[<matplotlib.lines.Line2D at 0x26d2543dfd0>]
```



Here we see clearly: There are oscillations and the maximum of the predator population is always a little bit after the maximum of the prey population. So, the first Lotka-Volterra rule is correctly reproduced in our model.

Lotka-Volterra rule 2

The average sizes of the two populations remain constant over long periods of time, although the maxima and minima are very different.

To check this, we have to look at longer periods of time. Within the first few oscillations it looks like the average population will remain constant, but we should look at longer periods to be on the safe side. Let us increase the simulation period to 100 years.

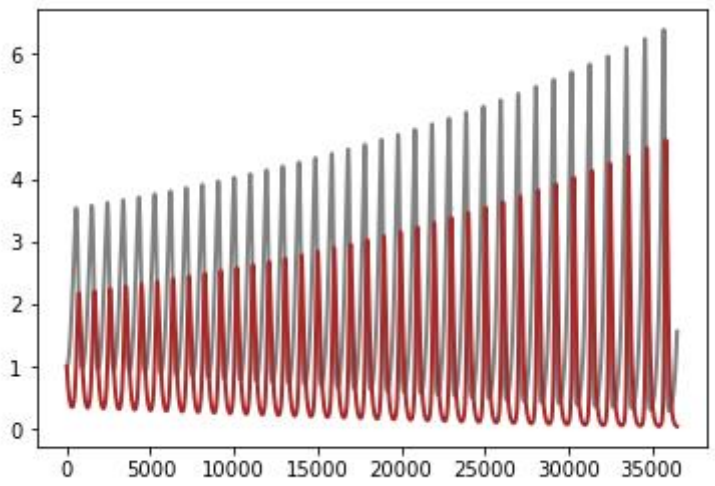
In [49]:

```
# Importing from matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np


# Starting conditions for rabbits
hasen = 1000           # how many rabbits are present at the beginning
hasen_liste = []       # empty list
hasen_liste.append(hasen) # saves the first entry in the list
hasen_wachstum = 0.005  # Growth rate, how fast the rabbits multiply
```

```
hasen_sterben = 0.005 / 100


# Starting conditions for lynxes
luchse = 100           # how many lynxes are present at the beginning
luchs_liste = []       # empty list
luchs_liste.append(luchse) # saves the first entry in the list
luchs_wachstum = 0.005 / 1000
luchs_sterben = 0.01


for it in range(1,100 * 365): # Loop over 10 * 365 days
    # Population equations:
    # population = population + growth - death
    hasen =  hasen_liste[it-1] + hasen_wachstum * hasen_liste[it-1] - hasen_sterben * hasen_liste[it-1] * luchs_liste[it-1]
    luchse = luchs_liste[it-1] + luchs_wachstum * luchs_liste[it-1] * hasen_liste[it-1] - luchs_sterben * luchs_liste[it-1]

    # per current population is added to the lists
    hasen_liste.append(hasen)
    luchs_liste.append(luchse)

# Listen werden zu Arrays konvertiert und normiert, d.h durch die Startpopulation dividiert.
hasen_array = np.array(hasen_liste)
hasen_array = hasen_array / 1000

luchs_array = np.array(luchs_liste)
luchs_array = luchs_array / 100
# Lists are plotted (lw indicates the thickness of the lines)
plt.plot(hasen_array, color = "gray", lw = 2)
plt.plot(luchs_array, color = "brown", lw = 2)
```

Out[49]:

```
[<matplotlib.lines.Line2D at 0x26d253ed208>]
```



Here we see very clearly that the maximum is getting bigger and bigger. So, the second Lotka-Volterra rule is violated in our model. What is the reason for this? The first suspicion is our simplification that instead of a differential equation we only use a difference equation that is accurate to one day. In order to determine whether this is really responsible for this behavior, we have to find a better method to graphically represent longer time developments. In the following we will use the so-called **phase space representation**.

Phase space representation

In our usual graphics we always plot time on one axis and population size on the other axis. But this is not the only possibility we have. We could also use one axis for the predators and the other axis for the prey. Then we no longer represent the time evolution, but much more the relation of predators and prey that we have encountered throughout the entire time evolution.

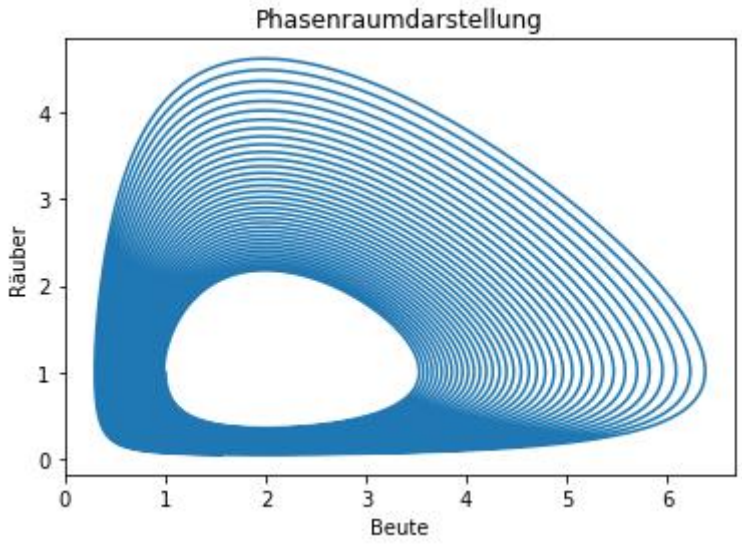
If we always have constant maxima and minima, the phase space representation should have a closed shape, similar to a circle. If we get always greater values, we see a spiral. So let us now make a phase space plot as a test:

In [50]:

```
plt.plot(hasen_array,luchs_array)
plt.title("Phase space representation ")
plt.xlabel("prey")
plt.ylabel("predators")
```

Out[50]:

```
<matplotlib.text.Text at 0x26d26a399b0>
```



So, we clearly see a spiral here, the maxima are getting bigger and bigger. To find out if our calculation accuracy really causes the problem, we can simply increase it. Instead of using whole days as time steps, we now use the hundredth of a day. So we have to make 100 times more time steps, but divide the changes per time step by 100.

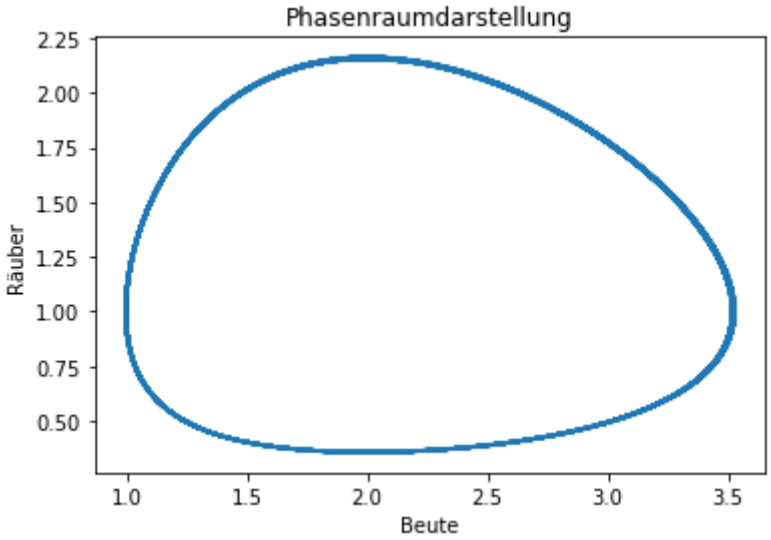
In [63]:

```
# Importing from matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np


# Starting conditions for rabbits
hasen = 1000           # how many rabbits are present at the beginning
```

```
hasen_liste = []       # empty list
hasen_liste.append(hasen) # saves the first entry in the list
hasen_wachstum = 0.005  # Growth rate, how fast the rabbits multiply
hasen_sterben = 0.005 / 100


# Starting conditions for lynxes
luchse = 100           # how many lynxes are present at the beginning
luchs_liste = []       # empty list
luchs_liste.append(luchse) # saves the first entry in the list
luchs_wachstum = 0.005 / 1000
luchs_sterben = 0.01


for it in range(1,100 * 365 * 100): # Loop over 10 * 365 days
    # Population equations:
    # population = population + growth - death
    hasen =  hasen_liste[it-1] + \
    1/100 * (hasen_wachstum * hasen_liste[it-1] - hasen_sterben * hasen_liste[it-1] * luchs_liste[it-1])
    luchse = luchs_liste[it-1] + \
    1/100 * (luchs_wachstum * luchs_liste[it-1] * hasen_liste[it-1] - luchs_sterben * luchs_liste[it-1])

    # per current population is added to the lists
    hasen_liste.append(hasen)
    luchs_liste.append(luchse)

# Lists are converted to arrays and normalized, i.e. divided by the starting population.
hasen_array = np.array(hasen_liste)
hasen_array = hasen_array / 1000

luchs_array = np.array(luchs_liste)
luchs_array = luchs_array / 100

plt.plot(hasen_array,luchs_array)
plt.title("Phase space representation ")
plt.xlabel("prey")
plt.ylabel("predator")
```

Out[63]:

```
<matplotlib.text.Text at 0x26d0b1e2d30>
```



Now the phase space representation is really a closed figure. So, the calculation accuracy really makes a difference. Now the second Lotka-Volterra rule is fulfilled with increased accuracy.

Lotka-Volterra rule 3

If predator and prey population are simultaneously decimated by the same percentage, the mean value of the prey population increases in the short term and the mean value of the predator population decreases in the short term.

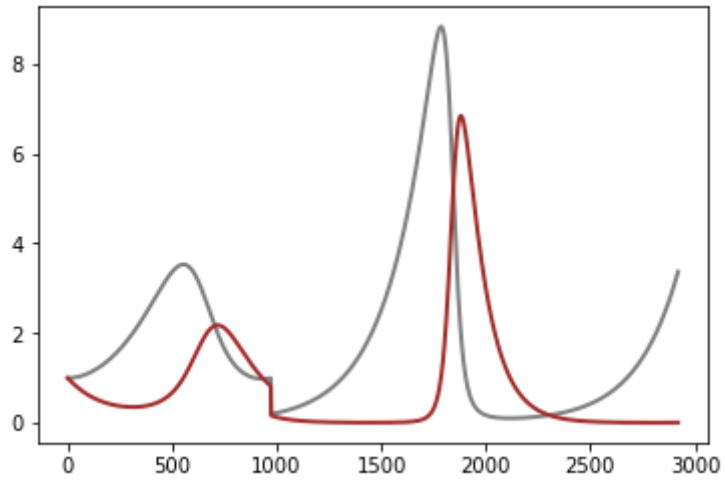
We can also try this effect very easily. In our model, a period (i.e. the time between two points, in which both predator and prey population have the normalized value 1) lasts 972 time units. So, we can calculate the average of the first 972 time units, then decimate the populations, and then calculate the average of the second 972 time steps. Mean values of arrays are best calculated with the numpy command `np.mean`. This command can also be used to calculate the mean value of certain parts of an array. To do this, write `np.mean(liste[anfang:ende])` for example `np.mean(hasen_array[0:972])`

In [1]:

```
# Importing from matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np


# Starting conditions for rabbits
hasen = 1000           # how many rabbits are present at the beginning
hasen_liste = []       # empty list
hasen_liste.append(hasen) # saves the first entry in the list
hasen_wachstum = 0.005  # Growth rate, how fast the rabbits multiply
hasen_sterben = 0.005 / 100


# Starting conditions for lynxes
luchse = 100           # wie viele Luchse sind am Anfang vorhanden
luchs_liste = []       # empty list
luchs_liste.append(luchse) # saves the first entry in the list
luchs_wachstum = 0.005 / 1000
luchs_sterben = 0.01
```

```
for it in range(1,8 * 365): # Loop over 8 * 365 days
    # Population equations:
    # population = population + growth - death
    hasen =  hasen_liste[it-1] + hasen_wachstum * hasen_liste[it-1] - hasen_sterben * hasen_liste[it-1] * luchs_liste[it-1]
    luchse = luchs_liste[it-1] + luchs_wachstum * luchs_liste[it-1] * hasen_liste[it-1] - luchs_sterben * luchs_liste[it-1]
    # Decimate the population:
    if it == 972:
        hasen = hasen * 0.2
        luchse = luchse * 0.2

    # per current population is added to the lists
    hasen_liste.append(hasen)
    luchs_liste.append(luchse)

# Lists are converted to arrays and normalized, i.e. divided by the starting population.
hasen_array = np.array(hasen_liste)
hasen_array = hasen_array / 1000

luchs_array = np.array(luchs_liste)
luchs_array = luchs_array / 100
# Lists are plotted (lw indicates the thickness of the lines)
plt.plot(hasen_array, color = "gray", lw = 2)
plt.plot(luchs_array, color = "brown", lw = 2)

Hmittel1=np.mean(hasen_array[0:972])
Hmittel2=np.mean(hasen_array[972:2*972])
Lmittel1=np.mean(luchs_array[0:972])
Lmittel2=np.mean(luchs_array[972:2*972])

print("Rabbit mean value changes from ")
print(Hmittel1)
print("to")
print(Hmittel2)
print("")
print("lynx mean value changes from ")
print(Lmittel1)
print("to")
print(Lmittel2)
```

```
Rabbit mean value changes from
1.960012221793669
to
2.71757133048013

Lynx mean value changes from
0.9988842085172154
to
0.855243907572326
```



We see that the third Lotka-Volterra rule is also fulfilled by our model.

Summary

Coupled differential equation systems

Predator-prey systems of animal populations are represented mathematically with coupled systems of differential equations and, because usually no analytical solution is possible, simulated on the computer as a system of difference equations.

Phasenraumdarstellung

In this representation, not several variables are plotted against time, but one variable against another. In predator-prey systems, for example, the prey population is displayed on the x-axis and the predator population on the y-axis. You can read the plot like this: For each value of the prey population on the x-axis, we find on the y-axis the size of the predator population under these conditions. But we do not see the time itself on this plot.

The Lotka-Volterra rules

- The predator and prey populations oscillate periodically and at different times. The predator population follows the prey population in time.
- The average sizes of the two populations remain constant over long periods of time, although maxima and minima are different.
- If predator and prey population are decimated by the same percentage at the same time, the mean value of the prey population increases for a short time and the mean value of the predator population decreases for a short time.

Arrays

In contrast to lists, arrays can be multiplied by numbers and divided by numbers. Each element in the array is treated individually, similar to a vector. Mean values of arrays or parts of arrays can be calculated with `np.mean(arrayname[anfang:ende])`.

Chapter 6 - Differentiating and integrating - a solar car

As we have already seen in the previous chapters, systems sciences are mainly interested in **dynamical systems**, i.e. systems whose variables or states change. To detect changes, movements or developments, mathematics provides an elaborate methodology, the **differential and integral calculus**. Beginning with this chapter, we will deal with this method, which is so central to systems sciences, not primarily from the perspective of mathematics, but rather from the possibilities offered by computers.

As examples, we look at the process of generating electricity using a solar panel and the operation of an electric vehicle. We assume - in order to get to know this in Python as well - that this requires reading and editing data from external files, i.e. from files that we have stored on the computer but have not yet considered in the context of the Jupyter notebook (the corresponding files can be found here (right click + save target as): `solargrob.txt`, `solarfein.txt` and `batterie.txt`).

Reading in data

First of all, we are interested here in how much energy a solar panel supplies. In the file `solargrob.txt` we find data about how much power the panel delivers at which hour of the day. So, we need a method how to load this file, respectively the content of this file into a Jupyter notebook to be able to work with Python. For the so-called reading of data there are packages in Python. One of them is called `csv` (for *comma separated values*), and allows to use contents from files.

In a first step we load this package into our notebook, together with the already known package `matplotlib` for scientific drawing.

```
In [1]:
import csv
import matplotlib.pyplot as plt
%matplotlib inline
```

In the next step we access the text file in question. (The file `solargrob.txt` must be saved in the same folder as the Jupyter notebook).

The file contains 24 entries, one for every full hour of a measurement day. Each entry indicates the power of the solar panel at that hour in Watts. To open a file in Python and give it a name (e.g. `inputfile`) you can use the command `with open(filename.txt) as inputfile`. All commands that should use this name `inputfile` must be indented, similar to a For-loop.

We open the file and try to output its contents with the print command:

```
In [2]:
import csv
import matplotlib.pyplot as plt
%matplotlib inline

with open('solargrob.txt') as inputfile: #solargrob.txt is opened and
    # called inputfile in this program
```

```
    print(inputfile)                    #inputfile shall be printed
```

```
<open file 'solargrob.txt', mode 'r' at 0x000000000A4DCA50>
```

We see: the `print` command does not deliver the desired result. We do not see 24 numbers, but only the memory address, where this file is stored inside the computer. To extract the content of the text file, we need to do another step. We need to iterate over the whole file, line by line, and output each line separately. To do this we use a For-loop again, combined with the new command `csv.reader`, which reads contents from files.

```
In [3]:
import csv
import matplotlib.pyplot as plt
%matplotlib inline

with open('solargrob.txt') as inputfile: #solargrob.txt is opened and
    # called inputfile in this program
    for row in csv.reader(inputfile):    #for every row inside of inputfile
        print(row)                       #the row is printed
```

```
['3.77513454428e-08']
['1.74472887359e-06']
['5.77774851942e-05']
['0.00137095908638']
['0.0233091011429']
['0.283962983903']
['2.47875217667']
['15.503853599']
['69.4834512228']
['223.130160148']
['313.417119033']
['446.481724891']
['1000.0']
['846.481724891']
['513.417119033']
['223.130160148']
['69.4834512228']
['15.503853599']
['2.47875217667']
['0.283962983903']
['0.0233091011429']
['0.00137095908638']
['5.77774851942e-05']
['1.74472887359e-06']
```

The result is more like what we expect. We can recognize numbers. But the first numbers are very small (e-08 at the end of a number means that the number in front is multiplied by 10^-8, i.e. by 0.00000001). The maximum of the numbers is at the 13th entry. This is the performance data of a solar panel.

Nevertheless, this output is not perfect yet, because the numbers are within some special characters that disturb us when editing, for example when drawing a performance curve. The fact that each entry is in square brackets tells us that the entries are lists with only one entry each. If

we had a file with several entries per line, this might be a reasonable notation. But in our case, we would prefer single entries, and not lists of length 1, so we specify that we need only the first (and only) entry of the whole line, i.e. the entry with index 0:

In [4]:

```
import csv
import matplotlib.pyplot as plt
%matplotlib inline

with open('solargrob.txt') as inputfile: #solargrob.txt is opened and
    # called inputfile in this program
    for row in csv.reader(inputfile):    #for every row inside of inputfile
        print(row[0])                    #the first entry of each row is
printed
3.77513454428e-08
1.74472887359e-06
5.77774851942e-05
0.00137095908638
0.0233091011429
0.283962983903
2.47875217667
15.503853599
69.4834512228
223.130160148
313.417119033
446.481724891
1000.0
846.481724891
513.417119033
223.130160148
69.4834512228
15.503853599
2.47875217667
0.283962983903
0.0233091011429
0.00137095908638
5.77774851942e-05
1.74472887359e-06
```

That looks better now. In the next step we should check the **data type**. When reading data, Python often assumes that it is text (simply put: letters, so-called `strings`).

Note: The data type indicates how a date - such as a number or letter - is stored on the computer. Usually programming languages cannot recognize the data type independently. For example, a '1' can be stored as a letter (`string`) or as an `integer` or as a real number (`float`). What looks almost identical for us and is easily understood in its meaning, are three completely different objects for the computer.

Therefore, we should use the `type` command to get the type of the variable. Since we need only a few entries to determine the type, we limit our output to five with the help of a counter `c` and an `If-query`.

In [5]:

```
import csv
import matplotlib.pyplot as plt
%matplotlib inline

c = 0
with open('solargrob.txt') as inputfile:
    for row in csv.reader(inputfile):
        if c < 5:
            print(row[0])
            print(type(row[0]))      #type identifies the type of a variable
        c = c + 1
```

```
3.77513454428e-08
<type 'str'>
1.74472887359e-06
<type 'str'>
5.77774851942e-05
<type 'str'>
0.00137095908638
<type 'str'>
0.0233091011429
<type 'str'>
```

As expected, the data are stored as text (i.e. as `<type 'str'>`), but not as numbers, which can be used for calculations. So we first have to transform this text into floating point (i.e. real) numbers, so that we can work with them. This is done in Python with the `float` command (we will demonstrate this again only for the first five entries and convert the remaining entries into real numbers in the next step below):

In [6]:

```
import csv
import matplotlib.pyplot as plt
%matplotlib inline

c = 0
with open('solargrob.txt') as inputfile:
    for row in csv.reader(inputfile):
        if c < 5:
            print(float(row[0]))    # float converts the text into a number
            print(type(float(row[0])))
        c = c + 1
```

```
3.77513454428e-08
<type 'float'>
1.74472887359e-06
<type 'float'>
5.77774851942e-05
<type 'float'>
0.00137095908638
<type 'float'>
0.0233091011429
<type 'float'>
```

As the last step of our data preprocessing, we don't want to just write the numbers on the screen, but save them in a list so that we can really work with them. So we create an empty list and then fill it with the converted entries. After this, we read out the entire list using the `print` command:

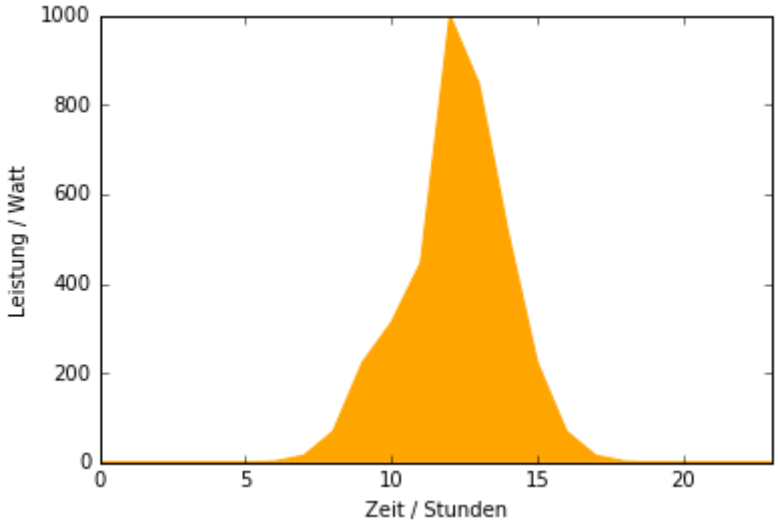
In [7]:
```
import csv
import matplotlib.pyplot as plt
%matplotlib inline

solargrob = [] # an empty list is created

with open('solargrob.txt') as inputfile:
    for row in csv.reader(inputfile):
        solargrob.append(float(row[0])) # every entry of each row is
        # appended to the list as a number
print(solargrob)    # the whole list is printed
```

```
[3.77513454428e-08, 1.74472887359e-06, 5.77774851942e-05, 0.00137095908638,
0.0233091011429, 0.283962983903, 2.47875217667, 15.503853599, 69.4834512228
, 223.130160148, 313.417119033, 446.481724891, 1000.0, 846.481724891, 513.4
17119033, 223.130160148, 69.4834512228, 15.503853599, 2.47875217667, 0.2839
62983903, 0.0233091011429, 0.00137095908638, 5.77774851942e-05, 1.744728873
59e-06]
```

Now we can work with these measurement data. For example, we can create a plot that shows how much power the solar panel produces at what time of day. For this we can use the plot command `plt.fill`, which in contrast to `plt.plot` also colors the area under a curve.

In [8]:
```
import csv
import matplotlib.pyplot as plt
%matplotlib inline

solargrob = []

with open('solargrob.txt') as inputfile:
    for row in csv.reader(inputfile):
        solargrob.append(float(row[0]))

plt.fill(range(24), solargrob, color = "orange") # fill is similar to the
# plt. command but fills the area under the curve
plt.ylabel("Performance / Watts")
plt.xlabel("Time / Hours")
plt.xlim(0, 23)
```

Out[8]:
```
(0, 23)
```



Numerical integration

We can now clearly see the performance over time. At peak time around 12h the panel delivers 1000 Watt (W), exactly 1 Kilowatt (kW). But what is its total output over 24 hours?

If the panel would always supply 1000 watts for 24 hours, calculating the total power would be easy. This would be

$$1\text{kW} * 24\text{h} = 24\text{kWh} \qquad 1\text{kW} * 24\text{h}=24\text{kWh}$$

But as we can see in the plot, the performance is different at every time of day. So, we cannot add up 24 times the same value to get the total power. Instead, we have to consider the value stored in our list for each hour. We have 24 entries in this list, so we know the power at each hour of the time of day. Obviously, we should be able to simply add up the entries in our list to determine the total daily power. In practice, summing up means that we approximate the yellow area under the curve. (This is only approximate, because we only consider the power data at every full hour and not, for example, at 12:30 pm. See below).

To calculate the area under a curve, mathematics uses **integral calculations**. Our "summing up a list" corresponds to this purpose. This is called the method of **numerical integration**.

Python provides the simple command `sum` for this summation process. In the following, we use this command to determine how much power the solar panel delivers over the day:

In [9]:

```
import csv
import matplotlib.pyplot as plt
%matplotlib inline

solargrob = []

with open('solargrob.txt') as inputfile:
    for row in csv.reader(inputfile):
        solargrob.append(float(row[0]))

# sum adds all elements of the list = numerical integration
gesamtleistung = sum(solargrob)
print(gesamtleistung)
```

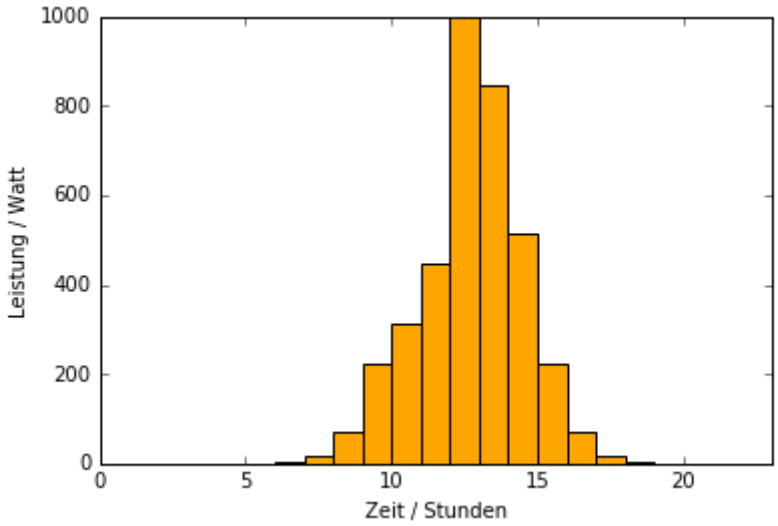
```
3741.60752731
```

In total, our solar panel produces about 3741 watt hours in one day. Of course, this result is still very inaccurate. Why? As already mentioned, we have so far assumed in this analysis that there is a fixed output value for every hour of the day. And we implicitly assume that this value remains constant within this hour. This is of course not the case in reality.

We can show the effect of our simplification with a so-called `Bar-Plot`. This allows us to display the data in the form of rectangles, just as our summing up works. You can see immediately that this can only be an approximate solution:

In [10]:
```
plt.bar(range(24), solargrob, width = 1.0, color = "orange")
plt.xlim(0, 23)
plt.ylabel("Performance / Watts")
plt.xlabel("Time / Hours")
```

Out[10]:
```
<matplotlib.text.Text at 0xad61630>
```



Increasing the accuracy

In order to increase the accuracy of our results, we have two basic options. Either we find better integration methods or we simply increase the time resolution of our measurement data.

Here we do the second option. Fortunately, we can fall back on more precise data. In addition to the file we have already read in, there is a file that contains data accurate to the minute, i.e. it has `24 * 60 = 1440` entries. In the following we will also add this file to our Python code and create a plot.

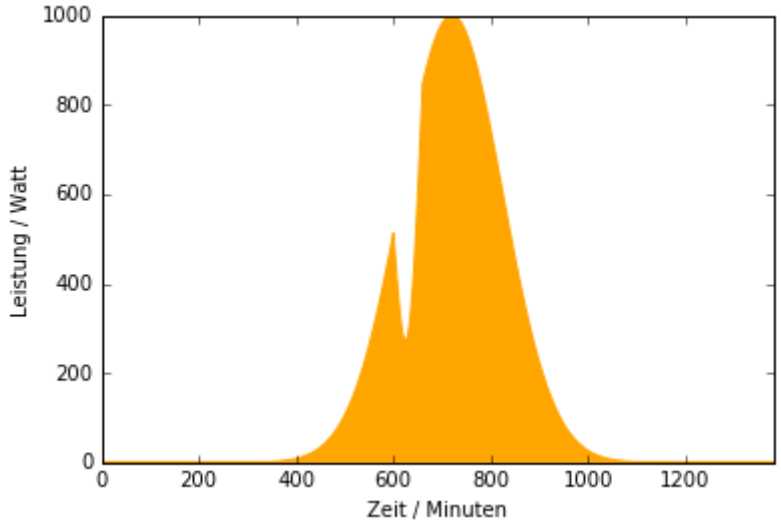
(The conversion from hours to minutes does not have to be brought forward here. The command `range(1440)` can also be written as `range(24 * 60)`.

In [11]:
```
import csv
import matplotlib.pyplot as plt
%matplotlib inline

solarfein = []
with open('solarfein.txt') as inputfile:
    for row in csv.reader(inputfile):
```

```
        solarfein.append(float(row[0]))

plt.fill(range(24 * 60),solarfein, color = "orange")
plt.ylabel("Performance / Watts")
plt.xlabel("Time / Minutes")
plt.xlim(0, 24 * 60)
```

Out[11]:

```
(0, 1380)
```



In this resolution we see a number of details, for example, the cloudiness that obviously briefly clouded the morning.

It is now obvious to perform a numerical integration with these data to calculate the overall performance. But beware! With the coarser data we had summed up 24 numbers, the largest of which was 1000. Now we want to sum up 1440 numbers, of which again the largest is 1000. This cannot give a similar result. Obviously, we have missed something?

For the coarser data our argumentation was as follows: If the solar panel delivers 1000 watts for one hour, it produces one kilowatt hour. Therefore, it was allowed to simply add up the values in the list.

But now we have data accurate to the minute. The statement "*If a solar panel delivers 1000 watts for one minute, it produces one kilowatt hour*" would be **wrong**. The correct statement is: "*If a solar panel delivers 1000 watts for one minute, it produces one kilowatt minute, i.e. one sixtieth of a kilowatt hour.”*

So, the values in our list have the wrong unit. Given are kilowatt-minutes, but we would like to have kilowatt-hours, so that we can compare the result of the numerical integration (the summation) with our original calculation. So, we have to include this conversion factor (1/60) and can only add up afterwards. To do this, we create a For-loop that converts every value in the original list and saves it in a new list. But how long must this loop run? That depends on the length of the file `solarfein`. We get this length with `len(solarfein)`:

In [5]:

```
import csv
import matplotlib.pyplot as plt
%matplotlib inline

solarfein = []
```

```
with open('solarfein.txt') as inputfile:
    for row in csv.reader(inputfile):
        solarfein.append(float(row[0]))

solarfeinumgerechnet = []

# each value is divided by 60 and then appended to the new list
for it in range(len(solarfein)):
    solarfeinumgerechnet.append(solarfein[it]/60)

gesamtleistungfein = sum(solarfeinumgerechnet)
print(gesamtleistungfein)
```

```
4086.1419317488526
```

So, we see that our original result (3.7 kW) was still relatively far from the more accurate result (4.1 kW). Furthermore, when we look at the plot we notice that most of the electricity is produced at noon. It would be interesting to know what percentage of the total power is actually produced between 11 and 13 o'clock.

To do this, we create a sum again, i.e. integrate numerically. But this time we do not want the integral over the whole period, but only over a part. Therefore, we select this part of the list. How is this done?

We already learned, that we can address single or several entries of a list. `listenname[4]` returns for example the entry with index 4, which is the 5th entry of a list. All elements from 4 (inclusive) to 20 (exclusive) can be obtained for example by `listenname[4:20]`. We can now use this knowledge to calculate the electricity production at noon:

```
In [13]:
import csv
import matplotlib.pyplot as plt
%matplotlib inline

solarfein = []
with open('solarfein.txt') as inputfile:
    for row in csv.reader(inputfile):
        solarfein.append(float(row[0]))

solarfeinumgerechnet = []

for wert in solarfein:
    solarfeinumgerechnet.append(wert/60)

gesamtleistungfein = sum(solarfeinumgerechnet) # total sum
print(gesamtleistungfein)

# Sum of the values between 11 and 13 o'clock
gesamtleistungmittag = sum(solarfeinumgerechnet[11 * 60 : 13 * 60])
print(gesamtleistungmittag)
# Calculate the percentage share
relation = gesamtleistungmittag / gesamtleistungfein * 100
```

```
print("Relation in percent:")
print(relation)
```

```
4086.14193175
1894.1770029
Relation in percent:
46.3561235644
```

So about 46% of the total power of our solar panel is produced at lunchtime. It does not seem advisable to operate electrical devices exclusively via the solar panel, as these devices would only work well at noon. It might be advantageous to store the electricity produced at noon in a battery and then give it back when it is needed.

For example, we could operate a small electric car.

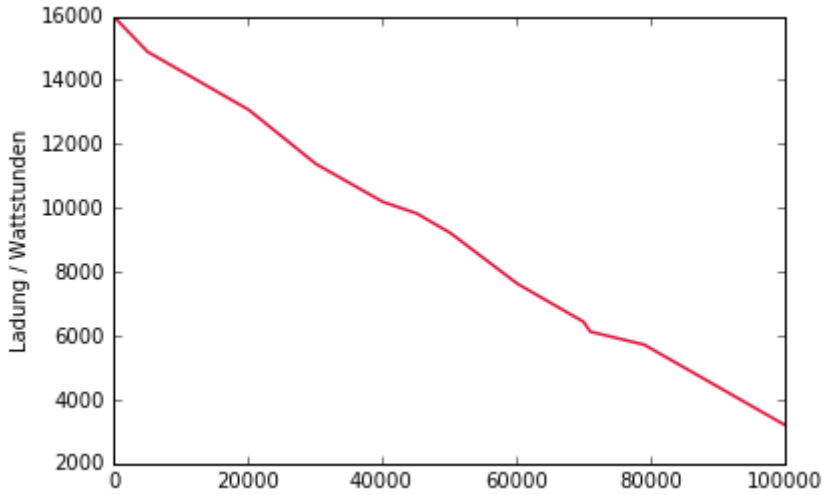
Let's take a closer look at this possibility and especially at the consumption of this vehicle.

Consumption of an electric vehicle

In addition to the power production of a solar panel, we also have the measured data of the power consumption of a small electric vehicle (download the file `batterie.txt` from the Moodle repository and save it in the notebook folder). For a test drive the car was fully charged and then after every meter driven the current battery level was logged. The result is tens of thousands of data points that we can no longer simply evaluate by hand. In a first step we want to read the data back into our Python code and display it graphically:

In [14]:
```
import csv
import matplotlib.pyplot as plt
%matplotlib inline

batterie = []
with open('batterie.txt') as inputfile:
    for row in csv.reader(inputfile):
        batterie.append(float(row[0]))


plt.plot(batterie, color = 'crimson', lw = 1.5)
plt.ylabel("Charge / watt hours ")
```

Out[14]:
```
<matplotlib.text.Text at 0xaf485c0>
```



On the basis of this plot you can already tell a lot about the test vehicle. With a full battery it has a charge of 16.000 watt hours, i.e. 16 kilowatt hours. So, to be fully charged, it would have to charge for about 4 days via our solar panel.

We can also see that the car covered exactly 100.000 meters, i.e. 100 kilometres during the test drive. After this distance the remaining charge is about 3 kilowatt hours, so 13 kilowatt hours were used. By simple division we can see that the average consumption of the electric car is about 13 kilowatt hours per 100 kilometres, i.e. 0.13 kilowatt hours per 1 kilometre.

Also striking: The consumption is not the same throughout. Sometimes the battery level drops faster, sometimes slower. This can have several reasons: accelerating strongly, but also maintaining a high speed increases consumption.

We could now ask ourselves on which sections of the test track the consumption was particularly high or particularly low. However, we have to define that consumption means the charge loss rate of the battery. Graphically expressed: How steep is the curve of the function that shows the battery consumption. Mathematically expressed: How large is the change or derivative of the function at any point.

That is, we are looking for the **derivation of the function** "battery charge". The function itself tells us how much charge is in the battery, the derivative tells us how this charge changes. It shows us the consumption. Mathematically, the method for determining this consumption is called **numerical differentiation**.

Numerical differentiation

Calculating the slope of a curve (a function) is not difficult on the computer. We know the battery charge at every point of our test track. The change in battery charge is then calculated from the charge at this point minus the charge at the point immediately following.

There is also a Python command for this kind of numerical differentiation, which can be found in the package `numpy` *(numerical python)*, together with many other commands for numerical calculation. We import this package and use the `diff` command to calculate the derivative of the charge, i.e. the consumption.

It is important to pay attention to the sign: the slope of our curve is of course negative (the charge is getting less and less), but the consumption is defined as a positive value. You say

"My car uses five liters per 100 kilometers."

and not

"The tank of my car changes by -5 liters per 100 kilometers."

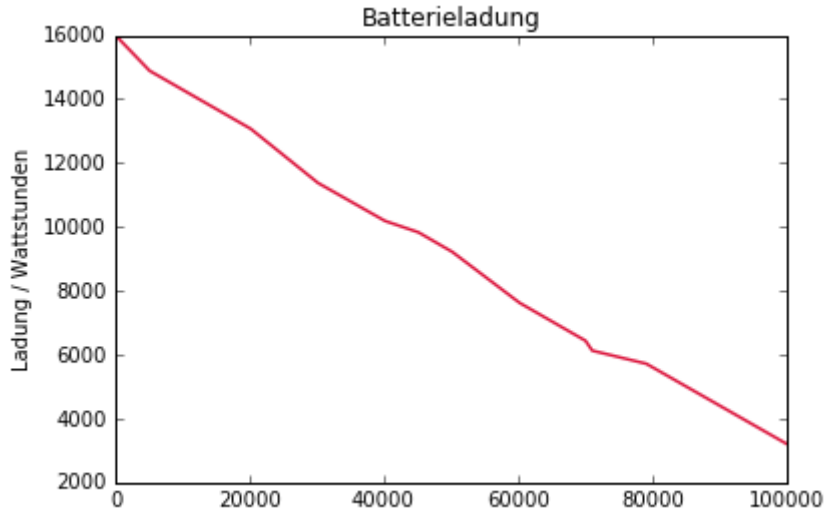
The consumption we are looking for is therefore calculated as "slope times -1". In the following we draw the plots for battery charge and battery consumption on top of each other.

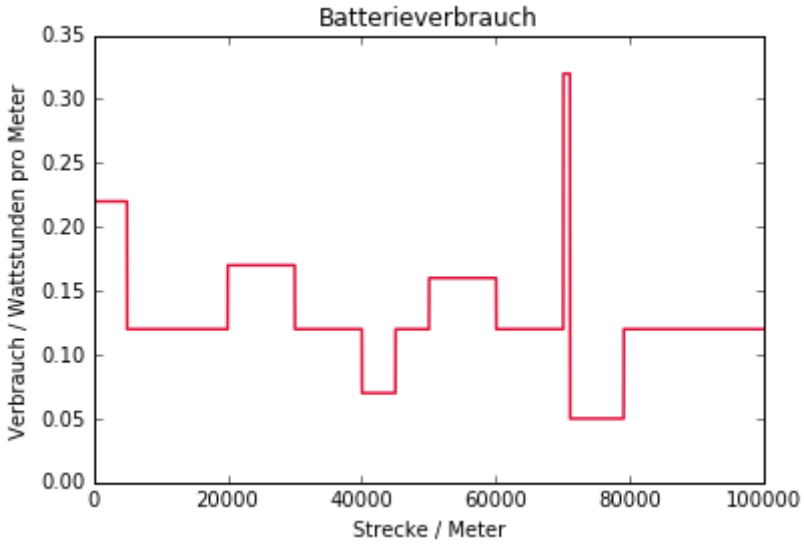
In [15]:

```
import numpy as np
import csv
import matplotlib.pyplot as plt
%matplotlib inline

batterie = []
with open('batterie.txt') as inputfile:
    for row in csv.reader(inputfile):
        batterie.append(float(row[0]))

plt.plot(batterie, color = 'crimson', lw = 1.5)
plt.ylabel("Charge / watt hours")
plt.title('Battery charge')
# np.diff calculates the derivative, i.e. the slope. The consumption is -1
# times slope
verbrauch = -1 * np.diff(batterie)
plt.figure()
plt.plot(verbrauch, color = 'crimson', lw = 1.5)
plt.ylabel("Consumption / watt hours per meter")
plt.xlabel("distance / meter")
plt.title(' Battery consumption')
```

Out[15]:

