Programming with Python
Part 1
A simple Actor-Environment Demo
emerging-mind.org eJournal ISSN 2567-6466
(info@emerging-mind.org)

Gerd Doeben-Henisch
gerd@doeben-henisch.de

October 14, 2017

Contents

Abstract

How to start with python? Programming actors within a simple environment by using simple graphics.

1 Introduction

This text is embedded in a twofold context. On one side we have the uffmm.org platform where we develop an online book project integrating subjects like philosophy of science, systems engineering, actor-actor interaction as well as intelligent machines; for this theoretical approach we need different software-tools. On the other side we have the philosophy-in-concert.org platform which supports art projects with the topic of the possible symbiosis of humans and intelligent machines in the future.

While the software for the uffmm.org platform will be based on ubuntu + ros (robot operating system) + tensorflow, the art project will additionally use windows (on account of the music software ableton-live with Max) with python.

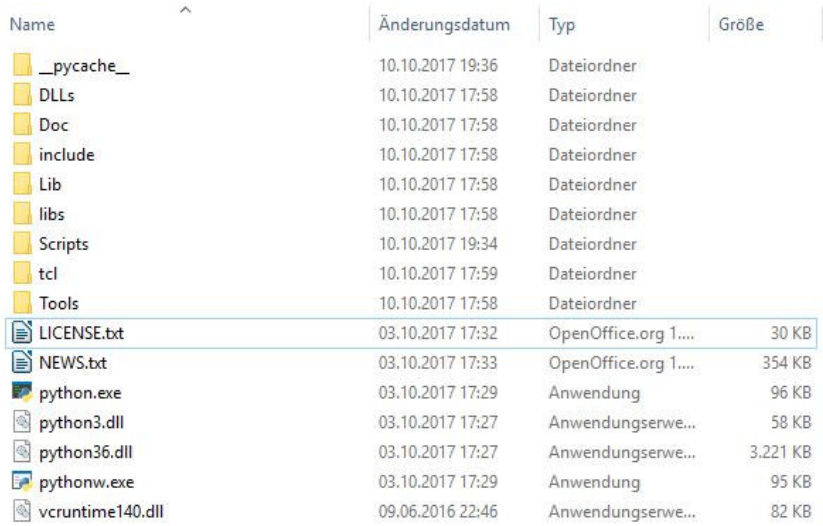
But python is a 'chameleon': it is part of ros and tensorflow too.

In the following text we show first steps in using the programming language python to program a first version of intelligent machines (= programs) within a simple environment for an art event.

2 Python Context

2.1 Python Language Source

To use python we need first the language sources. The sources for python can be found here: `https://www.python.org/`. We have downloaded python 3.6.3 for Windows. After downloading we have installed it with administrator rights. On my windows machine (win 10) it has been installed with this path:



Name	Änderungsdatum	Typ	Größe
__pycache__	10.10.2017 19:36	Dateiordner	
DLLs	10.10.2017 17:58	Dateiordner	
Doc	10.10.2017 17:58	Dateiordner	
include	10.10.2017 17:58	Dateiordner	
Lib	10.10.2017 17:58	Dateiordner	
libs	10.10.2017 17:58	Dateiordner	
Scripts	10.10.2017 19:34	Dateiordner	
tcl	10.10.2017 17:59	Dateiordner	
Tools	10.10.2017 17:58	Dateiordner	
LICENSE.txt	03.10.2017 17:32	OpenOffice.org 1....	30 KB
NEWS.txt	03.10.2017 17:33	OpenOffice.org 1....	354 KB
python.exe	03.10.2017 17:29	Anwendung	96 KB
python3.dll	03.10.2017 17:27	Anwendungserwe...	58 KB
python36.dll	03.10.2017 17:27	Anwendungserwe...	3.221 KB
pythonw.exe	03.10.2017 17:29	Anwendung	95 KB
vcruntime140.dll	09.06.2016 22:46	Anwendungserwe...	82 KB

Figure 1: Python Folder under Windows 10

`C:\Users\gerd_2\AppData\Local\Programs\Python`

Within this you will find the folder named 'Python36-32'. This means that the default installation is with the 32-Bit version of python. There is no chance to select the 64-Bit version, because you will not be asked to select it. The content of this folder is shown in figure 1.

2.2 Simple Graphics Library

Because we need for our first art projects a simple library for graphics operations we searched in the web and found a library called 'graphics.py' from John M. Zelle [Zel02]. You can download it from `http://mcsp.wartburg.edu`. You have to copy it inside a special sub-folder (cf. figure 2) :

`C:\Users\gerd_2\AppData\Local\Programs\Python\Python36-32\Lib\site-packages`

2.3 Python Console

To check whether python works and the graphics-library is 'known' to the python interpreter you can go back to the python-folder (see figure 1). Here you can either click directly on the 'python.exe' file to start a python console window or you click with the right mouse-button on the 'python.exe' file. A context-menu will pop up with the option, to fix this file to your task-bar. If

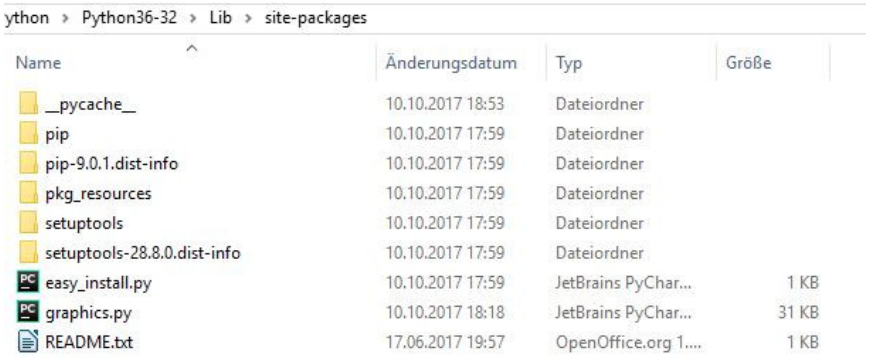


Figure 2: graphics.py in a special sub-folder of python

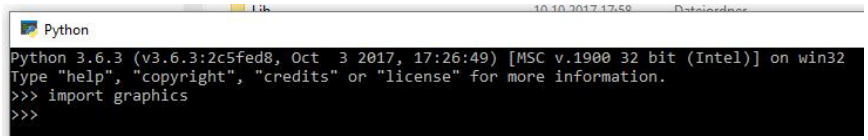


Figure 3: Python shell under windows 10

you select this then a small icon will appear on your task-bar showing a console window. This makes life easier if you want to use this python-console more often.

Whatever you will select, in the python-console you can then enter the command line: 'import graphics. This means that the python interpreter shall import the content of the graphics.py file into the interpreter for further usage. If the interpreter can find the graphics.py file a new line will appear without any comments. This means: everything is OK (cf. figure 3).

2.4 Integrated Programming Environment

2.4.1 PyCharm

Although working in the beginning with the python shell is possible (and surely very instructive for first steps), for more advanced work it is to cumbersome. Because millions of programmers have the same problem nice people (and companies) have developed some more comfortable environments supporting you with the generation of python-based projects. One of these environments has the name 'PyCharm'. It is an *integrated development environment (IDE)* for python developed by the company 'Jet Brains': `https://www.jetbrains.com/pycharm/`.