```
<matplotlib.text.Text at 0xae01978>
```





In a direct comparison we can see how meaningful the consumption is in comparison to the charge. In the consumption curve we can read directly that the consumption is usually about 0.12 watt hours per meter (= 0.12 kilowatt hours per kilometre). In addition, we can see very precisely at which points the consumption is increased or low. The highest consumption on this test track was over 0.30 watt hours per meter.

Numeric integral

The relation between charge and consumption of the battery in this example corresponds mathematically to the relation between **integral and differential**. The former indicates a **stock**, here the stock at certain points of the path, the latter indicates a **flow**, or a rate of change, over these points of the path.

Just as the consumption can be calculated from the information about the charge at different points of the route, the consumption can be used to draw conclusions about the charge. For this purpose, the system is integrated again. Conceptually, this is different from the integral used in the solar panel. For the solar panel we calculated the **area under the curve**. It was a so called **definite integral**, which provides a **number** as a solution (e.g. the total power production).

Here, however, we had given the current charge at any time. So, we are not looking for an area here, but for the so-called **primitive function** of consumption, i.e. the charge. This is an **indefinite integral**, which has a **function** as solution.

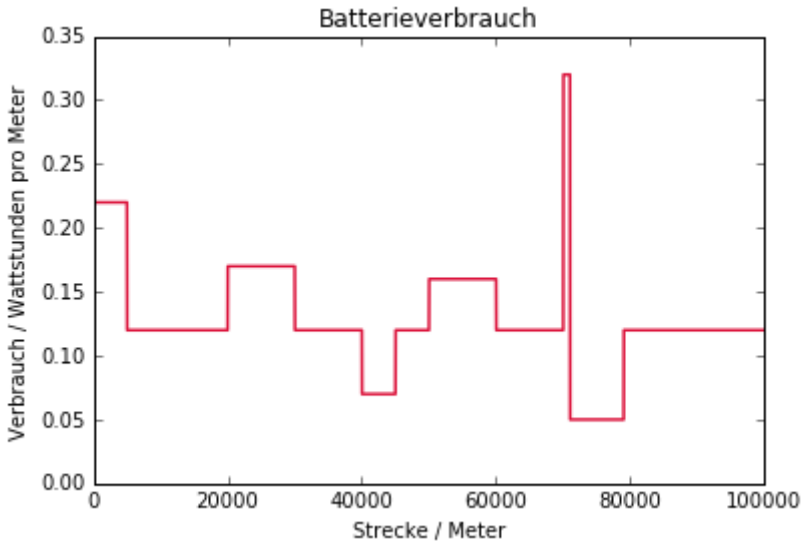
The `numpy` package contains a command that calculates the primitive function of a function using a method called "cumulative sum". In simple terms, it calculates for each time step how much of the total consumption was at all previous time steps. The command for this is `cumsum` and returns a list as the result. Let us try to calculate from the consumption back to the charge:

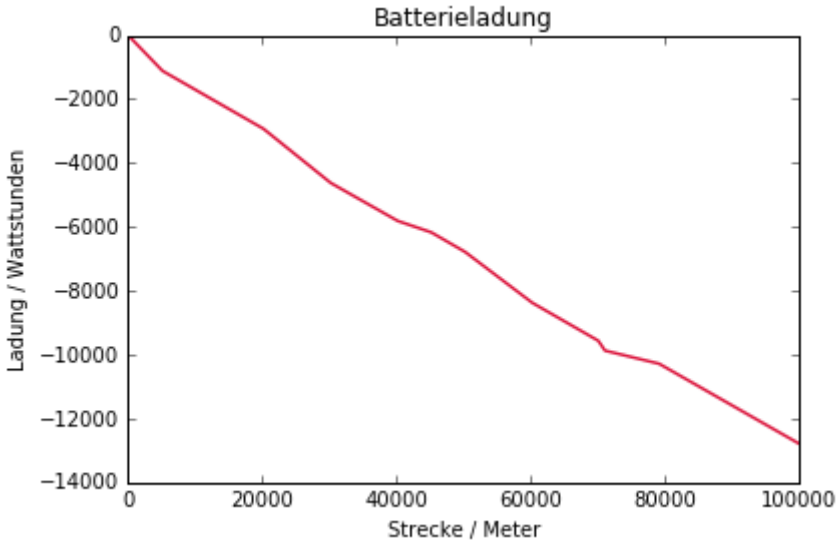
In [16]:
```
import numpy as np
import csv
import matplotlib.pyplot as plt
%matplotlib inline

batterie = []
with open('batterie.txt') as inputfile:
    for row in csv.reader(inputfile):
        batterie.append(float(row[0]))
```

```
# 1. the charge
# plt.plot(batterie, color = 'crimson', lw = 1.5)
# plt.ylabel("Charge / watt hours")
# plt.title('Battery charge')

# 2. the consumption (= verbrauch)
verbrauch = -1 * np.diff(batterie)
plt.figure()
plt.plot(verbrauch, color = 'crimson', lw = 1.5)
plt.ylabel("Consumption / watt hours per meter")
plt.xlabel("distance / meter")
plt.title('Battery consumption')

# 3. and again the charge, where np.cumsum calculates the cumulative sum to
# determine the primitive function
batterie2 = np.cumsum(-verbrauch)
plt.figure()
plt.plot(batterie2, color = 'crimson', lw = 1.5)
plt.ylabel("Charge / watt hours")
plt.xlabel("distance / meter")
plt.title(' Battery charge')
```

Out[16]:

```
<matplotlib.text.Text at 0xcc57ba8>
```





Based on the consumption, we have thus succeeded here in reconstructing the charge level of the battery at the individual points along the route.

However, we lost an important piece of information: if we only consider the consumption, we cannot say how much charge was in the battery at the beginning. In the present calculation, our integral assumes that the battery had a charge of 0 at the beginning of the journey, so the charge becomes negative overall.

This is not a fundamental error, but an intrinsic property of the integral calculus: when calculating an indeterminate integral, the result is always correct only up to a constant. This constant is called "**integration constant**" in mathematics. Note that such a constant exists and an indefinite integral always provides the correct function, but the starting value of this function can differ from zero.

Summary

Reading in data

To read data from a file, the package csv can be used as followed:

```
In [5]:
import csv
datenliste = []

with open('solargrob.txt') as inputfile:
    for row in csv.reader(inputfile):
        datenliste.append(float(row[0]))
```

The resulting list can then be further processed within a For-loop, for example.

Selecting list items

To select a part of the entries from a list, the command `listenname[anfang:ende]` is available. For example, to select the first 10 elements of the `solarstrom` list, write `solarstrom[0:10]` (caution: this will give you the entries *including* the element `solarstrom[0]`, but *excluding* the element `solarstrom[10]`).

Numerical integration

To calculate the area under a curve, i.e. a **definite integral**, the command `sum` is available. The result of a definite integral is a **number**. It is important to use the correct units, or in geometrical interpretation, to determine the width correctly in addition to the height of the rectangles that are summed (i.e. whether a rectangle is one watt-hour wide or only 1/60 watt-hour wide).

In addition to the definite integral, the **indefinite integral** can also be calculated. The indefinite integral always returns a **function** as solution, so in our case a list. To calculate an indefinite integral, i.e. a primitive function, the command `cumsum` from the package `numpy` is available. However, the resulting primitive function is always correct only up to a so-called integration constant.

Numerical differentiation

To calculate the derivative (i.e. the slope at each point) of a function or time series, i.e. to *differentiate*, the command `diff` from the package `numpy` is available. Just as in the present example the consumption was calculated from the battery charge, a distance travelled can be converted into speed or a speed into acceleration.

Chapter 7– Random numbers – an energy mix

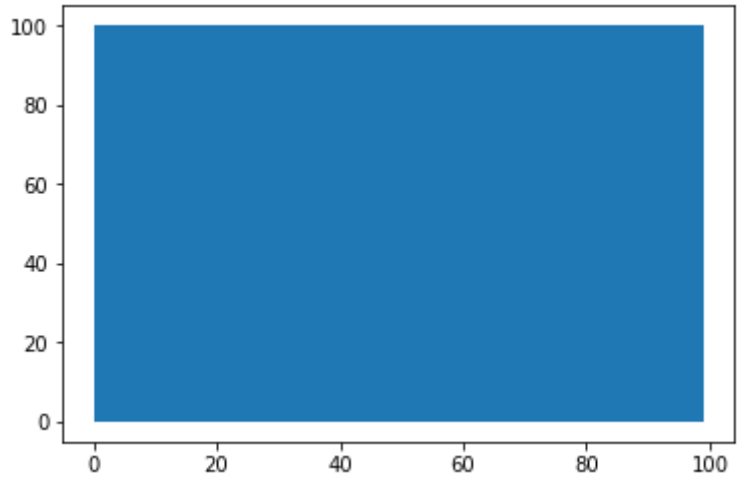
In the examples of systems science modeling considered so far, we have assumed that we know the data we need for this, for example, how large the growth rate or the initial size of an animal population is. Unfortunately, this is not always the case. Often, we have at best a more or less good idea about an interval within which the required data are located. In these cases, we can use **random numbers** to approximate our model to realistic conditions. Fortunately, Python provides us with the necessary tools for this as well.

In order to demonstrate this, in the following we will look at an energy network consisting of wind, hydro and coal-fired power plants. We will compare the production of these power plants with the consumption and find strategies to meet the demand as best as possible and to use as little fossil fuels as possible.

We simply start our model with a For-loop that iterates hour by hour over the simulation period (= Simulationszeit). In this basic framework, we already build a hydroelectric power (= Wasserkraft) plant that constantly supplies electricity. In order to graphically display the electricity produced, we use a so-called `Stack-Plot`:

In [6]:
```
import matplotlib.pyplot as plt
%matplotlib inline

simulationszeit = 100
wasserkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = 100
    wasserkraft_liste.append(wasserkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste)
```

Out[6]:
```
[<matplotlib.collections.PolyCollection at 0x1ef1f5463c8>]
```

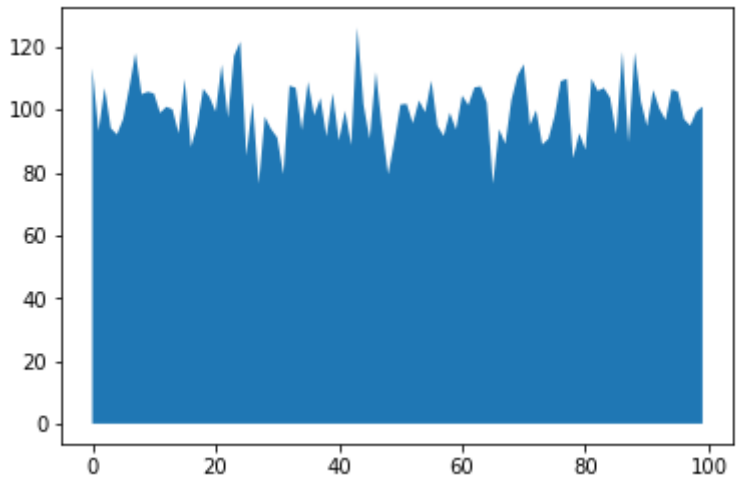


A hydroelectric power plant that constantly supplies the same amount of electricity is practical, but unfortunately not realistic. In reality, there will always be small fluctuations, which we should include in our simulation. For this purpose we use **random numbers**.

For our hydroelectric power plant we assume that the capacity is usually about 100 MW. Small deviations will be more frequent, large deviations less frequent. We are therefore dealing with a so-called **normal or Gaussian distribution**. Gaussian distributed random numbers are generated in Python with the command `random.gauss(mittelwert, standardabweichung)` from the package `random`, which we have to import first. As mean value (= Mittelwert) we use 1 (times 100 MW), as standard deviation (= Standardabweichung) we can use 0.1 (i.e. 10%). In simple terms, this means that the numbers generated with it are on average 10% away from the mean value, and the larger the deviation, the rarer.

In [7]:
```
import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
wasserkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste)
```

Out[7]:
```
[<matplotlib.collections.PolyCollection at 0x1ef1f5cfe10>]
```



In addition to a hydroelectric power plant, we now also want to install a wind power plant in our power grid. The performance of wind power plants is more dependent on chance: when there is no wind, they produce nothing at all, but too strong wind also causes problems. A Gaussian distribution is therefore not meaningful here if we want to illustrate that a wind power plant occasionally does not produce electricity either.

But we can use **equally distributed** random numbers. With equally distributed random numbers, every number is equally (= gleich) probable. The following plot illustrates the difference between Gaussian-distributed and equally distributed random numbers. We use the command `random.uniform(0, 20)` to generate real numbers between 0 and (exclusively) 20. These are

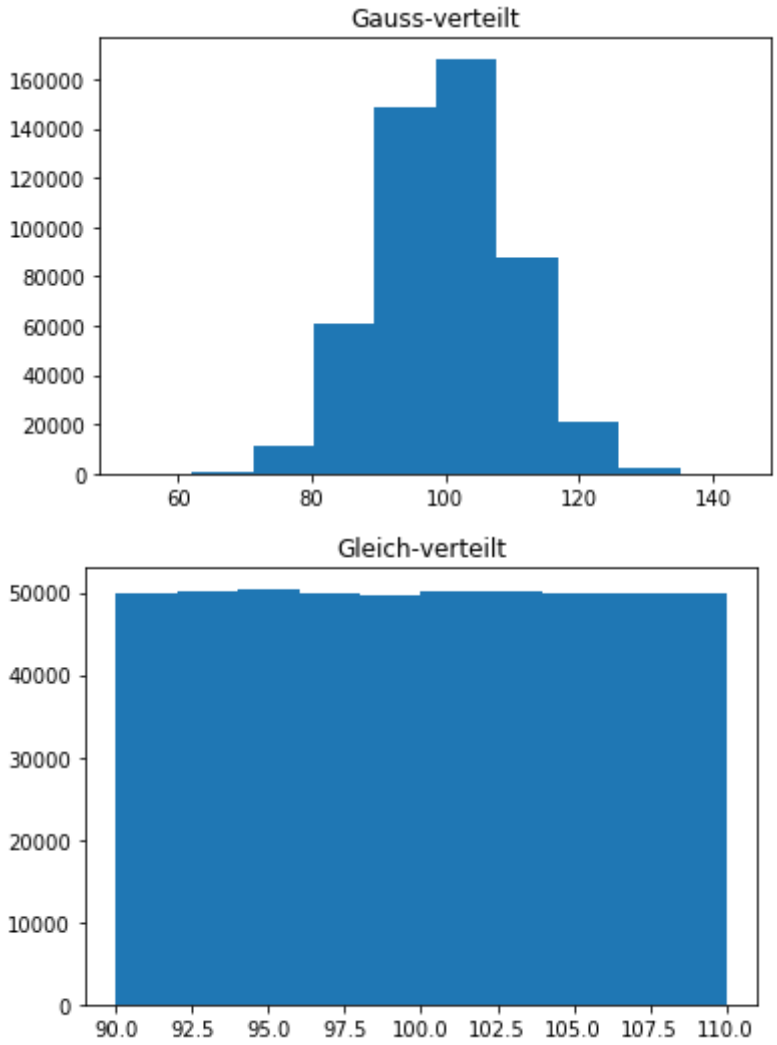
added to 90 to create a distribution that also varies by 100. For plotting, we use the `hist` command here, which draws a histogram of the generated data.

In [21]:

```
gauss=[]
gleich=[]

for it in range(500000):
    gauss.append(random.gauss(100, 10))
    gleich.append(90 + random.uniform(0, 20))

plt.hist(gauss)
plt.title('Gaussian distributed')

plt.figure()
plt.hist(gleich);
plt.title(' Equally distributed')
```

Out[21]:

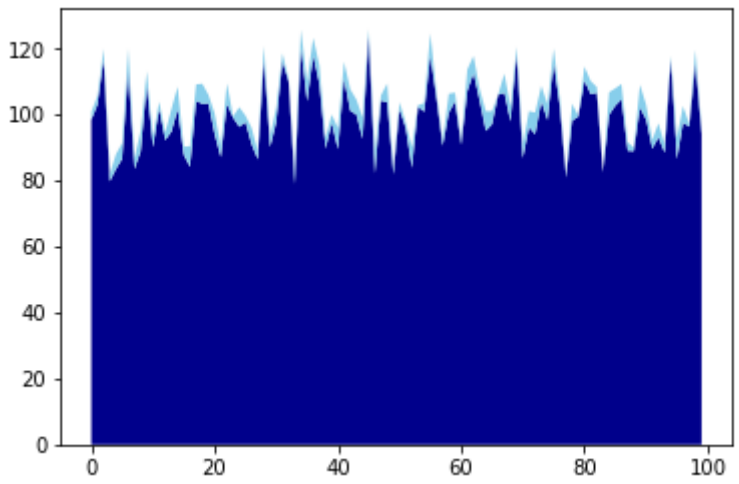
```
<matplotlib.text.Text at 0x178868bbc18>
```



Our wind power plant will have a maximum capacity of 7 MW. However, in case of no wind, it can happen that no electricity is supplied at all. We therefore use a random number between 0 and 7 for the power of the wind power plant, which we generate with the command

`random.uniform()`. Python's **stackplot** provides us with the convenient way to simply add the wind power (=Windkraft) generated in addition to the 100% hydro power graphically to the hydro power. To distinguish between the two types of energy we use two different shades of blue: `darkblue` and `skyblue`.

In [8]:
```
import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = random.uniform(0, 7)
    windkraft_liste.append(windkraft)

# '#cceeff' stands for a slightly lighter blue
plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, c
olors = ('darkblue', 'skyblue'))
```
Out[8]:
```
[<matplotlib.collections.PolyCollection at 0x1ef1f61fb38>,
 <matplotlib.collections.PolyCollection at 0x1ef1f66e860>]
```



We can see that the contribution of a single wind power plant is rather small compared to a hydroelectric power plant. We should therefore expand our model so that we can determine the number of power plants. This will later allow us to determine how many power plants of which type we need.

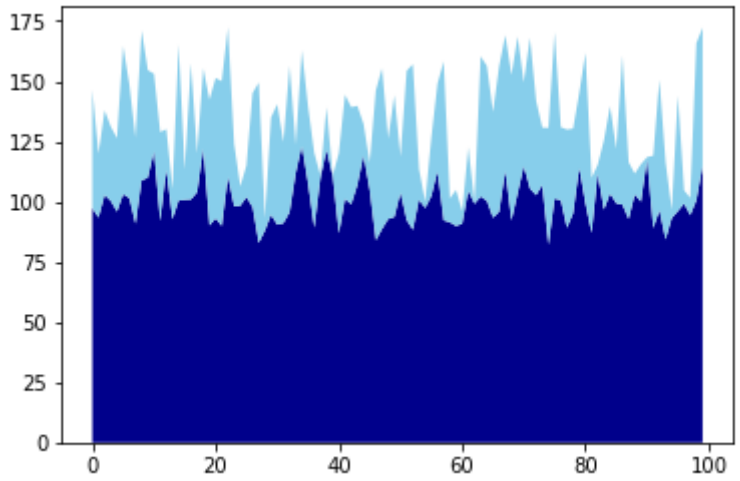
In the following we assume 10 wind power plants and one hydro power plant.

In [9]:
```
import matplotlib.pyplot as plt
import random
%matplotlib inline
```

```
simulationszeit = 100
windkraft_anzahl = 10
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = windkraft_anzahl * random.uniform(0, 7)
    windkraft_liste.append(windkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, c
olors=('darkblue', 'skyblue'))
```

Out[9]:

```
[<matplotlib.collections.PolyCollection at 0x1ef1f6c2828>,
 <matplotlib.collections.PolyCollection at 0x1ef1f705c88>]
```



If we now have as a guideline that at least 150 MW of electricity should be available at all times, then we see that the number of our wind power plants is not sufficient yet:

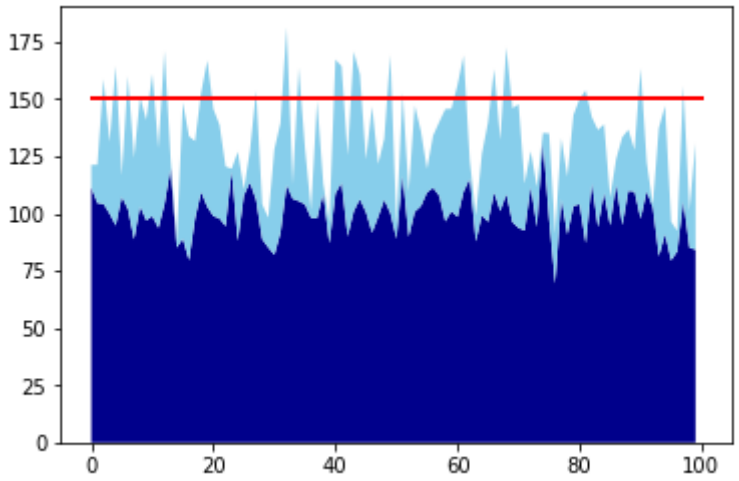
In [10]:

```
import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
windkraft_anzahl = 10
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
```

```
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = windkraft_anzahl * random.uniform(0, 7)
    windkraft_liste.append(windkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, c
olors = ('darkblue', 'skyblue'))
# draw a red line at the height of 150
plt.plot((0, 100),(150, 150), c = "red", lw = 2)
```

Out[10]:

```
[<matplotlib.lines.Line2D at 0x1ef1f7abb70>]
```



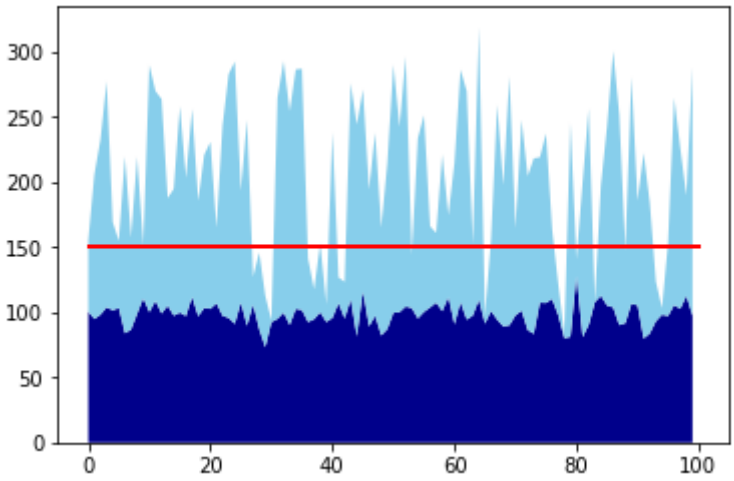
So, we have to increase the number of wind power plants a little more:

In [11]:

```
import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1



wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = windkraft_anzahl * random.uniform(0, 7)
    windkraft_liste.append(windkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, c
olors = ('darkblue', 'skyblue'))
plt.plot((0, 100), (150, 150), c = "red", lw = 2)
```

Out[11]:

```
[<matplotlib.lines.Line2D at 0x1ef1f846eb8>]
```



We can see that even if we plan to have 30 wind power plants, which means that at some times we produce twice as much electricity as we need, there are still times when we do not reach the 150 MW mark. What is the problem?

The problem lies in the specific nature of wind energy. When there is no wind, none of the 30 wind power plants produce electricity. When we created our model, we implicitly assumed that all wind power plants are located in the same region and thus affected by the same wind conditions. In our program code, all power plants use the same random number: So if "no wind" is rolled in one hour, this applies to all power plants.

But we could also assume that the power plants are located so far apart that there are different wind conditions for each power plant. But does that change anything? Won't the average amount of electricity produced be exactly the same?

Let's give it a try: Let's change our program so that each power plant has its own random number to determine how much electricity is currently being produced.

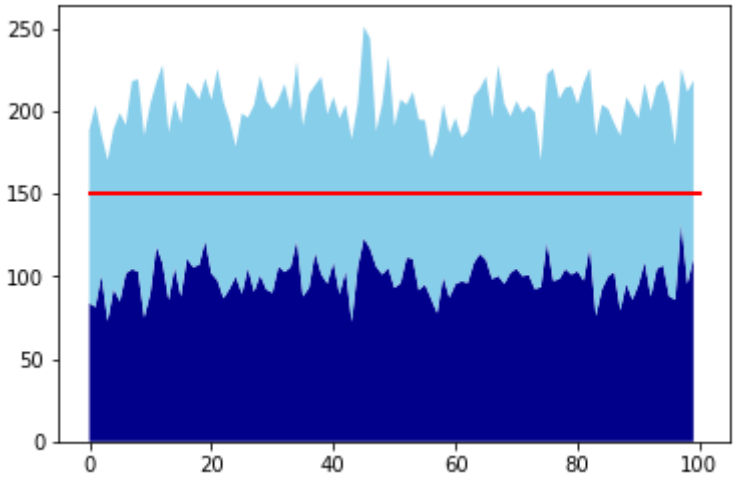
In [12]:

```
import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = 0
    for kraftwerke in range(windkraft_anzahl):
        windkraft = windkraft + random.uniform(0, 7)
    windkraft_liste.append(windkraft)
```

```
plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, c
olors = ('darkblue', 'skyblue'))
plt.plot((0, 100), (150, 150), c = "red", lw = 2)
```

Out[12]:

```
[<matplotlib.lines.Line2D at 0x1ef1f8e6940>]
```



We can see: although on average the same amount of electricity is produced, the assumption that wind power plants are affected by different wind conditions makes a big difference: if we generate a separate random number for each power plant, the result is better averaged. There are fewer peaks and fewer valleys, and electricity production is closer to the average.

This is a basic property of random numbers: If you use a lot of them, the result is always very close to the mean of the possible results.

So, with the model now available, it looks as if we can ensure that 150 MW of electricity is always available. But what happens if we take into account that wind power plants can become defective? How do we build such a random event into our simulation?

Random events with Python

So far we have learned how to create random numbers (=Zufallszahlen). But what if we don't want to generate a number but an event - e.g. a defect in a wind power plant - that occurs with a certain probability? We can combine random numbers with `If-queries` for this purpose.

For this we use the fact that 10% of the numbers from 1 to 100 are less than or equal to 10. And 50% of the numbers are less than or equal to 50, so we could write in code:

In [27]:

```
chance = 50
zufallszahl = random.uniform(0, 100)
if zufallszahl <= chance:
    print("HEAD!")
else:
    print("NUMBER!")
```

```
HEAD!
```

That means, we compare here the percentage chance of a random event with a random number and can therefore be sure that this event happens with exactly this chance.

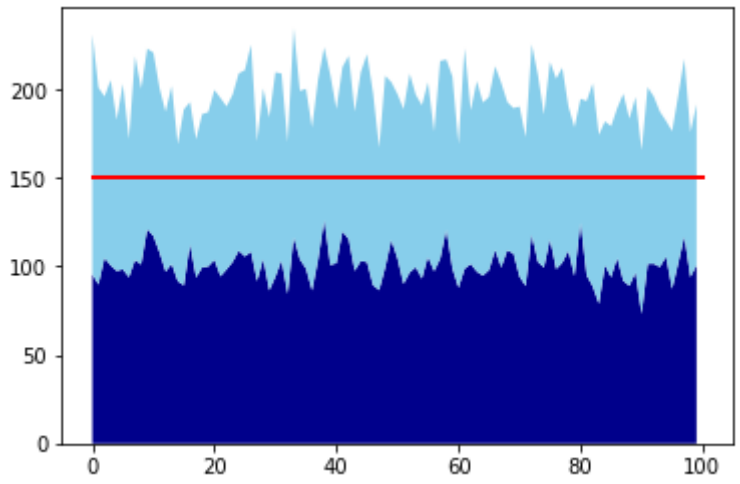
Let's build such an opportunity into our simulation. We assume that every hour there is a 10% chance that a defect in the wind turbine will cause a power failure:

In [13]:

```
import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = 0
    for kraftwerke in range(windkraft_anzahl):
        if random.uniform(0, 100) <= 10:
            # MALFUNCTION: no electricity is produced
            windkraft = windkraft + 0
        else:
            # Normal operation:
            windkraft = windkraft + random.uniform(0, 7)
    windkraft_liste.append(windkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, c
olors = ('darkblue', 'skyblue'))
plt.plot((0, 100), (150, 150), c = "red", lw = 2)
```

Out[13]:

```
[<matplotlib.lines.Line2D at 0x1ef1f98b198>]
```



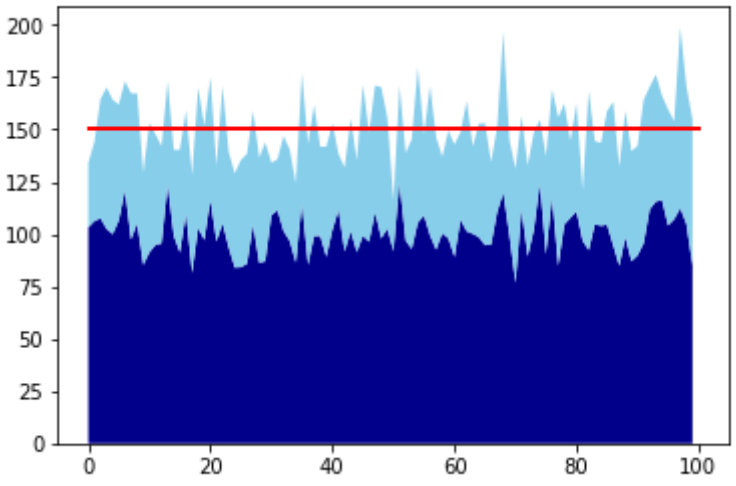
We can see that, although we are sometimes running short of power, we can usually still ensure the power supply. But if we increase the chance of failure to 50%, for example, things look different.

In [14]:

```
import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = 0
    for kraftwerke in range(windkraft_anzahl):
        if random.uniform(0, 100) <= 50:
            # MALFUNCTION: no electricity is produced
            windkraft = windkraft + 0
        else:
            # Normal operation:
            windkraft = windkraft + random.uniform(0, 7)
    windkraft_liste.append(windkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, c
olors = ('darkblue', 'skyblue'))
plt.plot((0, 100), (150, 150), c = "red", lw = 2)
```

Out[14]:

```
[<matplotlib.lines.Line2D at 0x1ef1fa1bcc0>]
```



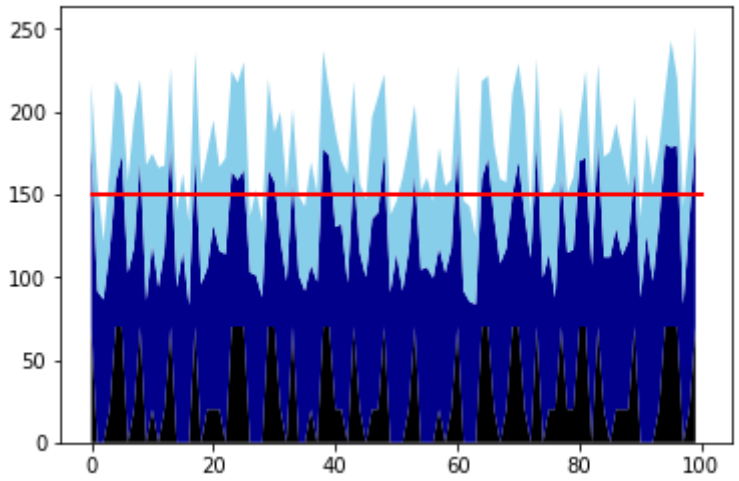
In order to learn a little more about random generators, we want to add another power plant to our power grid in the following. For this purpose, we assume a rather unreliable coal-fired power

plant, which has only three operating modes: "standstill" (0 MW), "normal operation" (20 MW) and "high operation" (70 MW). In our example, these three operating modes occur randomly, no other MW value can be produced.

How would we best implement this? A random number between 0 and 70 would be wrong, 3 consecutive 33% chances would not be right either, because in this case several operating modes would be possible at once. Fortunately, Pyhton offers a very simple solution for such problems: the command `random.choice((a, b, c))` makes a random choice between a, b and c, where a, b and c do not even have to be numbers. Let's use it to build the power plant into our simulation.

In [18]:
```
import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []
kohlekraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = 0
    for kraftwerke in range(windkraft_anzahl):
        if random.uniform(0, 100) <= 50:
            # MALFUNCTION: no electricity is produced
            windkraft = windkraft + 0
        else:
            # normal operation:
            windkraft = windkraft + random.uniform(0, 7)
    windkraft_liste.append(windkraft)

    kohlekraft = random.choice((0, 20, 70))
    kohlekraft_liste.append(kohlekraft)

plt.stackplot(range(simulationszeit), kohlekraft_liste, wasserkraft_liste,
windkraft_liste,
              colors = ('black', 'darkblue','skyblue'))
plt.plot((0, 100), (150, 150), c = "red", lw = 2)
```

Out[18]:
```
[<matplotlib.lines.Line2D at 0x1ef20c74dd8>]
```



Of course, there are still some problems that we cannot present with our present knowledge of random numbers: Let us assume, for example, that there is an hour within the simulation period in which the power grid experiences a particular load. The consumption increases to 200 MW. However, we do not know when exactly this hour will be, we only know that there is such an hour.

If this hour were at the very beginning of the simulation time, it would be a simple problem:

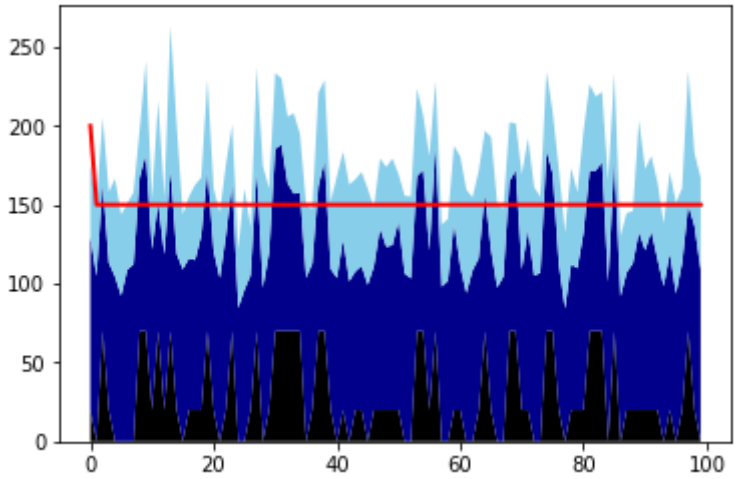
In [17]:

```
import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []
kohlekraft_liste = []
verbrauch_liste = []

for zeit in range(simulationszeit):
    verbrauch_liste.append(150)
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = 0
    for kraftwerke in range(windkraft_anzahl):
        if random.uniform(0, 100) <= 50:
            # MALFUNCTION: no electricity is produced
            windkraft = windkraft + 0
        else:
            # normal operation:
            windkraft = windkraft + random.uniform(0, 7)
    windkraft_liste.append(windkraft)

    kohlekraft= random.choice((0, 20, 70))
```

```
    kohlekraft_liste.append(kohlekraft)

plt.stackplot(range(simulationszeit), kohlekraft_liste, wasserkraft_liste,
windkraft_liste,
              colors = ('black', 'darkblue', 'skyblue'))

# set first element of consumption_list to 200:
verbrauch_liste[0] = 200

plt.plot(verbrauch_liste, c= "red", lw = 2)
```

Out[17]:

```
[<matplotlib.lines.Line2D at 0x1ef20bc7e80>]
```



But what if this excessive demand for energy simply occurs at some point? In this case the Python command `random.shuffle(list)` helps us. This command shuffles the entries of a list. You can not only shuffle playing cards with this command...

Note: Herz means heart, which is a playing card colour

In [32]:

```
karten = ['Herz 2', 'Herz 3', 'Herz 4','Herz 5','Herz 6','Herz 7']
random.shuffle(karten)
print(karten)
```

```
['Herz 3', 'Herz 6', 'Herz 7', 'Herz 2', 'Herz 5', 'Herz 4']
```

... but also solve our above mentioned problem:

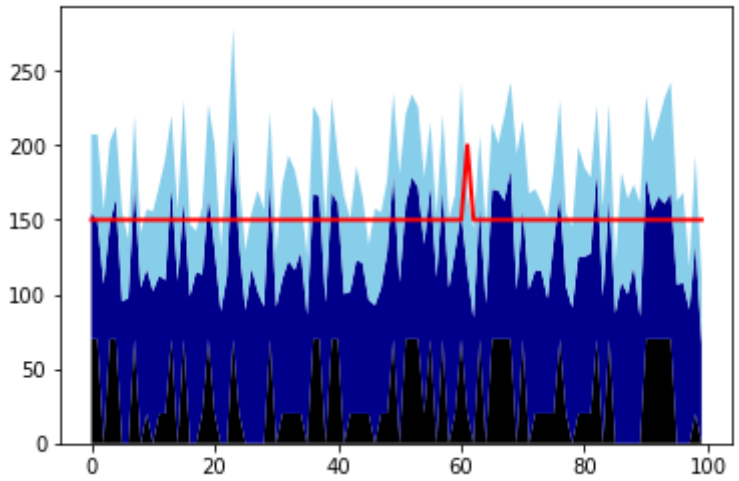
In [19]:

```
import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []
```

```
kohlekraft_liste = []
verbrauch_liste = []

for zeit in range(simulationszeit):
    verbrauch_liste.append(150)
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = 0
    for kraftwerke in range(windkraft_anzahl):
        if random.uniform(0, 100) <= 50:
            # MALFUNCTION: no electricity is produced
            windkraft = windkraft + 0
        else:
            # normal operation:
            windkraft = windkraft + random.uniform(0, 7)
    windkraft_liste.append(windkraft)

    kohlekraft= random.choice((0, 20, 70))
    kohlekraft_liste.append(kohlekraft)

plt.stackplot(range(simulationszeit), kohlekraft_liste, wasserkraft_liste,
windkraft_liste,
              colors=('black','darkblue','skyblue'))

# set first element of consumption_list to 200:
verbrauch_liste[0]=200
# shuffle the entire list
random.shuffle(verbrauch_liste)

plt.plot(verbrauch_liste, c = "red", lw = 2)
```

Out[19]:

```
[<matplotlib.lines.Line2D at 0x1ef20bf3550>]
```



Summary

Random numbers

Random numbers are generated in Python with the package `random`. There are several commands that generate different distributions of random numbers.

Equally distributed (real) random numbers are created with `random.uniform(a, b)`, where `a` is the smallest possible number and `b` marks the upper limit up to which (but excluding the) random numbers are generated. All numbers from `a` to (excluding) `b` are equally likely to occur. Thus, also percentage probabilities can be easily implemented: Since 10% of all numbers between 0 and 100 are smaller than 10, and in equally distributed random numbers all numbers are equally probable, the if query if `if random.uniform(0, 100) < 10` will return `true` in about 10% of cases.

Normally distributed random numbers are created with `random.gauss(m, s)`, where `m` is the mean value of the generated distribution and `s` the standard deviation. The exact range of values is therefore difficult to estimate, even with `random.gauss(100, 10)` it is theoretically possible to get a value smaller than 50. But the chance for this is negligible. The closer the number is to the mean value, the more likely it is to occur. About 70% of the values will be in the interval m ±± s, i.e. in this example between 90 and 110. About 95% will be in the interval m ±± 2s (i.e. 80 to 120) and over 99% in the interval m ±± 3s (i.e. 70 to 130).

Every time a random function is called, a new number is generated. If you run a program with random numbers several times in a row, you will generally get different results.

Random events

Random events are also best created with random numbers: Since x% of all numbers from 1 to 100 are smaller than x, we can simulate any percentage chance with `if random.uniform(0, 100) <= chance.`

Random Choice

`random.choice` selects a random element from several elements. Each element is equally probable and the elements do not necessarily have to be numbers.

Random Shuffle

`random.shuffle` merges a list. The elements remain the same, the position of the elements changes. This is helpful if it is known which value of the list occurs how often, but the exact position of the value should be random.

Histograms

Lists consisting of many numbers can be displayed as histograms. Histograms show us which number ranges are more frequent and which are less frequent. A large bar in a histogram does **not** indicate that the number is large, but that numbers from this range are **frequent**. Histograms are created with `plt.hist(NameDerListe).`

Chapter 8 – Vectors and matrices – The management of a forest

A very central method for handling data and content in systems science is computing with vectors and matrices. Python provides the necessary tools for this purpose. In order to get to know them, the following example deals with the sustainable management of a forest *(=Wald)*. We are especially interested in how the forest (re)grows.

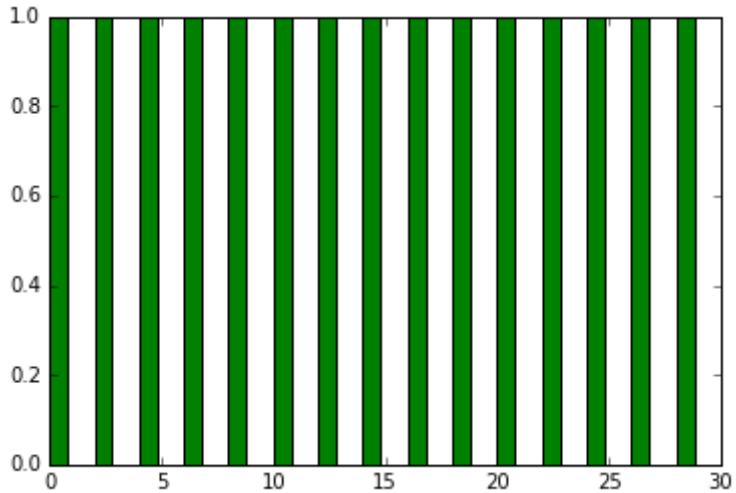
Let's start with a very simple, one-dimensional forest model: we are looking at just one "row" of trees. In our model, trees need space to grow well, so two trees must never be placed on adjacent patches. For this purpose, we create a list with Python, in which we write a one, if there is a tree at a corresponding position, and a zero, if not. For the sake of simplicity, we will first alternate: tree, no tree, tree, no tree,....

In [1]:

```
import matplotlib.pyplot as plt
%matplotlib inline

wald1d = []

for posx in range(15):
    wald1d.append(1)
    wald1d.append(0)

plt.bar(range(len(wald1d)), wald1d, color='green')
```

Out[1]:

```
<Container object of 30 artists>
```



But the forest still looks very "artificial". We could also let the forest grow a little more randomly: We could start on the left and then let a tree grow field by field with a certain probability:

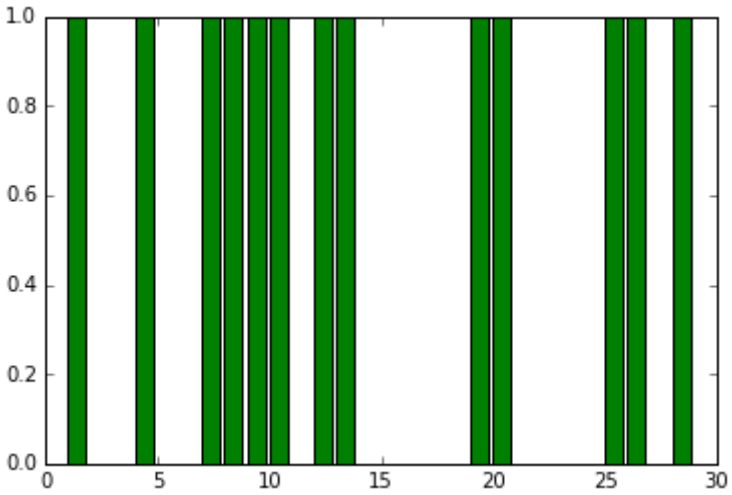
In [2]:

```
import matplotlib.pyplot as plt
# a package for generating and calculating with random numbers
import random
%matplotlib inline
```

```
wald1d = []


for posx in range(30):
    if random.uniform(0, 100) <= 50:
        wald1d.append(1)
    else:
        wald1d.append(0)

plt.bar(range(len(wald1d)), wald1d, color = 'green')
```

Out[2]:

```
<Container object of 30 artists>
```



This is how the forest looks like by chance, but not really natural yet: trees do not like to grow directly next to each other because they make each other shade. How can we incorporate this fact into our forest model? The probability that a tree can grow at a certain position seems to depend on whether or not there is already a tree at the position next to it. For the decision for a tree at position x, we need data for position x - 1.

This is not a problem. We have stored our whole forest in a list, mathematically speaking in a vector. So, we get this information by simply looking at the previous vector element. Let us assume that a tree has only a 20% chance to grow if there is already a tree at the previous position. In contrast, the chance could increase to 90% if the neighboring field is free. We could incorporate this into our Python code as follows:

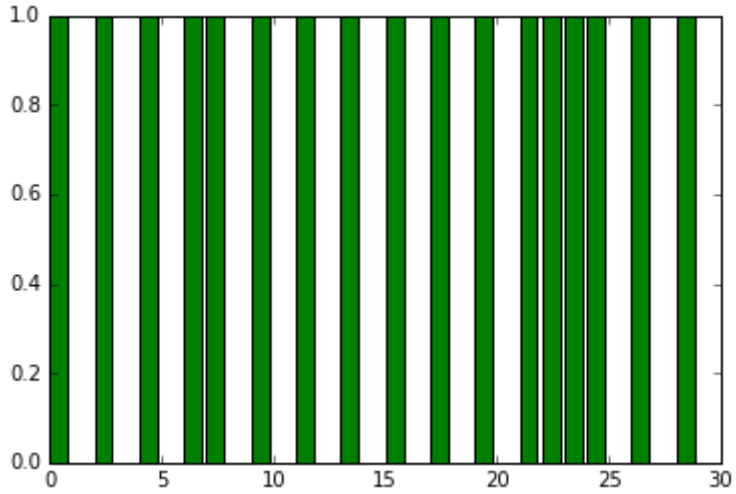
In [3]:

```
import matplotlib.pyplot as plt
import random
%matplotlib inline

wald1d = [1]

for posx in range(1, 30):
    # Note: the == symbol stands for actual equality in most programming
    # languages, while the = symbol stands for an assignment of a value to
    # a variable.
    if wald1d[posx - 1] == 1:
        if random.uniform(0,100) <= 20:
```

```
            wald1d.append(1)
        else:
            wald1d.append(0)
    else:
        if random.uniform(0,100) <= 90:
            wald1d.append(1)
        else:
            wald1d.append(0)
plt.bar(range(len(wald1d)), wald1d, color = 'green')
```

Out[3]:

```
<Container object of 30 artists>
```



So, this would be our one-dimensional forest. Now we want to extend it to two dimensions.

So far, we have stored our "forest" in a list, or mathematically in a vector. In mathematics, a **vector** is a scheme of numbers underneath each other. However, a whole forest usually consists of several rows of trees. To represent it, we need a separate vector for each row of trees. It is simpler and clearer if we write these rows of numbers next to each other. This creates a two-dimensional number scheme, a so-called **matrix**:

A 4x4 forest in vector notation:

$$\vec{v_1} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad \vec{v_2} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} \quad \vec{v_3} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \quad \vec{v_4} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

The same 4x4 forest in matrix notation:

$$\vec{v_1} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

A matrix is basically nothing else than a list of lists. With the Python package `numpy` we get access to numerous "tools" to facilitate working with matrices. For example, creating an empty 4x4 matrix would look like this without `numpy` :

```
wald2d = [[0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]]
```

which would become increasingly confusing for larger matrices. It is easier with `numpy`:

```
wald2d = np.zeros([4,4])
```

Reading and modifying matrix entries works similar to reading and modifying vector, i.e. list elements: The first entry of a vector (a list) is the entry `vektorname[0]`. In a matrix, the first entry in the upper left corner is `matrixname[0][0]`.

Let us now apply this knowledge about matrices to our forest model: First let's create an empty forest, i.e. a 10x10 matrix filled with zeros.

In the next step, a random number (either 1 or 0) should be placed at each position. `int(random.uniform(0, 2)` generates us exactly such a number (actually a number between 0 and 2, which is then rounded down). But how do we do this for each entry of the matrix?

The command is always the same, but the target of the command changes continuously, namely from element `[0][0]`, to `[0][1]`, to `[0][2]` and so on. So to fill all matrix positions with random numbers, it makes sense to use a so-called **double loop**.

A double loop is a loop within a loop. The inner loop executes the desired command for a whole row of the matrix, the outer loop ensures that the inner loop is executed for each row. It is important that the loop variables (`it`) must not have the same name.

When the forest is finished, we output it to the screen with a simple print command.

In [1]:

```
import matplotlib.pyplot as plt
import random
import numpy as np
%matplotlib inline

# create an 'empty', that is, a matrix filled with zeros
wald2d=np.zeros([10,10])

# first loop for the different rows
for it in range(10):
    # second loop for the elements in the rows
    for jt in range(10):
        wald2d[it][jt] = int(random.uniform(0, 2))

print(wald2d)
```

```
[[1. 1. 1. 0. 1. 1. 0. 1. 0. 1.]
 [1. 0. 0. 1. 0. 0. 0. 1. 0. 0.]
 [1. 1. 0. 0. 0. 1. 1. 1. 1. 0.]
 [1. 0. 1. 1. 1. 1. 1. 0. 1. 0.]
 [0. 1. 0. 0. 1. 0. 0. 0. 1. 1.]
 [1. 1. 0. 1. 0. 0. 0. 1. 0. 1.]
 [1. 1. 0. 1. 1. 0. 0. 0. 1. 0.]
```

```
 [1. 0. 1. 1. 0. 1. 1. 1. 1. 1.]
 [0. 0. 0. 1. 0. 0. 1. 0. 1. 0.]
 [1. 1. 1. 0. 1. 0. 0. 1. 0. 1.]]
```

One can already imagine a little bit how the forest could look like. But a graphical output would be better. One command we can use to plot matrices is called `plt.matshow()`.

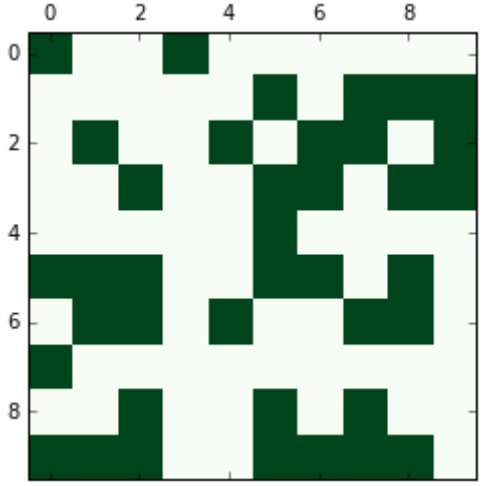
This allows us to specify which matrix is to be plotted on the one hand, and in which color on the other. The color representation in this kind of plots allows to design colors depending on the size of a matrix entry. For this purpose, a color gradient is defined. Although we have only zeros and ones as entries for the time being, we choose a gradient that goes from white to green. It is called `plt.cm.Greens`.

In [5]:

```
import matplotlib.pyplot as plt
import random
import numpy as np
%matplotlib inline

wald2d = np.zeros([10, 10])

for it in range(10):
    for jt in range(10):
        wald2d[it][jt] = int(random.uniform(0, 2))

plt.matshow(wald2d, cmap = plt.cm.Greens)
```

Out[5]:

```
<matplotlib.image.AxesImage at 0xb7c72b0>
```



Up to now we only consider the values 0 and 1 (no tree, tree) to represent our forest. But if we had for example also data about the size or the density of the trees in our forest, we could represent this with values between 0 and 1. Thanks to the gradient, we don't have to change anything else in our program for the representation.

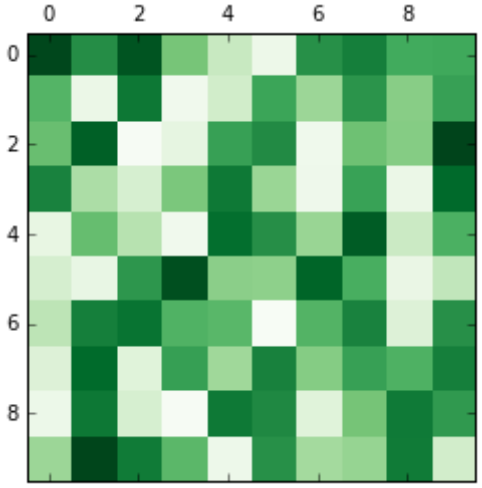
In [6]:

```
import matplotlib.pyplot as plt
import random
import numpy as np
%matplotlib inline
```

```
wald2d=np.zeros([10, 10])

for it in range(10):
    for jt in range(10):
        wald2d[it][jt] = random.uniform(0, 1)

plt.matshow(wald2d, cmap = plt.cm.Greens)
```

Out[6]:

```
<matplotlib.image.AxesImage at 0xb960b70>
```



In this plot we now see darker and lighter areas of the forest. Based on this plot, we could consider in which areas it would make sense to remove wood from the forest. It would be practical to clear the particularly dark areas of the forest by cutting down trees so that new trees can grow there again.

But how do we reliably find the dark areas of the forest? Simply looking for a large number in the matrix is not enough: It could be that we cut down a single large tree that has no neighbors at all. So, we have to make sure in our search that there are also large trees in the neighboring areas.

For this we first have to define a limit, from which we classify a tree as "big". In this example we assume the value 0.75. After we have generated the forest, we iterate in another double loop through all positions of the forest and check if a tree is "big" at this position. Then we additionally check all four neighbors at this position. If there is also a big tree at one of these neighboring places, the original, i.e. the middle tree is cut down.

In our matrix representation the four neighbors of the tree are `wald2d[it][jt]`

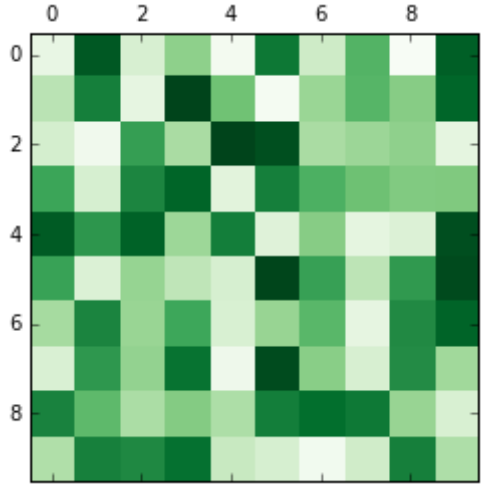
- `wald2d[it+1][jt]`
- `wald2d[it-1][jt]`
- `wald2d[it][jt+1]`
- `wald2d[it][jt-1]`

To view the forest before and after this thinning, we create two plots. We use the function `plt.show()`, which draws the plot before the rest of the program is executed.

In [7]:

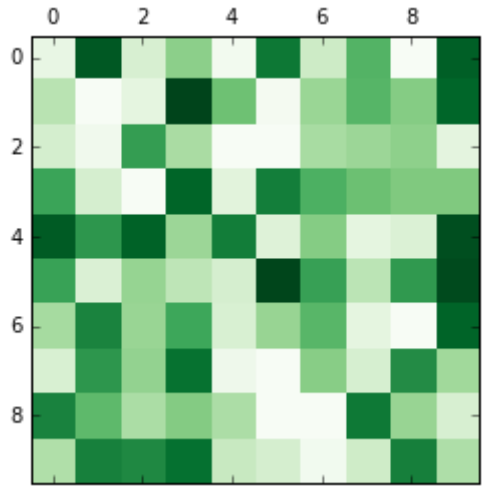
```
import matplotlib.pyplot as plt
import random
import numpy as np
```

```
%matplotlib inline

wald2d = np.zeros([10, 10])

# Forest is generated
for it in range(10):
    for jt in range(10):
        wald2d[it][jt] = random.uniform(0, 1)

# first presentation
plt.matshow(wald2d, cmap = plt.cm.Greens)
plt.show()

# Forest is thinned out
for it in range(1, 9):
    for jt in range(1, 9):
        if wald2d[it][jt] > 0.75:
            if wald2d[it+1][jt] > 0.75:
                wald2d[it][jt] = 0
            if wald2d[it-1][jt] > 0.75:
                wald2d[it][jt] = 0
            if wald2d[it][jt+1] > 0.75:
                wald2d[it][jt] = 0
            if wald2d[it][jt-1] > 0.75:
                wald2d[it][jt] = 0

# second representation
plt.matshow(wald2d, cmap = plt.cm.Greens)
```



Out[7]:

```
<matplotlib.image.AxesImage at 0xc4c04e0>
```



To see in these plots where exactly the trees were cut down, you have to look very closely. Is it easier to find the places where the trees were felled? What we are looking for is a method to show the difference between the first and second plot.

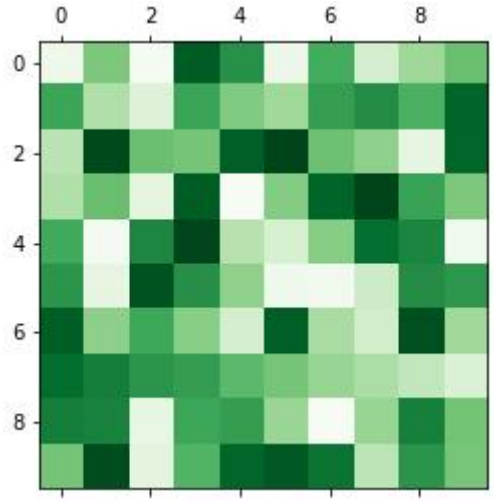
Differences can be expressed mathematically. Calculating with matrices offers the advantage of being able to subtract one matrix from another easily. We build this possibility into the program and create a plot that tells us where trees have been felled.

For this we have to cache the forest generated at first under an own name, so that we do not overwrite it when felling.

In [2]:

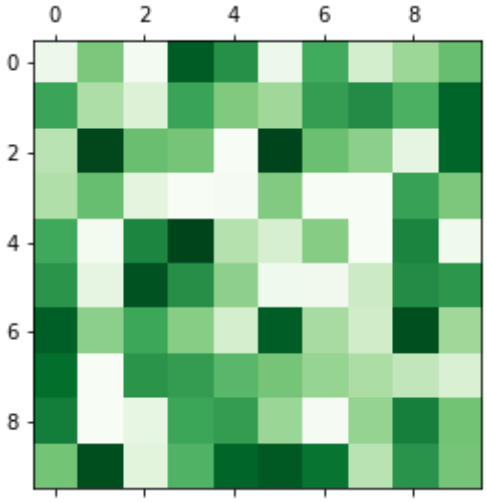
```
import matplotlib.pyplot as plt
import random
import numpy as np
%matplotlib inline

wald2d = np.zeros([10, 10])
for it in range(10):
    for jt in range(10):
        wald2d[it][jt] = random.uniform(0, 1)

# first representation, the generated forest
plt.matshow(wald2d, cmap = plt.cm.Greens)
plt.show()

# Cache
wald2dalt = np.array(wald2d)

for it in range(1, 9):
    for jt in range(1, 9):
        if wald2d[it][jt] > 0.75:
            if wald2d[it+1][jt] > 0.75:
                wald2d[it][jt] = 0
            if wald2d[it-1][jt] > 0.75:
                wald2d[it][jt] = 0
            if wald2d[it][jt+1] > 0.75:
```

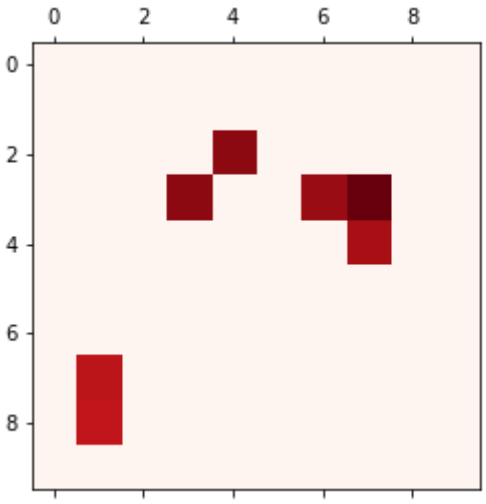
```
                wald2d[it][jt] = 0
            if wald2d[it][jt-1] > 0.75:
                wald2d[it][jt] = 0

# second representation, the cleared forest
plt.matshow(wald2d, cmap = plt.cm.Greens)

# third representation, the difference, this time in red tones
plt.matshow(wald2dalt - wald2d, cmap = plt.cm.Reds)
```



Out[2]:

```
<matplotlib.image.AxesImage at 0x218ebe3e5c0>
```





We could also try to find the biggest tree in the forest. For this we have to use a double loop again and go through all trees. We always remember the size of the biggest tree we have already encountered. If the current tree is bigger than what we have, it replaces the one that so far was biggest.

In [13]:

```
import matplotlib.pyplot as plt
import random
import numpy as np
%matplotlib inline

wald2d = np.zeros([10, 10])
for it in range(10):
    for jt in range(10):
        wald2d[it][jt] = random.uniform(0, 1)

maximum = 0
# We set the value for the largest tree to 0
# the first tree is guaranteed to be bigger than 0, it overwrites this value

for it in range(10):
    for jt in range(10):
    # Unlike the previous one, this loop now runs over all entries,
    # including the borders
        if wald2d[it][jt] > maximum:
            maximum = wald2d[it][jt]

# After the double loop we output the final value of the largest tree:

print("Biggest tree:")
print(maximum)
```

```
Biggest tree:
0.995210757744049
```

Summary

Vectors

Vectors are something like lists, whose entries are usually written over each other in mathematics.

$$\vec{v_1} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Vectors can be iterated as well as lists. Note that the first element (the first entry) in a vector is counted as 'zero' entry (`vectorname[0]`). Thus, the second entry has the name `vectorname[1]`, and so on.

Matrices

Matrices are vectors of vectors, or on the computer: lists of lists.

$$\vec{v_1} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Matrix entries are addressed mit `matrixname[0][0]`, `matrixname[0][1]`, ... `matrixname[n][m]` . A double loop can be used to iterate through a matrix.

A number of basic arithmetic operations (addition, subtraction, multiplication ...) can be applied to matrices.

Random

To generate random numbers the Python package (or module) `random` can be used.

Chapter 9 - Functions

In this chapter we will look at recycling raw materials and learn about functions in Python.
Let's assume there is a production process that requires a certain raw material. This raw material can be purchased in a purity of 90%. After the production process, the raw material can be recov ered and theoretically reused, but it loses a quarter of its purity (=`reinheit`). Mathematically, we can express it like this:

In [3]:

```
# the purity of the raw material equals 0.9 in the beginning
reinheit = 0.9
# the purity of the product equals ¾ of the initial purity
reinheit_neu = reinheit * 0.75
print(reinheit_neu)
```

```
0.675
```

The raw material now has a purity of only 67.5%. If we continue to use it in this way, it would very quickly reach a purity at which the process would no longer work. But what if we mix it with new raw material?

In [4]:

```
reinheit = 0.9
reinheit_rec = reinheit * 0.75

reinheit_neu = 0.5 * (reinheit + reinheit_rec) #Mittelwert
print(reinheit_neu)
```

```
0.7875000000000001
```

Now we would have a purity of over 78% after the first reuse. It would now be interesting to find out what happens if we keep repeating these steps, i.e. reusing and remixing the raw material. How quickly does the purity drop? Does it reach 0 at some point?

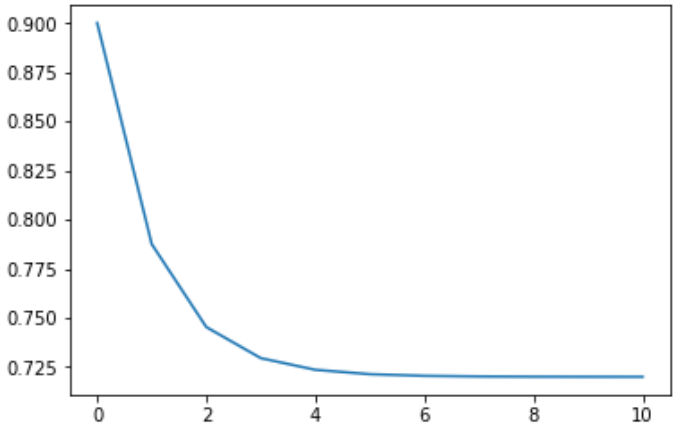
To find this out we can use a list and a for loop:

In [10]:

```
import matplotlib.pyplot as plt
%matplotlib inline

reinheit = 0.9
reinheit_liste = [reinheit] # a list with one entry (purity) is created

for it in range(10):
    reinheit = 0.5 * (0.9 + reinheit * 0.75)
    reinheit_liste.append(reinheit)


plt.plot(reinheit_liste)
```

Out[10]:

```
[<matplotlib.lines.Line2D at 0x1dd220b8b00>]
```



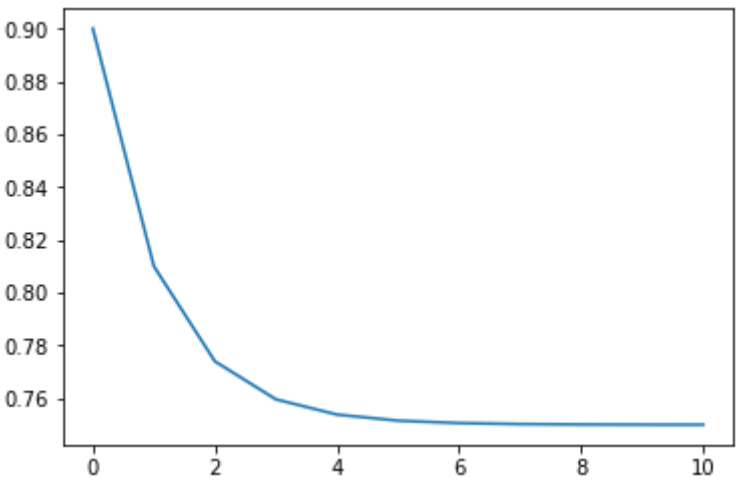
It looks like the purity would never fall below a certain value. Apparently, a balance is reached. No matter how often we repeat the process, we never get below 70 percent. Does this have anything to do with the fact that the process reduces the purity by 25%? What effect would a different parameter value have? We could easily check this by simply changing the parameter from, for example, to 0.8:

In [12]:

```
import matplotlib.pyplot as plt
%matplotlib inline

reinheit = 0.9
reinheit_liste = [reinheit]

for it in range(10):
    reinheit = 0.5 * (0.9 + reinheit * 0.8)
    reinheit_liste.append(reinheit)


plt.plot(reinheit_liste)
```

Out[12]:

```
[<matplotlib.lines.Line2D at 0x1dd221e6dd8>]
```



In fact the fixed point changes. We now want to compare several parameter values in one plot. The easiest solution would be to copy the research loop often and change the parameter each

time. We could also put the program into a For loop, but this makes changing the parameter very cumbersome. Fortunately, there is a better solution: we use **functions**.

Functions

Functions (not to be confused with mathematical functions) are programming structures that automatically execute more complicated processes. Similar to mathematical functions they can have an input and an output. We can give them a name and thus call them in a space-saving and clear way.

Functions are recommended whenever a process is needed at different places in the code. Functions are declared in Python with the word **def**. You write `def nameofthefunkction():`, the function itself is then indented. Now let's rebuild our recycling process into a function.

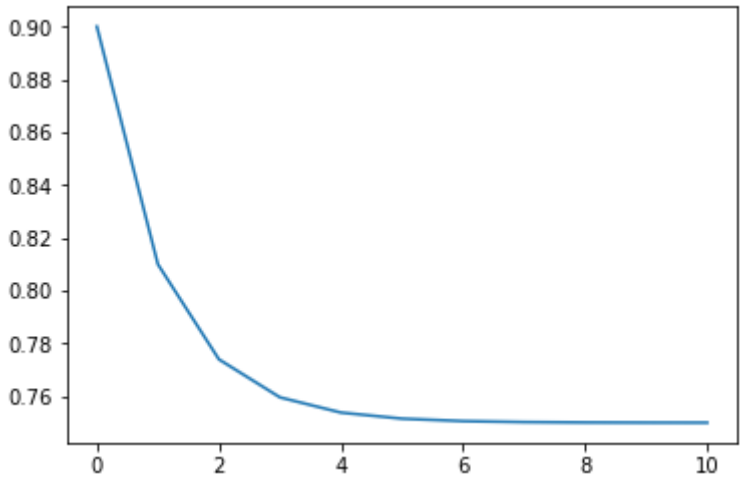
In [13]:

```
import matplotlib.pyplot as plt
%matplotlib inline


# a function (recyceln) is defined
def recyceln():
    reinheit = 0.9
    reinheit_liste = [reinheit]

    for it in range(10):
        reinheit = 0.5 * (0.9 + reinheit * 0.8)
        reinheit_liste.append(reinheit)


    plt.plot(reinheit_liste)
```

Please note: The function is not executed yet just by defining it. Therefore, we have to call it explicitely. This is done with the name of the function followed by round brackets:

In [14]:

```
import matplotlib.pyplot as plt
%matplotlib inline

def recyceln():
    reinheit = 0.9
    reinheit_liste = [reinheit]

    for it in range(10):
        reinheit = 0.5 * (0.9 + reinheit * 0.8)
        reinheit_liste.append(reinheit)


    plt.plot(reinheit_liste)

recyceln()
```



What did we actually gain from this function? An important property of functions is, that they can have an input value. Currently our function always uses a parameter of 0.8 to calculate how much purity is lost per step. We can rewrite this parameter to use it as an input parameter. Input parameters are implemented as follows:

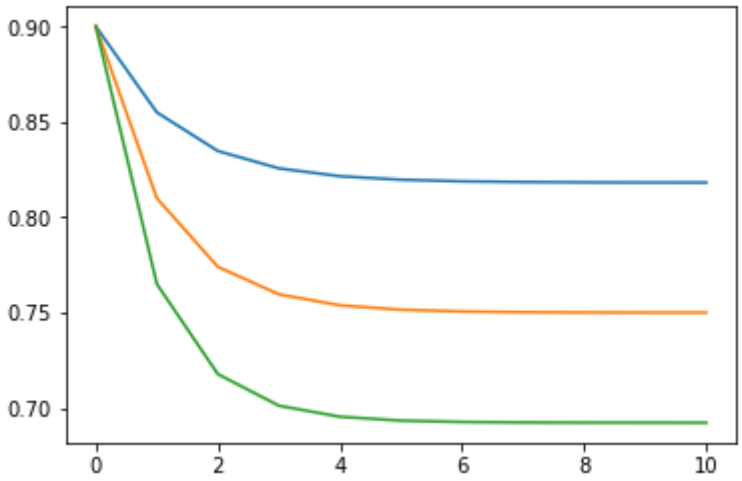
When defining the function, the name of the parameter is put in the round brackets. This name can then be used within the function definition.

When the function is then **called**, you specify the **value** that the parameter should have.

So for our recycling function it would look like this:

In [1]:

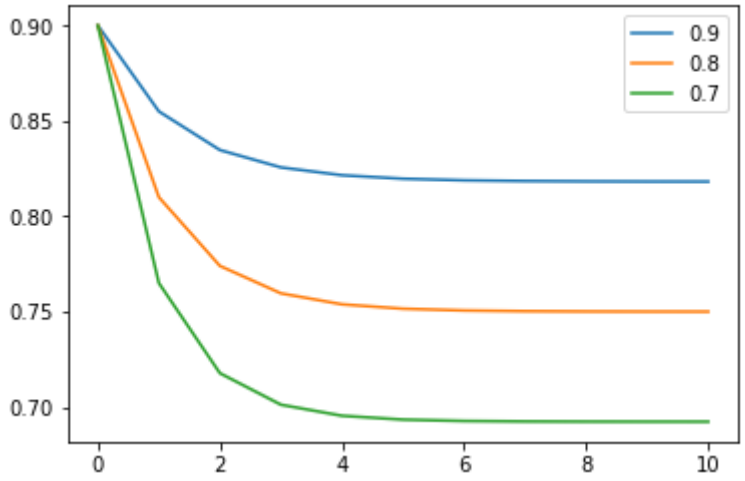
```
import matplotlib.pyplot as plt
%matplotlib inline

def recyceln(verlustparameter):
    reinheit = 0.9
    reinheit_liste = [reinheit]

    for it in range(10):
        reinheit = 0.5 * (0.9 + reinheit * verlustparameter)
        reinheit_liste.append(reinheit)


    plt.plot(reinheit_liste)

recyceln(0.9)
recyceln(0.8)
recyceln(0.7)
```



This allows us to compare different loss parameters very elegantly. In the plot, however, it would be nice if we label the lines so that we know which line belongs to which parameter value. We can easily adjust functions: If we change something in the definition, this change is valid for all calls.

In [20]:

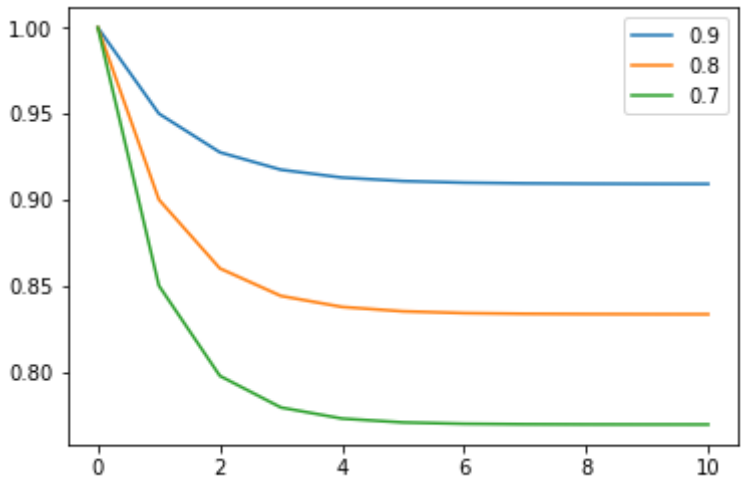
```
import matplotlib.pyplot as plt
%matplotlib inline

def recyceln(verlustparameter):
    reinheit = 0.9
    reinheit_liste = [reinheit]

    for it in range(10):
        reinheit = 0.5 * (0.9 + reinheit * verlustparameter)
        reinheit_liste.append(reinheit)


    plt.plot(reinheit_liste, label = verlustparameter) # we add a label
plt.legend() # the labels should be displayed in a legend
recyceln(0.9)
recyceln(0.8)
recyceln(0.7)
```



Functions can also be expanded as desired. For example, we can add a second input parameter. Further input parameters are simply separated with commas. Let's add the parameter, which tells us, that this is the fresh resource, as a parameter.

In [24]:

```
import matplotlib.pyplot as plt
%matplotlib inline

def recyceln(verlustparameter, startreinheit):
    reinheit = startreinheit
    reinheit_liste = [reinheit]

    for it in range(10):
        reinheit = 0.5 * (startreinheit + reinheit * verlustparameter)
        reinheit_liste.append(reinheit)


    plt.plot(reinheit_liste, label = verlustparameter)
    plt.legend()

recyceln(0.9,1.0)
recyceln(0.8,1.0)
recyceln(0.7,1.0)
```



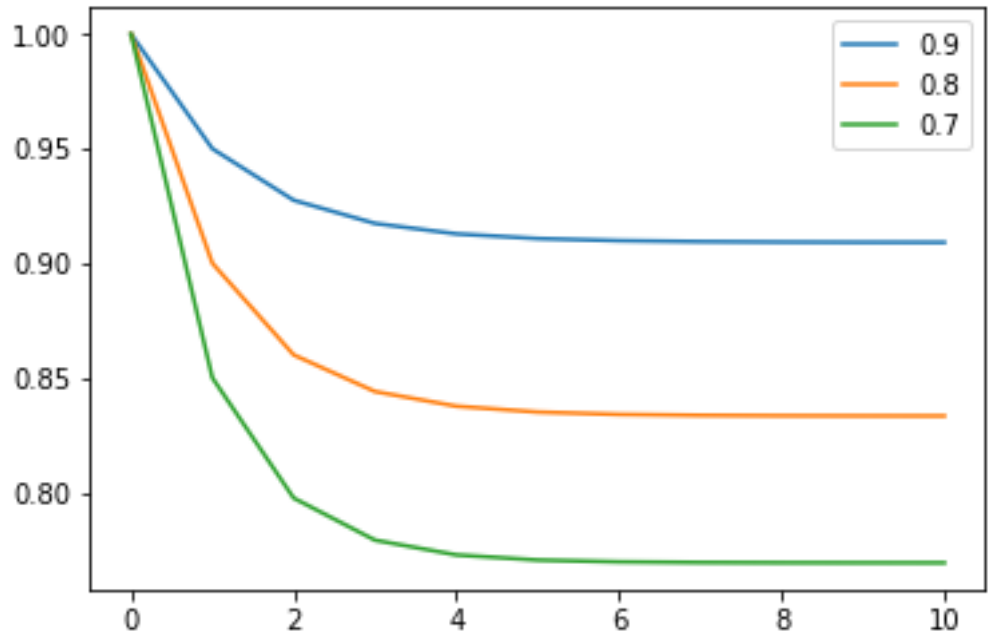
Additionally, to the input value a function can have an output value. This is the result of the function, which is the information that the function shall return to the main program. In our case this would be for example the last purity value of our time series. The output value is defined by the word **return** at the end of the definition. Whenever we apply a function (for example `print`) to another function, we actually apply it to the output value.

In [26]:

```
import matplotlib.pyplot as plt
%matplotlib inline

def recyceln(verlustparameter, startreinheit):
    reinheit = startreinheit
    reinheit_liste = [reinheit]

    for it in range(10):
```

```
        reinheit = 0.5 * (startreinheit + reinheit * verlustparameter)
        reinheit_liste.append(reinheit)


    plt.plot(reinheit_liste, label = verlustparameter)
    plt.legend()
    # the element with the index -1 is the last element of the list,
    # no matter how long this list is
    return reinheit_liste[-1]

print(recyceln(0.9,1.0))
print(recyceln(0.8,1.0))
print(recyceln(0.7,1.0))
```

```
0.9091218642081055
0.8333508096
0.7692371351092773
```



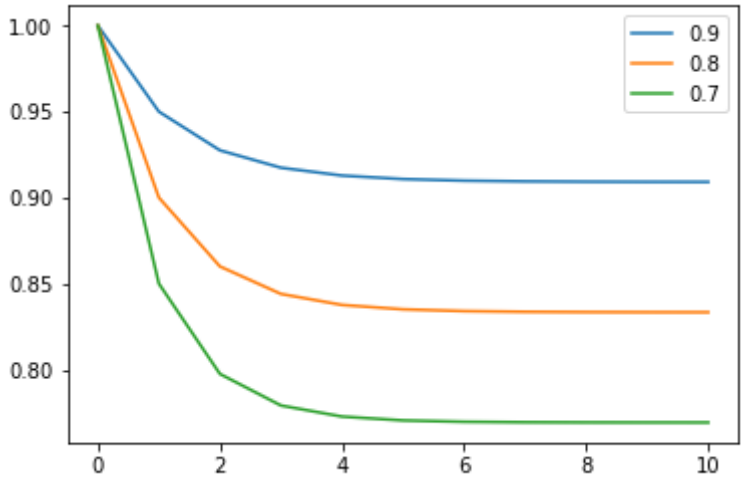
It is also possible to use functions within functions. We can outsource the mixing of raw materials into a separate Function:

In [1]:

```
import matplotlib.pyplot as plt
%matplotlib inline

def mischen(aktuell,start,verlust): # definition of a new function
    neu = 0.5 * (start + aktuell * verlust)
    return neu


def recyceln(verlustparameter, startreinheit):
    reinheit = startreinheit
    reinheit_liste = [reinheit]
```

```
    for it in range(10):
        reinheit = mischen(reinheit,startreinheit,verlustparameter)
        reinheit_liste.append(reinheit)


    plt.plot(reinheit_liste, label = verlustparameter)
    plt.legend()

    return reinheit_liste[-1]

print(recyceln(0.9,1.0))
print(recyceln(0.8,1.0))
print(recyceln(0.7,1.0))
```

```
0.9091218642081055
0.8333508096
0.7692371351092773
```



Now let's analyze another recycling process. We produce glass bottles, which are then used. During use, a total of 10 milligrams of harmful substances are deposited in the glass. The bottles are melted down and new bottles are made from them. During this process 15% of the harmful substances are removed. How long does it take until more than 50 milligrams are in a bottle?

Let us try to solve this problem with Functions. Let's first write a function that describes the contamination process and a function that describes the melting process.

Then we can put these processes in a for-loop and plot what happens.

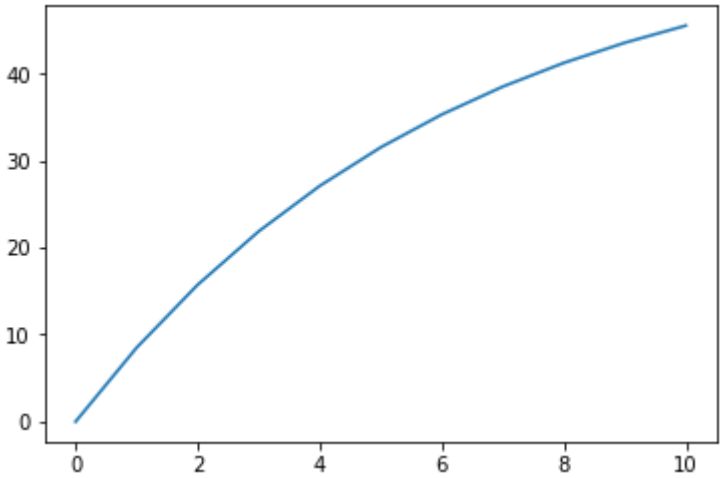
In [2]:

```
import matplotlib.pyplot as plt
%matplotlib inline

# verschmutzen = pollute
def verschmutzen(startwert):
    return startwert + 10

# schmelzen = melt
def schmelzen(startwert):
    return startwert * 0.85
```

```
schadstoff = 0      # schadstoff = pollutant
schadstoff_liste = [schadstoff]

for it in range(10):
    schadstoff = verschmutzen(schadstoff)
    schadstoff = schmelzen(schadstoff)
    schadstoff_liste.append(schadstoff)

plt.plot(schadstoff_liste)
```

Out[2]:

```
[<matplotlib.lines.Line2D at 0x2066550ed68>]
```



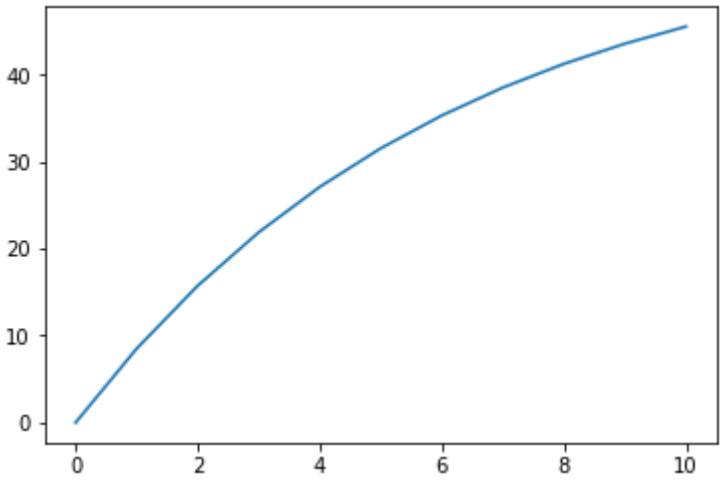
If we nest the functions, it becomes much more compact. The functions are processed from inside to outside:

In [7]:

```
import matplotlib.pyplot as plt
%matplotlib inline

def verschmutzen(startwert):
    return startwert + 10

def schmelzen(startwert):
    return startwert * 0.85

schadstoff = 0
schadstoff_liste = [schadstoff]

for it in range(10):
    schadstoff = schmelzen(verschmutzen(schadstoff))
    schadstoff_liste.append(schadstoff)

plt.plot(schadstoff_liste)
```

Out[7]:

```
[<matplotlib.lines.Line2D at 0x2674436fe80>]
```



So we see: after 10 steps the limit value is not yet exceeded. But how can we now find out when exactly this value is exceeded for the first time? We could simply let the For-loop run longer and keep trying until we find a suitable value. But there is a better solution: Whenever we don't know how often a loop should be run, but only **under which condition it should be stopped** (in our case when 50 mg of pollutant is reached), we can use a **While-loop**.

The While-Loop

With the While-loop, we do not specify a number of loop passes, but a condition. As long as this condition is fulfilled the loop will continue. You use While-loops with

```
while condition:

      instructions
```

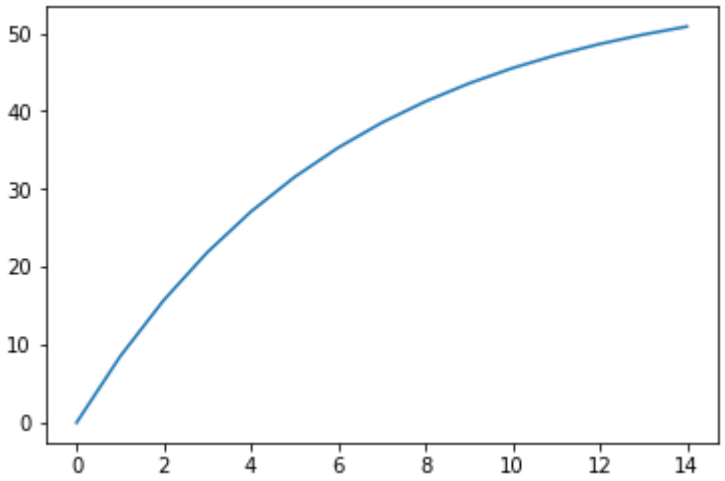
In our example:

In [10]:

```
import matplotlib.pyplot as plt
%matplotlib inline

def verschmutzen(startwert): # startwert = start value
    return startwert + 10

def schmelzen(startwert):
    return startwert * 0.85

schadstoff = 0
schadstoff_liste = [schadstoff]

while schadstoff < 50:
    schadstoff = schmelzen(verschmutzen(schadstoff))
    schadstoff_liste.append(schadstoff)

plt.plot(schadstoff_liste)
```

Out[10]:

```
[<matplotlib.lines.Line2D at 0x267444a3a20>]
```

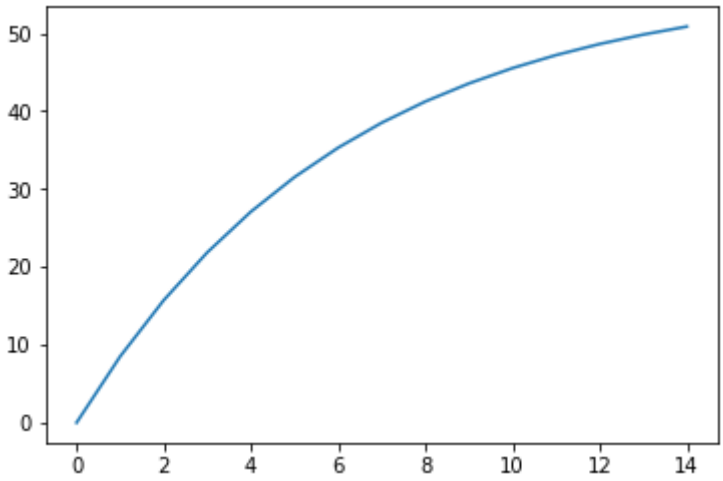


While-loops have no built-in variable that counts how many times the loop has been run through. But if we need something like that (like here to find the number of reuses) we can easily add it:

In [5]:

```
import matplotlib.pyplot as plt
%matplotlib inline

def verschmutzen(startwert):
    return startwert + 10

def schmelzen(startwert):
    return startwert * 0.85

schadstoff = 0
schadstoff_liste = [schadstoff]
counter = 0

while schadstoff < 50:
    schadstoff = schmelzen(verschmutzen(schadstoff))
    schadstoff_liste.append(schadstoff)
    counter = counter + 1

plt.plot(schadstoff_liste)

print("After")
print(counter)
print("Processes the pollutant value the pollutant limit value is exceeded.
")
```

```
After
14
Processes the pollutant limit value is exceeded.
```



Be careful with While-loops: If the condition of the While-loop is badly chosen, it may remain true forever. Then the While-loop runs on infinitely long.

So you should use while-loops only if you are sure that the condition will be false one time. If we don't know what will happen to our system we should avoid While loops and use For loops.

Summary

Functions

With functions we can define processes that we need often once and then reuse them many times. We can build our own Python commands, so to speak. Functions can have **input values**, so you can adjust them. There are also **output values** that are passed back to the main program. Other variables, which are created inside the function, exist only inside the function and are not callable by the main program.

Fuctions are defined with

```
def functionname(input1,input2,input3):

      instructions

      return outputvariable
```

While-Loops

While-loops, similar to For-loops, are used to **repeat** a statement many times. With the For-loop, you must specify how often it must be run through. The While-loop, on the other hand, runs until a **condition** that must be determined **is no longer met**.

Thus, using the While-loop may be **risky**: if the condition **always remains true** (due to a mistake in thinking or programming), the loop will never finish and the computer may **crash** if you don't exit the kernel fast enough.

Chapter 10 – Recursions and Iterations

In the last chapter we got to know **Functions**. So we could program our own commands, which can have input and output values.

We also saw that a function can call another function. This could bring us to an interesting question: What happens, if a function **calls itself**? Can it work at all or do we automatically build an endless loop?

In fact, it is possible that we build functions that call themselves. Strictly speaking, this is a **recursion**. A recursion occurs whenever you perform an operation and then apply the same operation to the output of the first operation and so on. Similar to While-loops we have to be careful not to produce endless loops. Functions that are guaranteed to call themselves will always produce endless loops. Functions that call themselves only under certain conditions are suitable for recursion.

Let us now try to program such a recursion. We want to have a function, which counts down a **countdown**, starting from any number. So, if the input is 0 the function shall output the number 0. If it gets another input, it shall on the one hand output this input, on the other hand it shall call itself. The input, with which it calls itself, shall be its own input - 1, so that it counts down.

In [80]:

```
# we define the function countdown
def countdown(number):
    # we want to print the input
    print(number)
    # if the value 0 has not been reached yet
    # the function calls itself
    # with an input decreased by 1

    if number != 0: # if the input equals 0
        countdown(number-1)

# after the definition the function has to be called
# the input can be chosen freely
countdown(10)
```

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

Our countdown function seems to work. It counts down from any number by calling itself again and again. So we do not need a loop. Nevertheless, it can happen, that we build an **infinite loop**, namely if we never get to input 0, because we start the function for example with another input: (the print command is commented out here for safety reasons)

In [5]:

```
def countdown(number):
    # we want to print the input
    # print(number)
    # if the value 0 has not been reached yet
    # the function calls itself
    # with an input decreased by 1

    if number != 0: # if the input equals 0
        countdown(number-1)

# after the definition the function has to be called
# the input can be chosen freely
countdown(10.5)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-5-9b03a94dcc7f> in <module>()
     14 # after the definition the function has to be called
     15 # the input can be chosen randomely ---> 16 countdown(10.5)

<ipython-input-5-9b03a94dcc7f> in countdown(zahl)
      9
     10     if number != 0: # if the input equals 0
---> 11         countdown(number-1)
     12
     13

... last 1 frames repeated, from the frame below ...