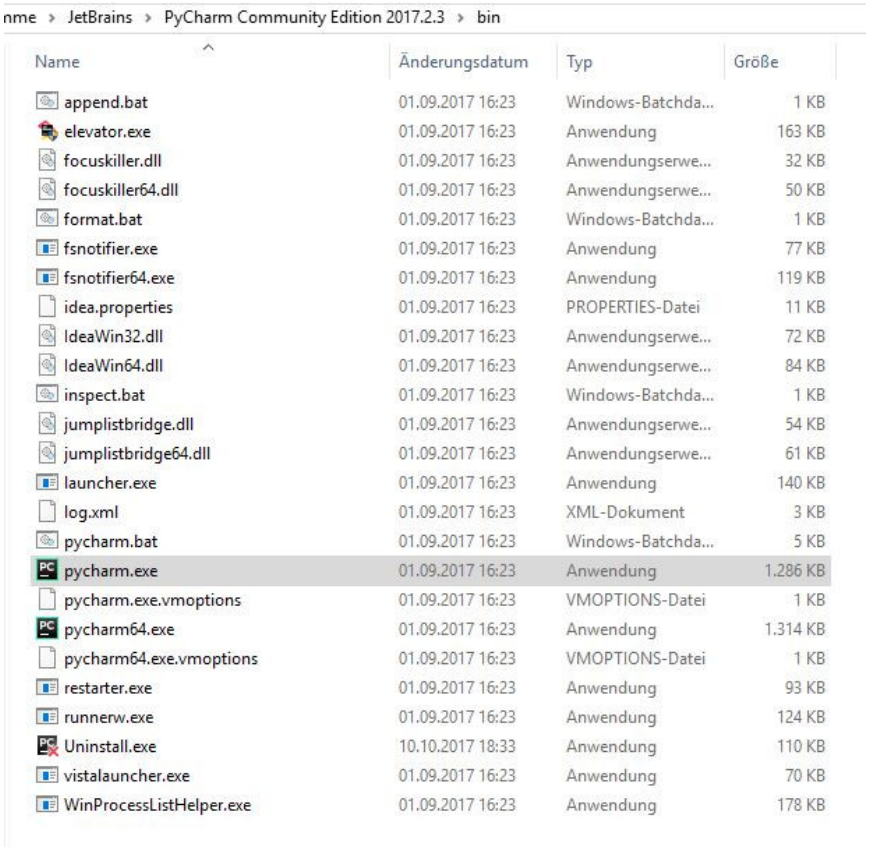


Figure 4: PyCharm Folder under Windows 10

If you scroll down the web-page with interesting informations about the features of this IDE you will find a download-option for the community, which is for free (with not all nice features). If you select the '.exe' download option than you can download the IDE-software for Windows and install it with administrator rights afterwards. On my machine it is installed with the following path:

```
C:\Program Files\JetBrains\PyCharm Community Edition 2017.2.3\bin
```

The content of this folder is shown in figure 4

Although I have for python only the 32-Bit version it is possible and makes sense to use for the pycharm-IDE the 64-Bit version. While the python programs can only address an address-space of 4GB, the pycharm-projects as such can be larger (in principle).

2.4.2 WinPython (With spyder and numpy)

The pyCharm IDE is a very nice tool and the python source with python 3.6.3 installed so far is the most new version available for windows. But a problem popped up during first tests: the standard python does not include the data-type 'matrix'. For most problems the data-types available like 'list', 'tuple', 'set' etc are very strong, but for many mathematical applications one needs definitely matrices and typical mathematical functions and constructs. This can by find in a library like 'numpy'.[1]

To include 'numpy' into your windows based python installation is not easy going. The best solution is to download a python-distribution which already has integrated numpy and is fitting windows. After some search I detected 'WinPython'.[2]

The installation was quite straightforward. After downloading the Winpython 3.6.2 64-Bit version for Windows in the target folder, it extracted itself and prepared everything. See the final folder in figure 5.

This distribution contains as an integrated development environment the spyder software. This software looks a bit 'poorer' then the pycharm IDE but it works nicely and has the integrated numpy package.

In the following experiments I often use both implementations (pycharm with python 3.6.2 32-Bit and spyder with python 3.6.2 64-Bit with numpy).

The special graphics library from John Zelle 'graphics.py' has to be introduced to the WinPythin distribution again manually in the following folder:

```
C:\Users\gerd_2\Documents\EMProjekt\SW\WinPython-64bit-3.6.2.0Qt5...
\python-3.6.2.amd64\Lib\site-packages
```

3 Problem to be Solved

To start and exercise programming with python we have a real project which has to be programmed.

We want to demonstrate the behavior of simple *input-output system (IO-SYS)* which live in a simple, flat (2D) *environment (ENV)*. We want to investigate three types of systems: behaving (i) with a fixed rule, what to do; (ii) following a random behavior, and (iii) showing a random behavior which can be enhanced with a simple memory of realized paths from start to some

[1] See here: `http://www.numpy.org/`

[2] See here: `https://winpython.github.io/`

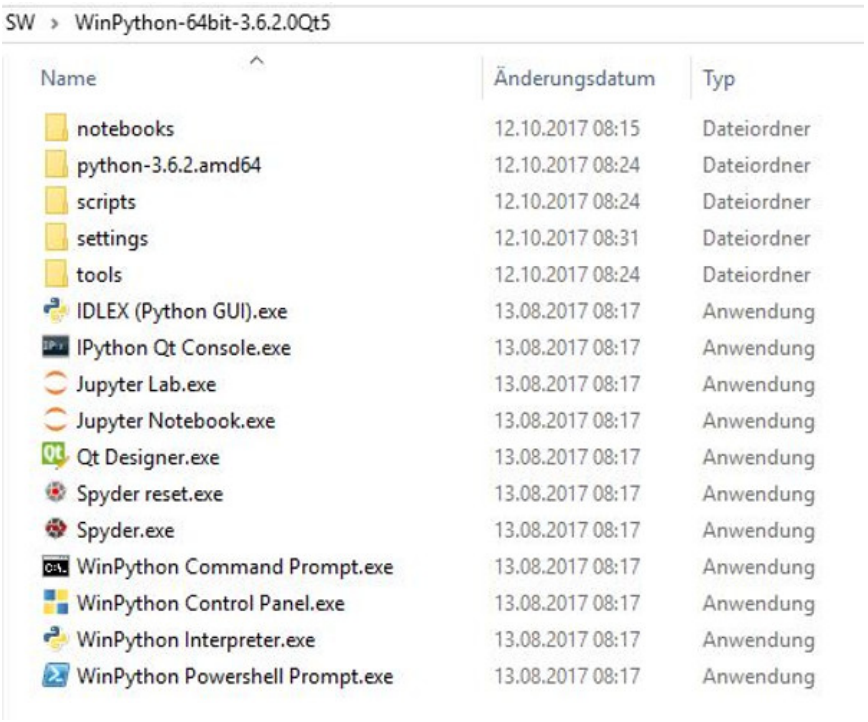


Figure 5: The WinPython Folder as separate folder

goal.

The systems have a minimal 'visual perception' of the environment; they consume 'energy' depending on the time; and they have to 'eat' some 'food' before some threshold of energy, otherwise they will die. After a death the experiment can be repeated with everything set back to default values.

To be able to survive they can move around in the four main directions of their quadratic environment. The borders of the environment can not be crossed and within the borders there are only two kinds of objects: (i) 'obstacles' which can not be passed directly and 'food' which can be 'eaten'. In the simple version the food will regenerate immediately after it has been consumed.

If there is more than one input-output system in the environment then a system A is like an obstacle for another system B.

There should be 'log-files' from every experiment allowing the repetition of the behavior of a system in that environment.

4 Programming with Python

We will proceed in two steps: first we will make some experiments with parts of the whole problem. Then we will put everything together in a more generalized setting. As our 'guide' for the programming we will use the book from John Zelle [Zel02], especially chapter 5 dealing with graphics.

4.1 Testing an Environment

One important part of the problem is the environment. It should be presented as a graphical object showing a flat, 2D-area with regular fields (a 'grid'), and these fields can be occupied either by 'empty space', by 'obstacles' or by 'food-objects'.