<ipython-input-5-9b03a94dcc7f> in countdown(zahl)
      9
     10     if number != 0: # if the input equals 0
---> 11         countdown(number-1)
     12
     13

RecursionError: maximum recursion depth exceeded in comparison
```

After some time, Python returns an error message:

```
RecursionError: maximum recursion depth exceeded
```

In contrast to While-loops, recursions are better secured in Python. If there are too many self-calls, Python recognizes them as endless loops and aborts the calculation before the computer freezes.

But this is only one of the advantages of recursions over loops. Recursions can also perform much more complicated tasks, which are difficult to solve with loops. Let us think back to the Fibonacci sequence, i.e.

0,1,2,3,5,8,13,...

If we now want to know, for example, which is the 20th number in this sequence, we could build a loop that calculates 20 Fibonacci numbers. This can be done more elegantly with a recursion: For this we only need to know the following three things.

- The first Fibonacci number is 0.
- The second Fibonacci number is 1.
- The n^{th} Fibonacci number can be calculated by adding the $(n-1)^{th}$ to the $(n-2)^{th}$ Fibonacci number .

Written as a recursion:

In [78]:

```
def fibonacci(n):
    ret = 0 # we define a value and set it to zero

    # we know the first Fibonacci number
    if n == 0:
        ret = 0

    # we know the second Fibonacci number
    if n == 1:
        ret = 1

    # all other Fibonacci numbers can be calculated recursively
    if n > 1:
        ret = fibonacci(n-1) + fibonacci(n-2)

    return ret

fibonacci(20)
```

Out[78]:

```
6765
```

Thanks to recursion, we do not need a loop, nor a list, nor variables to store old Fibonacci numbers.

In a similar way we can find more complicated sets of numbers. The **Mandelbrot set**, for example, is particularly exciting. To understand what the Mandelbrot set is exactly, we first have to deal with the **Mandelbrot sequence**. The Mandelbrot progression for the number c is calculated as follows:

$$z_{n+1} = z_n^2 + c$$

Similar to the Fibonacci sequence we need the n^{th} element to calculate the $(n+1)^{th}$.

The Mandelbrot set is the set of all numbers, where none of the elements of this set becomes greater than 2.

For real numbers we can calculate this sequence in our heads:

- For c = 1

$$1^2 + 1 = 2, \quad 2^2 + 1 = 3, \quad 3^2 + 1 = 10, \quad \ldots$$

For c = 0

$$0^2 + 0 = 0, \quad 0^2 + 0 = 0, \quad 0^2 + 0 = 0, \quad \dots$$

For c = -1

$$-1^2 - 1 = 0, \quad 0^2 - 1 = -1, \quad -1^2 - 1 = 0, \quad \dots$$

c = 1 thus is not in the Mandelbrot set, because the elements of the set become infinite.

c = 0 is in the Mandelbrot set, because the elements are all 0.

c = -1 is in the Mandelbrot set, because the elements are always alternating 0 and -1.

But what about the numbers in between? And with complex numbers? That becomes very complex in our heads, so we write a program that does the work for us.

First, we need a **double loop** that moves us over an interesting area of the **complex plane**. Later we want to determine for every number that occurs whether it is in the Mandelbrot set or not. For now, we are content to simply display the absolute value of the number. In Python you can define complex numbers with `complex(real part, imaginary part)` and calculate the absolute value with `abs`.

We investigate complex numbers with a real part between -2 and 0.5 and an imaginary part between -1.25 and 1.25. We would like to have an accuracy of 0.01. To create **numbers from these intervals** the range command is insufficient. We need the command `np.arange` with which we can specify start value, end value and step size.

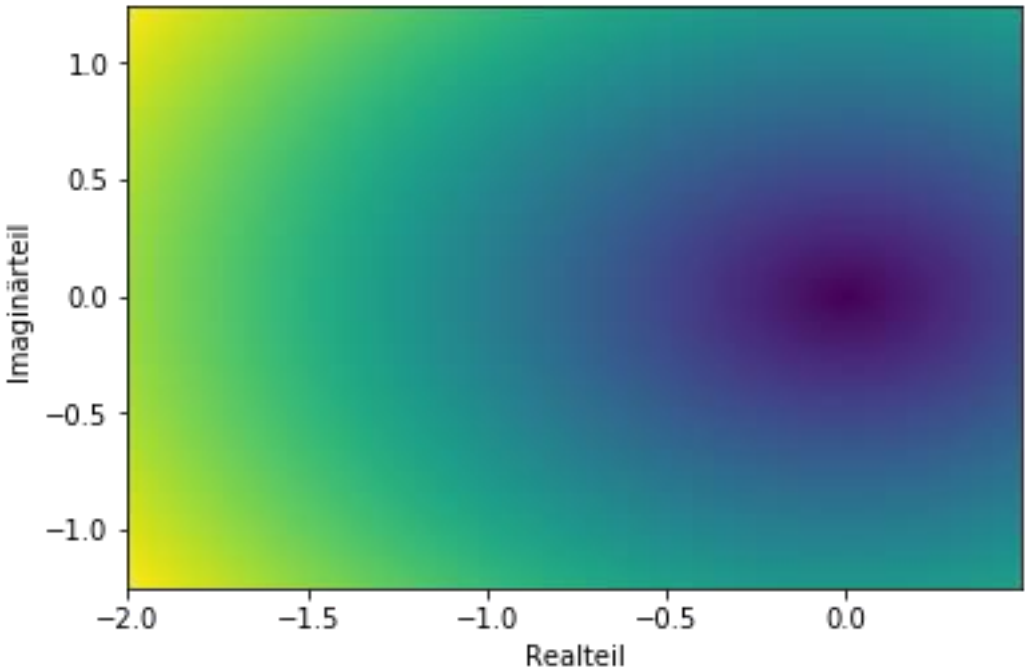
In [2]:

```
import matplotlib.pyplot as plt
%matplotlib inline

import numpy as np

# we define the real part (= r_bereich) of the numbers we want to
# investigate
r_bereich = np.arange(-2,0.5,0.01)
# and the imaginary part (= i_bereich) of the numbers we want to
# investigate.
i_bereich = np.arange(-1.25,1.25,0.01)

# we define a matrix of an appropriate size filled with zeros.
mandelbrot = np.zeros([len(i_bereich),len(r_bereich)])


# we go over all numbers with a double loop
for it in range(len(i_bereich)):
    for jt in range(len(r_bereich)):
        # and save the absolute value of the number in the matrix
        mandelbrot[it,jt]= abs(complex(r_bereich[jt],i_bereich[it]))

plt.pcolormesh(r_bereich, i_bereich, mandelbrot)
```

```
# the command pcolormesh works similar to imshow,
# but we can add x and y intervals for the labelling

plt.ylabel("Imaginärteil")    # = imaginary part
plt.xlabel("Realteil")        # = real part
```

Out[2]:

```
Text(0.5,0,'Realteil')
```



Now we have the **basic framework** of our program. We walk over an area of the complex plane and draw the result of an operation. At the moment the operation is simply the absolute value of the number. But we will replace it with another information in the next step: Namely how many steps it takes until an element of the Mandelbrot set becomes greater than 2.

First, we have to build a **function**, which calculates the **Mandelbrot sequence**. Another recursion is made here. If we want to have the n^{th} element of the Mandelbrot set of the number *cc*, we can easily calculate this by squaring the $(n\text{-}1)^{th}$ element and calculating + c. The "zeroth" element of the Mandelbrot set is always the number cc itself.

Mathematically expressed:

$$M_0 = c$$
$$M_n = (M_{n-1})^2 + c$$

We can write this as a function as follows:

In [2]:

```
def mandelbrotelement(c, it):
    # the zeroth element is the number itself
    if it == 0:
        return c
    else:
    # otherwise we have to calculate it from the previous element
```

```
        return mandelbrotelement(c, it - 1) ** 2  + c

mandelbrotelement(complex(-1,0),10)
```

Out[2]:

```
(-1+0j)
```

Our recursion seems to work. The 10th element of the Mandelbrot-Set of the number `-1` is `-1` again. With this function we can write another function, which calculates the first 50 elements of the Mandelbrot-Set

In [4]:

```
def mandelbrotfolge(c):
    ret = [] # generate empty list
    for it in range(50):
        # fill this list with elements of the Mandelbrot set
        z = mandelbrotelement(c, it)
        ret.append(z)
    return ret

def mandelbrotelement(c, it):
    if it == 0:
        return c
    else:
        return mandelbrotelement(c, it - 1) ** 2  + c


mandelbrotsequence(complex(0.1,0.1))
```

Out[4]:

```
[(0.1+0.1j),
 (0.1+0.12000000000000001j),
 (0.0956+0.12400000000000001j),
 (0.09376336+0.12370880000000001j),
 (0.0934877004810496 1+0.123198705499136j),
 (0.0935620291045716+0.123035127358712 54j),
 (0.0936162107259900 9+0.123022832333641 09j),
 (0.0936293776353018 2+0.123033862791708 58j),
 (0.093629128962925+0.1230391680025096j),
 (0.0936277692760627+0.123040100256795 93j),
 (0.0936272943412032+0.123039942119987 2j),
 (0.0936272428887645+0.123039793753185 3j),
 (0.0936272697641253 3+0.123039753309425 94j),
 (0.0936272847490399+0.123039752349626 11j),
 (0.0936272877912204 8+0.123039755857379 6j),
 (0.0936272874976964 6+0.123039757262840 78j),
 (0.0936272870968775 4+0.123039757453789 56j),
 (0.0936272869748337 7+0.123039757390912 27j),
 (0.0936272869674533 3+0.123039757349105 74j),
 (0.0936272869763590 4+0.123039757339461 1j),
 (0.0936272869804000 3+0.123039757339846 61j),
```

```
 (0.09362728698106185+0.1230397573409132j),
 (0.0936272869809233 2+0.12303975734127579j),
 (0.09362728698080815+0.12303975734130959j),
 (0.09362728698077827+0.12303975734128758j),
 (0.09362728698077809+0.12303975734127612j),
 (0.09362728698078088+0.12303975734127393j),
 (0.09362728698078193+0.1230397573412742j),
 (0.09362728698078207+0.12303975734127451j),
 (0.09362728698078202+0.12303975734127459j),
 (0.09362728698078199+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j),
 (0.09362728698078197+0.12303975734127459j)]
```

Such functions, which call themselves recursively, look very elegant, but in practice they usually have some disadvantages. Thus, for many applications, especially in Python, it makes sense that we do not recursively call a function, but by iteration. So, we do not start at the last element and work our way to element 0, but we start at the beginning and iterate until we reach the element we are interested in. The same recursive process as above, now expressed as iteration:

In [82]:

```
def mandelbrotfolge(c):
    ret = [c]
    for it in range(1, 50):
        z = ret[it - 1]**2 + c
        ret.append(z)
    return ret

mandelbrotsequence(complex(0.1,0.1))
```

Out[82]:

```
[(0.1+0.1j),
 (0.1+0.12000000000000001j),
 (0.0956+0.12400000000000001j),
 (0.09376336+0.12370880000000001j),
```

```
(0.09348770048104961+0.123198705499136j),
(0.0935620291045716+0.12303512735871254j),
(0.09361621072599009+0.12302283233364109j),
(0.09362937763530182+0.12303386279170858j),
(0.093629128962925+0.1230391680025096j),
(0.09362777692760627+0.12304010025679593j),
(0.0936272943412032+0.1230399421199872j),
(0.0936272428887645+0.1230397937531853j),
(0.09362726976412533+0.12303975330942594j),
(0.0936272847490399+0.12303975234962611j),
(0.09362728779122048+0.1230397558573796j),
(0.09362728749769646+0.12303975726284078j),
(0.09362728709687754+0.12303975745378956j),
(0.09362728697483377+0.12303975739091227j),
(0.09362728696745333+0.12303975734910574j),
(0.09362728697635904+0.1230397573394611j),
(0.09362728698040003+0.12303975733984661j),
(0.09362728698106185+0.1230397573409132j),
(0.09362728698092332+0.12303975734127579j),
(0.09362728698080815+0.12303975734130959j),
(0.09362728698077827+0.12303975734128758j),
(0.09362728698077809+0.12303975734127612j),
(0.09362728698078088+0.12303975734127393j),
(0.09362728698078193+0.1230397573412742j),
(0.09362728698078207+0.12303975734127451j),
(0.09362728698078202+0.12303975734127459j),
(0.09362728698078199+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j),
(0.09362728698078197+0.12303975734127459j)]
```

But actually, we are not interested in the whole sequence, but only in the elements before the first element, which is greater than 2. Thus, we do not need to run through the whole For-loop for

many numbers: Whenever we find a number greater than 2, we could abort the loop to save computing time. But how do we **interrupt** a loop, which actually runs until 50?

For this purpose, Python provides the command `break`. `break` is used inside loops to **abort** the loop **prematurely**. Now let us break our loop as soon as we calculate an element greater than 2:

In [84]:

```
def mandelbrotfolge(c):
    ret = [c]
    for it in range(1, 50):
        z = ret[it - 1]**2 + c
        ret.append(z)
        if abs(z)>2:
            break
    return ret

mandelbrotsequence(complex(0.5,0.5))
```

Out[84]:

```
[(0.5+0.5j), (0.5+1j), (-0.25+1.5j), (-1.6875-0.25j), (3.28515625+1.34375j)]
```

This is basically all we need to draw the Mandelbrot set. For each number we examine, we look at the Mandelbrot set. Since we automatically abort as soon as a number is too large, the **length of the list** returned by our function is automatically the number of digits in the sequence, which are smaller than 2. For numbers, which are part of the Mandelbrot set, our function returns the full length of 50.

Let us write a function, which returns the length of the list. We then build this function into our **original framework** to investigate the complex number plane by numbers, which are part of the Mandelbrot set. The form that is created is amazing.

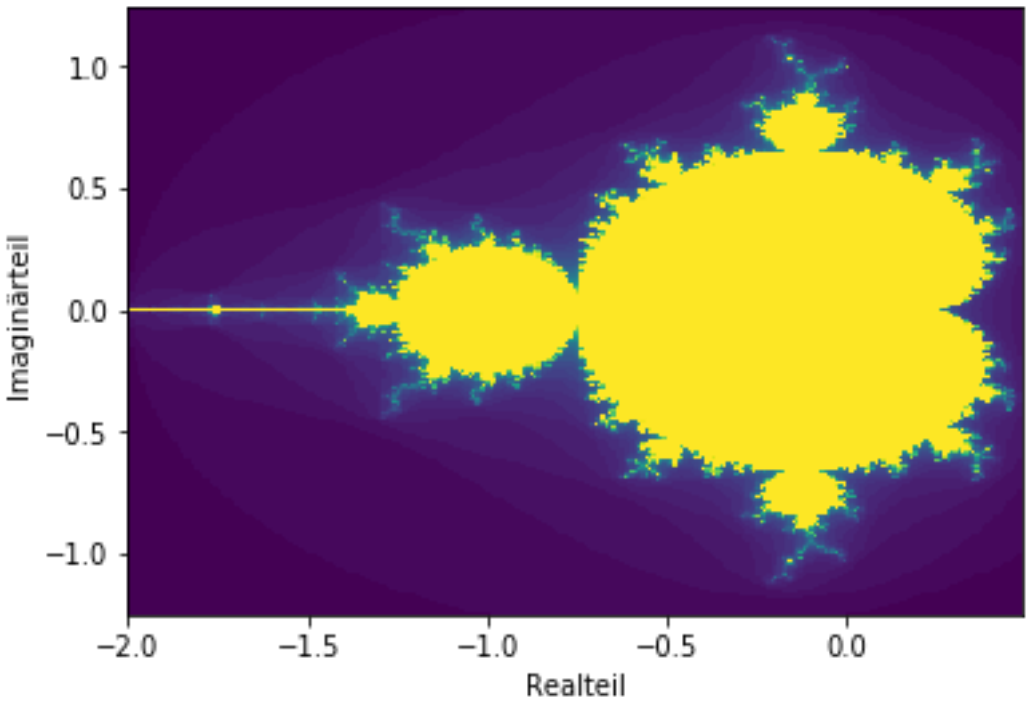
In [85]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np

def mandelbrotfolge(c):
    ret = [c]
    for it in range(1,50):
        z = ret[it - 1]**2 + c
        ret.append(z)
        if abs(z)>2:
            break
    return ret

def check_mandelbrot(c):
    # returns the length of the list
    return len(mandelbrotsequence(c))


r_bereich = np.arange(-2,0.5,0.01)
i_bereich = np.arange(-1.25,1.25,0.01)
```

```
mandelbrot = np.zeros([len(i_bereich),len(r_bereich)])

for it in range(len(i_bereich)):
    for jt in range(len(r_bereich)):
        mandelbrot[it,jt]= check_mandelbrot(complex(r_bereich[jt],i_bereich[it]))

plt.pcolormesh(r_bereich,i_bereich,mandelbrot)
plt.ylabel("Imaginärteil")  # = imaginary part
plt.xlabel("Realteil")      # = real part
```

Out[85]:

```
Text(0.5,0,'Realteil')
```



We can also zoom in and look at just a part of the complex plane. The interesting thing is that no matter how much we zoom in, we will see more and more fine structures.

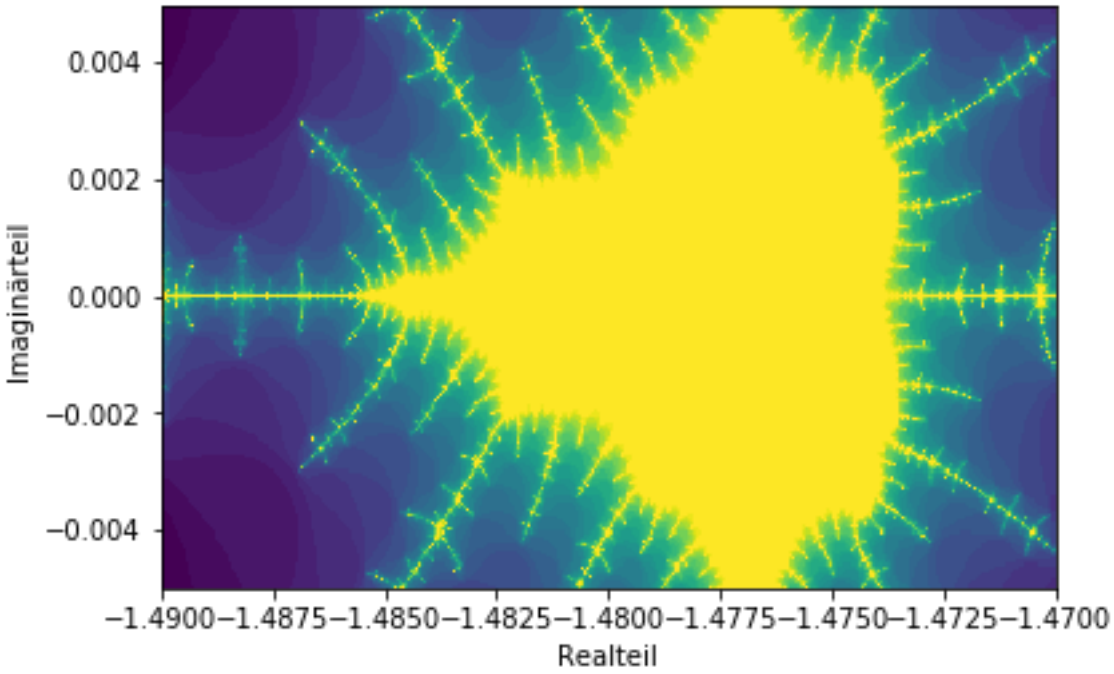
In [86]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np

def mandelbrotfolge(c):
    ret = [c]
    for it in range(1,50):
        z = ret[it - 1]**2 + c
        ret.append(z)
        if abs(z)>2:
            break
    return ret

def check_mandelbrot(c):
```

```
    return len(mandelbrotsequence(c))

r_bereich = np.arange(-1.49,-1.47,0.00005)
i_bereich = np.arange(-0.005,0.005,0.00005)

mandelbrot = np.zeros([len(i_bereich),len(r_bereich)])

for it in range(len(i_bereich)):
    for jt in range(len(r_bereich)):
        mandelbrot[it,jt]= check_mandelbrot(complex(r_bereich[jt],i_bereich[it]))

plt.pcolormesh(r_bereich,i_bereich,mandelbrot)
plt.ylabel("Imaginärteil")    # = imaginary part
plt.xlabel("Realteil")        # = real part
```

Out[86]:

```
Text(0.5,0,'Realteil')
```



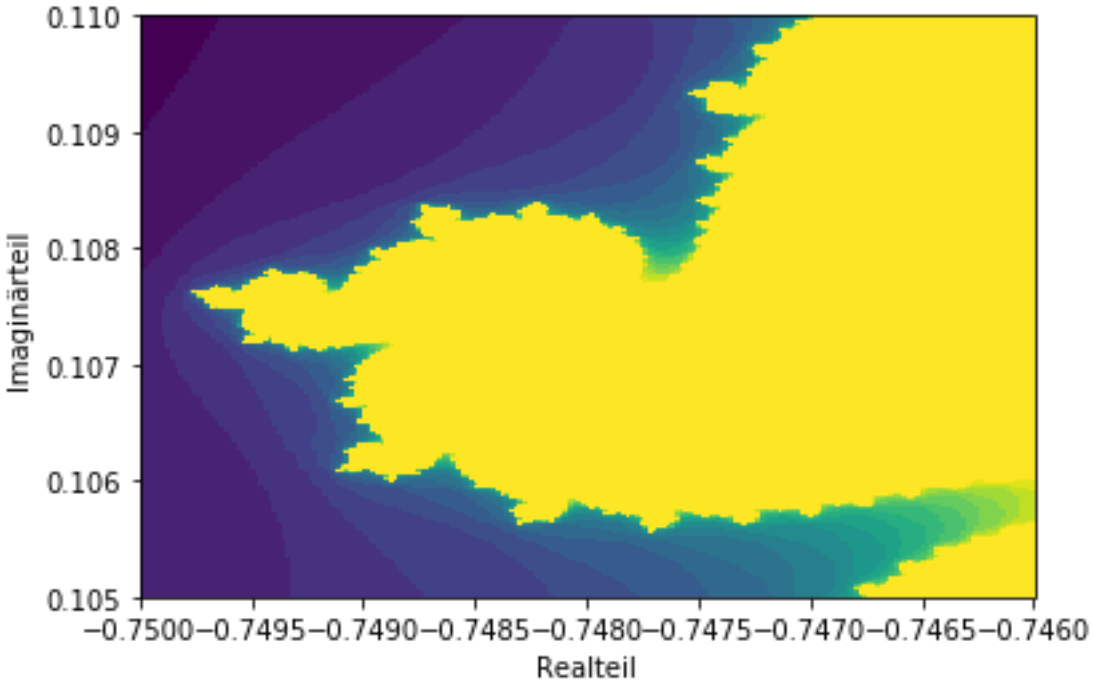
In [87]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np

def mandelbrotfolge(c):
    ret = [c]
    for it in range(1,50):
        z = ret[it - 1]**2 + c
        ret.append(z)
        if abs(z)>2:
            break
    return ret
```

```
def check_mandelbrot(c):

    return len(mandelbrotsequence (c))

r_bereich = np.arange(-0.75,-0.746,0.00002)
i_bereich = np.arange(0.105,0.11,0.00002)

mandelbrot = np.zeros([len(i_bereich),len(r_bereich)])

for it in range(len(i_bereich)):
    for jt in range(len(r_bereich)):
        mandelbrot[it,jt]= check_mandelbrot(complex(r_bereich[jt],i_bereich[it]))

plt.pcolormesh(r_bereich,i_bereich,mandelbrot)
plt.ylabel("Imaginärteil")     # = imaginary part
plt.xlabel("Realteil")         # = real part
```

Out[87]:

```
Text(0.5,0,'Realteil')
```



Why the Mandelbrot set has exactly this fractal form cannot be explained. But it serves as an impressive example that very simple recursions can lead to extraordinary complexity and that complexity is not caused by humans, but is present in nature. It only needs to be discovered.

Summary

Recursions

One speaks of recursions, if a function **is called by itself**. But you have to take care, similar to while-loops, **not to generate endless loops**. Therefore, it is important that there is at least one condition, where the function does not call itself and this condition is guaranteed to be reached at some point. In most cases, it is better to avoid recursive calls and to rewrite the recursion to an iteration.

Complex Numbers

Complex numbers can be entered in Python with `complex(real part, imaginary part)`. Then the normal arithmetic operations can be used (+, *, abs(), ...) and Python handles complex numbers correctly. Instead of *i*, which is common in mathematics, Python uses the *j* for the imaginary unit $\sqrt{-1}$, which is more common in technology.

np.arange

With `np.arange` you can create number intervals of arbitrary precision, over which you can iterate. The syntax is `np.arange(start value, target value, step)`.

break

In many cases one would like to **interrupt loops prematurely**, although the primary termination condition has not yet been reached. In Python there is the command `break` to do this. `break` interrupts the current loop. The code directly beneath the loop is executed as usual.

Chapter 11 – Classes and Objects

Modern programming is mostly object-oriented. This means that the focus is not so much on fixed procedures and arithmetic operations that are executed, but on objects that are defined, have properties and capabilities and can be used in a variety of ways.

Also, in Python you can easily create your own **objects**. An object always belongs to a **class**, which you can also define yourself.

In the following we will use objects and classes to simulate a traffic system. This traffic system will be very rudimentary. There will be people who have a home and a place of work, and they will commute between these places. A person will always be an object of a class and will have properties (e.g. hometown) and skills - called **methods** in programming - (e.g. driving to work).

Let's just get started: We define the class Person with the word `class`. Then we can define methods for this class. One method that all classes have is the **initialization method**. With this method you can create new objects of this class. In Python this initialization method always has the same name as the class itself. So if we call the class `Person`, we can create new persons with the `Person()` function. The name of this initialization method is uniform in Python and is `__init__`, all other methods we may name ourselves. The initialization method usually defines the properties of this object. The property `home` would be called `self.home`. This means that every person has a home, but not necessarily the same. When we initialize a property we set the value that this property has for the current object, so `self.home`.

So now we create the class Person and define the initialization method. There the persons get a random home address. **Methods are defined similar to functions**, the only difference is, that methods always have `self` as the first input variable, which is itself. So, every time a method is called, Python knows, which object of this class (for us which person) called the method.

In [88]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import random


class Person:
    def __init__(self):
        # a new person is initialized

        # they get a random home address
        self.home = [random.randint(0,10),random.randint(0,10)]
```

Like defining a function, defining classes does not do anything by itself. They must always be called in the main program first. Now let's create a person as a sample and draw their home address as a scatterplot.

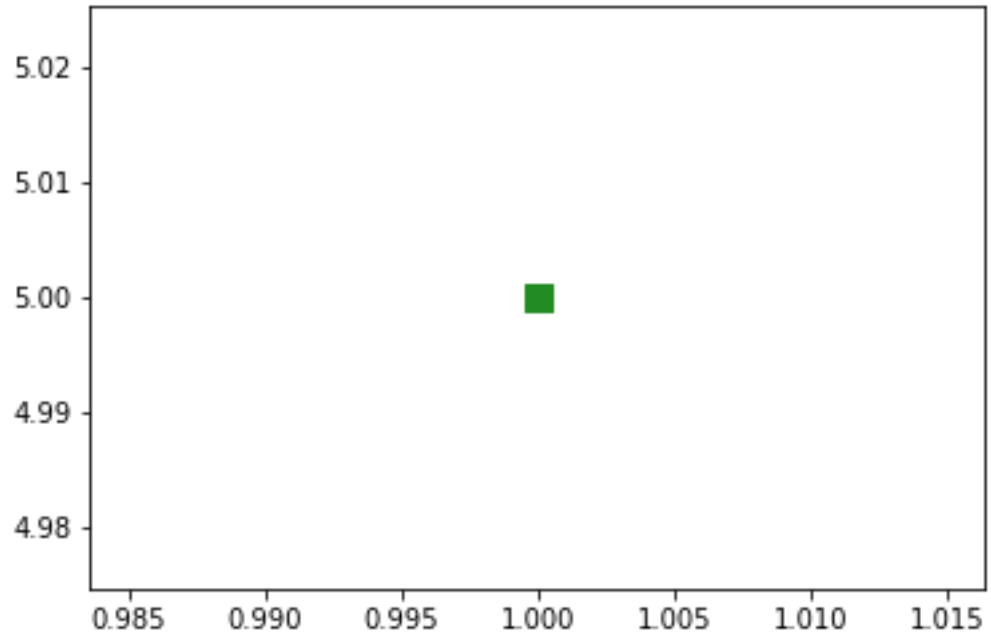
In [89]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import random


class Person:
    def __init__(self):
```

```
        # a new person is initialized

        # they gets a random home address
        self.home = [random.randint(0,10),random.randint(0,10)]


#----Here the main program starts----


# we create an object of class person with the name person1
person1 = Person()
# Then we draw the home of person1
# s is the size of the marker, the shape "s" stands for square
plt.scatter(person1.home[0],person1.home[1], s = 100, marker = "s", c = "fo
restgreen")
```

Out[89]:

```
<matplotlib.collections.PathCollection at 0x2ab3c21b198>
```

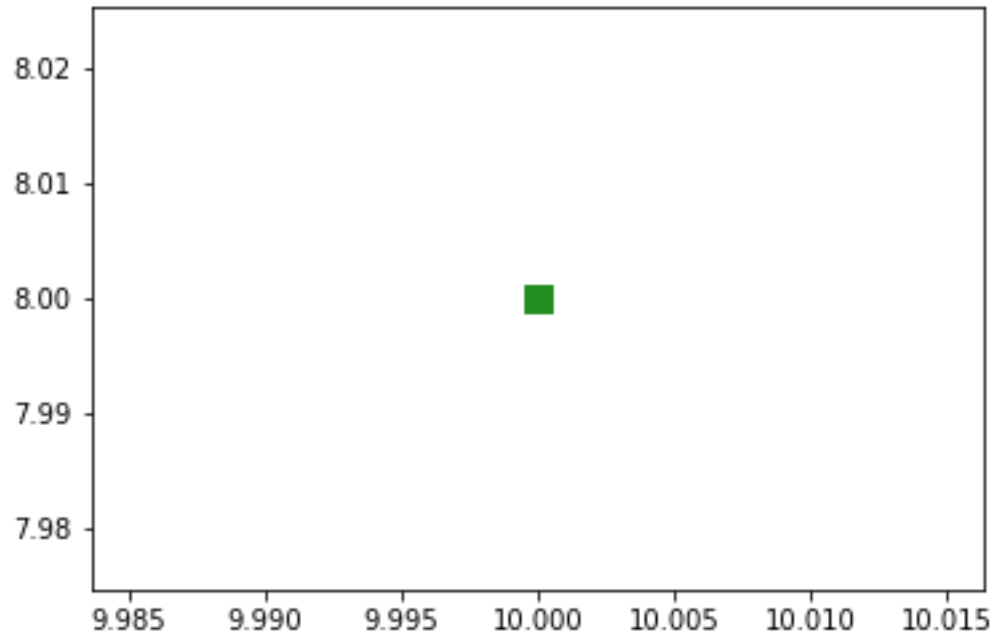


We have now successfully created an object of class Person and drawn the home. But it would be more elegant if we did not do the drawing in the main program, but created it as a **method**: All persons should get the ability to draw themselves. We call this method `zeichnen` and define it directly after `__init__`. Whenever we want to call this method, we write `NameDerPerson.zeichnen()`.

In [90]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import random


class Person:
    def __init__(self):
        # a new person is initialized
```

```
        # they get a random home address
        self.home = [random.randint(0,10),random.randint(0,10)]

    def zeichnen(self):
        # The persons draw their home address
        # s is the size of the marker, the shape "s" stands for square
        plt.scatter(self.home[0],self.home[1], s = 100, marker = "s", c = "
forestgreen")


#----Here the main program starts----

# we create an object of class person with the name person1
person1 = Person()
# Then we draw the home town of person1
person1.zeichnen()
```

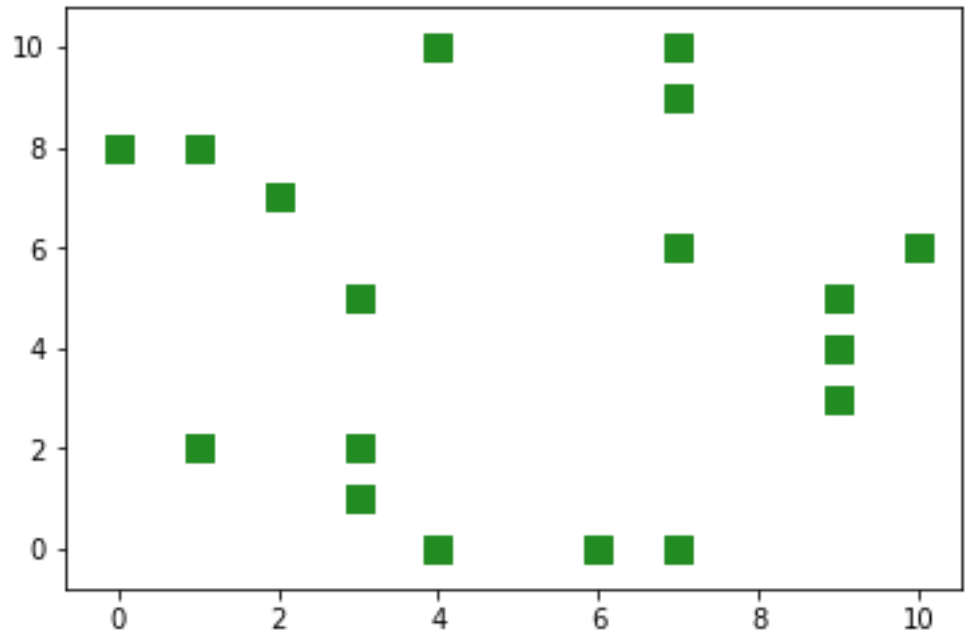


With this elegant solution, we can now easily create and manage **multiple objects** of the same class. We create an empty list and then add 20 objects to the `Person` class. We then call them all in a loop and let them draw.

In [92]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import random


class Person:
    def __init__(self):
        # a new person is initialized

        # they get a random home address
        self.home = [random.randint(0,10),random.randint(0,10)]
```

```
    def zeichnen(self):
        # The persons draw their home address
        # s is the size of the marker, the shape "s" stands for square
plt.scatter(self.home[0],self.home[1], s = 100, marker = "s", c = "forestgr
een")


#----Here the main program starts----

# We create an empty list
allepersonen=[]

# We add new people
for it in range(20):
    allepersonen.append(Person())

for it in range(20):
    allepersonen[it].zeichnen()
```



The last loop runs over a list of objects and calls the it-th object in the it-th step to do something with it. This structure is so common that there is a **shorthand** for it in Python. Instead of

```
for it in range(20):
      Command for object stored under listenname[it]
```

you can write shortened

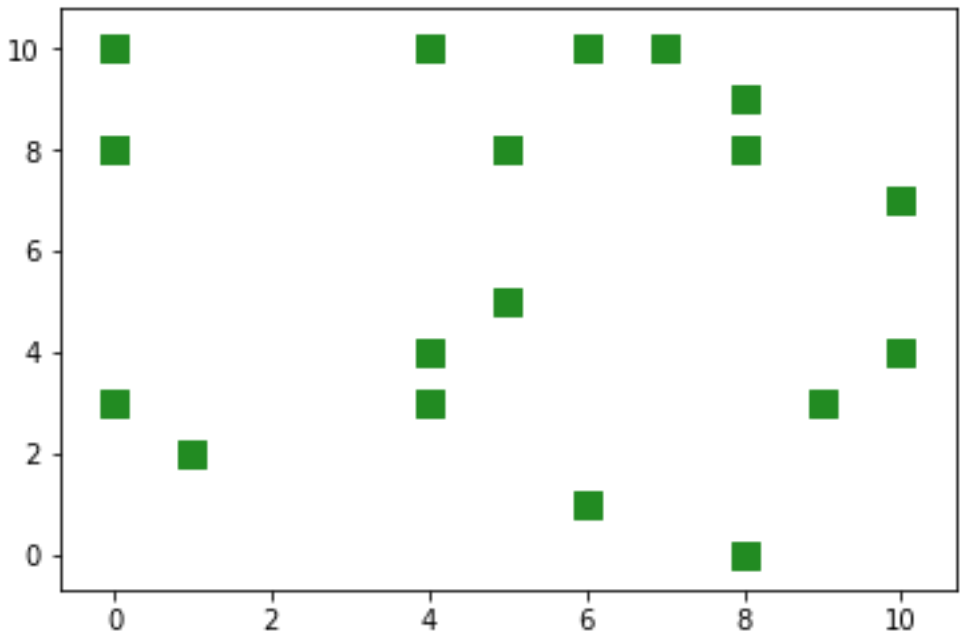
```
for obj in listenname:
      Command for obj
```

, where `obj` is a freely selectable name.

So, for our program this would be a shorthand way of writing:

In [93]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import random


class Person:
    def __init__(self):
        # a new person is initialized

        # she gets a random home address
        self.home = [random.randint(0,10),random.randint(0,10)]

    def zeichnen(self):
        # The persons draw their home address
        # s is the size of the marker, the shape "s" stands for square
        plt.scatter(self.home[0],self.home[1], s = 100, marker = "s", c = "
forestgreen")


#----Here the main program starts----

# We create an empty list
allepersonen=[]

# We add new people
for it in range(20):
    allepersonen.append(Person())

for p in allepersonen:
    p.zeichnen()
```



In addition to a home, all persons now also get a **place of work**. In our model there are 3 working places, which are different in size. One work place is at position [7,9], one at [4,1] and one at [5,5].

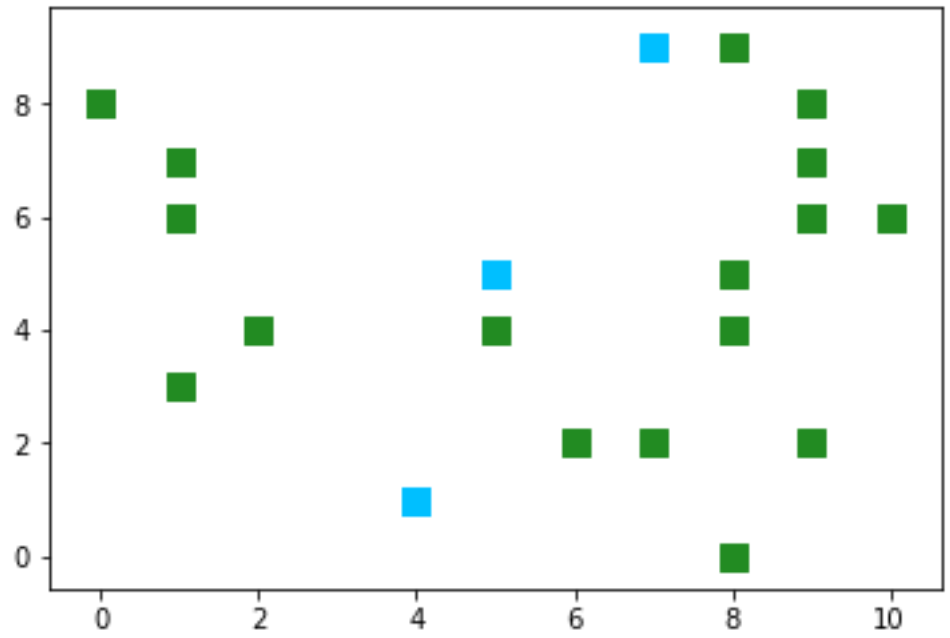
When initializing, all persons are assigned a work location from a list of work locations. Since not all work places are visited equally often, not all places appear equally often in the list.

We also make a change in the `zeichnen` method: The workspace should also be drawn.

In [94]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import random


class Person:
    def __init__(self):
        # a new person is initialized

        # they get a random home address
        self.home = [random.randint(0,10),random.randint(0,10)]
        # and a work address from a list
        self.work = random.choice([[5,5],[5,5],[5,5],[4,1],[4,1],[7,9]])

    def zeichnen(self):
        # The persons draw their home address
        # s is the size of the marker, the shape "s" stands for square
        plt.scatter(self.home[0],self.home[1], s = 100, marker = "s", c = "
forestgreen")
        # And their place of work
        plt.scatter(self.work[0],self.work[1], s = 100, marker = "s", c = "
deepskyblue")
```

```
#----Here the main program starts----

#We create an empty list
allepersonen=[]

#We add new people
for it in range(20):
    allepersonen.append(Person())

for p in allepersonen:
    p.zeichnen()
```



Now we would like to add another important method to our traffic system. People should have the ability **to drive from their home to their place of work**. This way we can see which streets of the model city are used frequently and which are not.

For driving we use the so-called **Manhattan metric**: In this metric it is only allowed to drive along the x-axis and along the y-axis, but not diagonally. So our people drive first in x-direction at constant y, then in y-direction at constant x.

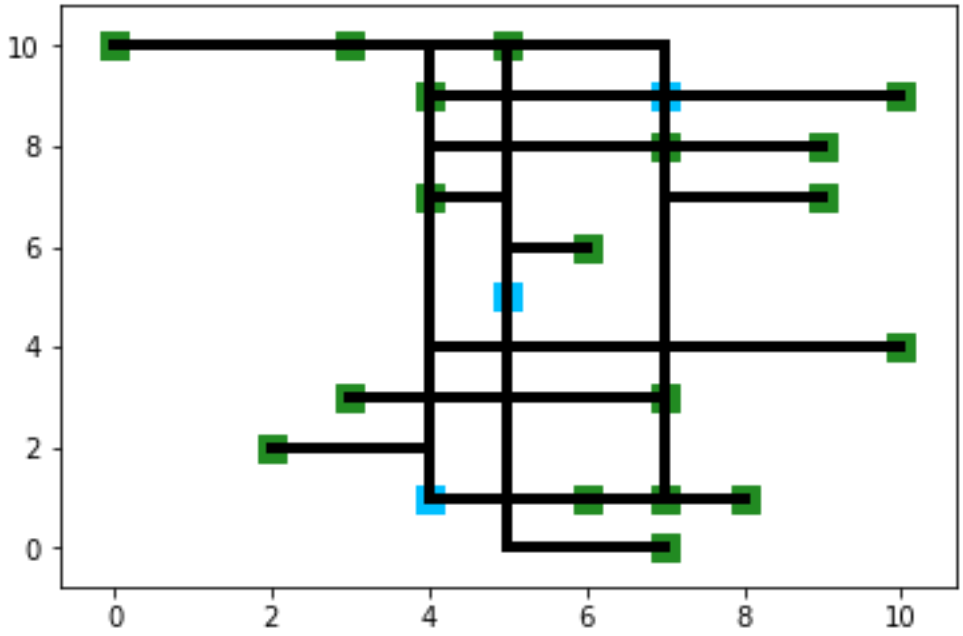
We simply implement the driving itself as line plots in addition to our existing plot.

Now let's create the method `fahren` and call it up for all persons.

In [95]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import random


class Person:
    def __init__(self):
        # a new person is initialized
```

```
        # they get a random home address
        self.home = [random.randint(0,10),random.randint(0,10)]
        # and a work address from a list
        self.work = random.choice([[5,5],[5,5],[5,5],[4,1],[4,1],[7,9]])

    def zeichnen(self):
        # The persons draw their home address
        # s is the size of the marker, the shape "s" stands for square
        plt.scatter(self.home[0],self.home[1], s = 100, marker = "s", c = "
forestgreen")
        # And their place of work
        plt.scatter(self.work[0],self.work[1], s = 100, marker = "s", c = "
deepskyblue")

    def fahren(self):
        # We draw two lines:
        # Line 1 goes in the x-direction from home to work, y remains consta
nt at home
        # Line 2 goes in y-direction from home to work, x remains constant a
t work
        plt.plot([self.home[0],self.work[0]],[self.home[1],self.home[1]],c
= "black", lw = 4)
        plt.plot([self.work[0],self.work[0]],[self.home[1],self.work[1]],c
= "black", lw = 4)


#----Here the main program starts----

# We create an empty list
allepersonen=[]

# We add new people
for it in range(20):
    allepersonen.append(Person())

for p in allepersonen:
    p.zeichnen()
    p.fahren()
```



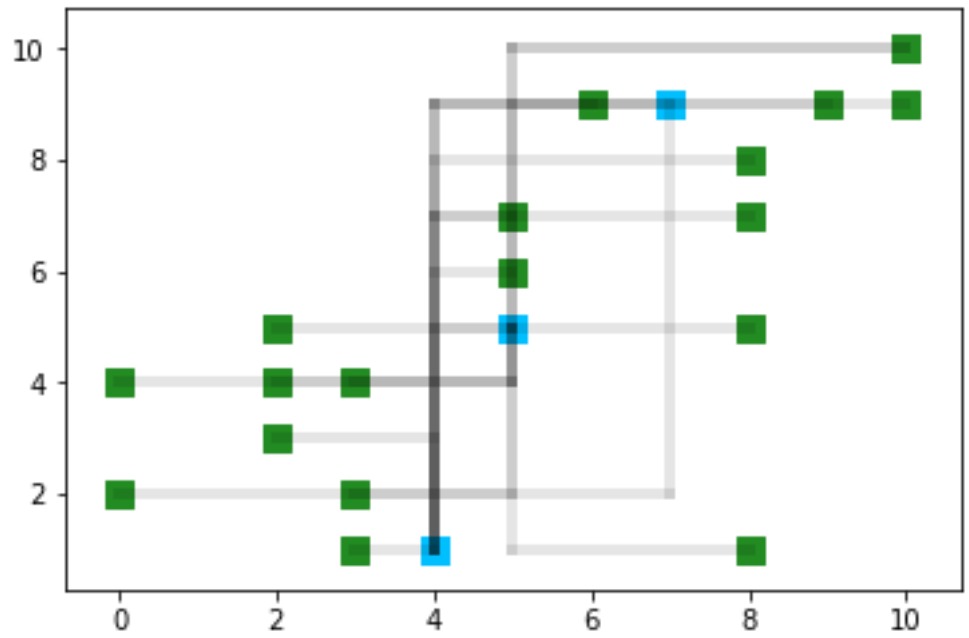
Now we see which lines are used, but not necessarily which ones are used how often, because we often draw lines on top of each other and the plot does not give any information about how many lines were drawn on top of each other.

We can solve this problem with the `alpha` attribute. The `alpha` of a line tells us the **opacity**. The value 0.1 tells us that only 10 percent opacity is used, so only when we draw 10 lines on top of each other does the line look really black, otherwise parts of the white background will shimmer through. This should help us to learn more about our traffic system.

In [96]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import random


class Person:
    def __init__(self):
        # a new person is initialized

        # they get a random home address
        self.home = [random.randint(0,10),random.randint(0,10)]
        # and a work address from a list
        self.work = random.choice([[5,5],[5,5],[5,5],[4,1],[4,1],[7,9]])

    def zeichnen(self):
        # The persons draw their home address
        # s is the size of the marker, the shape "s" stands for square
        plt.scatter(self.home[0],self.home[1], s = 100, marker = "s", c = "
forestgreen")
        # And their place of work
        plt.scatter(self.work[0],self.work[1], s = 100, marker = "s", c = "
deepskyblue")
```

```
    def fahren(self):
        # We draw two lines:
        # Line 1 goes in the x-direction from home to work, y remains consta
nt at home
        # Line 2 goes in y-direction from home to work, x remains constant a
t work
        plt.plot([self.home[0],self.work[0]],[self.home[1],self.home[1]],c
= "black", alpha = 0.1, lw = 4)
        plt.plot([self.work[0],self.work[0]],[self.home[1],self.work[1]],c
= "black", alpha = 0.1, lw = 4)


#----Here the main program starts----

# We create an empty list
allepersonen=[]

# We add new people
for it in range(20):
    allepersonen.append(Person())

for p in allepersonen:
    p.zeichnen()
    p.fahren()
```



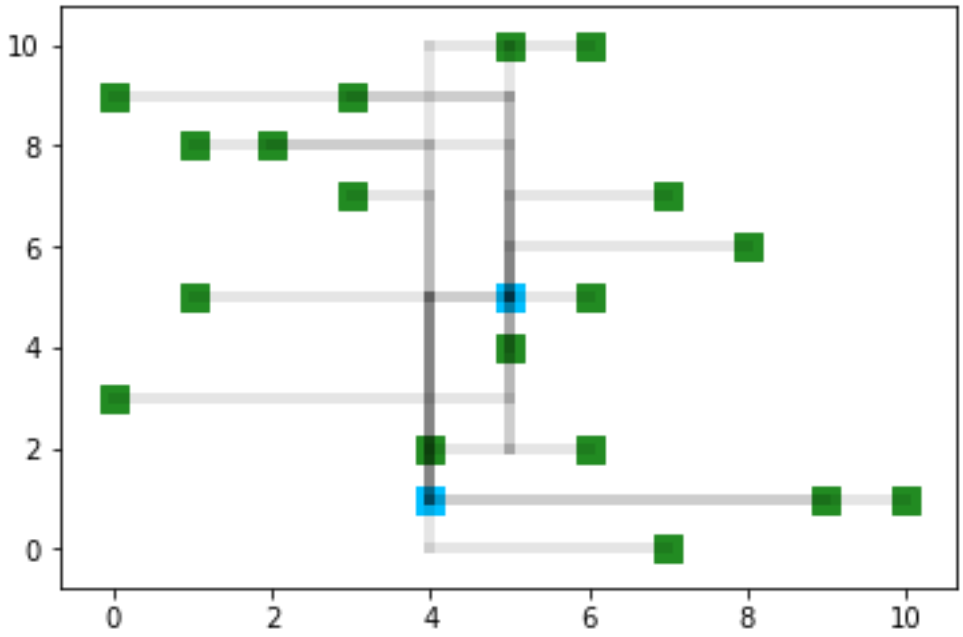
We would get even more information about the traffic system if we calculated the **total distance** travelled. In a first step, we will implement a method that outputs the length of the way to work for each person.

In [97]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import random
```

```
class Person:
    def __init__(self):
        # a new person is initialized

        # they get a random home address
        self.home = [random.randint(0,10),random.randint(0,10)]
        # and a work address from a list
        self.work = random.choice([[5,5],[5,5],[5,5],[4,1],[4,1],[7,9]])

    def zeichnen(self):
        # The persons draw their home address
        # s is the size of the marker, the shape "s" stands for square
        plt.scatter(self.home[0],self.home[1], s = 100, marker = "s", c = "
forestgreen")
        # And their place of work
        plt.scatter(self.work[0],self.work[1], s = 100, marker = "s", c = "
deepskyblue")

    def fahren(self):
        # We draw two lines:
        # Line 1 goes in the x-direction from home to work, y remains consta
nt at home
        # Line 2 goes in y-direction from home to work, x remains constant a
t work
        plt.plot([self.home[0],self.work[0]],[self.home[1],self.home[1]],c
= "black", alpha = 0.1, lw = 4)
        plt.plot([self.work[0],self.work[0]],[self.home[1],self.work[1]],c
= "black", alpha = 0.1, lw = 4)

    def arbeitsweg(self):
        # the path is calculated here from the distance in x-direction + the
distance in y-direction
        dist = abs(self.home[0]-self.work[0]) + abs(self.home[1]-self.work[
1])
        return dist


#----Here the main program starts----

# We create an empty list
allepersonen=[]

# We add new people
for it in range(20):
    allepersonen.append(Person())

for p in allepersonen:
```

```
    p.zeichnen()
    p.fahren()
```



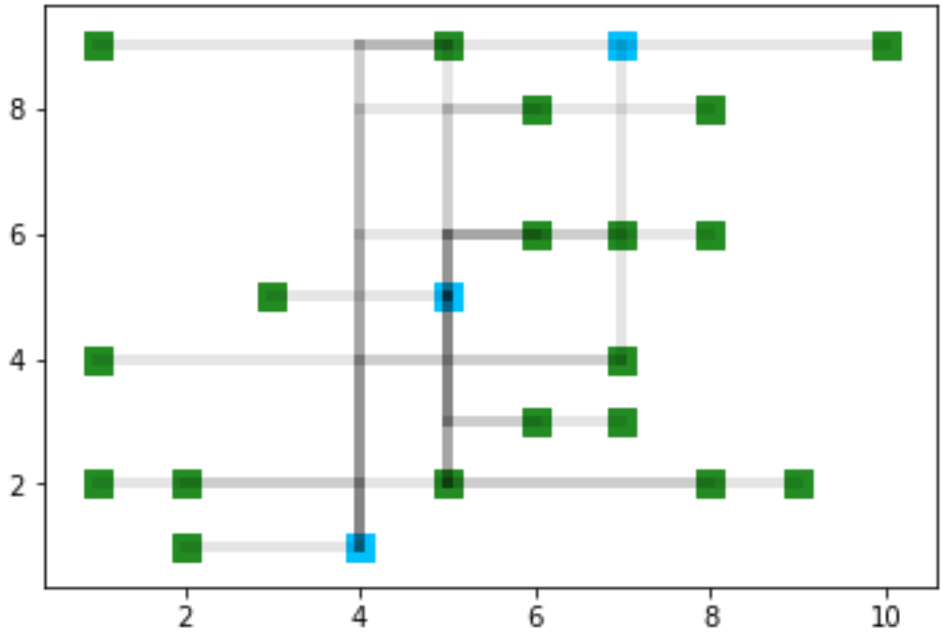
Now we write another function, which **adds up all distances** and displays the sum on the screen. Attention: This function is **not a method of the class**, because a single person cannot know about the routes of all other persons. Therefore, the definition of this function is not indented, so it is not part of the class:

In [98]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import random


class Person:
    def __init__(self):
        # a new person is initialized

        # they get a random home address
        self.home = [random.randint(0,10),random.randint(0,10)]
        # and a work address from a list
        self.work = random.choice([[5,5],[5,5],[5,5],[4,1],[4,1],[7,9]])

    def zeichnen(self):
        # The persons draw their home address
        # s is the size of the marker, the shape "s" stands for square
        plt.scatter(self.home[0],self.home[1], s = 100, marker = "s", c = "
forestgreen")
        # And their place of work
        plt.scatter(self.work[0],self.work[1], s = 100, marker = "s", c = "
deepskyblue")

    def fahren(self):
        # We draw two lines:
```

```
        # Line 1 goes in the x-direction from home to work, y remains consta
nt at home
        # Line 2 goes in y-direction from home to work, x remains constant a
t work
        plt.plot([self.home[0],self.work[0]],[self.home[1],self.home[1]],c
= "black", alpha = 0.1, lw = 4)
        plt.plot([self.work[0],self.work[0]],[self.home[1],self.work[1]],c
= "black", alpha = 0.1, lw = 4)

    def arbeitsweg(self):
        # the path is calculated here from the distance in x-direction + the
distance in y-direction
        dist = abs(self.home[0]-self.work[0]) + abs(self.home[1]-self.work[
1])
        return dist

def gesamtstrecke():
    # we calculate the total distance
    gesamt = 0
    # We go in a loop over all persons
    for p in allepersonen:
        # and add up the work paths
        gesamt = gesamt + p.arbeitsweg()

    print("Gesamtstrecke:")
    print(gesamt)

#----Here the main program starts----

# We create an empty list
allepersonen=[]

# We add new people
for it in range(20):
    allepersonen.append(Person())

for p in allepersonen:
    p.zeichnen()
    p.fahren()

gesamtstrecke()
```

```
Gesamtstrecke:
116
```

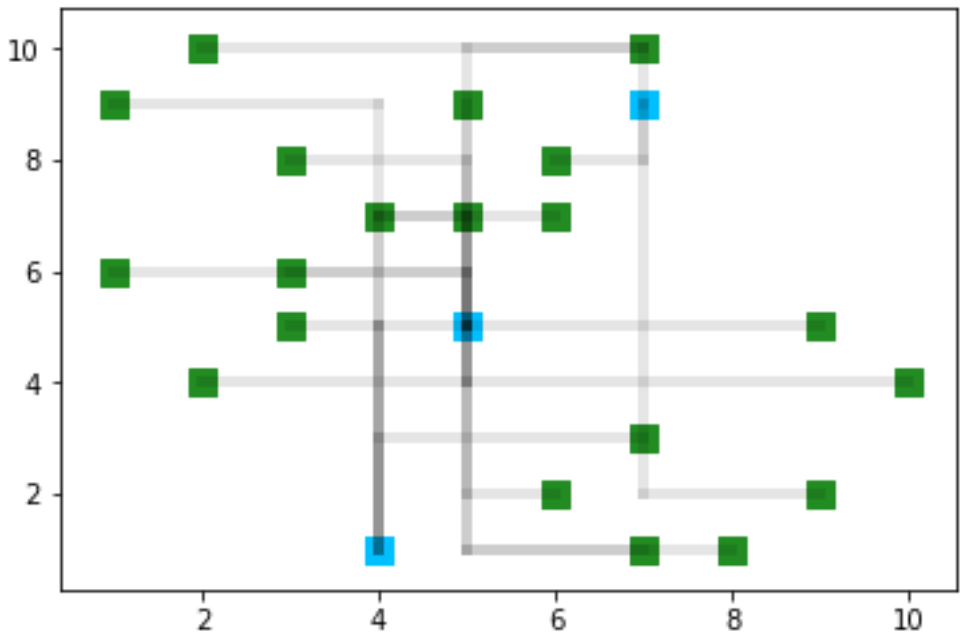


Here, the total distance is now above the plot, although we only call up the `gesamtstrecke` afterwards. This has to do with the fact that by default Python waits with the plots and does not show them on the screen until all other calculations are finished. With the function `plt.show()` we can **display a plot immediately**.

In [99]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import random


class Person:
    def __init__(self):
        # a new person is initialized

        # they get a random home address
        self.home = [random.randint(0,10),random.randint(0,10)]
        # and a work address from a list
        self.work = random.choice([[5,5],[5,5],[5,5],[4,1],[4,1],[7,9]])

    def zeichnen(self):
        # The persons draw their home address
        # s is the size of the marker, the shape "s" stands for square
        plt.scatter(self.home[0],self.home[1], s = 100, marker = "s", c = "
forestgreen")
        # And their place of work
        plt.scatter(self.work[0],self.work[1], s = 100, marker = "s", c = "
deepskyblue")

    def fahren(self):
        # We draw two lines:
        # Line 1 goes in the x-direction from home to work, y remains consta
nt at home
```

```
        # Line 2 goes in y-direction from home to work, x remains constant a
t work
        plt.plot([self.home[0],self.work[0]],[self.home[1],self.home[1]],c
= "black", alpha = 0.1, lw = 4)
        plt.plot([self.work[0],self.work[0]],[self.home[1],self.work[1]],c
= "black", alpha = 0.1, lw = 4)

    def arbeitsweg(self):
        # the path is calculated here from the distance in x-direction + th
e distance in y-direction
        dist = abs(self.home[0]-self.work[0]) + abs(self.home[1]-self.work[
1])
        return dist

def gesamtstrecke():
    # we calculate the total distance
    gesamt = 0
    # We go in a loop over all persons
    for p in allepersonen:
        # and sum up the work paths
        gesamt = gesamt + p.arbeitsweg()

    print("Gesamtstrecke:")
    print(gesamt)

#----Here the main program starts----

# We create an empty list
allepersonen=[]

# We add new people
for it in range(20):
    allepersonen.append(Person())

for p in allepersonen:
    p.zeichnen()
    p.fahren()

plt.show()

gesamtstrecke()
```




```
Gesamtstrecke:
111
```

Now we would like to try if we can still **optimize** this traffic system a little bit. Many people live far away from their place of work and could save a lot of distance by moving. So, let's write a method that **changes the home of a person on a trial basis**. If the commute to work has become longer, the change will be reversed.

This method should consist of 3 parts: First we remember the current route and home. Then we change the home to a new random value. In the last step we undo the change, if it was disadvantageous (for this we need the information about old distance and home).

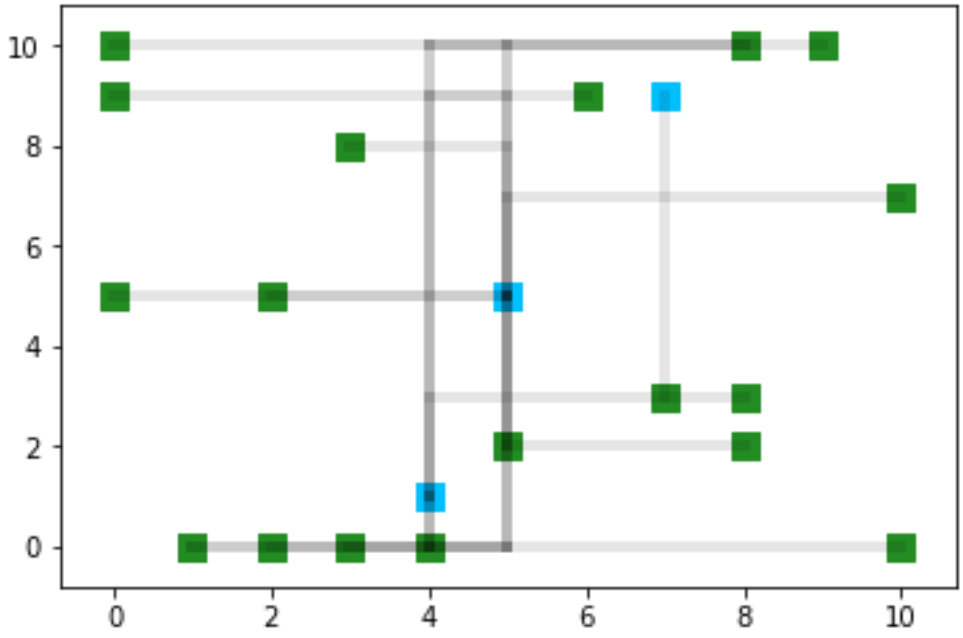
We define this method and then give our people 10 times the chance to move. Then we repeat the plot and the distance calculation to see what effect this change had.

In [100]:

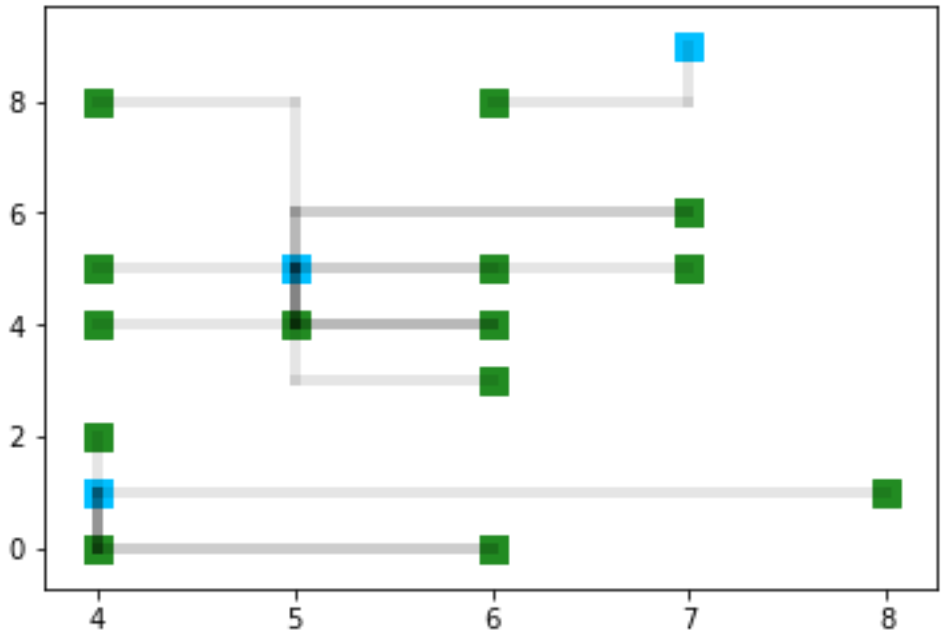
```
import matplotlib.pyplot as plt
%matplotlib inline
import random


class Person:
    def __init__(self):
        # a new person is initialized

        # they get a random home address
        self.home = [random.randint(0,10),random.randint(0,10)]
        # and a work address from a list
        self.work = random.choice([[5,5],[5,5],[5,5],[4,1],[4,1],[7,9]])

    def zeichnen(self):
        # The persons draw their home address
        # s is the size of the marker, the shape "s" stands for square
```

```
        plt.scatter(self.home[0],self.home[1], s = 100, marker = "s", c = "
forestgreen")
        # And their place of work
        plt.scatter(self.work[0],self.work[1], s = 100, marker = "s", c = "
deepskyblue")

    def fahren(self):
        # We draw two lines:
        # Line 1 goes in the x-direction from home to work, y remains consta
nt at home
        # Line 2 goes in y-direction from home to work, x remains constant a
t work
        plt.plot([self.home[0],self.work[0]],[self.home[1],self.home[1]],c
= "black", alpha = 0.1, lw = 4)
        plt.plot([self.work[0],self.work[0]],[self.home[1],self.work[1]],c
= "black", alpha = 0.1, lw = 4)

    def arbeitsweg(self):
        # the path is calculated here from the distance in x-direction + the
distance in y-direction
        dist = abs(self.home[0]-self.work[0]) + abs(self.home[1]-self.work[
1])
        return dist

    def umziehen(self):
        # Method to change the home location

        # we save the current route and the current home location
        alterweg = self.arbeitsweg()
        oldhome = self.home

        # the person moves to a random position
        self.home = [random.randint(0,10),random.randint(0,10)]

        # If the path has become longer as a result...
        if self.arbeitsweg() > alterweg:
            #... the change is undone
            self.home = oldhome


def gesamtstrecke():
    # we calculate the total distance
    gesamt = 0
    # We go in a loop over all persons
    for p in allepersonen:
        # and sum up the work paths
        gesamt = gesamt + p.arbeitsweg()

    print("Gesamtstrecke:")
```

```
    print(gesamt)


#----here the actual program starts----


# We create an empty list
allepersonen=[]

# We add new people
for it in range(20):
    allepersonen.append(Person())

# We iterate over all persons
for p in allepersonen:
    # Each person uses the drawing and driving method
    p.zeichnen()
    p.fahren()

# The plot is displayed
plt.show()
# The total distance is calculated
gesamtstrecke()

# We let the people move
for it in range(10):
    for p in allepersonen:
        p.umziehen()


# A new figure is created
plt.figure()

# Draw again and calculate the total distance
for p in allepersonen:
    p.zeichnen()
    p.fahren()
plt.show()
gesamtstrecke()
```




```
Gesamtstrecke:
141
```




```
Gesamtstrecke:
42
```

Of course, we could now extend this model as we wish. We can **add new methods and properties**, or refine the existing ones. Methods can (just like functions) have multiple input values. Of course, a program can also contain several different classes.

A big advantage of classes and objects: The main program always remains **very readable** and there is a strict separation of what should happen (main program) and how it should happen (methods and functions). So you can use a main program without having to understand the functions and methods in detail.

This concept is more or less the **basic concept of modern programming**: Right from the start, for example, we created graphics with the function `plot` and added elements to an object of the

class list without ever having to deal with the exact definition of these classes. Using ready-made functions, classes and even whole libraries and packages makes modern programming possible.

Summary

Classes and objects

In Python we can define **classes** and then create **objects** that belong to these classes. Each class is defined by `class NameDerKlasse:` and has an **initialization method** called `__init__` which is then called by the main program with the name of the class. This method creates a new object of this class, which can be stored under a name or in a list.

Methods and properties

Properties are variables that all objects of a class have. For example, every object of the class "Person" in our example has a home location. These properties are usually **created** and stored **in the initialization method**. Properties are called and changed within the definitions with `self.NameDerEigenschaft`. Outside the definition, you must also specify which object of the class you are referring to, i.e., `NameDesObjekts.NameDerEigenschaft`.

Methods are so to speak **functions within classes**. They can have input and output variables and are defined similar to common functions. The first input variable of a method is always `self`

Methods are defined within a class definition with `NameDerMethode(self,inputA,inputB,inputC,...):` and then called in the main program with `NameDesObjekts.NameDerMethode(inputA,inputB,inputC,...)`. Note that you do not need to use `self` when calling the method, because the name of the object is already in front of the dot.

Fast Iteration

Loops that run over a **list of objects** and in the i-th step call the i-th object to do something with it can be written in a shortened form. Instead of

```
for it in range(20):
      Command for object stored under listenname[it]
```

you can write shortened

```
for obj in listenname:
      Command for obj
```

,where `obj` is a free selectable name.

Chapter 12 – A simple macroscopic traffic model

In this chapter we use our knowledge of classes and other Python structures to create a simple traffic model. We want to find out how many direct (and for e-Cars also indirect) emissions vehicles produce and calculate some scenarios.

We will start with a simple class called `Car`. For now, this class should have only one initialization method and 3 properties: `age`, `size` and `distance`. These three sizes are the arguments of the initialization method. After that we create a test object of this class and let us output the properties.

In [215]:

```
class Car:
    def __init__(self, age, size, distance):
        self.age = age
        self.size = size
        self.distance = distance
testcar = Car(2,"m",5000)

print(testcar.age,testcar.size,testcar.distance)
```

```
2 m 5000
```

The next thing that would be relevant for us are the CO2 emissions the car produces per kilometer. However, we will not simply read in this property, but estimate it from the available information (i.e. size and age). We will create our own method for this. The numerical values we have assumed are based on Hofer, Jäger, Füllsack: Large scale simulation of CO2 emissions caused by urban car traffic.

In [217]:

```
class Car:
    def __init__(self, age, size, distance):
        self.age = age
        self.size = size
        self.distance = distance
        self.calcemissions()


    def calcemissions(self):
        if self.size == "s":
            self.emissions = 120 * (1 + age * 0.02) #g/km
        elif self.size == "m":
            self.emissions = 140 * (1 + age * 0.02)
        elif self.size == "l":
            self.emissions = 180 * (1 + age * 0.02)

testcar = Car(2,"m",5000)

print(testcar.age,testcar.size,testcar.distance,testcar.emissions)
```

```
2 m 5000 151.24681834894383
```

But here we have to be careful again: If there is something under Size that we don't expect (like "large" or "👶"), we skip all ifs and the emissions property is not set. So we need an error message pointing to this problem to avoid that we continue working with missing or wrong data.

In [227]:

```
class Car:
    def __init__(self, age, size, distance):
        self.age = age
        self.size = size
        self.distance = distance
        self.calcemissions()


    def calcemissions(self):
        if self.size == "s":
            self.emissions = 120 * (1 + age * 0.02) #g/km
        elif self.size == "m":
            self.emissions = 140 * (1 + age * 0.02)
        elif self.size == "l":
            self.emissions = 180 * (1 + age * 0.02)
        else:
            raise Exception("Unknown car size: " + str(self.size) )

testcar = Car(2,"👶",5000)

print(testcar.age,testcar.size,testcar.distance,testcar.emissions)
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-227-e17ed964b143> in <module>()
     17             raise Exception("Unknown car size: " + str(self.size) )
     18
---> 19 testcar = Car(2,"👶",5000)
     20
     21 print(testcar.age,testcar.size,testcar.distance,testcar.emissions)

<ipython-input-227-e17ed964b143> in __init__(self, age, size, distance)
      4         self.size = size
      5         self.distance = distance
----> 6         self.calcemissions()
      7
      8

<ipython-input-227-e17ed964b143> in calcemissions(self)
     15             self.emissions = 180 * (1 + age * 0.02)
     16         else:
---> 17             raise Exception("Unknown car size: " + str(self.size) )
     18
     19 testcar = Car(2,"👶",5000)

Exception: Unknown car size: 👶
```

Now that we can assign an emission value to each vehicle, we can also build a method that returns this value. This is all we need to calculate the total emissions of a vehicle fleet.

To test the method, we also create a population of vehicles.

In [230]:

```
import random


class Car:
    def __init__(self, age, size, distance):
        self.age = age
        self.size = size
        self.distance = distance
        self.calcemissions()


    def calcemissions(self):
        if self.size == "s":
            self.emissions = 120 * (1 + age * 0.02) #g/km
        elif self.size == "m":
            self.emissions = 140 * (1 + age * 0.02)
        elif self.size == "l":
            self.emissions = 180 * (1 + age * 0.02)
        else:
            raise Exception("Unknown car size: " + str(self.size) )

    def reportemissions(self):
        return self.distance * self.emissions



def makepopulation (distmean,diststd):
    allcars = []


    for it in range(10000):
        age = random.uniform(0,20)
        dist = random.gauss(distmean,diststd)
        size = random.choice(["s","m","l"])
        allcars.append(Car(age,size,dist))
    return allcars


distmean = 5500
diststd = 50
```

```
#overall emissions
allcars = makepopulation(distmean, diststd)
em_sum = 0
for car in allcars:
    em_sum = em_sum + car.reportemissions()

print(em_sum / 1000 / 10000, "kg pro Person")
```

```
869.6788068315942 kg pro Person
```

This model only becomes interesting when we can calculate different scenarios and compare them with each other. Here we have many possibilities. For example, we could look at what happens if all people whose car is older than 10 years buy a new car. Alternatively, we could look at what happens if 25% of the people change to more bus trips, which reduces their annual driving distance to 10%. For both scenarios we need new methods: One that sets the age of a vehicle to 0 and one that changes the distance travelled.

With these new methods the creation and evaluation of scenarios is easy:

In [236]:

```
import random


class Car:
    def __init__(self, age, size, distance):
        self.age = age
        self.size = size
        self.distance = distance
        self.calcemissions()


    def calcemissions(self):
        if self.size == "s":
            self.emissions = 120 * (1 + age * 0.02) #g/km
        elif self.size == "m":
            self.emissions = 140 * (1 + age * 0.02)
        elif self.size == "l":
            self.emissions = 180 * (1 + age * 0.02)
        else:
            raise Exception("Unknown car size: " + str(self.size) )

    def reportemissions(self):
        return self.distance * self.emissions

    def buynewcar(self):
        self.age = 0
        self.calcemissions()

    def usebus(self):
        self.distance = self.distance / 10


def makepopulation (distmean,diststd):
```

```
    allcars = []


    for it in range(10000):
        age = random.uniform(0,20)
        dist = random.gauss(distmean,diststd)
        size = random.choice(["s","m","l"])
        allcars.append(Car(age,size,dist))
    return allcars



distmean = 5500
diststd = 50


#overall emissions
allcars = makepopulation(distmean, diststd)
em_sum = 0
for car in allcars:
    em_sum = em_sum + car.reportemissions()

print("Baseline:  ", em_sum / 1000 / 10000, "kg pro Person")

# Scenario 1: Cars older than 10 years are replaced
allcars = makepopulation(distmean, diststd)
em_sum = 0
for car in allcars:
    if car.age > 10:
        car.buynewcar()
    em_sum = em_sum + car.reportemissions()

print("Szenario 1:",em_sum / 1000 / 10000, "kg pro Person")

# Scenario 2: 25% use a bus
allcars = makepopulation(distmean, diststd)
em_sum = 0
for car in allcars:
    if random.uniform(0,100)<20:
        car.usebus()
    em_sum = em_sum + car.reportemissions()

print("Szenario 2:",em_sum / 1000 / 10000, "kg pro Person")
```

```
Baseline:   869.7388177600628 kg pro Person
Szenario 1: 872.9046779718418 kg pro Person
Szenario 2: 713.6243753931545 kg pro Person
```

Now we can still build electric cars into our model. This raises an interesting question: Can we also use the `Car` class for electric cars? On the one hand, we could use most of the properties

and methods in exactly the same way. On the other hand, the calculation of emissions works differently for electric cars. This is where the concept of **inheritance** helps us. In Python, we can create a subclass based on a class, which inherits all properties and methods of the original class, but can also have additional properties and methods. It is also possible to adapt existing methods for the subclass.

So in our model we can create the subclass `Ecar`, which inherits everything from Car, but has a different method of `reportemissions`:

In [ ]:

```
class Ecar(Car):
    def reportemissions(self):
        return "unknown yet"
```

As you can see, this class definition is very small, since we get most of it from `Car`. Now we just have to think about how to calculate the emissions of the e-car. Of course, the matter is a bit more complicated, because the emissions are not produced directly on the road, but wherever the electricity was generated that was used to charge the car. So it depends very much on the energy mix that was used.

So the method we are looking for will be to multiply the distance by a number that tells us how many emissions per kilometer have been produced by the power generation. Of course it is difficult to do this correctly, but with a **Life Cycle Assessment (LCA)** we can at least compare different ways of producing electricity. This also gives us a rough estimate of the produced emissions. Details of this process can be found for example in Turconi, Boldrin, Astrup.

Here we assume a consumption of 0.2 kWh per kilometer and compare the energy mix of Graz, Vienna and the USA. We find the following values:

- Graz: 2.5 g/km | 90% water (10g/kWh), 5% wind (30g/kWh), 5% biomass (50g/kWh)|
- Vienna: 45 g/km | 45% water, 45% natural gas (500g/kWh), 5% wind, 5% biomass|
- USA: 100 g/km | 37% oil (700g/kWh), 30% gas (500g/kWh), 15% coal (800g/kWh), 10% nuclear (20g/kWh), 8% renewable (30g/kWh)|

We now use these values in the method `reportemissions`.

To test the e-cars we have to adjust our function `makepopulation` a little bit. We also need to know the percentage of the existing electric cars. We can estimate this at 0.4% for Austria.

In [239]:

```
import random


class Car:
    def __init__(self, age, size, distance):
        self.age = age
        self.size = size
        self.distance = distance
        self.calcemissions()


    def calcemissions(self):
        if self.size == "s":
            self.emissions = 120 * (1 + age * 0.02) #g/km
        elif self.size == "m":
```

```
            self.emissions = 140 * (1 + age * 0.02)
        elif self.size == "l":
            self.emissions = 180 * (1 + age * 0.02)
        else:
            raise Exception("Unknown car size: " + str(self.size) )

    def reportemissions(self):
        return self.distance * self.emissions

    def buynewcar(self):
        self.age = 0
        self.calcemissions()

    def usebus(self):
        self.distance = self.distance / 10

class Ecar(Car):
    def reportemissions(self):
        return self.distance * emix

def makepopulation (echance,distmean,diststd):
    allcars = []


    for it in range(10000):
        age = random.uniform(0,20)
        dist = random.gauss(distmean,diststd)
        size = random.choice(["s","m","l"])
        if random.uniform(0,100)<echance:
            allcars.append(Ecar(age,size,dist))
        else:
            allcars.append(Car(age,size,dist))
    return allcars



emix = 2.5  #GRAZ: 90% water (10g/kWh), 5% wind (30g/kWh), 5% biomass (50g/
kWh)
#emix = 45  #VIENNA: 45% water, 45% natural gas (500g/kWh), 5% wind, 5% bio
mass
#emix = 100 #USA: 37% oil (700g/kWh), 30% gas (500g/kWh), 15% coal (800g/kW
h), 10% nuclear(20g/kWh), 8% renewable (30g/kWh)

echance = 0.4
distmean = 5500
diststd = 50
```

```
#overall emissions
allcars = makepopulation(echance, distmean, diststd)
em_sum = 0
for car in allcars:
    em_sum = em_sum + car.reportemissions()

print("Baseline:  ", em_sum / 1000 / 10000, "kg per person")

#scenario 1: Cars older than 10 years are replaced
allcars = makepopulation(echance, distmean, diststd)
em_sum = 0
for car in allcars:
    if car.age > 10:
        car.buynewcar()
    em_sum = em_sum + car.reportemissions()

print("Scenario 1:",em_sum / 1000 / 10000, "kg per person")

#Szenario 2: 25% use the bus
allcars = makepopulation(echance, distmean, diststd)
em_sum = 0
for car in allcars:
    if random.uniform(0,100) < 20:
        car.usebus()
    em_sum = em_sum + car.reportemissions()

print("Scenario 2:",em_sum / 1000 / 10000, " kg per person ")
```

```
Baseline:   866.7936403430951 kg per person
Scenario 1: 869.8223364537947 kg per person
Scenario 2: 712.3889638111342 kg per person
```

This way we have succeeded in building a very simple, macroscopic traffic model in Python. Of course, this can be extended at will. The commands and structures that appear, however, will remain the same even in extremely complex programs: If you understand classes, functions, loops, queries and data types, you can write any program you can think of.

Summary

Initialization methods with arguments

Initialization methods can require arguments. This is completely analogous to functions using

```
def __init__(self,arg1,arg2,arg3)
```

These arguments (without `self`) must be passed when creating an object:

```
obj = NameDerKlasse(arg1,arg2,arg3)
```

Inheritance in classes

If you want to create a new class based on an existing class, there is the concept of **inheritance**. This allows us to inherit all methods and properties of the base class, but change and extend them.