4.1.1 A First Window

Therefore we make the following assumptions:

1. We open a window with the size 700 $\times$ 700 Pixel and the label 'ENVIRONMENT 1'.
2. The window will close if you give an answer to the input question.

A possible simple source code could read as follows:[3]

```
# -*- coding: utf-8 -*-
# gdh_win0.py
# author: Gerd Doeben-Henisch
# Trying a first environment
# Using book from John Zelle (2002) and his simple graphics.py library

def main():
import graphics
win = graphics.GraphWin('ENVIRONMENT 1', 700, 700)
answer = input("Shall we finish? (y/n)")
win.close()
main()
```

[3]Comment: LaTeX does not preserve the leading spaces in the python source code. Thus, if you want to use this program with copy and paste you have to insert these manually.

This simple program imports the 'graphics.py' module from John Zelle and then opens a window 'win' with label 'ENVIRONMENT 1' with the size 700 $\times$ 700 pixels. The windows waits to close until you input something (because there is no check of agreement of input and question.)

This works fine. But even in this simple code (compared to the original commands in Zelle it has been changed) one can detect differences between python at the time of the book 2002 and today 2017. This gives you a chance to learn a bit more about the language.

4.1.2 Rectangles, Lines, Circles, Text-Labels

In the next example we have introduced some elements: a rectangle, two lines, a circle and a text-label for the circle.

```
# -*- coding: utf-8 -*-
# gdh_win1.py
# author: Gerd Doeben-Henisch
# Trying a first environment
# Using book from John Zelle (2002) and his simple graphics.py library

def main():
import graphics
win = graphics.GraphWin('ENVIRONMENT 1', 700, 700)
rect1 = graphics.Rectangle(graphics.Point(130,70), graphics.Point(190,130))
rect1.draw(win)

line = graphics.Line(graphics.Point(70,0), graphics.Point(70,699))
line.draw(win)

line2=line.clone()
line2.move(70,0)
line2.draw(win)

center = graphics.Point(100,100)
circ = graphics.Circle(center, 30)
circ.setFill('red')
circ.draw(win)

label = graphics.Text(center, "Red Circle")
label.draw(win)

answer = input("Shall we finish? (y/n)")
```

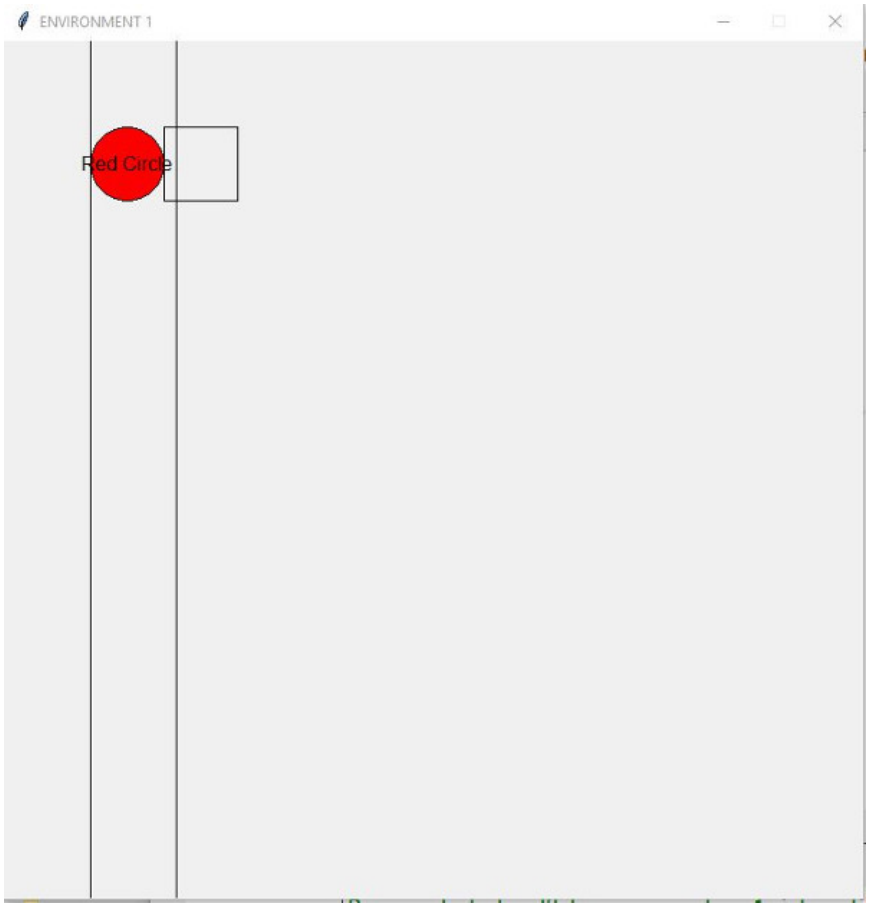


Figure 6: Simple window with rectangle, lines, circle and text

```
win.close()
main()
```

This produces the following graphical output (cf. figure 6):

4.1.3 Producing a Grid

This shows how such a window could work 'in principle'. This is not yet, what we want. Let us make the requirements a bit more axplicit:

1. We open a window with the size 700 $\times$ 700 Pixel and the label 'ENVIRONMENT 1'.

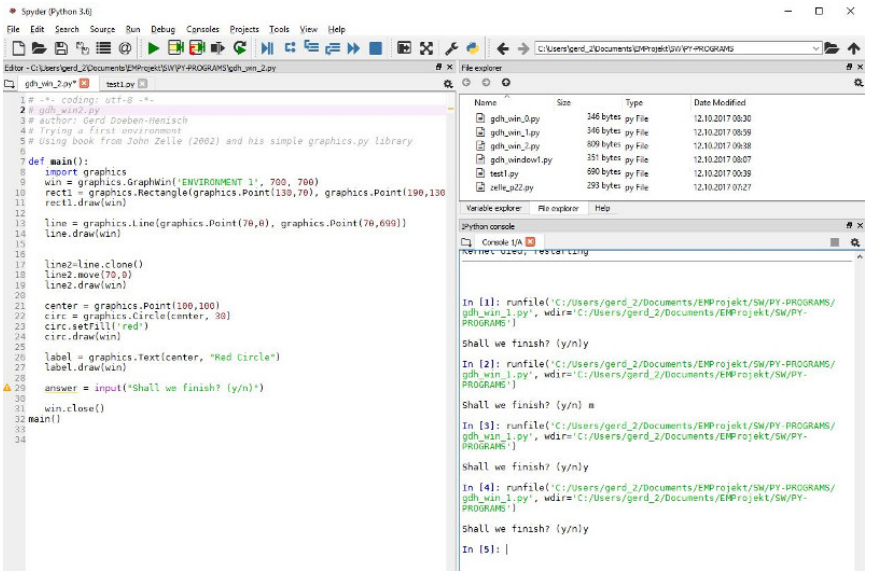


Figure 7: The spyder IDE starting editing gdh_win2.py

2. The window will be written with vertical and horizontal lines generating a grid with 7 $times$ 7 'cells' each with the size 100 $\times$ 100.

3. The window will close if you give an answer to the input question.

Before we continue here a snapshot of the spyder IDE just beginning to write the next version of the test program' gdh_win2.py' (cf. figure 7).