```
Class NameOfTheNewClass(NameOfTheBaseClass):
```

Chapter 13 - Visualizations

No matter what is to be programmed, whether a model is to be created, a simulation is to be carried out or statistical data is to be evaluated: in the end, the results should mostly be visualized. The presentation of results - the "plotting" - is often as important as the results themselves and there are countless ways of visualizing.

Some important rules for creating graphics should always be observed, regardless of the type of presentation:

- all axes should be labeled
- if possible the plot should show which units are used
- all relevant values should be recognizable
- Colors should be chosen so that they are easily distinguishable even in grayscale. Alternatively, markings can be used.

In the following we give an overview of the most common ways to visualize results. We already know some of them, but there are others we have not encountered yet.

An overview of all colors that have a proper name in Matplotlib can be found here: https://matplotlib.org/examples/color/named_colors.html

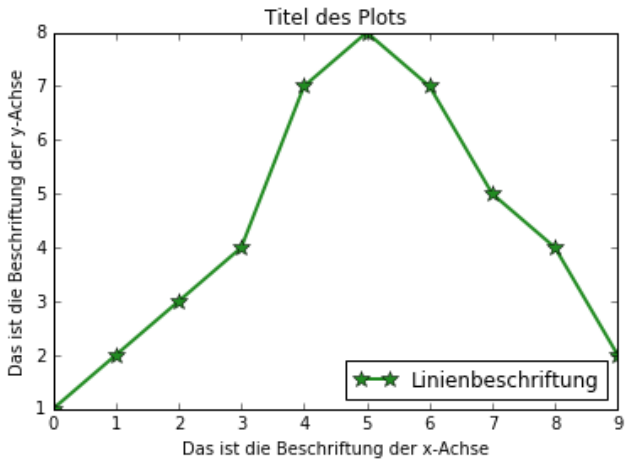
The line plot

In [1]:

```
import matplotlib.pyplot as plt
%matplotlib inline

testdata = [1, 2, 3, 4, 7, 8, 7, 5, 4, 2]
plt.xlabel("This is x-axis labeling")
plt.ylabel("This ist he y-axis labeling")
plt.title("Plot title")

plt.plot(testdata, color = "forestgreen", lw = 2, marker = "*", markersize
= 10, label = "line labeling")
plt.legend(loc = "best")
```

Out[1]:

```
<matplotlib.legend.Legend at 0xa988d30>
```



A line plot is one of the easiest ways to visualize data. Whenever you have one size (e.g. number of frogs) that you want to plot against another size (e.g. time), a line plot is a good choice.

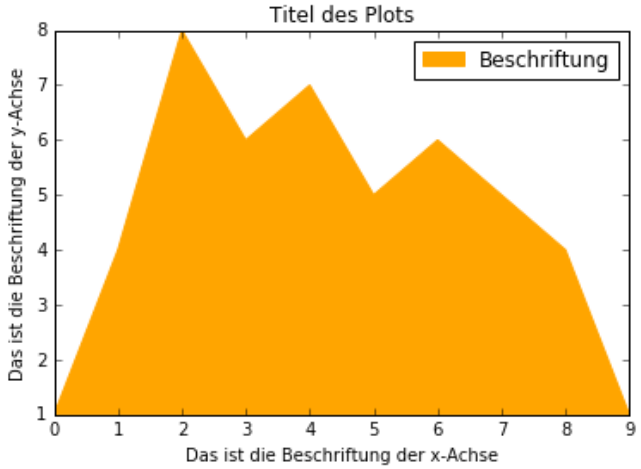
The fill plot

In [2]:

```
import matplotlib.pyplot as plt
%matplotlib inline

testdata = [1, 4, 8, 6, 7, 5, 6, 5, 4, 1]
plt.xlabel("This is x-axis labeling")
plt.ylabel("This ist he y-axis labeling")
plt.title("Plot title")

plt.fill(testdata, color = "orange", label = "labeling")
plt.legend(loc = "best")
```

Out[2]:

```
<matplotlib.legend.Legend at 0xa988cf8>
```



The Fill Plot is conceptually similar to the Line Plot, except that here you want to point to the area under the curve rather than the curve itself.

The stack plot

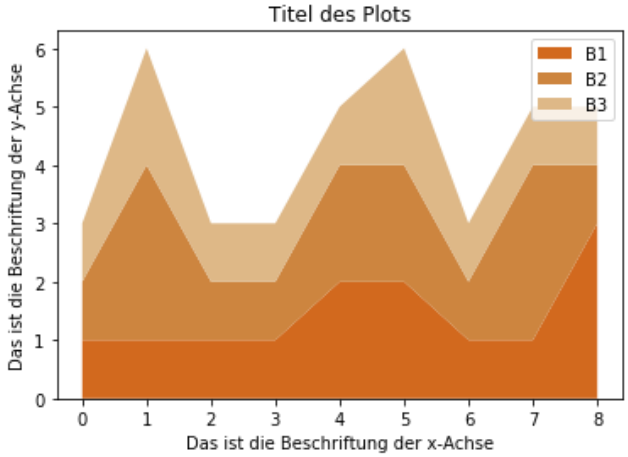
In [2]:

```
import matplotlib.pyplot as plt
%matplotlib inline

testdata = [1, 1, 1, 1, 2, 2, 1, 1, 3]
testdata2 = [1, 3, 1, 1, 2, 2, 1, 3, 1]
testdata3 = [1, 2, 1, 1, 1, 2, 1, 1, 1]

plt.xlabel("This is x-axis labeling")
plt.ylabel("This ist he y-axis labeling")
plt.title("Plot title")

plt.stackplot(range(len(testdata)), testdata, testdata2, testdata3, labels
= ("B1","B2","B3"),
              colors=["chocolate","peru","burlywood"])
```

```
plt.legend(loc = "best")
```

Out[2]:

```
<matplotlib.legend.Legend at 0x1de1222d908>
```



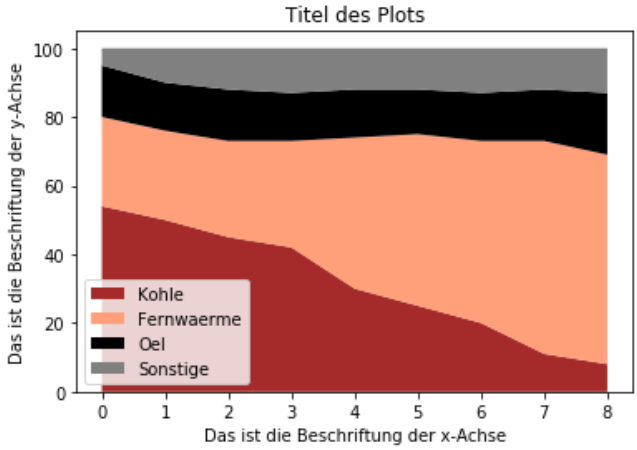
With stackplot we can stack several fill plots on top of each other. The stackplot is relatively difficult to read, because it is easily confused with a line plot. So in our example, you would think that the size B3 has the value 3 at the beginning. But in reality the plot can be read like this: B3 goes from 2 to 3, so it has the value 1. Also gradients cannot be read correctly in a stackplot. Stack plots should be used with caution. But where they do make sense is when you have sizes that always add up to a common sum. For example the percentage of a certain heating technology. All technologies should always add up to 100%:

In [3]:

```
import matplotlib.pyplot as plt
%matplotlib inline

tec1 = [54,50,45,42,30,25,20,11,8]
tec2 = [26,26,28,31,44,50,53,62,61]
tec3 = [15,14,15,14,14,13,14,15,18]
tec4 = [ 5,10,12,13,12,12,13,12,13]
plt.xlabel("This is x-axis labeling")
plt.ylabel("This ist he y-axis labeling")
plt.title("Plot title")

plt.stackplot(range(len(tec1)), tec1, tec2, tec3, tec4, labels=("coal","district heating","oil","various"),
              colors = ["brown","lightsalmon","black","gray"])
plt.legend(loc = "lower left")
```

Out[3]:

```
<matplotlib.legend.Legend at 0x1de122dbeb8>
```



The scatter plot

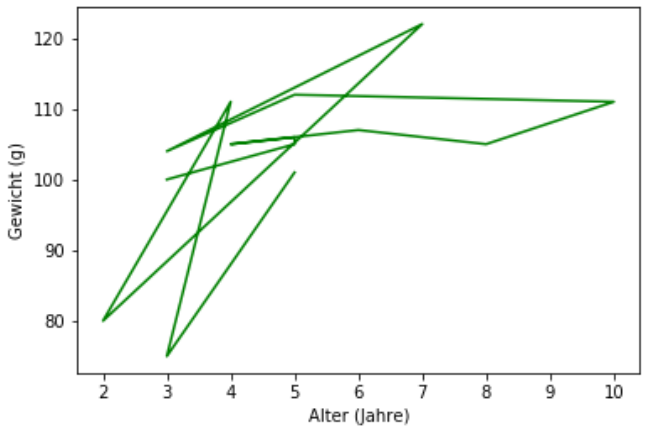
All previous plots had more or less the same field of application: We want to clearly assign a size yy (number of frogs) to another size xx (time). But what if this assignment is not unique. What if we want to weigh the frogs and determine their age. In this case it is not so that we can assign an exact weight to each age: Frogs of the same age may have different weights, and frogs of different ages may have the same weight. So what happens if we weigh all the frogs in the pond and want to graph both their age and their weight? Let's first try a line plot.

In [4]:

```
import matplotlib.pyplot as plt
%matplotlib inline

# The data is structured as follows:
# frogweight = (weight of frog1, weight of frog2, weight of frog3, ...)
# frogage = (age of frog1, age of frog2, age of frog3, ...)

frogweight = (100,105,106,105,107,105,111,112,104,122,80,111,75,101)
frogage = (3, 5, 5, 4, 6, 8, 10, 5, 3, 7, 2, 4, 3, 5)
plt.plot(frogage, frogweight, color = "green")
plt.xlabel("age (years)")
plt.ylabel("weight(g)")
```

Out[4]:

```
<matplotlib.text.Text at 0x1de121d9a90>
```



This plot is not wrong in principle, it is just very confusing. The connecting lines suggest that the frogs, which were randomly weighed one after the other, have something to do with each other.

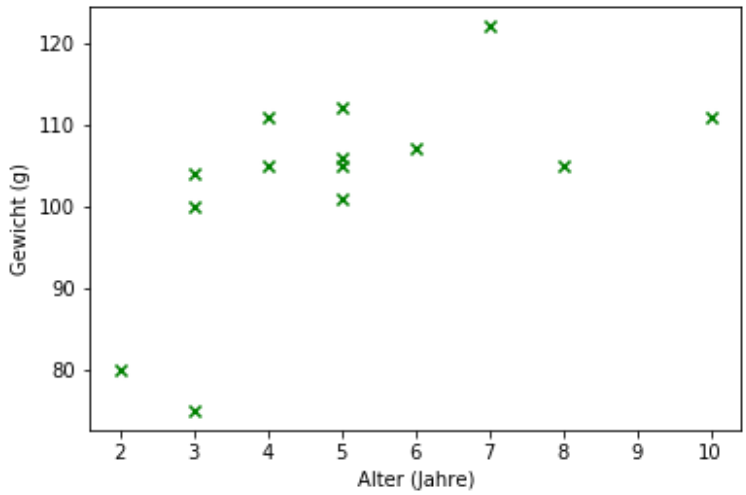
But the order of the frogs has no meaning in this plot, the frogs could have been weighed in a completely different order. So to repair this plot, we should remove the connection lines and just mark each frog. This creates a so-called **scatter plot**.

In [5]:

```
import matplotlib.pyplot as plt
%matplotlib inline

# The data is structured as follows:
# frogweight = (weight of frog1, weight of frog2, weight of frog3, ...)
# frogage = (age of frog1, age of frog2, age of frog3, ...)

frogweight = (100,105,106,105,107,105,111,112,104,122,80,111,75,101)
frogage= (3, 5, 5, 4, 6, 8, 10, 5, 3, 7, 2, 4, 3, 5)
plt.scatter(frogage, frogweight, color = "green", marker = "x")
plt.xlabel("age (years)")
plt.ylabel("weight (g)")
```

Out[5]:

```
<matplotlib.text.Text at 0x1de123d8c88>
```



The pie chart

A classic among the visualizations, which is very well suited to display percentages, is the cake or pie chart. As a scientific diagram it is only conditionally suitable, since concrete sizes are difficult to read. For presentations, or to get a rough overview of data, it is however quite suitable:

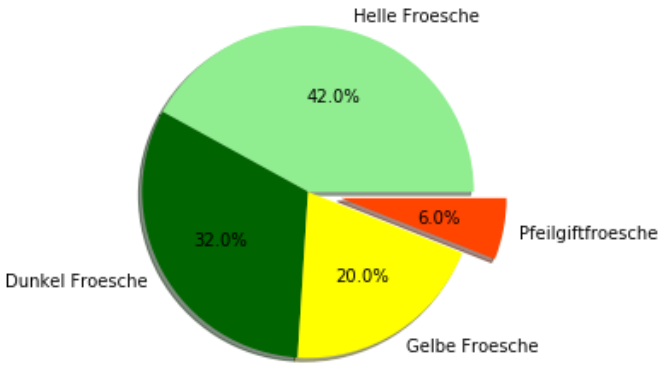
In [6]:

```
import matplotlib.pyplot as plt
%matplotlib inline

labels = ('light frogs', 'dark frogs', 'yellow frogs', 'poison dart frog')
colors = ("lightgreen","darkgreen","yellow", "orangered")
number = [42, 32, 20, 6]
explode = (0, 0, 0, 0.2)  # Only poison dart frogs are highlighted

plt.pie(number, explode = explode, labels = labels, autopct = '%1.1f%%',
        shadow = True, colors = colors)
```

```
plt.axis('equal')  # squares the plot
```

Out[6]:

```
(-1.1098228888537658,
 1.3062803190020018,
 -1.1130371722185557,
 1.1255162745181304)
```



Sub plotsSubplots

You often want to display several graphics next to each other. So-called `subplots` are suitable for this. The syntax is a little bit difficult to get used to. Subplots are always defined with three numbers, where for example the sequence of numbers (2, 3, 4) - or alternatively simply (234) - means that all plots to be displayed are arranged in 2 rows and 3 columns and currently (with the last number 4) the 4th of 6 plots (2 times 3) is addressed.

When defining subplots, `fig = plt.figure(figsize=(10, 3))` is often used to define a general drawing area. `figsize` defines the size of the entire drawing area, and the `subplots_adjust(hspace = 0.5)` command allows to define the distances between subplots, in this example the height distances (`hspace`). Commands such as `ax1 = fig.add_subplot(2, 3, 1)` are used to insert the respective subplot into the general drawing area - in this case, subplot `ax1` is placed at the first position of the plots.

Note: if you define an own plot name instead of the `matplotlib.pyplot` abbreviation `plt`, like `ax1` in this example, all plot labels must be `set_`, for example `ax1.set_title('Plot 1')`.

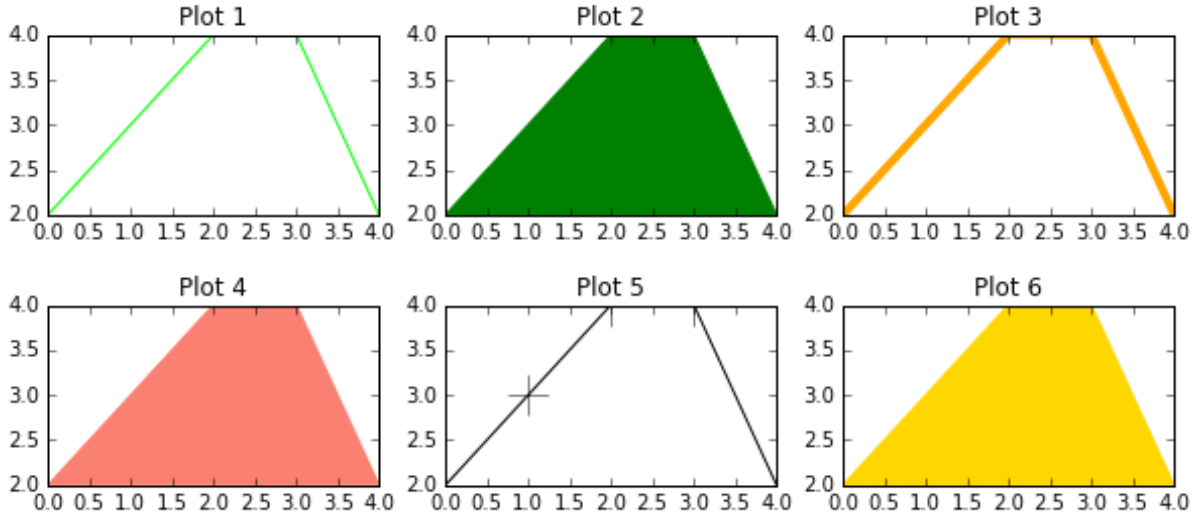
In [8]:

```
import matplotlib.pyplot as plt
%matplotlib inline

testdata=[2,3,4,4,2]

# define a general drawing area
fig = plt.figure(figsize=(10, 4))
# define the distance in height between the sub plots
plt.subplots_adjust(hspace = 0.5)

# add the first sub plot tot he general drawing area
ax1 = fig.add_subplot(2, 3, 1)
ax1.plot(testdata, color = "lime")
ax1.set_title('Plot 1')
```

```
# second sub plot
ax2 = fig.add_subplot(2, 3, 2)
ax2.fill(testdata, color = "green")
ax2.set_title('Plot 2')
# third sub plot
ax3 = fig.add_subplot(2, 3, 3)
ax3.plot(testdata, color = "orange", lw=4)
ax3.set_title('Plot 3')
# fourth sub plot
ax4 = fig.add_subplot(2, 3, 4)
ax4.fill(testdata, color = "salmon")
ax4.set_title('Plot 4')
# fifth sub plot
ax5 = fig.add_subplot(2, 3, 5)
ax5.plot(testdata, color = "black", marker="+", ms=20)
ax5.set_title('Plot 5')
# sixth sub plot
ax6 = fig.add_subplot(2, 3, 6)
ax6.fill(testdata, color = "gold")
ax6.set_title('Plot 6')
```

Out[8]:

```
<matplotlib.text.Text at 0xc54c8d0>
```



The polar diagram

Although the polar diagram looks similar to a pie chart, it has a fundamentally different meaning. Important are the angles: each angle (0 - 360 degrees) is assigned a certain size. In physics, for example, this plot is used to show in which direction an object emits how much radiation. As a simpler example, you could also compare the field of view of different animals:

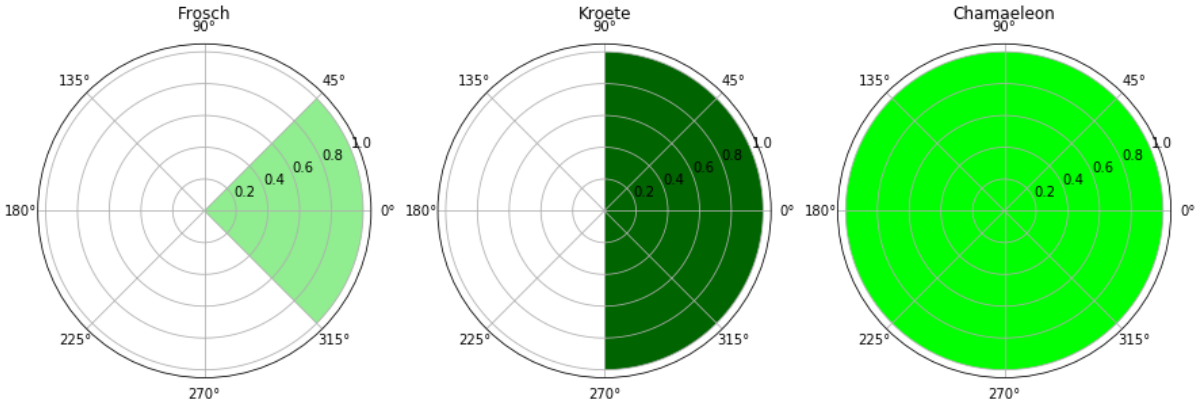
In [7]:

```
import matplotlib.pyplot as plt
%matplotlib inline

# define a general drawing area
fig = plt.figure(figsize=(15,5))
```

```
# add a sub plot in the general drawing area, the first one in a single-row
line with three places = (1, 3, 1)
ax1 = fig.add_subplot(1, 3, 1, projection='polar')
# the plot is initialized as polar plot
ax1.bar(0, 1, 3.1415/2, bottom = 0.0, color = "lightgreen")
# the syntax for a polar plot is as follows:
# plt.bar(By how many degrees is the surface centered, radius of the surfac
e, angle of the surface)
# Note: The angles here are not given in degrees, but as radians
# so 360° = 2 * pi ~ 2 * 3.1415
ax1.set_title("Frog")  # Note: The labeling must be done with set_ when usi
ng sub plots, for example with set_title

# add another sub plot, the second one in a single-row line with three plac
es (1, 3, 2)
ax2 = fig.add_subplot(1, 3, 2, projection='polar')
ax2.bar(0, 1, 3.1415, bottom = 0.0, color = "darkgreen")
ax2.set_title("toad")

# add another sub plot, the third one in a single-row line with three place
s (1, 3, 3)
ax3 = fig.add_subplot(1, 3, 3, projection='polar')
ax3.bar(0, 1, 2 * 3.1415, bottom = 0.0, color = "lime")
ax3.set_title("Chameleon")
```

Out[7]:

```
<matplotlib.text.Text at 0x1de1248ca58>
```



Histograms

Histograms are useful for showing how often a certain value occurs in a list. Suppose we know that there was a population explosion in our frog pond, but we do not know exactly in which year it occurred. To answer this question, we catch all frogs in the pond and determine their age. To answer our research question, we then present the resulting list as a histogram:

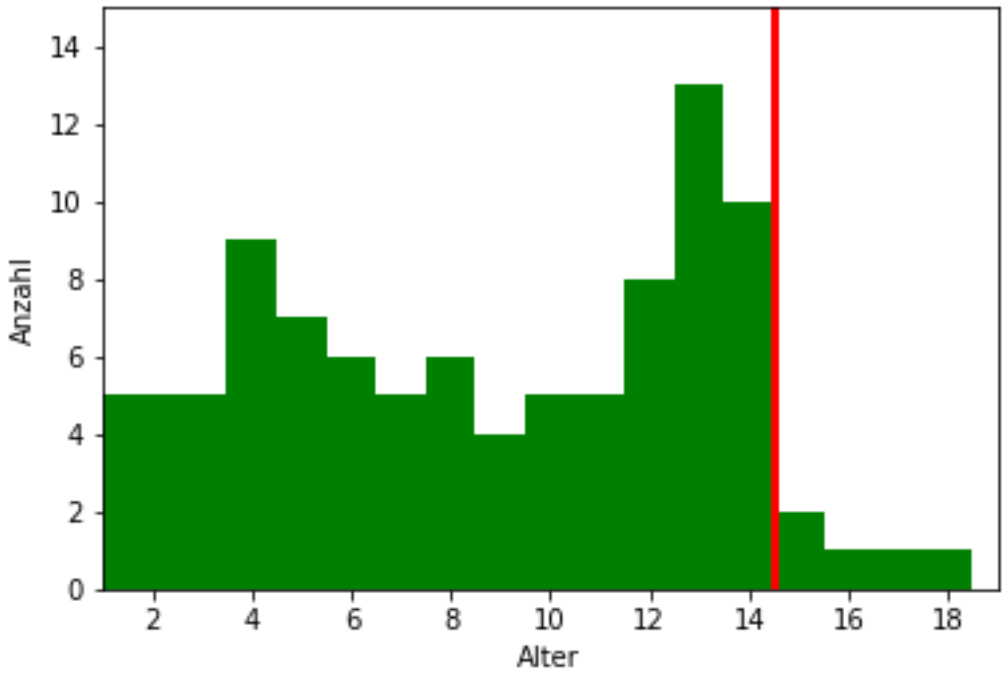
In [8]:

```
import matplotlib.pyplot as plt
%matplotlib inline

frogage=(15,16,15,17,14,13,11,14,14,12,13,1,1,2,3,4,4,5,6,2,1,1,1,3,4,5,14,
14,10,10,14,14,11,11,14,
```

```
             14,13,12,14,13,12,11,11,10,7,6,8,9,13,13,13,13,5,4,3,2,4,5,6,7
,8,12,12,9,4,12,2,3,4,5,6,7,8,9,10,10,
            6,7,8,4,5,9,8,2,3,4,5,6,7,8,13,13,12,12,13,13,13,19,18)

# Syntax: plt.hist(list, number of bars, smallest and largest value, color)
plt.hist(frogage, 18, range = (0.5, 18.5), color = "green")
plt.xlim(1,19)
plt.ylim(0,15)

# draw a red vertical line to mark the population explosion about 15 years
ago
plt.plot((14.5, 14.5),(0, 16), color = "red", lw = 3)
plt.ylabel("number")
plt.xlabel("age")
```

Out[8]:

```
<matplotlib.text.Text at 0x1de12481588>
```



We can see from the histogram that many frogs in the pond are currently 14 years old, but only very few are 15 years old. So 14 years ago there were much more frogs born than 15 years ago. So the population explosion obviously took place about 15 years ago.

Box and violin plots

Box and violin plots, similar to histograms, are suitable for displaying distributions. Furthermore, box plots allow to compare several distributions with each other. They show the mean value of a distribution (red line), the range in which the majority of the values lie (black box), but also the extreme values and outliers.

Violin plots are very similar, with the difference that they provide more detailed information about the distribution. Similar to the histogram, they show (highly smoothed) in which areas there are many and in which there are few values.

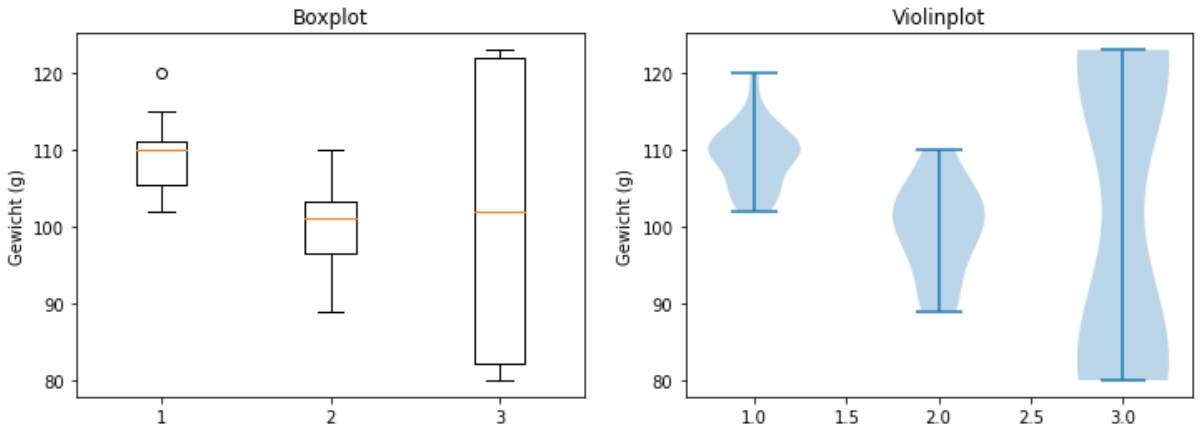
In the following we compare the weight of the frogs in 3 different ponds, once with box plots and once with violin plots:

In [9]:

```
import matplotlib.pyplot as plt
%matplotlib inline

pond1=(110,105,106,104,110,115,113,112,113,103,104,110,109,108,105,102,110,
110,110,111,109,111,120)
pond2=(90,99,100,102,103,103,104,102,110,90,95,98,97,93,98,110,105,106,102,
89)
pond3=(120,122,123,122,120,122,123,122,119,80,85,80,81,82,83,82,84,85)

fig = plt.figure(figsize=(12,4))
# first plot
ax1 = fig.add_subplot(1, 2, 1)
ax1.boxplot((pond1, pond2, pond3))
ax1.set_title("box plot")
ax1.set_ylabel("weight (g)")

# second Plot
ax2 = fig.add_subplot(1, 2, 2)
ax2.violinplot((pond1, pond2, pond3))
ax2.set_title("violin plot")
ax2.set_ylabel("weight (g)")
```

Out[9]:

```
<matplotlib.text.Text at 0x1de12898eb8>
```



Heat maps

When plotting three-dimensional data, one usually wants to assign an additional zz value to each x/yx/y data pair, just as geo-coordinates on topological maps are assigned a sea level. In order to display these plots in an impressive way, in the following we import ready-made sample data from the `matplotlib.cbook` using the command `get_sample_data`.

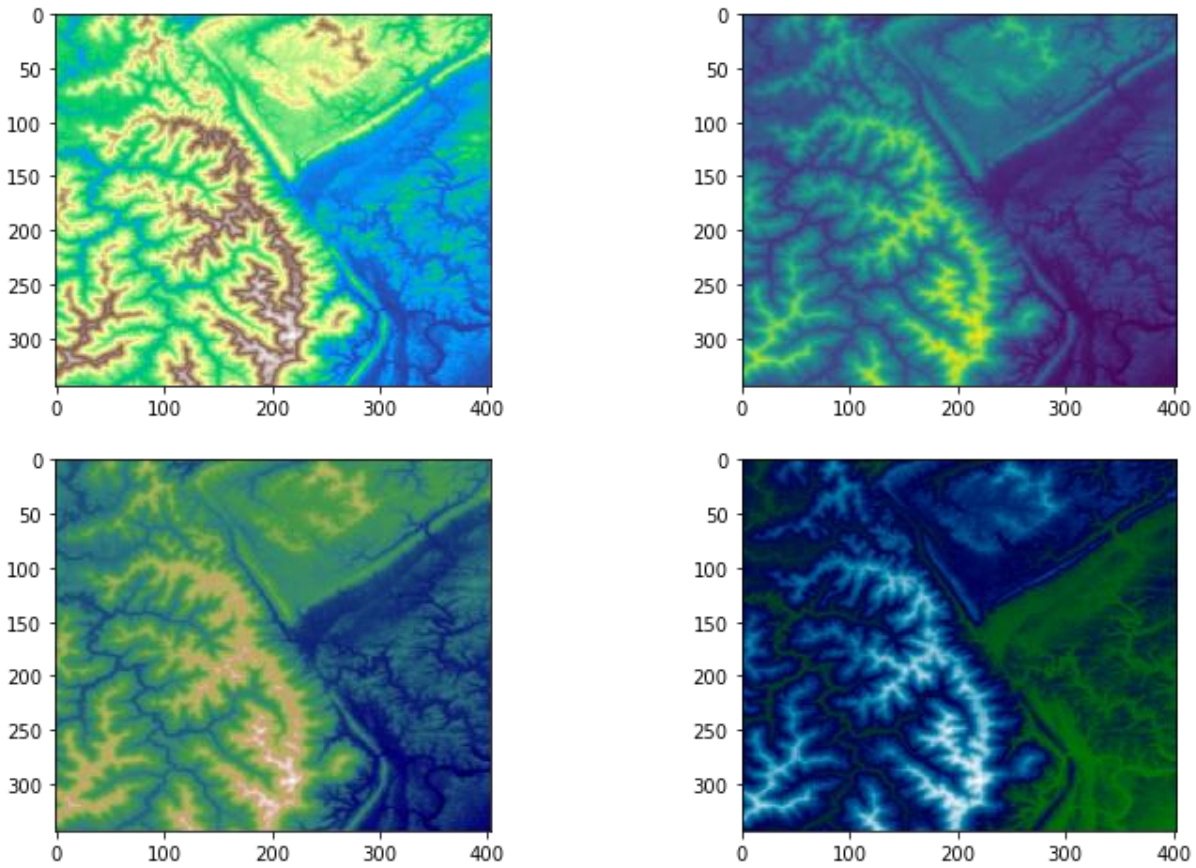
In [10]:

```
import matplotlib.pyplot as plt
%matplotlib inline
# import sample data
```

```
from matplotlib.cbook import get_sample_data
import numpy as np

data = np.load(get_sample_data('jacksboro_fault_dem.npz')) # loads the samp
le data
z = data['elevation'] # savest he information on the hight oft he data as v
ariable z

fig = plt.figure(figsize=(12, 8))
ax1 = fig.add_subplot(2, 2, 1)
ax1.imshow(z, cmap = plt.cm.terrain)     # colormap terrain: for example for
maps
ax2 = fig.add_subplot(2, 2, 2)
ax2.imshow(z, cmap = plt.cm.viridis)     # colormap viridis: for abstract da
ta
ax3 = fig.add_subplot(2, 2, 3)
ax3.imshow(z, cmap = plt.cm.gist_earth) # colormap earth: for realistic map
s
ax4 = fig.add_subplot(2, 2, 4)
ax4.imshow(z, cmap = plt.cm.ocean)       # colormap ocean: shades of blue fo
r marine images
```

Out[10]:

```
<matplotlib.image.AxesImage at 0x1de12a8f2b0>
```



Other visualizations

Besides the possibilities presented here, there are many more ways to visualize data. A good overview is provided by the `matplotlib` plot gallery at https://matplotlib.org/gallery.html, which also provides the code used to create the plot for each plot.

There are also other specialized Python packages dedicated to data visualization. Examples are `seaborn`, for displaying statistical data, or `bokeh`, which allows interactive visualizations in the browser.

Summary

Line plots

Line plots are simple plots that are used whenever a size is to be clearly assigned to another size.

Scatter plots

Scatterplots are used to display data pairs (e.g. age and weight) that belong together without a clear assignment. Example: a three-year-old frog does not always have to weigh exactly 111 grams.

Histograms, box and violin plots

Histograms as well as box or violin plots are used to display distributions. While a histogram shows an exact distribution, box and violin plots allow to compare several distributions. Box plots show important characteristics clearly (mean value, standard deviation, outliers), violin plots show more details of the distribution.

Sub plots

Subplots allow to display several graphics next to each other. Subplots are always denoted by three numbers: `subplot(a, b, c)`, where the graphics are organized in a matrix with a rows and b columns and the currently addressed plot has the index c.

Chapter 14 – Data processing with Pandas

Python can also be used to analyze and process large amounts of data. No matter how the data is available (csv, excel, ...), many data types can be imported and processed. The Python package that is usually used for this is called `pandas`.

We'll get to know pandas a little bit by examining the World Happiness Report. This data set, and many many more, can be found for free for example on Kaggle. There you can also find data sets about all apps in the Google Play Store, movies, stock market data development indicators or the surface temperature of the earth.

First, we will import the data set. This works on the one hand with data on your own hard disk, but also with files on the internet:

In [3]:

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

url="https://s3.amazonaws.com/happiness-report/2018/WHR2018Chapter2OnlineData.xls"

df=pd.read_excel(url)
```

Now the data of the World-Happiness-Report 2018 are stored under `df`. This abbreviation stands for DataFrame, which is the data type that pandas use. You can imagine it like an Excel spreadsheet: The data is sorted by rows and columns and can have as entries either numbers or strings.

Since the data set is relatively large, it makes little sense to print all data at once. But to get an insight into the dataset and to confirm that the import worked, you can use the command `head` to display the first few entries of the dataset.

In [4]:

```
df.head()
```

Out[4]:

	country	year	Life Ladder	Log GDP per capita	Social support	Healthy life expectancy at birth	Freedom to make life choices	Generosity	Perceptions of corruption
0	Afghanistan	2008	3.723590	7.168690	0.450662	49.209663	0.718114	0.181819	0.881686
1	Afghanistan	2009	4.401778	7.333790	0.552308	49.624432	0.678896	0.203614	0.850035
2	Afghanistan	2010	4.758381	7.386629	0.539075	50.008961	0.600127	0.137630	0.706766
3	Afghanistan	2011	3.831719	7.415019	0.521104	50.367298	0.495901	0.175329	0.731109
4	Afghanistan	2012	3.782938	7.517126	0.520637	50.709263	0.530935	0.247159	0.775620

Positive affect	Negative affect	Confidence in national government	Democratic Quality	Delivery Quality	Standard deviation of ladder by country-year	Standard deviation/Mean of ladder by country-year	GINI index (World Bank estimate)	GINI index (World Bank estimate), average 2000-15	gini of household income reported in Gallup, by wp5-year
0.517637	0.258195	0.612072	-1.929690	-1.655084	1.774662	0.476600	NaN	NaN	NaN
0.583926	0.237092	0.611545	-2.044093	-1.635025	1.722688	0.391362	NaN	NaN	0.441906
0.618265	0.275324	0.299357	-1.991810	-1.617176	1.878622	0.394803	NaN	NaN	0.327318
0.611387	0.267175	0.307386	-1.919018	-1.616221	1.785360	0.465942	NaN	NaN	0.336764
0.710385	0.267919	0.435440	-1.842996	-1.404078	1.798283	0.475367	NaN	NaN	0.344540

This way we can already see what data appears here. Each line contains the name of the country and the year, followed by many possible indicators of happiness, such as life expectancy or trust in the government.

We get a more detailed overview with the command `describe`:

In [5]:

```
df.describe()
```

Out[5]:

	year	Life Ladder	Log GDP per capita	Social support	Healthy life expectancy at birth	Freedom to make life choices	Generosity	Perceptions of corruption	Positive affect
count	1562.000000	1562.000000	1535.000000	1549.000000	1553.000000	1533.000000	1482.000000	1472.000000	1544.000000
mean	2011.820743	5.433676	9.220822	0.810669	62.249887	0.728975	0.000079	0.753622	0.708969
std	3.419787	1.121017	1.184035	0.119370	7.960671	0.145408	0.164202	0.185538	0.107644
min	2005.000000	2.661718	6.377396	0.290184	37.766476	0.257534	-0.322952	0.035198	0.362498
25%	2009.000000	4.606351	8.310665	0.748304	57.299580	0.633754	-0.114313	0.697359	0.621471
50%	2012.000000	5.332600	9.398610	0.833047	63.803192	0.748014	-0.022638	0.808115	0.717398
75%	2015.000000	6.271025	10.190634	0.904329	68.098228	0.843628	0.094649	0.880089	0.800858
max	2017.000000	8.018934	11.770276	0.987343	76.536362	0.985178	0.677773	0.983276	0.943621

Negative affect	Confidence in national governmen t	Democratic Quality	Delivery Quality	Standard deviation of ladder by country-year	Standard deviation/M ean of ladder by country-year	GINI index (World Bank estimate)	GINI index (World Bank estimate), average 2000-15	gini of household income reported in Gallup, by wp5-year
1550.000000	1401.000000	1391.000000	1391.000000	1562.000000	1562.000000	583.000000	1386.000000	1205.000000
0.263171	0.480207	-0.126617	0.004947	2.003501	0.387271	0.372846	0.386948	0.445204
0.084006	0.190724	0.873259	0.981052	0.379684	0.119007	0.086609	0.083694	0.105410
0.083426	0.068769	-2.448228	-2.144974	0.863034	0.133908	0.241000	0.228833	0.223470
0.204116	0.334732	-0.772010	-0.717463	1.737934	0.309722	0.307000	0.321583	0.368531
0.251798	0.463137	-0.225939	-0.210142	1.960345	0.369751	0.349000	0.371000	0.425395
0.311515	0.610723	0.665944	0.717996	2.215920	0.451833	0.433500	0.433104	0.508579
0.704590	0.993604	1.540097	2.184725	3.527820	1.022769	0.648000	0.626000	0.961435

This command calculates relevant indicators for each column, such as the number of entries (count), the mean value (mean), the standard deviation (std), but also more detailed information about the distribution. Columns where these calculations are not possible (e.g. country name) are omitted. So, we see for example that the mean healthy life expectancy is about 62 years, but there is also at least one entry in the database where this value is below 40 years. Of course, it would be especially interesting to find out which entries these are.