The main idea of 'gdh_win2.py' is to draw the lines of the intended grid with some automatic (and scalable) mechanisms. The first idea is to produce a list of possible objects for the vertical lines. This can be done with three steps: (i) generate a line object for the first vertical line in the window, then (ii) generate a list for 7 possible objects (in the range from 1 to 7) and then assign the line-objects to the list-objects. This is done here for the first two list objects (including the 'clone' operation). Then one can use the 'move' operation to move the second line-object from the original start position 100 pixels to the right. As the output shows, this idea works.

```
# -*- coding: utf-8 -*-
# gdh_win2.py
# author: Gerd Doeben-Henisch
# Trying a first environment
# Using book from John Zelle (2002) and his simple graphics.py library

def main():
```

```
import graphics
win = graphics.GraphWin('ENVIRONMENT 1', 700, 700)

line = graphics.Line(graphics.Point(100,0), graphics.Point(100,699))

lx=list(range(1,8))

lx[1]=line
lx[1].draw(win)

lx[2]=line.clone()
lx[2].move(100,0)
lx[2].draw(win)

answer = input("Shall we finish? (y/n)")

win.close()
main()
```

4.1.4 Grid with Vertical Lines

The following small program 'gdh_win3.py' draws successfully 6 vertical lines with a distance of 100 pixels each. There is one base-line at the border of the windows (which is not drawn), and there is a list of possible objects. Each of these possible objects clones the first line-object and moves it accordingly i-times 100 pixels to the right.

```
# -*- coding: utf-8 -*-
# gdh_win3.py
# author: Gerd Doeben-Henisch
# Trying a first environment
# Using book from John Zelle (2002) and his simple graphics.py library

def main():
import graphics
win = graphics.GraphWin('ENVIRONMENT 1', 700, 700)

lx=list(range(1,8))
line = graphics.Line(graphics.Point(0,0), graphics.Point(0,699))

for i in range(1,7):     #the workable range ends up with '6'!
lx[i]=line.clone()
```

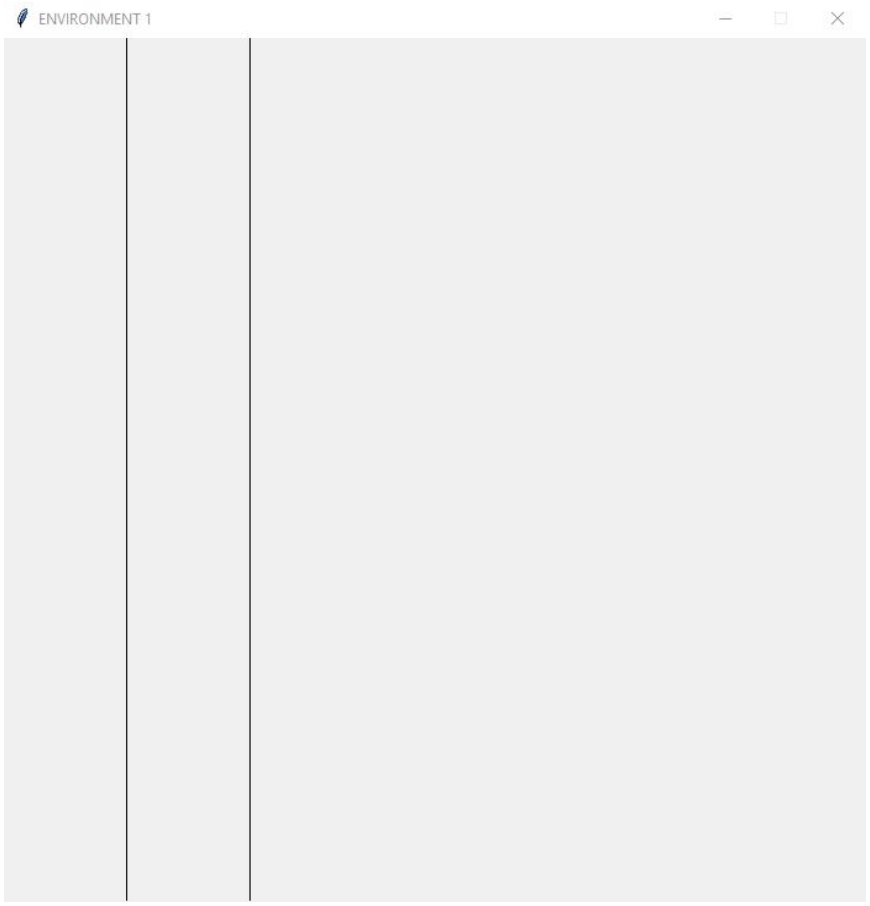


Figure 8: Windows with two lines from gdh_win2.py

```
lx[i].move(i*100,0)
lx[i].draw(win)
print(i)

answer = input("Shall we finish? (y/n)")

win.close()
main()
```

4.1.5 Grid with Vertical and Horizontal Lines

From here it is a small step to a full grid. See the result on the screen in figure 9.

```
# -*- coding: utf-8 -*-
# gdh_win4.py
# author: Gerd Doeben-Henisch
# Trying a first environment
# Using book from John Zelle (2002) and his simple graphics.py library

def main():
import graphics
win = graphics.GraphWin('ENVIRONMENT 1', 700, 700)

lx=list(range(1,8))
line = graphics.Line(graphics.Point(0,0), graphics.Point(0,699))

for i in range(1,7):
lx[i]=line.clone()
lx[i].move(i*100,0)
lx[i].draw(win)
print(i)

ly=list(range(1,8))
liney = graphics.Line(graphics.Point(0,0), graphics.Point(699,0))

for i in range(1,7):
ly[i]=liney.clone()
ly[i].move(0,i*100)
ly[i].draw(win)
print(i)

answer = input("Shall we finish? (y/n)")
```

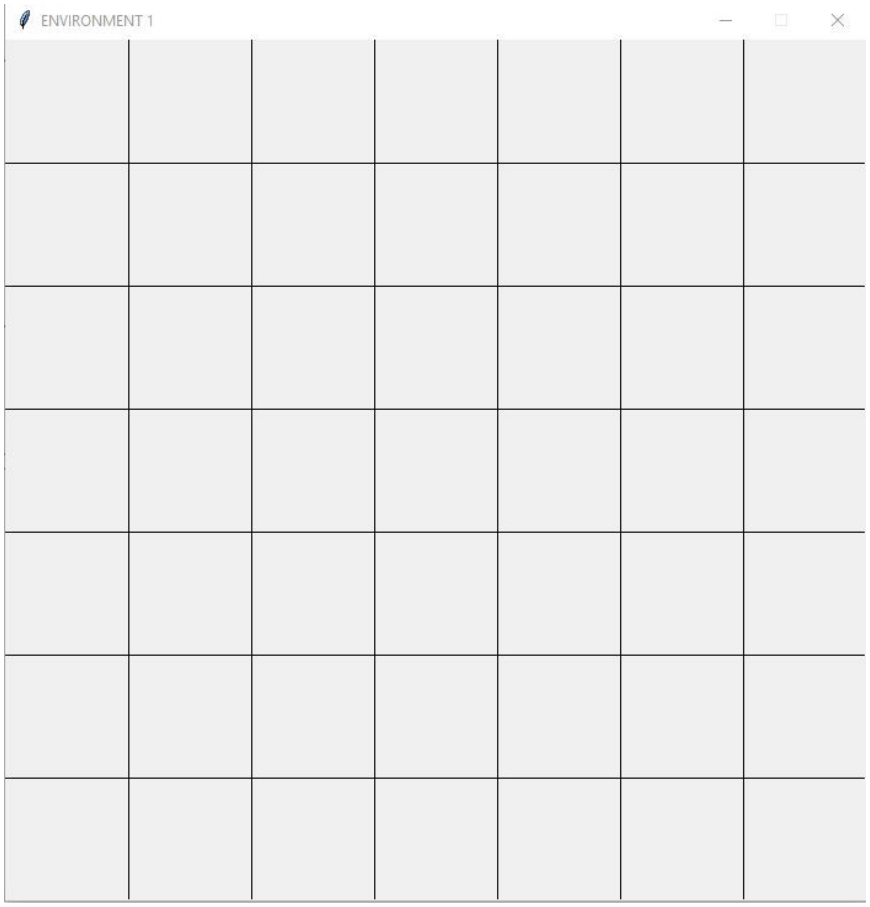


Figure 9: A full 7 x 7 grid with gdh-win4.py

```
win.close()
main()
```

4.1.6 Introduce Helper Functions

Because we know that our environment program will grow in size in the future we can consider already here to 'condense' the code a little bit by asking, which part of the code can be 'outsourced' into a separate file and which will be called 'on demand' with only a single function call. As one can see (below), one can easily take out the lines with the production of the lines in the window.

To do this we generate a new file called 'environment.py' which shall become the place of all the helping functions, define there our first helpfer

function 'grid()' and then modify our main program as 'gdh-win5.py'.

Running the main() program with this new helper function 'grid()' produces the same output as shown before (cf. figure 9).

```
# -*- coding: utf-8 -*-
# environment.py
# author: Gerd Doeben-Henisch
# Collection of functions to support simple enviroments
# Using book from John Zelle (2002) and his simple graphics.py library

def grid(win,xmax,ymax,first,last,distance):

import graphics

lx=list(range(first,last+1))
linex = graphics.Line(graphics.Point(0,0), graphics.Point(0,ymax-1))

for i in range(first,last):
lx[i]=linex.clone()
lx[i].move(i*distance,0)
lx[i].draw(win)
print(i)

ly=list(range(first,last+1))
liney = graphics.Line(graphics.Point(0,0), graphics.Point(xmax-1,0))

for i in range(first,last):
ly[i]=liney.clone()
ly[i].move(0,i*distance)
ly[i].draw(win)
print(i)
```

```
# -*- coding: utf-8 -*-
# gdh_win4.py
# author: Gerd Doeben-Henisch
# Trying a first environment
# Using book from John Zelle (2002) and his simple graphics.py library

def main():

import graphics         #Graphics objects from Zelle
import environment      #Functions to generate grids
```

```
win = graphics.GraphWin('ENVIRONMENT 1', 700, 700)

xmax = 700
ymax = 700
first = 1    #First element of the list
last = 7     # Last element of the list used
distance = 100

environment.grid(win,xmax, ymax, first, last, distance)

answer = input("Shall we finish? (y/n)")

win.close()
main()
```

4.2 Introducing Objects

Now with a first simple grid we want to introduce first *objects*. We distinguish two big categories: actor-objects and non-actor-objects. Before we are going into the details we stay with objects in general.

We distinguish between the *graphical presentation* of the grid with objects and the *object data structure* which is somehow 'behind' the scene. Everything which is important to know about the objects will be kept in this object data structure and only that what should be visible in some moment of time should be 'mapped' from the data structure into the visual representation. There are other representations possible like 'sounds', 'smells' etc.; all the needed data are hidden in the objects data structure.

There are many strategies possible to manage this. We start here with the following simple structure:

1. *OBJEKTDISTRIBUTION (OD):* There is a *n-m-array (a 'matrix')* as a 1-to-1 mapping of the graphics grid which contains numbers representing the kind of object which shall be present in the world. '0' represents empty space, '1' obstacles, '2' food, and '3' are agents.
2. *OBJECT GENERATION (OG):* This object distribution will in the beginning be generated by random as a percentage 'nobj' of the available array space.
3. *FOOD GENERATION (FG):* The food generation will be handled similar to the object generation.

The following helper function from the file 'environment.py' produces first an array of thre wanted dimension and then fills this array with zeros. Afterwards the coordinates of the wanted objects will randomly be computed. On the console you can see some printing of the final distribution, but be aware. numpy translates the rows of the array into a string and the string shows the rows as columns!

```
def fillgridobj2(win,first, last, maxXcoord,maxYcoord,nobj):

#nobj gives the percentage how many  objects shall be inserted into the matrix

import graphics as grph
import numpy as np

#Compute number of wanted objects by percentage

number=int(((maxXcoord*maxYcoord)/100)*nobj)
if number <1:
number=1

gr2=np.zeros((maxXcoord,maxYcoord),int)

# Produce randomly the coordinates of the wanted objects

for i in range(1,number):
p=np.random.randint(first,last+1,size=2) #coordinates for the fillgridobj
print(p)

gr2[p[0]-1,p[1]-1]=1    # Set a marker in the data array
x=p[0]-1
y=p[1]-1

#translating the fillgridob coordinates into the graphics coordinates

rect1 = grph.Rectangle(grph.Point(x*100,y*100), grph.Point((x+1)*100,(y+1)*100))
rect1.setFill('black')
rect1.draw(win)

print('MAP OF ARRAY \n')
print(str(gr2))
print('ATTENTION: Columns represent Rows on screen!\n')
```

```
return gr2
```

After providing an area with spaces and obstacles the next function computes randomly food positions according to the percentage 'nfood'. On the console you can again see some printing of the final distribution with the same oddity of translating rows into columns.

```
def fillgridfood(win,first,last, maxXcoord,maxYcoord,nfood,gr2):

#n gives the numer of food inserted into the matrix

import graphics as grph
import numpy as np

number=int(((maxXcoord*maxYcoord)/100)*nfood)
if number <1:
number=1


for i in range(1,number+1):
p=np.random.randint(first,last+1,size=2) #coordinates for the fillgridobj
print(p)
gr2[p[0]-1,p[1]-1]=2
#translating the fillgridob coordinates into the graphics coordinates
rect1 = grph.Rectangle( grph.Point( (p[0]-1)*100,(p[1]-1)*100 ),grph.Point( p[0]*100
rect1.setFill('green')
rect1.draw(win)

print('OBSTACLES with FOOD \n')

# This concersion shows a matrix whose columns correspond to the
# rows of the original matrix !

gr2s=str(gr2)
print(gr2s)
print('anders \n')

#This conversion shows rows after the conversion which correspnd
#to the columns in the original array !

gprint(gr2,last)

return gr2
```

You can see an example of the columns-row mapping below. This shows a printout on the console from the small world shown in figure 10.

```
OBSTACLES with FOOD

[[0 0 0 1 0 0 0]<--- translate columns here into rows in the figure
[0 1 0 1 0 0 0]
[0 0 0 0 0 0 0]
[1 0 0 0 0 0 0]
[0 0 0 1 1 0 0]
[0 0 2 1 0 0 0]
[0 0 0 0 1 0 0]]
anders

00010000101000000000010000000001100002100000000100

0001000<--- translate these rows into columns in the figure

0101000

0000000

1000000

0001100

0021000

0000100

Shall we finish? (y/n)y
```

Here is the file with the main() program labeled 'gdh-win7.py'. After the import commands and some value-settings a first call initiates a graphic window 'win'.

Then this graphic window 'win' will be filled with horizontal and vertical lines to produce optical a grid.

With the graphical grid given a data structure is generated in the background which represents an array 'gr2' with the size 7 x 7 representing the cells of the graphical grid.

This data array will be filled with spaces ’0’ and obstacles ’1’ (black colored squares).

Then food objects ’2’ are inserted in the data array (green colored squares).