Such queries (i.e. all rows where the column X has a value of Y or less) are very easily possible with pandas. Wir schreiben:

```
df[df[NameDerSpalte] < Mindestwert]
```

In [10]:

```
df[df["Healthy life expectancy at birth"] < 40]
```

Out[10]:

	country	year	Life Ladder	Log GDP per capita	Social support	Healthy life expectancy at birth	Freedom to make life choices	Generosity	Perceptions of corruption
248	Central African Republic	2007	4.160130	6.761289	0.532297	38.385059	0.662871	0.099451	0.782131
1207	Sierra Leone	2006	3.628185	6.975739	0.561356	37.766476	0.679001	0.113010	0.836166
1208	Sierra Leone	2007	3.585127	7.025132	0.686471	38.560307	0.720373	0.259907	0.830483
1209	Sierra Leone	2008	2.997251	7.053099	0.590737	39.351990	0.716396	0.160101	0.924901
1550	Zimbabwe	2006	3.826268	7.366704	0.821656	39.087681	0.431110	-0.053216	0.904757

Positive affect	Negative affect	Confidence in national government	Democratic Quality	Delivery Quality	Standard deviation of ladder by country-year	Standard deviation/Mean of ladder by country-year	GINI index (World Bank estimate)	GINI index (World Bank estimate), average 2000-15	gini of household income reported in Gallup, by wp5-year
0.567980	0.329995	0.623566	-1.468937	-1.397169	1.656170	0.398105	NaN	0.499	NaN
0.505072	0.380655	0.541412	-0.314346	-1.065230	1.819301	0.501436	NaN	0.371	NaN
0.581781	0.289842	0.561873	-0.139665	-1.031073	1.793122	0.500156	NaN	0.371	NaN
0.533604	0.369601	0.687578	-0.189854	-1.007436	1.688531	0.563360	NaN	0.371	NaN
0.715229	0.297147	0.317073	-1.236102	-1.570760	2.013538	0.526241	NaN	0.432	NaN

Similarly, we can restrict the data set to a specific year, for example, the most recent value of the data set, the year 2018:

In [4]:

```
df[df["year"] == 2017].head()
```

Out[4]:

	country	year	Life Ladder	Log GDP per capita	Social support	Healthy life expectancy at birth	Freedom to make life choices	Generosity	Perceptions of corruption
9	Afghanistan	2017	2.661718	7.460144	0.490880	52.339527	0.427011	-0.106340	0.954393
19	Albania	2017	4.639548	9.373718	0.637698	69.051659	0.749611	-0.035140	0.876135
25	Algeria	2017	5.248912	9.540244	0.806754	65.699188	0.436670	-0.194670	0.699774
41	Argentina	2017	6.039330	9.843519	0.906699	67.538704	0.831966	-0.186300	0.841052
53	Armenia	2017	4.287736	9.034711	0.697925	65.125687	0.613697	-0.132166	0.864683

Positive affect	Negative affect	Confidence in national government	Democratic Quality	Delivery Quality	Standard deviation of ladder by country-year	Standard deviation/Mean of ladder by country-year	GINI index (World Bank estimate)	GINI index (World Bank estimate), average 2000-15	gini of household income reported in Gallup, by wp5-year
0.496349	0.371326	0.261179	NaN	NaN	1.454051	0.546283	NaN	NaN	0.286599
0.669241	0.333884	0.457738	NaN	NaN	2.682105	0.578096	NaN	0.303250	0.410488
0.641980	0.288710	NaN	NaN	NaN	2.039765	0.388607	NaN	0.276000	0.527556
0.809423	0.291717	0.305430	NaN	NaN	2.409329	0.398940	NaN	0.476067	0.394176
0.625014	0.437149	0.246901	NaN	NaN	2.325379	0.542333	NaN	0.325067	0.478877

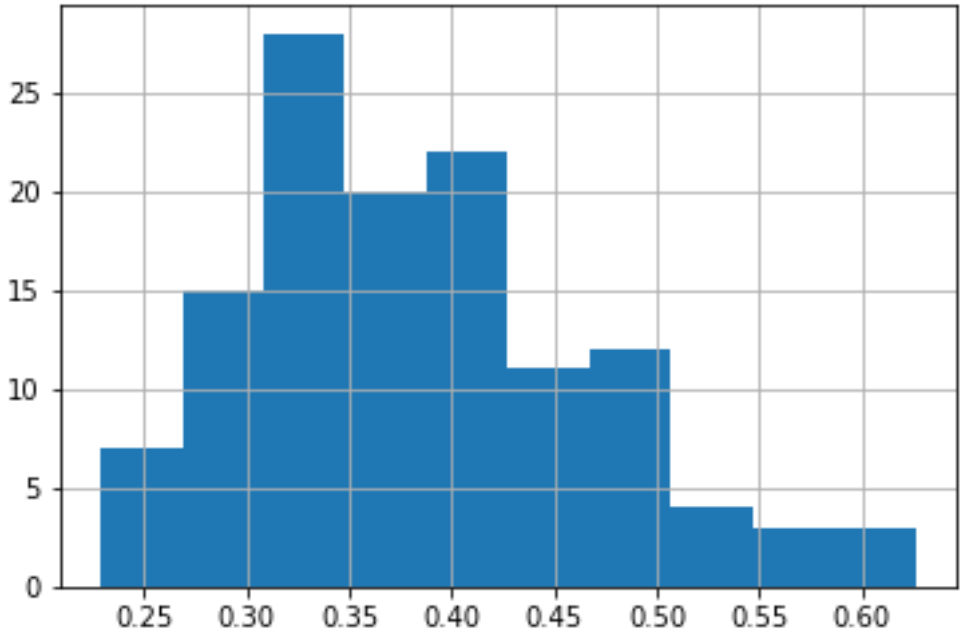
Often it is also useful to use visualizations to display the data. For example, we could display the GINI index as a histogram:

In [5]:

```
df[df["year"] == 2017]['GINI index (World Bank estimate), average 2000-15']
.hist()
```

Out[5]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x18350516978>
```



Scatterplots can be used to determine correlations and relationships. This makes it possible to plot one value on the y-axis and one on the x-axis. If a point cloud is created there is no correlation between these values. The sooner the distribution resembles a straight line, the stronger the correlation.

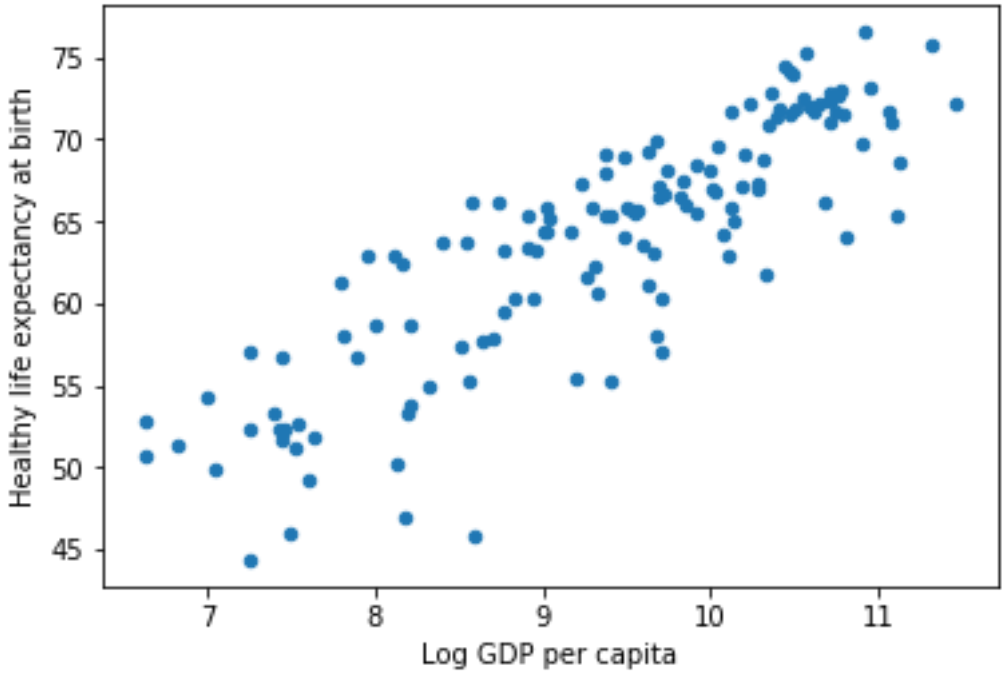
Let us consider the relationship between GDP/person and life expectancy on the one hand and the relationship between "social support" and life expectancy on the other.

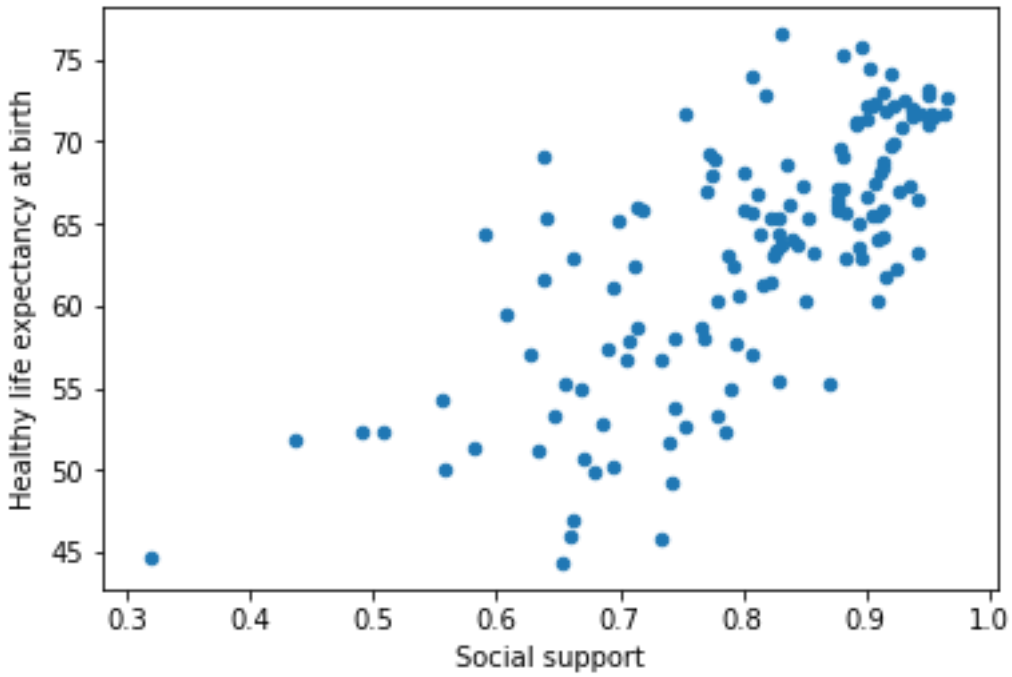
In [6]:

```
df[df["year"] == 2017].plot.scatter(x="Log GDP per capita", y = "Healthy li
fe expectancy at birth")

df[df["year"] == 2017].plot.scatter(x="Social support", y = "Healthy life e
xpectancy at birth")
```

Out[6]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x18350616278>
```





Of course you don't need the complete data set for every analysis. For more detailed considerations it makes sense to limit the data. So we could only look at Germany, Austria and Switzerland.

First we look at the average level of corruption for these countries over all years. For this we use the command `mean()` and write the result directly to the screen with `print`.

In [7]:

```
print("Corruption Austria:    ",df[df["country"]=="Austria"]["Perceptions of corruption"].mean())
print("Corruption Germany:    ",df[df["country"]=="Germany"]["Perceptions of corruption"].mean())
```

```
print("Corruption Switzerland:",df[df["country"]=="Switzerland"]["Perceptio
ns of corruption"].mean())
```

```
Corruption Austria:     0.596848064661026
Corruption Germany:     0.614804724852244
Corruption Switzerland: 0.3119955360889435
```

Since we have limited ourselves to three countries, we can now carry out detailed analyses without the results becoming confusing. Instead of the mean value, we can therefore look at the corruption index over time.

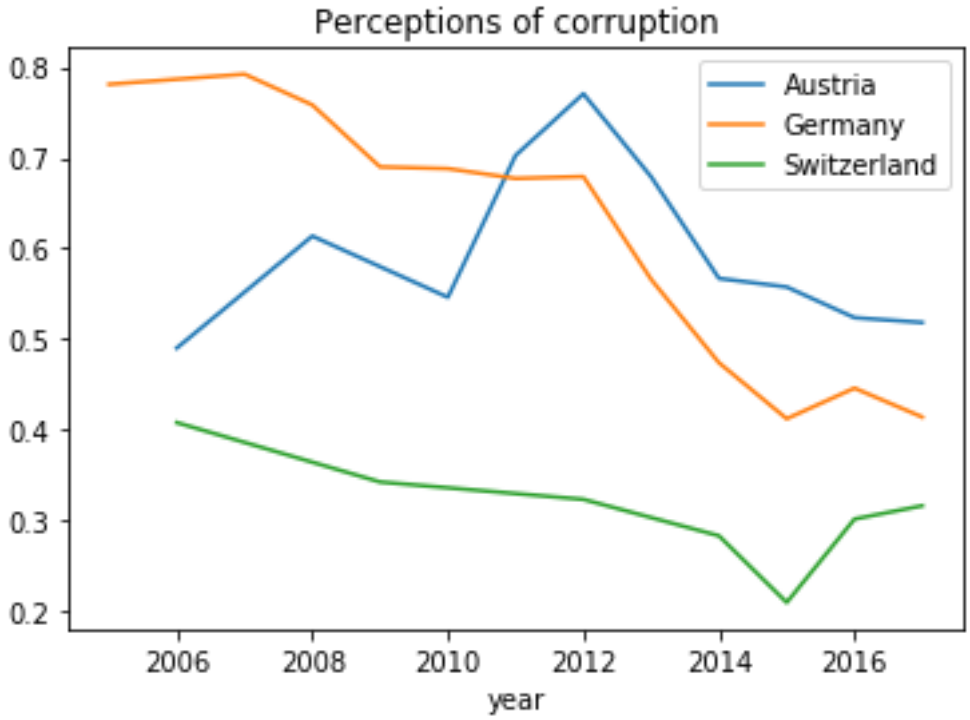
For this we use the `plot` command from pandas, which is a bit different from the one from matplotlib. As x and y, for example, we don't pass a list, but only the name of the column under which we find the relevant data points. Furthermore, this command always creates a new axis. To suppress this behavior so that we can see all data in a figure, we have to give the axis a name as soon as it is created (ax1) and tell all further plot commands with `ax=ax1` to continue using the existing axis.

In [8]:

```
ax1=df[df["country"]=="Austria"].plot(x="year",y="Perceptions of corruption
",label="Austria")
df[df["country"]=="Germany"].plot(x="year",y="Perceptions of corruption",la
bel="Germany",ax=ax1)
df[df["country"]=="Switzerland"].plot(x="year",y="Perceptions of corruption
",label="Switzerland",ax=ax1)
plt.title("Perceptions of corruption")
```

Out[8]:

```
Text(0.5,1,'Perceptions of corruption')
```



Pandas offers of course many more possibilities to analyze, process and prepare data. A complete overview can be found in the Pandas User Guide.

Summary

Reading Data with Pandas

Pandas can read a variety of data types and convert them into dataframes. The most important commands are `pd.read_excel` and `pd.read_csv`.

Selecting entries with certain properties

You can restrict a dataframe to lines with certain properties by using

```
df[df[spaltenname]==wert]
```

Analogously this works also with <= and >=.

To not select the whole row, but only the entry in a certain column, write

```
df[df[spaltenname]==wert][spaltenname2]
```

Plotting with Pandas

There are many ways to visualize data with pandas:

- Histograms (`df.hist()`)
- Scatterplots (`df.plot.scatter(x = spalte1,y = spalte2)`)
- Line plots (`df.plot(x = spalte1,y = spalte2)`)

Chapter 15 - Analytical differentiation and integration with the `sympy` package

So far we have always done numerical calculations with Python. This means that all differentials were approximated by the computer as small differences and irrational numbers, such as roots, were simply rounded. This is the simplest way a computer can do mathematics, but not the only one. With Python it is also possible to solve mathematical problems analytically, i.e. the way a human being would solve them. This way we can calculate primitives, solve equations and solve complex mathematical problems that would take a human being days to solve.

Let's start by taking a closer look at the difference between numerical and analytical calculations. As an example, let's take a simple process, drawing a square root. Let's calculate the square root of 9 with the standard math package `math`.

In [1]:

```
import math
math.sqrt(9)
```

Out[1]:

```
3.0
```

This works without any problems, exactly the same as with any calculator. As we all know, the square root of 9 is exactly 3, because 3 times 3 is 9, so there is no need to round and we can be satisfied with our result. But what if we are interested in the root of 8, which is known to have no integer solution?

In [2]:

```
import math
math.sqrt(8)
```

Out[2]:

```
2.8284271247461903
```

Here we get an approximate solution, but this result is no longer correct. The root of 8 is not exactly 2.8284271247461903, there would follow an infinite number of decimal places. Why can this become a problem? Let us try to multiply the square root of 8 by the square root of 8. We (as humans) do not need to know the root of 8 for this, we know that the root of 8 times the root of 8 must be 8 again.

$$\sqrt{8} * \sqrt{8} = \sqrt{8 * 8} = \sqrt{8^2} = 8$$

Root and square cancel each other out, so to speak. If we understand this connection without being able to draw the square root in our heads, then a computer should have no problem with this task, right?

In [3]:

```
import math
math.sqrt(8) * math.sqrt(8)
```

Out[3]:

```
8.000000000000002
```

Apparently so. Python claims rock-solid that the square root of 8 squared is a bit larger than 8. The error is small, but you have to take it seriously. In a large program, such calculations are often performed one after the other, and the error builds up. And of course such errors occur not

only when drawing the root, but in every operation that has to round in some way. Even the number 1/3 can no longer be stored exactly, because you would need an infinite number of digits to be exact if you wanted to display it as 0.3333333333333.....

How can this problem be solved? With analytical calculations! If Python would understand what 1/3 means, i.e. not only a number with a finite number of digits, but the arithmetic operation, which divides the number 1 into 3 equal parts, we would no longer need rounds. It would be even nicer if Python would also understand the operation of drawing a root, so that the square of a root would automatically always result in the number itself. All this, and much more is possible with the Python package `sympy`.

We will take a closer look at this package in the following and use it first to draw a root:

In [4]:

```
import sympy
sympy.sqrt(9)
```

Out[4]:

```
3
```

Our first pleasant result: The root of 9 is still 3, but what happens to the root of 8?

In [5]:

```
import sympy
sympy.sqrt(8)
```

Out[5]:

```
2*sqrt(2)
```

This result looks strange, but is absolutely correct. The root of 8 is exactly twice as large as the root of 2.
For mathematicians:

$$\sqrt{8} = \sqrt{4 * 2} = \sqrt{4} * \sqrt{2} = 2 * \sqrt{2}$$

The advantage we have here is obvious: This result is exact, we do not have to round. And if we want, we can still convert the result to a decimal number:

In [6]:

```
import sympy
float(sympy.sqrt(8))
```

Out[6]:

```
2.8284271247461903
```

Now we could check if squaring a root works better this way:

In [7]:

```
import sympy
sympy.sqrt(8) * sympy.sqrt(8)
```

Out[7]:

```
8
```

Obviously. The advantage of this calculation is still small, because the square root of 8 can also be transformed in the head (see the calculation above). This is where the advantage of a computer comes into play: If the principle works, more complicated calculations are not really more difficult than simple calculations, as the following example shows:

In [8]:

```
import sympy
sympy.sqrt(1264)
```

Out[8]:

```
4*sqrt(79)
```

Symbolic Programming

So, the root of 1264 is 4 times the root of 79. Even with a lot of practice, such calculations cannot easily be done in the head. The `sympy` package can do this - and much more. It becomes especially exciting when we define variables ourselves. For this there is the command `symbols`, which explains to Python that these variables are general variables, in which no value is stored. Nevertheless, Python can calculate with them. We can store an expression that contains x and y and Python can work with it, although we have not assigned values to x and y, but only use them as general **symbols**:

In [9]:
```
from sympy import *
x = symbols("x")
y = symbols("y")
ausdruck = 3 * x - 4 * y
ausdruck
```

Out[9]:
```
3*x - 4*y
```

In [10]:
```
ausdruck - 1
```

Out[10]:
```
3*x - 4*y - 1
```

In [11]:
```
ausdruck + 3 * y
```

Out[11]:
```
3*x - y
```

In [12]:
```
ausdruck * x # the solution is not yet multiplied out here
```

Out[12]:
```
x*(3*x - 4*y)
```

In [13]:
```
expand(ausdruck * x) # expand performs the mutliplication
```

Out[13]:
```
3*x**2 - 4*x*y
```

You see immediately: Python now really understands the arithmetic operations, and although no value has been assigned to the variable x, it is clear that x times x is the same as x-square. We can use this, for example, to simplify very complicated expressions with the command `factor`:

In [14]:
```
from sympy import *
x = symbols("x")
y = symbols("y")
z = symbols("z")
ausdruck = -x * y * z**2 + 2 * x * y * z - x * z**2 + 2 * x * z + y * z**2
- 2 * y * z + z**2 - 2 * z
factor(ausdruck)
```

Out[14]:
```
-z*(x - 1)*(y + 1)*(z - 2)
```

The complicated expression

```
-x * y * z**2 + 2 * x * y * z - x * z**2 + 2 * x * z + y * z**2 - 2 * y
* z + z**2 - 2 * z
```

can therefore be written compactly as

```
-z*(x - 1)*(y + 1)*(z - 2)
```

While this is helpful, in practice one will very rarely find oneself in the situation of having to carry out such simplifications (except perhaps in the context of mathematics courses). Much more often one would like to solve equations, for example.

Solving equations

While there are still formulas available to solve quadratic equations, 3rd-order equations, such as

$$3x^3 + x^2 - x = 0$$

are much more difficult to solve on paper. Faster is the `sympy`-command `solve`. This command only needs the equation and as second argument the variable to be calculated:

In [15]:

```
from sympy import *

x = symbols("x")
solve(3 * x**3 + x**2 - x , x)
```

Out[15]:

```
[0, -1/6 + sqrt(13)/6, -sqrt(13)/6 - 1/6]
```

The three solutions of the equation

$$3x^3 + x^2 - x = 0$$

are thus

$$x = 0, x = -\frac{1}{6} + \frac{\sqrt{13}}{6} \text{ and } x = -\frac{1}{6} - \frac{\sqrt{13}}{6}$$

Differentiation and integration

From the perspective of systems science, `sympy` shows its greatest advantage in the calculation of differentials and integrals. We have already discussed numerical differentiation and integration in previous chapters. However, the solutions there were always numbers or lists of numbers. General, analytical insights, such as the fact that the integral of cos(x) is exactly sin(x), could not be obtained in this way. With the help of sympy's `diff` and `integrate` commands, we can analytically perform very complicated derivations and integrals and get as solutions not numbers but complete, generally valid functions:

In [16]:

```
from sympy import *

x = symbols("x")
```

In [17]:

```
diff(sin(x))
```

Out[17]:

```
cos(x)
```

In [18]:

```
integrate(cos(x))
```

Out[18]:

```
sin(x)
```

In [19]:

```
diff(sin(3 * x) * x**2)
```

Out[19]:

```
3*x**2*cos(3*x) + 2*x*sin(3*x)
```

In [20]:

```
integrate(sin(x) * cos(x))
```

Out[20]:

```
sin(x)**2/2
```

In [21]:

```
diff(x**3 + log(x) * sin(x) + exp(x**2))
```

Out[21]:

```
3*x**2 + 2*x*exp(x**2) + log(x)*cos(x) + sin(x)/x
```

But attention: integrals, which are analytically not solvable, cannot be solved even by Python. Differentiation always follows very simple rules, and in principle you can derive any function with these rules. Integrating, however, is more a creative process, and this can lead to correlations that simply cannot be calculated. This has nothing to do with the fact that the right "calculation rules" have just not been found yet. It is rather a property of integral calculus itself. Unsolvable integrals need not be terribly complicated. Already the simple example $\frac{\sin(x)}{\log(x)}$ is not to integrate:

In [22]:

```
integrate(sin(x) / log(x))
```

Out[22]:

```
Integral(sin(x)/log(x), x)
```

If the above cell is executed, it shows that Python calculates for a while and obviously tries to solve the integral. However, if no partial integration, substitutions or other tricks help anymore, Python (and any other math software and the human brain) capitulates. As solution you get the statement:

$$\int \frac{\sin(x)}{\log(x)} dx = \int \frac{\sin(x)}{\log(x)} dx$$

This is of course (by definition) not wrong, but it doesn't really help us either. If we want to solve this integral in a certain range, we are again dependent on numerical methods. These do not provide nice functions, but only rounded numbers. But they really always work. So, you can see, that numerical as well as analytical methods have their right to exist, and you should choose one of the two approaches depending on the problem. Especially in the mathematical field, working with analytical functions has great advantages.

The application areas of `sympy` do not end with differentiation and integration. You can also use it to calculate limits, solve differential equations, calculate eigenvalues and eigenvectors or (if you really want to do so ;-) transform a Bessel function into a spherical Bessel function.

Matrix Calculations

`Sympy` also allows you to calculate with matrices, for example. All important mathematical operations are supported. Matrices are written in square brackets, where each line itself is written in square brackets. So Sympy considers matrices as lists of lists. The matrix

$$M = \begin{pmatrix} 1 & 3 & 2 \\ 1 & 1 & 1 \\ 2 & 3 & 1 \end{pmatrix}$$

can therefore be written as:

In [23]:

```
M = Matrix([[1,3,2],[1,1,1],[2,3,1]])
M
```

Out[23]:

```
Matrix([
[1, 3, 2],
[1, 1, 1],
[2, 3, 1]])
```

Basic arithmetic operations work quite intuitively with the corresponding operators +, * and -:

In [24]:

```
M + M
```

Out[24]:

```
Matrix([
[2, 6, 4],
[2, 2, 2],
[4, 6, 2]])
```

In [25]:

```
3 * M
```

Out[25]:

```
Matrix([
[3, 9, 6],
[3, 3, 3],
[6, 9, 3]])
```

In [26]:

```
M - M
```

Out[26]:

```
Matrix([
[0, 0, 0],
[0, 0, 0],
[0, 0, 0]])
```

Even matrix multiplication (often tedious by hand) can be handled easily with `sympy`.

In [27]:

```
M = Matrix([[1,3,2],[1,1,1],[2,3,1]])
N = Matrix([[4,1,2],[1,2,1],[1,2,4]])
```

```
M * N
```

Out[27]:

```
Matrix([
[ 9, 11, 13],
[ 6,  5,  7],
[12, 10, 11]])
```

In [28]:

```
N * M
```

Out[28]:

```
Matrix([
[ 9, 19, 11],
[ 5,  8,  5],
[11, 17,  8]])
```

As a quick check, if this is really a real matrix multiplication, and not just multiplying the elements together, you can have `M * N` calculated and compare the result with `N * M`. In our example we see that `M * N` is not the same as `N * M`, just as it should be for matrix multiplications.

But also extended arithmetic operations are possible. For example the calculation of determinants, which is needed to solve systems of equations with Cramer's rule (https://en.wikipedia.org/wiki/Cramer%27s_rule):

In [29]:

```
M.det()
```

Out[29]:

```
3
```

The calculation of eigenvalues is also possible:

In [30]:

```
M.eigenvals()
```

Out[30]:

```
{-1: 1, -sqrt(7) + 2: 1, 2 + sqrt(7): 1}
```

Summary

The `sympy` package allows to perform analytical calculations with Python. This has the advantage that numbers are not rounded, but fractions or roots can be used exactly. This not only avoids rounding errors, but also allows many operations which are numerically impossible or only approximate.

Solving equations

Solving equations numerically usually requires tricks and creativity. But with `sympy`, equations can be solved exactly. The symbols used in these equations are first defined with the command `symbols` and then solved with command `solve(gleichung, variable)`.

Derivatives

The derivation of a function is calculated with `diff`, where the variables have to be declared with `symbols` before.

Integration

Integrating, i.e. searching for the root function, is done with the command `integrate`.

Note: Not all functions can be integrated analytically. For many, there is no analytical solution. For such problems numerical methods have to be used. These require longer computing time and provide solutions only approximately and in certain intervals. But they usually work all the time.

Chapter 16 - Networks

Network research offers a particularly fruitful way of investigating the interactions between the components of a system. Similar to systems science, network research is applied in a wide variety of areas. Computers that communicate with each other form computer networks. People who are in contact with each other form social networks. Animals and plants that share a habitat form ecological networks. All these networks, although they actually seem to be fundamentally different, have common characteristics and follow similar rules.

In recent years, the concept of networks has gained enormously in importance in the sciences. It is therefore hardly surprising that Python, too, now offers specialized packages dedicated to the representation and analysis of networks. In the following we will take a closer look at one of these packages.

What are networks?

We speak of networks whenever objects exist as **nodes** (**_nodes_** or **_vertices_**), which are connected to other such objects in any way. The **connections** are called **_links_** or **_edges_**.

Networks are characterized by their connection structure, i.e. by the information about which nodes are connected to which other nodes in which way. This connection structure results in a variety of different network types, some of which are discussed in more detail below.

NetworkX

But first we will have a look at some basics of the Python package `NetworkX`, which is specialized on the representation and analysis of networks and is already included in the installed Anaconda package.

For the beginning we consider a very simple network: The network should consist of three people: Alice, Bob and Carol. These three people are the *nodes* of our network. All of them are friends with each other, which we represent in our "friendship network" through connections (*edges*).

We start by importing the `NetworkX` packet and first create an empty network. Then we fill it with *nodes* and *edges*:

In [5]:

```
import networkx as nx


G = nx.Graph() # create a new network with the name G
```

The next step is to add three nodes: Alice, Bob and Carol.

In [6]:

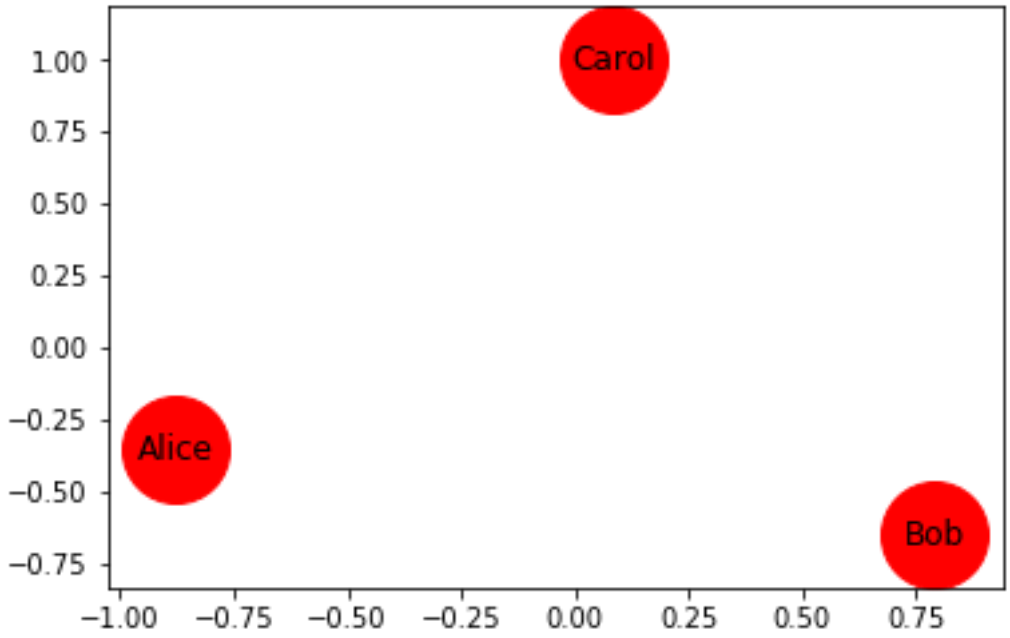
```
G.add_node("Alice") # add the node "Alice" to network G
G.add_node("Bob")
G.add_node("Carol")
```

To display the network graphically, we again need the `matplotlib` package, which works very well with `networkx`.

In [7]:

```
import matplotlib.pyplot as plt
%matplotlib inline
```

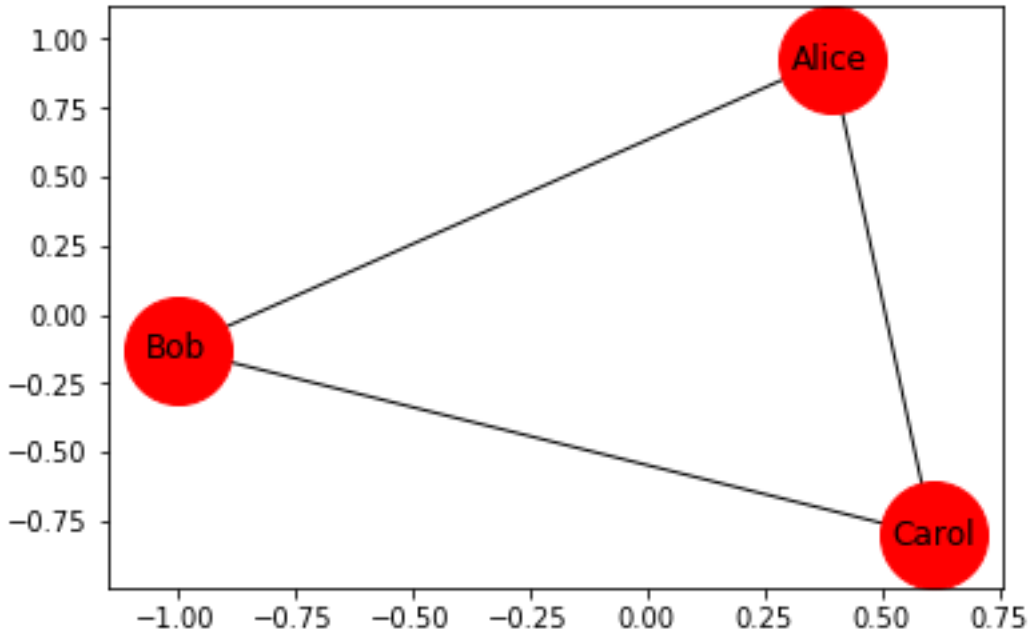
```
nx.draw_networkx(G, node_size = 1600) # draw the network G. The nodes shoul
d have the size 1600.
```



We see: the nodes are displayed correctly. What is still missing are the connecting lines between the nodes. These are added with the command `G.add_edge(from,to)`:

In [8]:

```
G.add_edge("Alice","Bob") # add a new connection between Alice and Bob to n
etwork G
G.add_edge("Bob","Carol")
G.add_edge("Carol","Alice")

nx.draw_networkx(G, node_size = 1600)
```

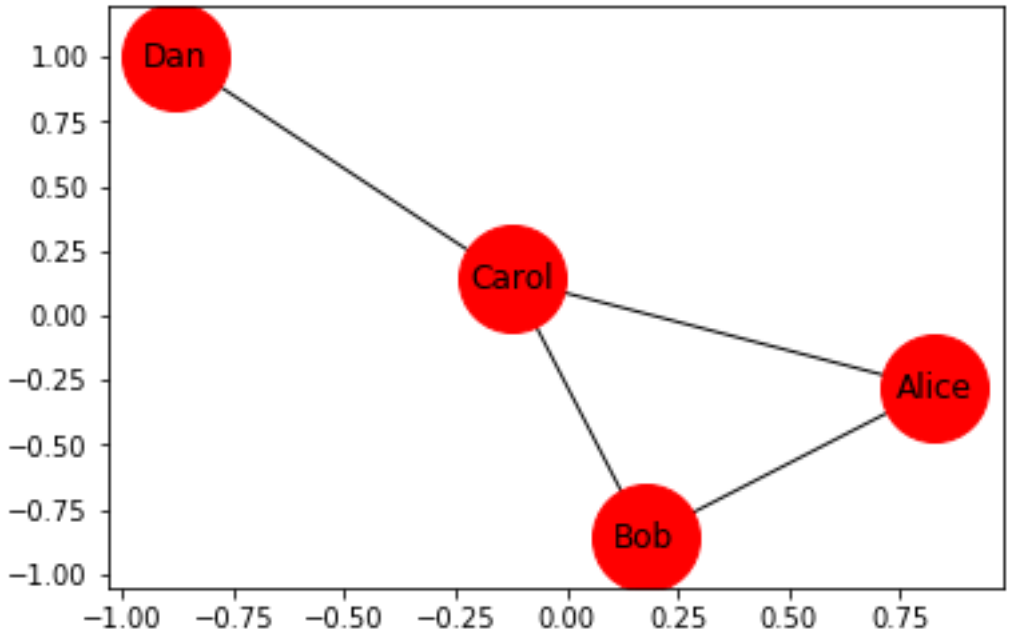


So, our (very simple) network is displayed correctly. If we execute the above code cell several times, we notice that the positions of the individual nodes are reassigned each time. They are generated randomly. In this representation they have no meaning, what is important is who is connected to whom, not where they are located.

Let's add one more person to our network: Dan. Dan is only friends with Carol, but he doesn't know anyone else from the network. So we make only one connection between Carol and Dan:

In [9]:

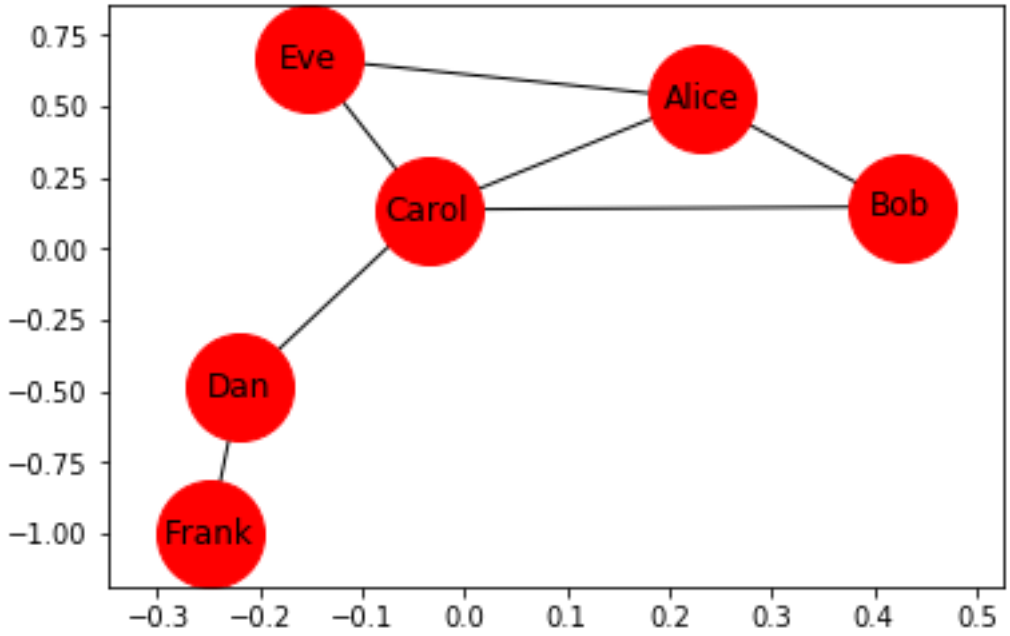
```
G.add_node("Dan")
G.add_edge("Carol","Dan")

nx.draw_networkx(G, node_size = 1600)
```



To expand our network a little bit more, we add two more people: Eve is a friend of Alice and Carol. Frank is only friends with Dan:

In [10]:

```
G.add_node("Eve")
G.add_edge("Eve","Alice")
G.add_edge("Eve","Carol")

G.add_node("Frank")
G.add_edge("Frank","Dan")

nx.draw_networkx(G, node_size = 1600)
```



This social network is now already sufficiently complex to try out different types of representation. Until now, the positions of the nodes in the graph were left to chance. But they can also be arranged with so-called "layouts". The nodes are then arranged according to certain rules, which can clarify special network properties, or simply make the networks generally clearer.

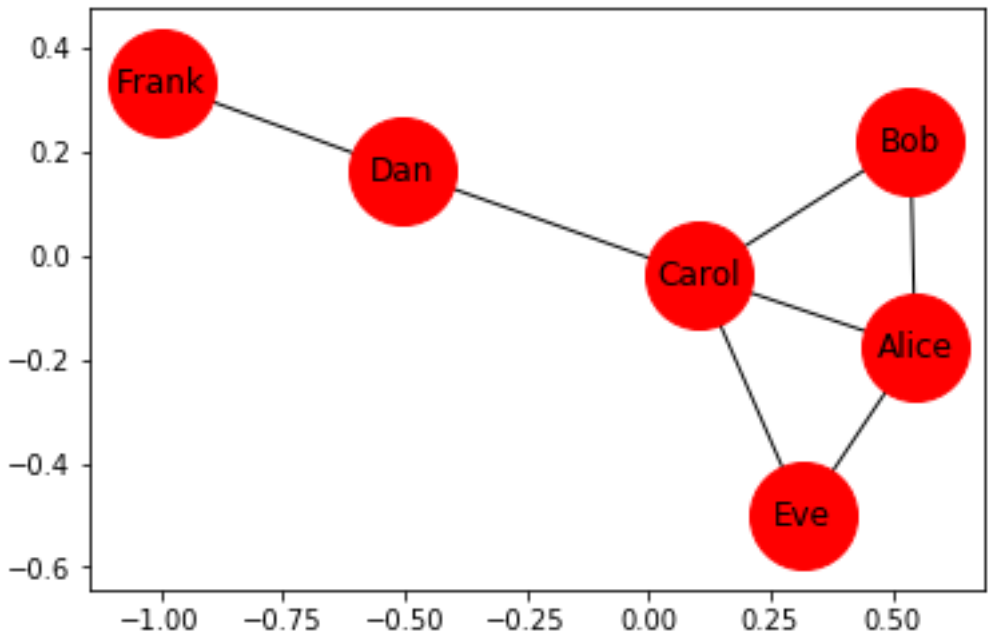
The Spring-Layout

One of the clearest layouts (and therefore the default setting in NetworkX) is the Spring layout. It is so called because the connecting lines are interpreted as *springs*, which means that two connected nodes that are too close to each other are pulled apart by the algorithm that generates the network, and if they are too far apart, they are pulled together. Most of the time this gives a quite clear picture.