In parallel are the objects from the data array mapped into the graphics window.

```
# -*- coding: utf-8 -*-
# gdh-win7.py
# author: Gerd Doeben-Henisch
# Trying a first environment
# Using book from John Zelle (2002) and his simple graphics.py library

def main():
import graphics as grph        #Graphics onbjects from Zelle
import environment as env      #Functions to generate grids
import numpy as np             #numerical laibrary numpy

xmax = 700
ymax = 700

win = grph.GraphWin('ENVIRONMENT 1', xmax, ymax)


first = 1    #First element of the list
last = 7     # Last element of the list used
distance = 100   #distance of two lines in thr grid
maxXcoord = 7  #size of the data array in the background
maxYcoord = 7
nobj = 20    #objects: percentage of the array space !
nfood = 1    #food: percentage of the array space !

#generating the lines of the grid

env.grid(win,xmax, ymax, first, last, distance)

# Fill the grid with spaces and objects
gr2=env.fillgridobj2(win,first, last, maxXcoord,maxYcoord,nobj)

# Fill the grid with food
gr2=env.fillgridfood(win,first, last, maxXcoord,maxYcoord,nfood,gr2)
```

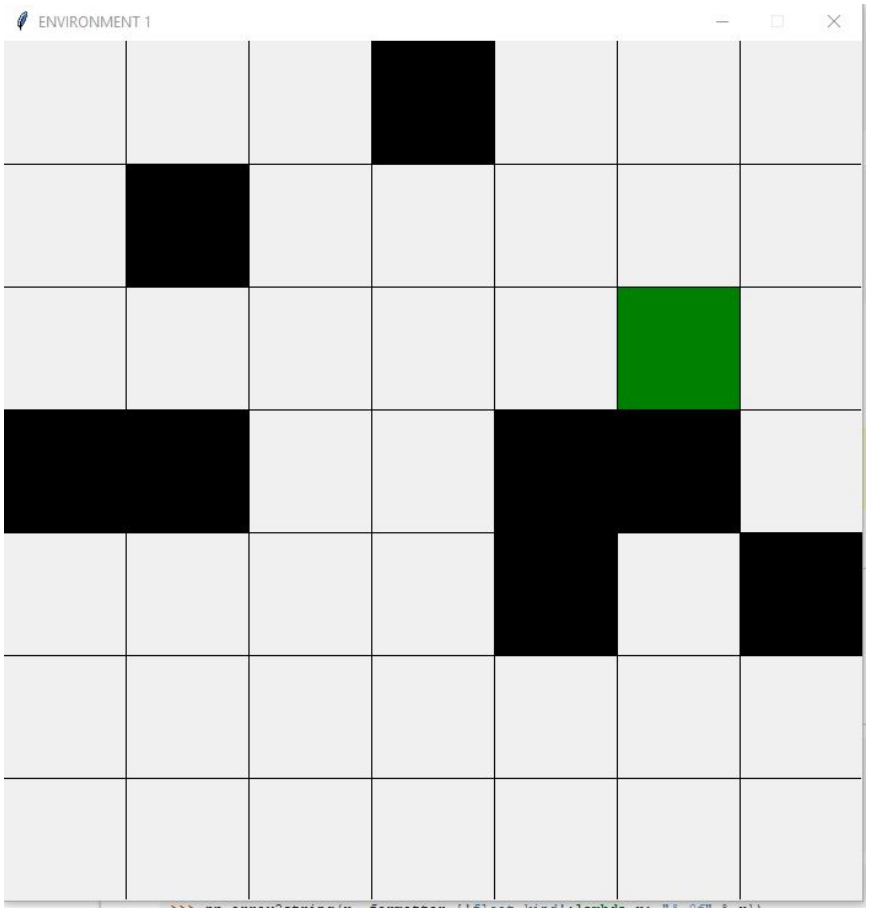


Figure 10: Graphic window with spaces, obstacles (black squares), and food (green squares).

```
answer = input("Shall we finish? (y/n)")

win.close()
main()
```

4.3 Introducing Actors

4.3.1 First Requirements for an Actor

Although the environment is still very simple we will not improve it now but we will have some first look to some simple actors.

From the view of the environment is an actor nothing else than another object having some shape (here a 'circle') with some color (here 'red'), but with at least wo additional properties: an actor has some form of *perception*, taking some kind of 'input' from the external environment 'into' his *internal states (IS)*, and he has some kind of a *response* or *reaction* to the environment. Such a response does some change to the environment. In our case we assume that the actor can *move* on the plane in four directions which are parallel to the y and x axis. He can move 'forward' or he can 'turn' by 90^o left or right, and this as often he wants. Thus this simple actor can turn around in a circle and can move in every of the four possible directions.

Because there can be more than one actor (not directly in the beginning but soon :-)) we assume here that an actor has a 'label' which gives him some unique shape-identity that he can distinguished from other actors.

Our first actor will need some amount of energy when he is alive, depending alone from the time running; if this energy level would go below some threshold θ_{Energy} then the actor would start dying. There is some critical period where the actor can not move any more but another actor could him bring some food. If this critical period has passed over the actor will die.

As long as an actor has enough energy to move he will travel around in his environment. If he finds some food then he will eat it. This can increase his energy level depending from the amount of energy kept in the food.

4.3.2 First Considerations How to Program

Because the actor has 'two faces': being an object and being an input-output system with inputs (perceptions) and outputs (responses) we will consider both aspects separately.

The actor as object needs some basic layout as an object. For this we take the simple object-class so far and extend this class a bit. See below:

```
class ENVOBJ:

import graphics as grph

def __init__(self, kind,x,y,x2,y2,icolor):
```

```
self.kind = kind
self.position = ((x,y),(x2,y2))
self.color = icolor

def getType(self):
return self.kind

def setType(self,intype):
self.kind = intype

def getPos(self):
return self.position

def setPos(self,x,y,x2,y2):
self.position = ((x,y),(x2,y2))

def getColor(self):
return self.color

def setColor(self,incolor):
self.color = incolor
```

This class allows some basic properties like 'type' of object, 'color', and 'position'. Because this was mainly related to 'obstacle' objects these positions where directly given with the coordinates of the graphic window. The planned actor is a movable object, thus we should give the coordinates first for the data structure and only later, if needed, this will be mapped to the graphic window. Furthermore we need some label for the identification. Another important information is the actual direction, because the actual direction influences the field of the actor vision. We do not allow that the actor has a 360^o view. Instead he has only a fragment of his environment and he must find some way (in the long run) to built some more general views of his environment.

Here is a first class-definition for actor-objects.

```
class ACTOBJ:
#The actor object has more propeties as a simple environment object.

def __init__(self,x,y,label,dir,energy):

self.kind = 'actor'
self.position = (x,y)
```

```
self.color = 'red'
self.label = label
self.dir = dir
self.energy = energy

def getType(self):
return self.kind

def setType(self,inkind):
# Basically an actor will stay an actor. But it is conceivable
# that there are more different types of actors in the future
self.kind = inkind

def getPos(self):
# This is the position in the data array with simple (x,y) coordinates
return self.position

def setPos(self,x,y):
# This is the position in the data array with simple (x,y) coordinates
self.position = (x,y)

def setDir(self,dir):
# There are four basic directions as {1,2,3,4}
self.dir = dir

def getDir(self):
# There are four basic directions as {1,2,3,4}
return self.dir

def setEnergy(self,energy):
# Show actual energy level
self.energy = energy

def getEnergy(self):
# Show actual energy level
return self.energy

def setLabel(self,label):
#This should be a short name different from others; a number
# is enough ...
self.label = label

def getLabel(self):
#This should be a short name different from others; a numer
```

```
# is enough ...
return self.label

def getColor(self):
# All actors have the color red (for now; perhaps more later)
return self.color

def setColor(self,incolor):
# All actors have the color red (for now; perhaps more later)
self.color = incolor
```

Then we need a function to introduce an actor in the environment. A first sketch is here:

```
def introduceActor(x,y,dir,label,energy,distance,win):

# Convert array coordinates (x,y) into window coordinates (xwin,ywin)
import graphics as grph
xwin = ((x-1)*distance)+(distance/2)
ywin = ((y-1)*distance)+(distance/2)

#gnerate a red circle
center = grph.Point(xwin,ywin)
circ = grph.Circle(center, distance/2)
circ.setFill('red')
circ.draw(win)

#gnerate a name as a lable
name = grph.Text(center, "A1")
name.draw(win)
```

Within the main program 'gdh-win8.py' we have written a simple call for this function:

```
x=5
y=5
dir=1
label= "A"
energy=200

env.introduceActor(x,y,dir,label,energy,distance,win)
```

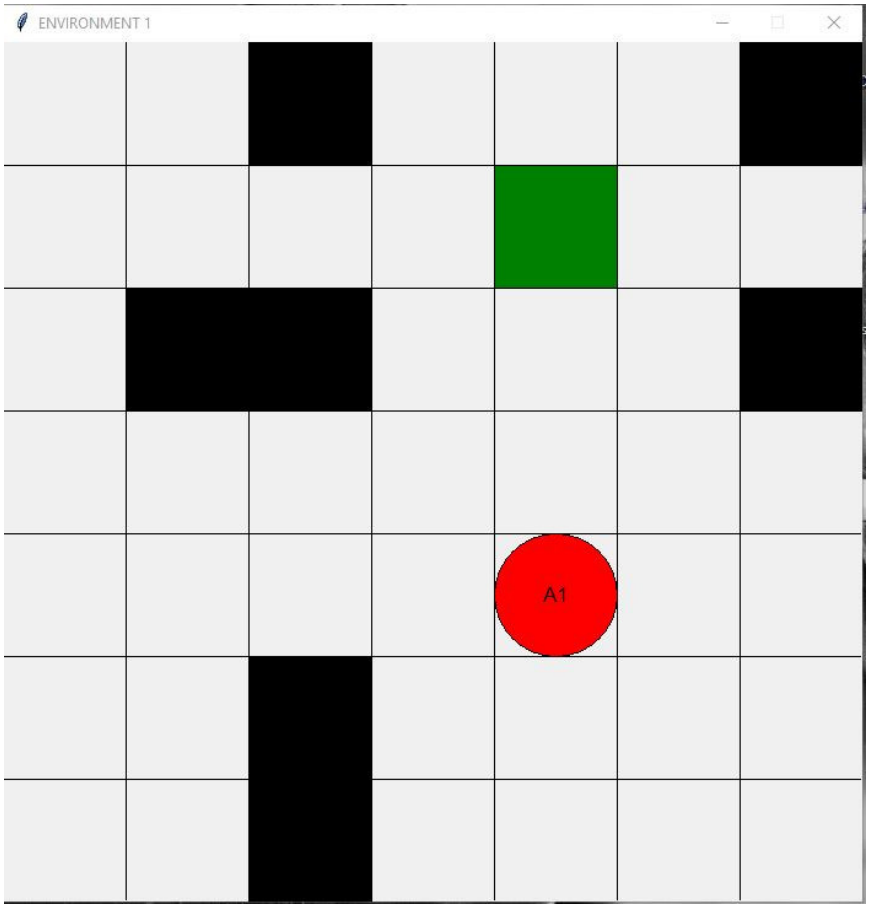


Figure 11: Environment with objects (black squares), food (green squares), and a first actor (red circle)

In the figure 11 one can see how the actor appears as a red circle with a name-label in the grid.

4.3.3 Let an Actor Move: Environment

Now where we have a first idea how we can introduce an actor inside the grid-environment, we want to clarify how we can enable an actor to move around.

The basic idea sounds simple: (i) take a *direction*, (ii) compute the *new coordinates*, (iii) *check*, if the intended cell of the grid is *not occupied* by another object; if free then *do a move*; if not free, then take a *new direction*.

Because we can assume by default that an actor is always placed in the grid with at least one free cell in his neighborhood there will always be at least one move possible. The only theoretical limits can be the exhaustion of the energy or that the free space is a small, encircled area of the grid-world with no path to some food.

In the new version of the python main program one can see that the main program has now been partitioned in two parts:

1. SET UP OF THE ENVIRONMENT
2. EVENT LOOP

Until now the program did set up a grid-environment as an intended test-bed for possible actors. Now, when we have this, we can start an *event loop* which allows the actors to do some work in this grid environment.

For this to work we have to distinguish between two perspectives:

1. ENVIRONMENT MANAGER
2. ACTOR BEHAVIOR FUNCTION

The part of the actor will be managed by the *behavior function* of an *actor*. This function takes some input from the world, computes some response, and then sends back the planned response. Every actor can have his own behavior function!

The *environment manager* provides all actors with the necessary input from the world and takes account of all responses. It is the duty of the environment manager to clarify whether a planned action is possible or not. If the actor e.g. wants to make a move onto a cell of the grid which is already occupied this will not be allowed (an exception would be, that the planning actor has some powerful action at hand, which would allow him, to enable such a move).

Thus what is needed is an *event loop* managed by the *environment manager*, which delivers the necessary messages to the actors (their input) and asks back what responses are known.[4].

In the following code of main you can detect a first outline of this new partitioning in 'SET UP OF ENVIRONMENT' and 'EVENT LOOP'.

[4] In a distributed real-time system this would be done within time-frames; in our simple scenario the environment manager sets up some scheduling rule for this

In the Setup an environment is generated with window, data array and a first actor.

Then the beginning of the event loop is visible. The environment manager looks up for the actual direction and position of the introduced actor a1. Then he looks whether the intended new position is possible. If Yes then he changes the position of the actor a1 in the data array as well in the window. This includes the deletion of the old positions and the update of the position in the actor class. The actor class works here like an interface between the environment and the behavior function which is internal to the actor.

```
# -*- coding: utf-8 -*-
# gdh-win7.py
# author: Gerd Doeben-Henisch
# Trying a first environment
# Using book from John Zelle (2002) and his simple graphics.py library

def main():
import graphics as grph        #Graphics onbjects from Zelle
import environment as env      #Functions to generate grids
import numpy as np             #numerical laibrary numpy
import acctor as acc           #Acctor related functions

##
## SET UP THE ENVIRONMENT
##

xmax = 700
ymax = 700

win = grph.GraphWin('ENVIRONMENT 1', xmax, ymax)
win.setBackground('HoneyDew')

first = 1    #First element of the list
last = 7     # Last element of the list used
distance = 100   #distance of two lines in thr grid
maxXcoord = 7  #size of the data array in the background
maxYcoord = 7
nobj = 20    #objects: percentage of the array space !
nfood = 1    #food: percentage of the array space !

#generating the lines of the grid
```

```
env.grid(win,xmax, ymax, first, last, distance)

# Fill the grid with spaces and objects
gr2=env.fillgridobj2(win,first, last, maxXcoord,maxYcoord,nobj)

# Fill the grid with food
gr2=env.fillgridfood(win,first, last, maxXcoord,maxYcoord,nfood,gr2)

# Introducing an actor for the event-loop

dir=1
label="A1"
energy=200

a=input("Please an x-coordinate (1-7) for the actor ")
b=input("Please an y-coordinate (1-7) for the actor ")
x=int(a)
y=int(b)

a1=acc.ACTOBJ(x,y,dir,label,energy,distance)

a1.setPos(x,y)

env.introduceActor(x,y,dir,label,energy,distance,win)

##
## START AN EVENT LOOP FOR THE ENVIRONMENT
##
## ASK ACTOR BEHAVIOR FUNCTIONS WHAT TO DO
## CHECK WETHER ACTIONS ARE POSSIBLE
## MODIFY THE ENVIRONMENTR IF NECESSARY
## REPEAT AS LONG AS ACTORS ARE ALIVE
##

# condition for while

FINISH=10

while FINISH >0:

# Look to the actor class for new responses

dir=a1.getDir()
```

```
print('Direction = ', dir)
p=a1.getPos()
print('Actual Position = ',p)
xold=p[0]
yold=p[1]

# If there are responses then compute the next possible move

Err, xnew, ynew = env.newPosition(dir,xold,yold)
print('Err = ', Err)
print('New Position = ', (xnew,ynew))

#Check whether this new position  is possible

if gr2[xnew-1,ynew-1] != 0:
a1.setInputMessage("NoMove")
print('type =',type,' has not moved')
else:
type=3 #actor
#First change position in data array
env.moveArray(type,xold,yold,xnew,ynew,gr2)
a1.setPos(xnew,ynew)
#Next change position in window
env.introduceActor(xnew,ynew,dir,label,energy,distance,win)
env.deleteActor(p,win)


# send the actor a feedback by changing the actor class values

# call the actor behavior function

# The actor will compute i t's response and will update the class

#counting FINISH down for to stop after finitely many steps
FINISH=FINISH-1

# repeat loop

answer = input("Shall we finish? (y/n)")

win.close()
main()
```

In the figure 12 you can see the actor A1 after he has been moved

from the environment manager from start-position (6,6) to the position (6,1) where he came to stop on account of an obstacle ahead of him. In the full program the actor A1 will interact with the environmanager and change his direction, but here the behavior function of the actor is not yet implemented.