Layouts are defined within the `draw` command with the argument `pos` (note: we have already defined the network `G` above in all its parts and therefore we only need to call it in its different layouts):

In [11]:

```
# nx.spring_layout(G) calculates the spring layout for the network
nx.draw_networkx(G, node_size = 1600, pos = nx.spring_layout(G))
```

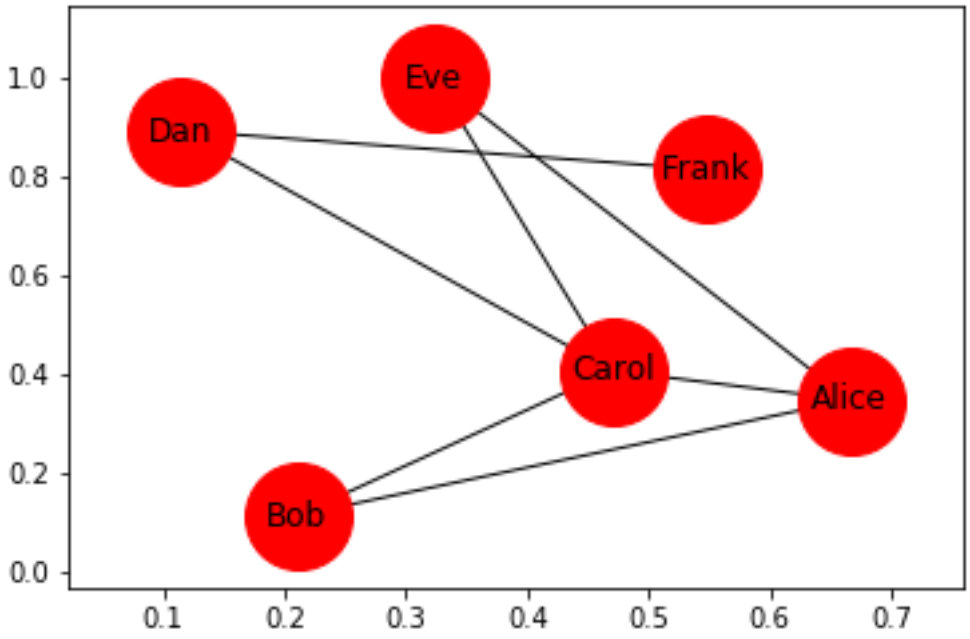


The Random-Layout

Somewhat less clear is the layout "random". Here, the positions of the nodes are chosen absolutely randomly, which means that the connecting lines can run criss-cross. The advantage of this layout, however, is that it requires hardly any computing time and is therefore often used for large networks where it is difficult to follow the individual connecting lines anyway.

In [18]:

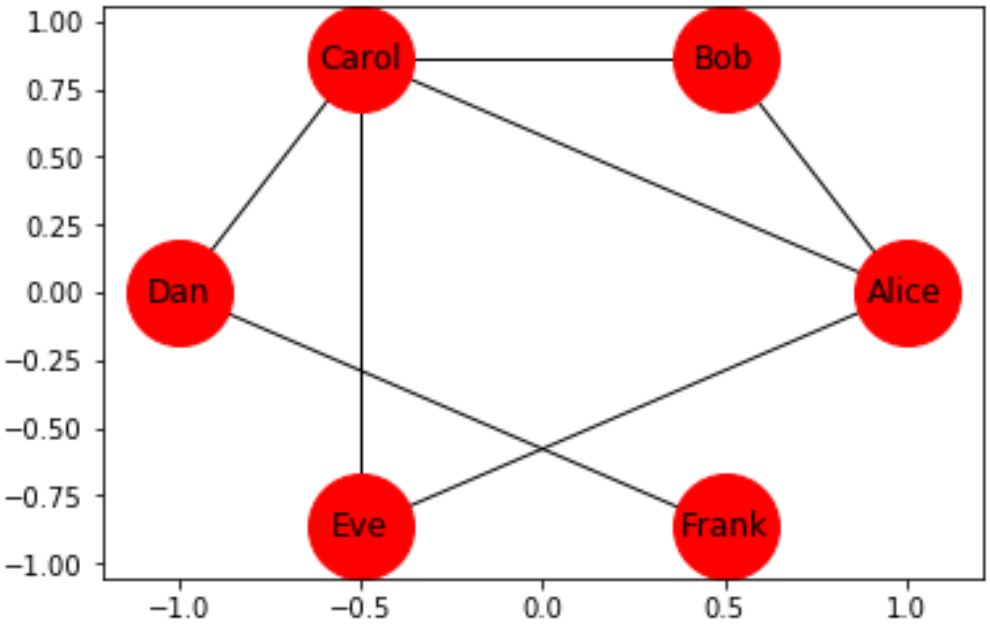
```
nx.draw_networkx(G, node_size = 1600, pos = nx.random_layout(G))
```



The Circular-Layout

In circular layout, all nodes are arranged in a circle. The advantage here is that you can quickly see who is connected to whom.

In [29]:




```
nx.draw_networkx(G, node_size = 1600, pos = nx.circular_layout(G))
```

Network Analysis

What can be done with such a network? First, we could try to determine certain network properties. For our social network, for example, we could find out how friendships are distributed in the network. So, we want to know how many connecting lines the individual nodes have. In English we speak of the **degree** of a node. We get this information with the command `nx.degree(G)`:

In [19]:

```
nx.degree(G)
```

Out[19]:

```
DegreeView({'Alice': 3, 'Bob': 2, 'Carol': 4, 'Dan': 2, 'Eve': 2, 'Frank':
1})
```

The answer should be read like this: The node named "Alice" has degree 3 (three friends), "Bob" has degree 2 (two friends), and so on.

But often you are only interested in the numbers themselves, not necessarily in the names of the nodes. For this purpose, the `.values()` command can be added as follows:

In [20]:

```
list(dict(nx.degree(G)).values())
```

Out[20]:

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

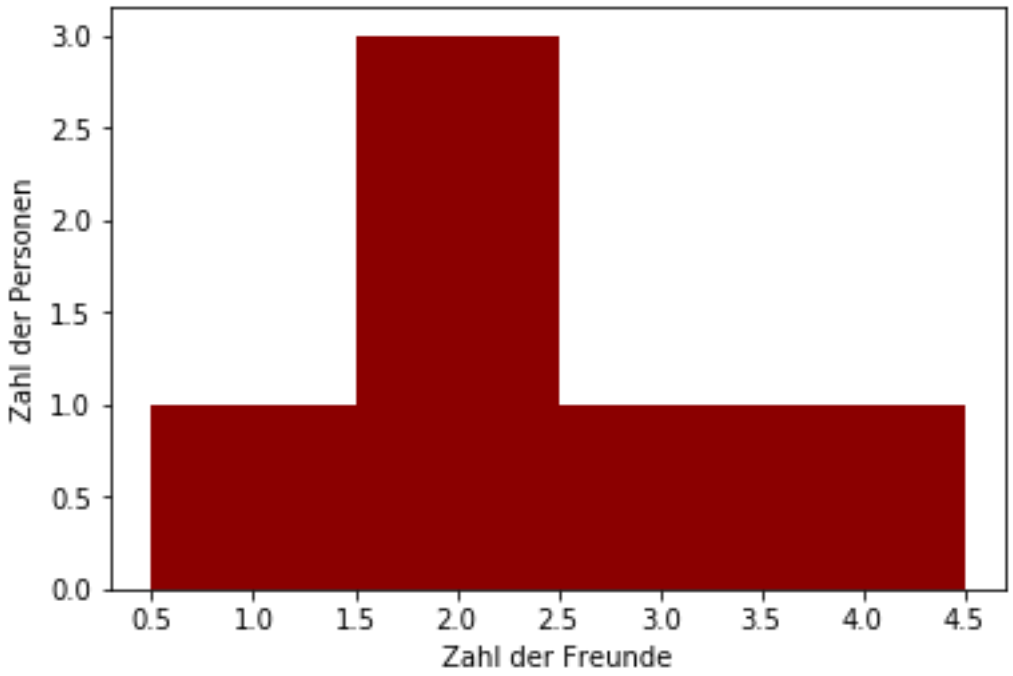
The answer is a list that can easily be transformed into a histogram:

In [21]:

```
plt.hist(list(dict(nx.degree(G)).values()), 4, range = (0.5, 4.5), color =
"darkred")
plt.xlabel('Number of friends')
plt.ylabel('Number of persons')
```

Out[21]:

```
Text(0,0.5,'Number of persons')
```



The histogram shows that there are three people who have exactly two friends and one person who has one, three and four friends. Such an analysis becomes really interesting only with large networks with several hundreds or thousands of nodes. Such networks are usually no longer entered manually. So, instead of defining each node and each connection individually, a so-called network generator is used.

Network generators

Network generators are particularly suitable for creating certain archetypes of networks that occur repeatedly in nature, but also in society or in technical contexts. These archetypes are characterized by very specific network topologies, which occur in a similar way in different contexts.

The Small-World Network

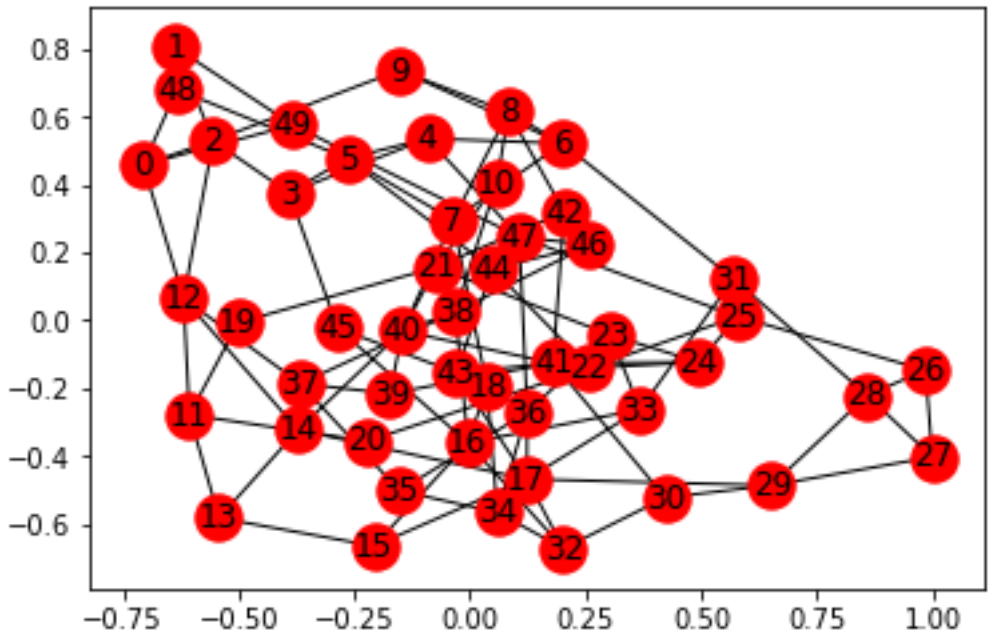
One of the most common network archetypes is the so-called Small World Network, which owes its name to a famous experiment by Stanley Milgram (see: http://systems-sciences.uni-graz.at/etextbook/networks/networks_2.html). It is characterized by a relatively low average network density, which, however, is interrupted by regions of high network density, which in turn are connected to other such dense regions via short paths. Most of the nodes in such networks have very few connections. A few nodes have many connections and very few nodes have even more connections. These very few, extremely well connected nodes are called **hubs**. Many social networks, but also the Internet or many gene networks have this small-world structure.

The Python command for creating such networks is a bit bulky. The names of the developers of the algorithm for generating this type of network can be found in it: `connected_watts_strogatz_graph`. As arguments, this generator needs the number of nodes in the network, the average number of connections per node, and a number that defines how big the hubs that are created may become.

In [22]:

```
# creates a small-world network with 50 nodes, and an average of 5 connecti
ons per node
```

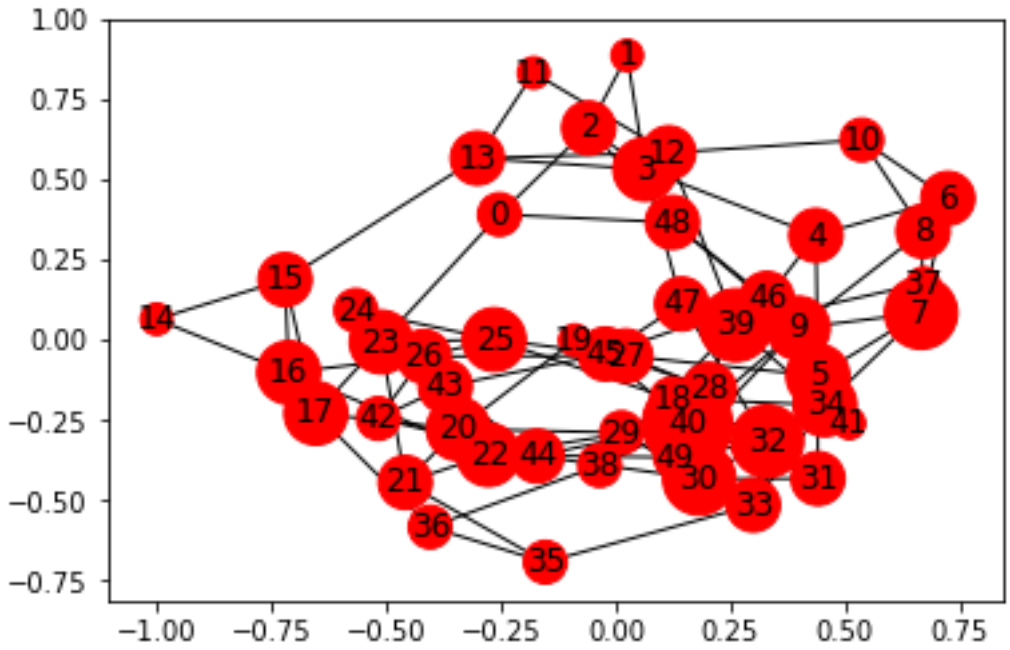
```
# the third number indicates the chance that a node will dock with one of i
ts connections to a hub.
G = nx.connected_watts_strogatz_graph(50, 5, 0.40)
nx.draw_networkx(G)
```



Due to the size of the network, the display has now become a bit confusing, and we can no longer clearly see who is connected to whom, or which node has many connections and which few. This could become clearer if we make the size of the nodes in the display dependent on their `degree` of connection:

In [23]:

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

G = nx.connected_watts_strogatz_graph(50, 5, 0.40)
deg = dict(nx.degree(G)).values()
size = [] # empty list is created

# for each value in the degree list an entry is added to the sizelist
for wert in deg:
    size.append(50 * wert**1.5)

nx.draw_networkx(G, node_size = size)
```



Now you can see from the size of the nodes that some nodes have significantly more connections than others. These hubs of the network can have a great influence on the overall system.

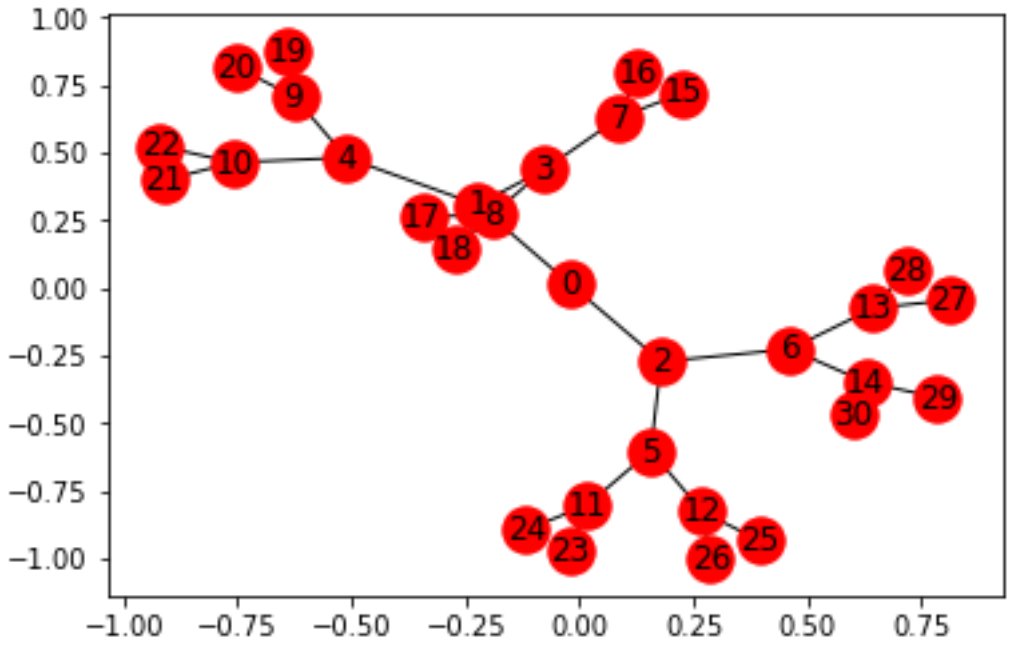
The hierarchical network

In hierarchical networks usually only one node has a special position. This most central account is connected to less important nodes, which in turn are connected to other, even less central nodes.

Such hierarchical structures are often found in organizations, but also for example in food chains in the animal kingdom. The NetworkX command for creating hierarchical networks is `balanced_tree`. The arguments are the number of node connections to the lower level of the hierarchy and the number of levels.

In [24]:

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

# a hierarchical network with 2 connections "down" and 4 levels
G = nx.balanced_tree(2, 4)
nx.draw_networkx(G)
```

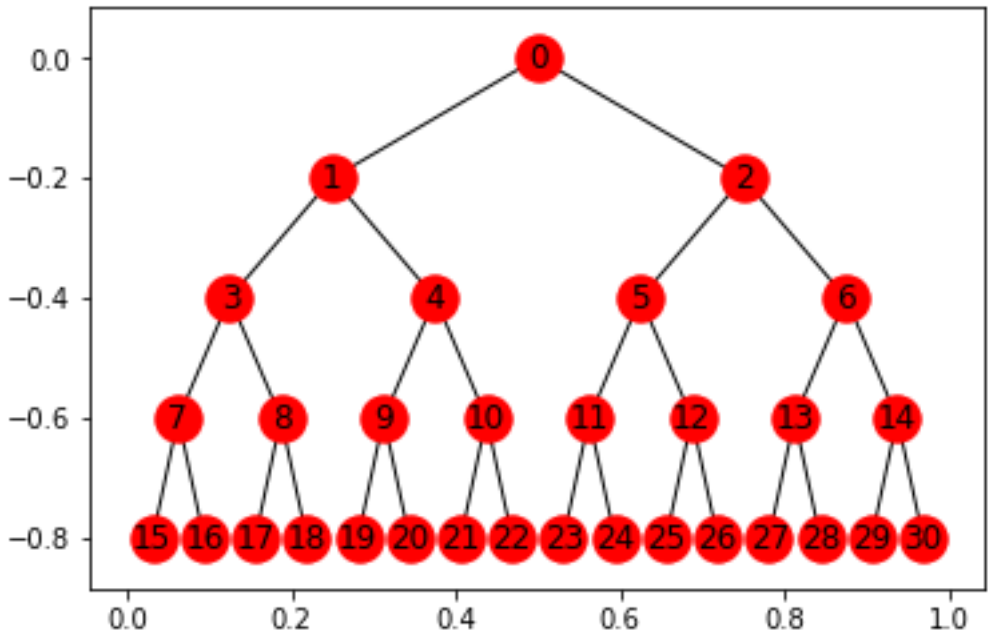


The representation of a hierarchical network with a special layout is particularly clear. Since the exact structure of the function for this layout is beyond the scope of this chapter, we will treat it as a "black box" in the following. That means, we will show the function and its output in the following, but we will not explain how it works exactly.

In [25]:

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

# BLACKBOXFUNCTION that you do not need to understand
def hierarchy_pos(G, root, width=1., vert_gap = 0.2, vert_loc = 0, xcenter
= 0.5,
                  pos = None, parent = None):
    if pos == None:
        pos = {root:(xcenter,vert_loc)}
    else:
        pos[root] = (xcenter, vert_loc)
    neighbors = list(G.neighbors(root))
    if parent != None:
        neighbors.remove(parent)
    if len(neighbors) != 0:
        dx = width/len(neighbors)
        nextx = xcenter - width/2 - dx/2
        for neighbor in neighbors:
            nextx += dx
            pos = hierarchy_pos(G,neighbor, width = dx, vert_gap = vert_gap
,
                                vert_loc = vert_loc-vert_gap, xcenter = nex
tx, pos = pos,
                                parent = root)
    return pos
# END OF THE BLACKBOXFUNTCION
```

```
G=nx.balanced_tree(2, 4)
nx.draw_networkx(G, pos = hierarchy_pos(G, 0))
```



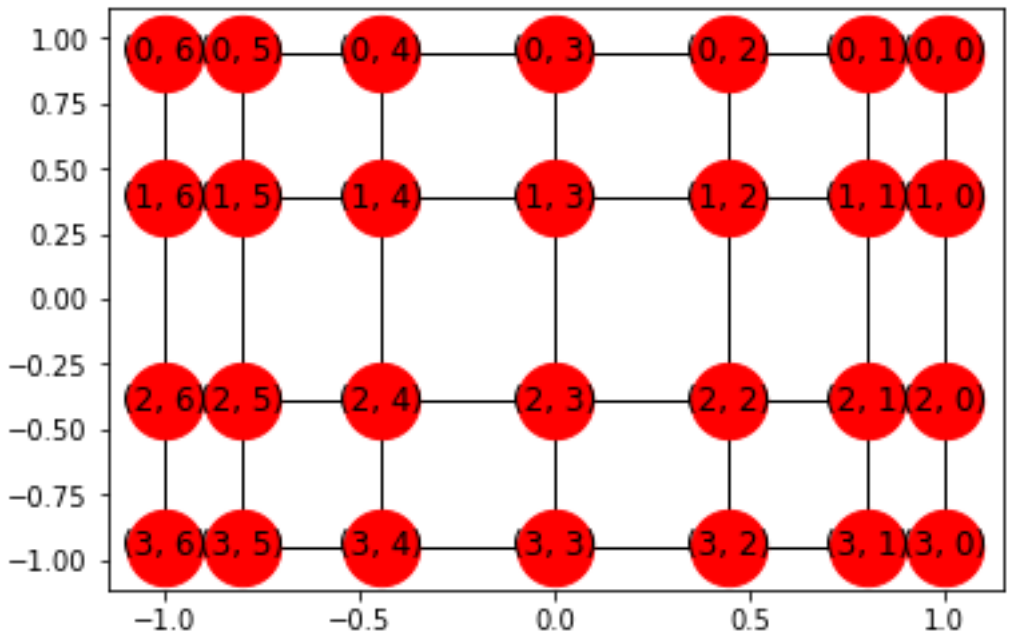
The Grid-Network

Even a simple grid structure can be considered as a network. Such structures occur, for example, in crystals, but also everywhere else where objects prefer to interact with their nearest neighbors.

The command to create grid networks is `grid_2d_graph` and needs the length and width of the grid as arguments.

In [29]:

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

G = nx.grid_2d_graph(4, 7) # creates a 4 by 7 grid
# Spectral-layout for suitable grid shape
nx.draw_networkx(G,pos = nx.spectral_layout(G), node_size = 800)
```



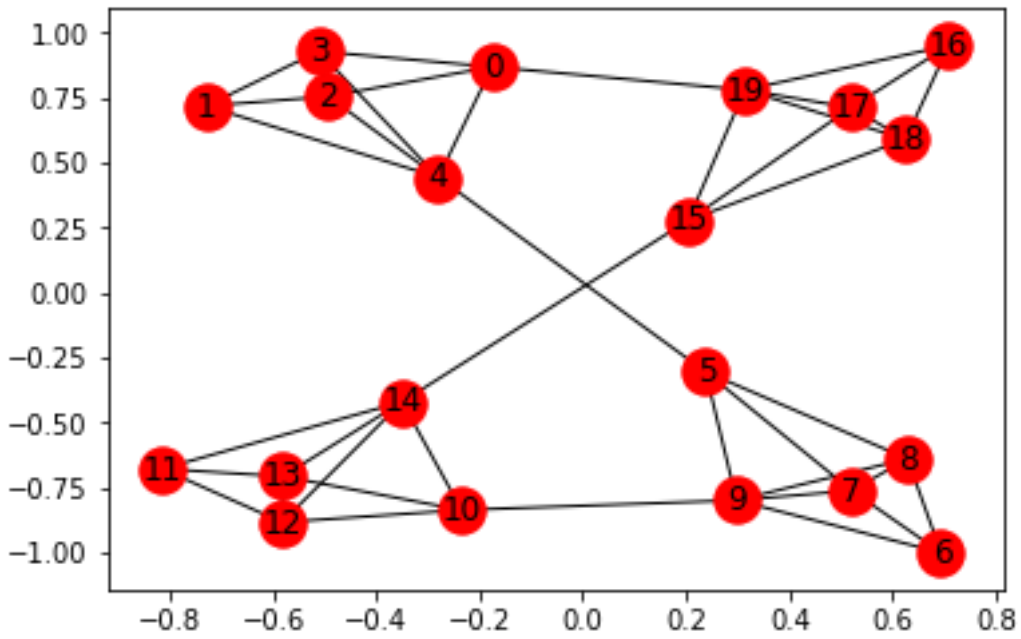
The Caveman-Network

A so-called Caveman network is characterized by several densely networked groups of nodes, so-called **clusters**, which are only slightly interconnected.

The command to create such networks is `connected_caveman_graph` and needs as arguments the number of clusters and the size of the clusters.

In [30]:

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

G=nx.connected_caveman_graph(4, 5)  # Network of 4 clusters with 5 nodes ea
ch
nx.draw_networkx(G)
```



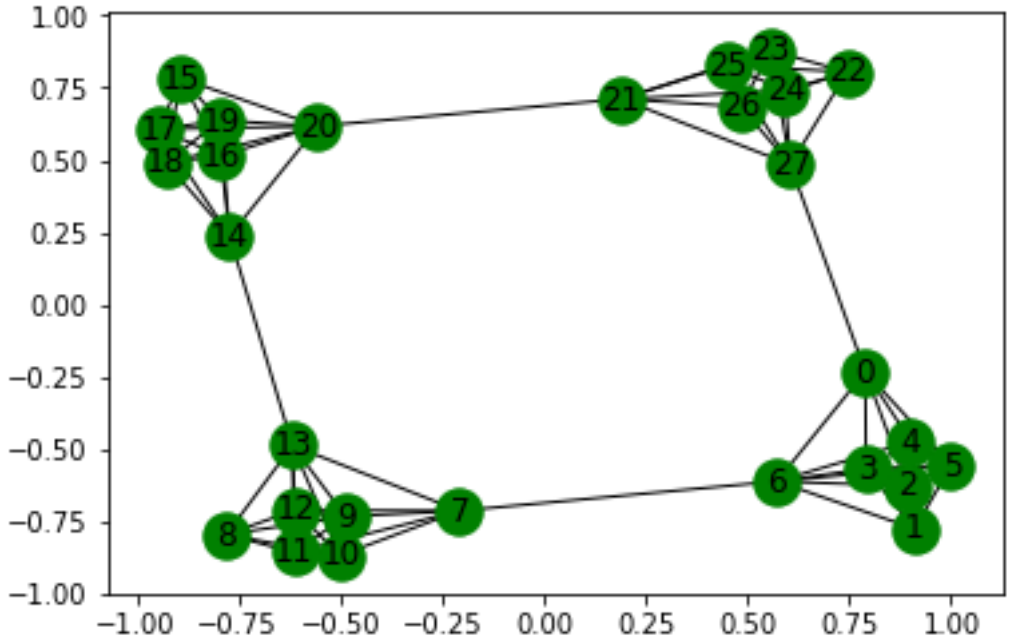
Example: The spread of a computer virus

Let us now try to use our previous knowledge about networks in a simple application example. We create a fictitious computer network and investigate how a software virus would spread in this network.

As a network type we choose a Caveman network, in which four companies connected via the Internet each operate seven computers in a company-owned intranet.

In [27]:

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

G = nx.connected_caveman_graph(4, 7)
# save the node positions to see the same network in all following views
position = nx.spring_layout(G)
nx.draw_networkx(G, pos = position, node_color = "green")
```



With the following code, we add an attribute for each node, which later stores whether the respective computer is infected by the software virus or not. To do this, we iterate with a loop over the number of nodes of the entire network (`G.number_of_nodes()`), define the new attribute "infected" for each node and set it to "no". This is done with `G.node[it]["infected"] = "no"`.

In [28]:

```
# iterate over all nodes and give them a new attribute called "infected"
# set this new attribute to "no"
for it in range(G.number_of_nodes()):
    G.node[it]["infected"] = "no"
```

We check this operation by asking any node - the node with index 0 - for its attributes:

In [29]:

```
G.node[0] # ask which data is stored in the node with index 0
```

Out[29]:

```
{'infected': 'no'}
```

In a similar way, any amount of data can be stored in a node. But here we are happy with only one attribute.

The next step is to infect a node (a computer) with the software virus, for example the node with index 8:

In [30]:

```
# set "infected" to "yes" for node 8
G.node[8]["infected"] = "yes"
# ask which data is stored in node 8
G.node[8]
```

Out[30]:

```
{'infected': 'yes'}
```

If we were to check the entire network for viruses, we would probably ask all nodes using a For loop:

In [31]:

```
for it in range(G.number_of_nodes()):
```

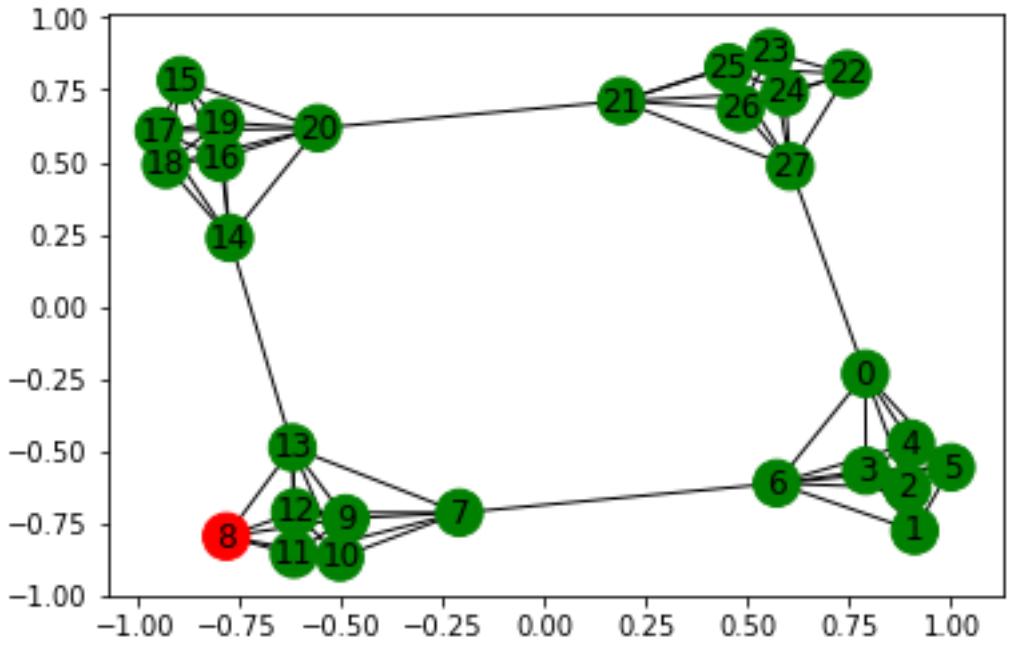
```
    print(G.node[it]),  # the comma at the end of the print command writes
output continuously to the same line
```

```
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'yes'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
{'infected': 'no'}
```

Of course, this is not that clear. It would be nicer to reflect this information with the color of the nodes. To do this, we create a new list in which the colors of the nodes are stored: green for "not infected" and red for "infected":

In [32]:

```
farben = []
for it in range(G.number_of_nodes()):
    # ask the value of the attribute "infected":
    if G.node[it]["infected"] == "yes":
        farben.append("red")
    if G.node[it]["infected"] == "no":
        farben.append("green")

# instead of the one color "green", we now plot with the list "colors".
# for each node there is exactly one entry
nx.draw_networkx(G, pos = position, node_color = farben)
```



In this display the infected node is clearly visible.

How would such a software virus spread? Epidemiology knows many models of infection spread. Here we use the simplest one at first: in each time step all nodes connected to an infected node are infected.

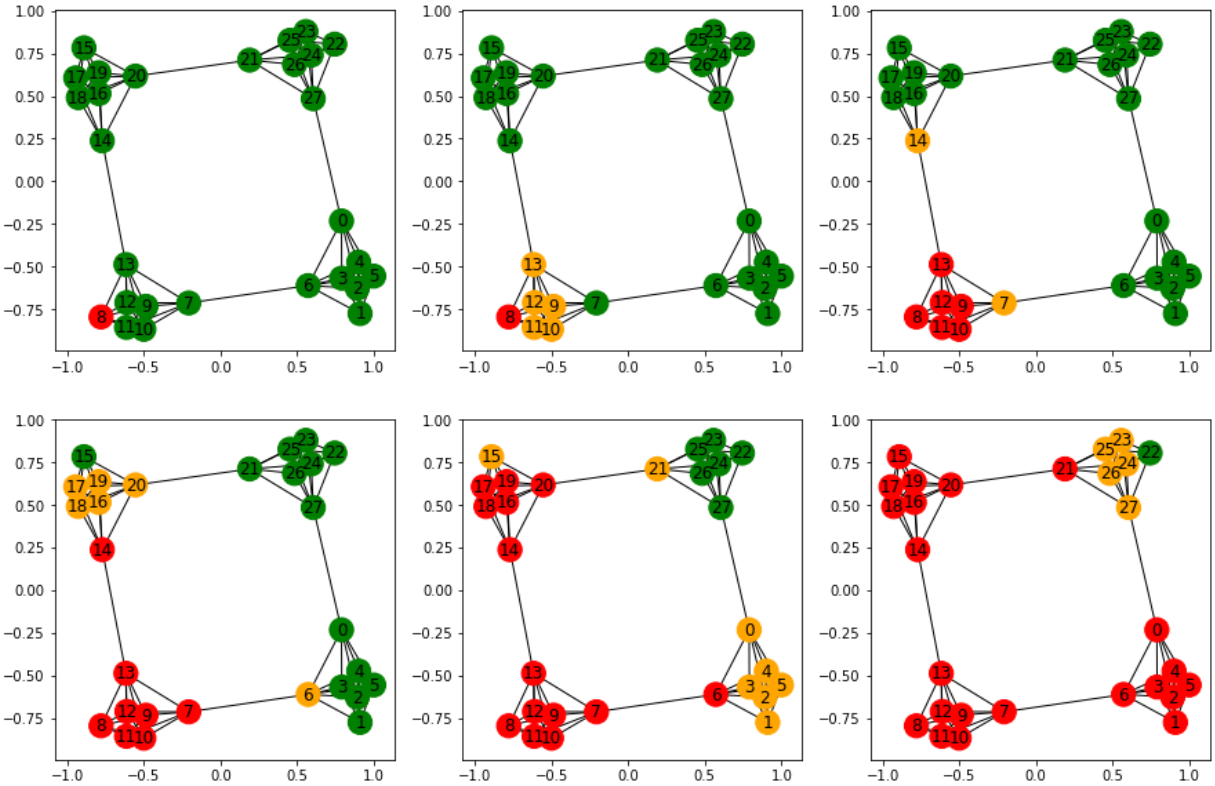
For this we use a new command: `G.neigbors(n)`. It gives us all neighbors (i.e. connected nodes) of a node `n`.

If we now want to simulate five time steps of this simple infection model, some complex nesting of If queries and For loops will result. We take a closer look at these in the following code segment:

In [33]:

```
# define a general drawing area
fig = plt.figure(figsize=(15, 10))

n = 1    # Counter for counting the subplots
# first subplot
ax = fig.add_subplot(2, 3, n)
nx.draw_networkx(G, pos = position, node_color = farben)

# simulate 5 time steps
for zeit in range(5):
    n += 1 # set counter for counting the subplots up by 1
    # for loop over all nodes
    # we call the run variable inf, because we are only looking at the infe
cted nodes
    for inf in range(G.number_of_nodes()):
        # if the node is infected,
        if G.node[inf]["infected"] == "yes":
            # we go in another loop over all neighbors of the infected node
:
            for opfer in G.neighbors(inf):
```

```
                # and if this node is not yet infected
                if G.node[opfer]["infected"] == "no":
                    # it is newly infected at this time step:
                    G.node[opfer]["infected"] = "new"
                # End of the if-query
            # End of the for loop about the victims
        # End of the if-query, which determines whether a node was infected
    # End of the for loop over all nodes
    # here we are again in the for loop over 5 time steps
    farben = []
    for it in range(G.number_of_nodes()):
        # the newly infected nodes get the color orange, so that we can bet
ter track their spread
        if G.node[it]["infected"] == "yes":
            farben.append("red")
        if G.node[it]["infected"] == "no":
            farben.append("green")
        if G.node[it]["infected"] == "new":
            farben.append("orange")
    # new subplot
    ax = fig.add_subplot(2, 3, n)
    nx.draw_networkx(G, pos = position, node_color = farben)

    # here we now change all "newly" infected nodes to infected ("yes"),
# so that they appear red instead of orange in the next step and in turn in
fect other nodes
    for it in range(G.number_of_nodes()):
        if G.node[it]["infected"] == "new":
            G.node[it]["infected"] = "yes"
```



With this very simple model, we can already begin to analyze aspects of the spread of a network virus. Try to extend this model by yourself, for example by generating larger or different networks, or by defining nodes with additional attributes - antivirus programs, firewalls etc. - firewalls, etc.

Summary

Networks

In network research, objects are understood as *nodes*, which are connected to other nodes. The decisive factor here is not so much the special properties of the nodes, but rather the connection structure (the toplogy of a network), which results from the number of connections (*links, edges*) of the nodes - the so-called *degree* - and all neighboring nodes. Certain types of networks, such as small-world or hierarchical networks, are based on phenomena in a wide variety of contexts.

Networks in Python

In Python the package `NetworkX` is available for the representation and analysis of networks.

Manual generation of Networks

With the function `G = nx.Graph()` a new, empty network named G is generated. New nodes can be added with the function `G.add_node("Name_of_the_new_node")` and connected to other nodes with `G.add_edge("node_1","node_2")`.

Netzwork Generators

To create special network types, so-called network generators are available, which generate partly random, partly deterministic networks. Commonly used generators are those for small

world networks (`connected_watts_strogatz_graph`) and for hierarchical networks (`balanced_tree`).

Programming with Network

To iterate over all nodes in a network, the loop command `for it in range(G.number_of_nodes()):` is available. To assign an attribute to a node that contains additional information, the command `G.node["Name_of_node"]["Name_of_attribute"] = "value_of_attribute"` is available.

Chapter 17 – and what´s next?

This ends this introduction to (scientific) programming. We now know the most important structures and concepts:

- Lists
- For Loops
- If-queries
- Functions
- Classes

But beyond that there is still a lot more to learn.

Programming in Python

To learn more about programming with Python, there are many useful online resources:

https://docs.python.org/3/tutorial/index.html

https://www.w3schools.com/python/

https://www.learnpython.org/

https://www.codecademy.com/learn/learn-python-3

An overview of these and other ways to learn about Python can be found here:

https://docs.python-guide.org/intro/learning/

References

If you are only looking for a specific structure, the easiest way to find it is in the Python Language Reference. There all structures of the language are listed with short explanations.

https://docs.python.org/3/reference/index.html

All functions included in Python can be found in the Functions Reference:

https://www.w3schools.com/python/python_ref_functions.asp

Algorithms

In addition to pure programming techniques, one can also acquire knowledge about algorithms. These can then be applied in other programming languages and complicated problems can be solved. An introduction to this topic can be found here for example:

https://www.khanacademy.org/computing/computer-science/algorithms

Further programming languages

Besides Python there are of course many other programming languages. If you know Python, and therefore have a basic understanding of programming itself, you can easily learn a second language. In a scientific context there are particularly relevant:

C++ (universal, widely used programming language)

http://www.learn-cpp.org/

Matlab (for numerical calculations with matrices in technology and engineering)

https://learntocode.mathworks.com/

Mathematica (for analytical calculations and mathematical derivations)

http://www.wolfram.com/language/fast-introduction-for-programmers/en/

R (For statistics and data analysis)

https://swirlstats.com/students.html

Learning by Doing

Once the first step has been taken, programming is also very suitable for learning by doing. The best way is to find a topic or project you are really interested in and try to create a program for it. Once the basic framework is working, you can extend it as much as you like and will practice and learn a lot of programming techniques without losing the joy of working.

Command Glossary

Command	Relevant Chapter
`and`	Chapter 4
`arange`	Chapter 10
`break`	Chapter 10
`class`	Chapter 11
`csv.reader`	Chapter 6
`def`	Chapter 9
`else`	Chapter 4
`except`	Chapter 4
`for`	Chapter 2
`if`	Chapter 4
`len`	Chapter 4
`liste.append`	Chapter 1
`np.array`	Chapter 5
`np.cumsum`	Chapter 6

Command	Relevant Chapter
`np.diff`	Chapter 6
`np.zeros`	Chapter 8
`or`	Chapter 4
`plt.hist`	Chapter 7
`plt.plot`	Chapter 1
`random.gauss`	Chapter 7
`random.normal`	Chapter 7
`range`	Chapter 1
`sum`	Chapter 6
`try`	Chapter 4
`while`	Chapter 9