Nevertheless shows this the different views of the environment and the actor. The environment-manager takes the actor as he appears to the environment by direction and position and moves the actor as long as there is no change.

This power of the environment-manager is strong. We can limit it by introducing an additional condition saying to the 'outside' besides actual position and direction whether the actor 'wants' to move or not. Then the environment manager would move the actor only if he wants to move (independent of the real possibility to move or not).

```
Please an x-coordinate (1-7) for the actor 6

Please an y-coordinate (1-7) for the actor 6

Direction =  1
Actual Position =  (6, 6)
Err =  0
New Position =  (6, 5)
Type = 3  has moved to (6, 5)
Direction =  1
Actual Position =  (6, 5)
Err =  0
New Position =  (6, 4)
Type = 3  has moved to (6, 4)
Direction =  1
Actual Position =  (6, 4)
Err =  0
New Position =  (6, 3)
Type = 3  has moved to (6, 3)
Direction =  1
Actual Position =  (6, 3)
Err =  0
New Position =  (6, 2)
Type = 3  has moved to (6, 2)
Direction =  1
Actual Position =  (6, 2)
Err =  0
```

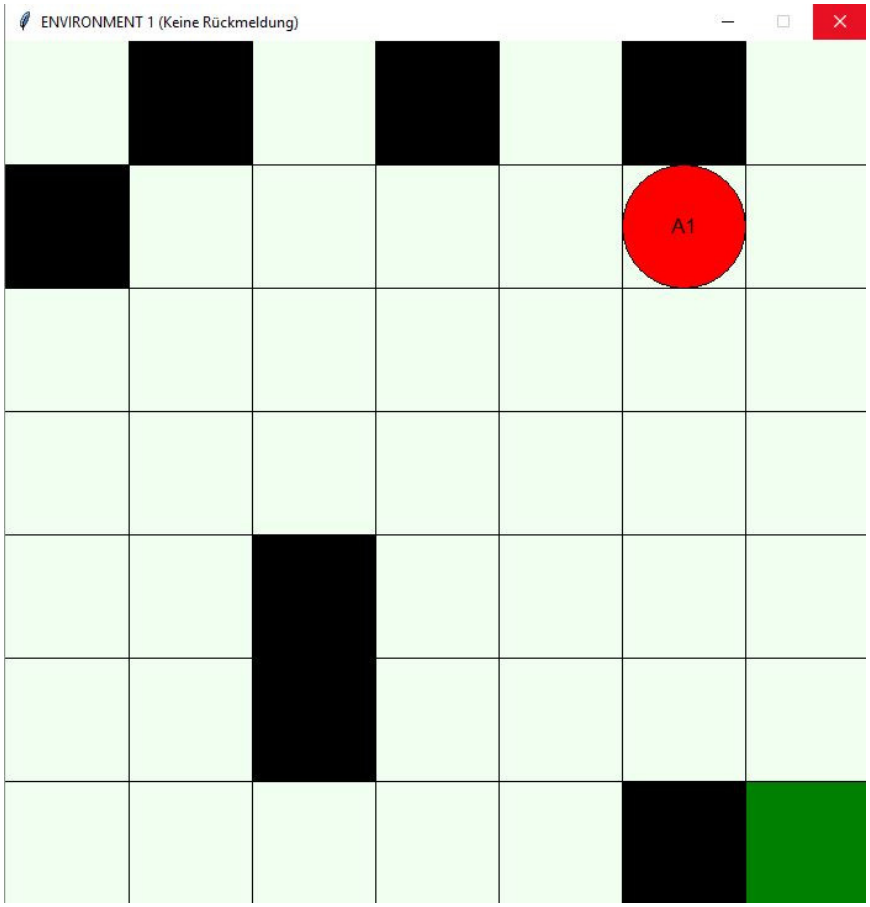


Figure 12: Actor A1 after moving from the start until he has been blocked

```
New Position =  (6, 1)
type = 3  has not moved
```

Here you can find the new helper functions to calculate the new planned position 'newPosition(dir,xold,yold)', a move in the data array 'moveArray(type,xold,yold,xnew,ynew,g as well as the deletion of the old position in the window ' deleteActor(p,win)'.

```
def deleteActor(p,win):

# Convert array coordinates (x,y) into window coordinates (xwin,ywin)
import numpy as np
import graphics as grph

x=p[0]-1
y=p[1]-1

#translating the fillgridob coordinates into the graphics coordinates

rect1 = grph.Rectangle(grph.Point(x*100,y*100), grph.Point((x+1)*100,(y+1)*100))
rect1.setFill('HoneyDew')
rect1.draw(win)

#Move to new position in data array

def moveArray(type,xold,yold,xnew,ynew,gr2):
import numpy as np
gr2[xold-1,yold-1]=0
gr2[xnew-1,ynew-1]=type
print('Type =',type,' has moved to',(xnew,ynew))


# Compute a new position starting with an old position and a direction

def newPosition(dir,xold,yold):

xnew=xold
ynew=yold
Err=0

if dir == 1:
ynew=yold-1
elif dir == 2:
```

```
xnew=xold+1
elif dir == 3:
ynew=yold+1
elif dir == 4:
xnew=xold-1
else:
print('ERROR : Wrong Direction ! ')
Err=-1

return Err, xnew, ynew
```

Finally the actual actor-class definition, which is in use for actor a1 (A1).

```
# -*- coding: utf-8 -*-
# actor.py
# author: Gerd Doeben-Henisch
# Collection of functions to support simple actors

class ACTOBJ:
#The actor object has more propeties as a simple environment object.

def __init__(self,x,y,dir,label,energy,distance):
self.kind = 'actor'
self.position = (x,y)
self.color = 'red'
self.label = label
self.dir = dir
self.energy = energy

def getType(self):
return self.kind

def setType(self,inkind):
# Basically an actor will stay an actor. But it is conceivable
# that there are more different types of actors in the future
self.kind = inkind

def getPos(self):
# This is the position in the data array with simple (x,y) coordinates
return self.position

def setPos(self,x,y):
# This is the position in the data array with simple (x,y) coordinates
```

```
# The array counts from 0 ... n-1
# The user counts from 1 ... n
self.position = (x,y)

def setDir(self,dir):
# There are four basic directions as {1,2,3,4}
self.dir = dir

def getDir(self):
# There are four basic directions as {1,2,3,4}
return self.dir

def setEnergy(self,energy):
# Show actual energy level
self.energy = energy

def getEnergy(self):
# Show actual energy level
return self.energy

def setLabel(self,label):
#This should be a short name different from others; a numer
# is enough ...
self.label = label

def getLabel(self):
#This should be a short name different from others; a number
# is enough ...
return self.label

def getColor(self):
# All actors have the color red (for now; perhaps more later)
return self.color

def setColor(self,incolor):
# All actors have the color red (for now; perhaps more later)
self.color = incolor

def setInputMessage(self,inputMessage):
#Some message from the environment
self.inputMessage=inputMessage

def getInputMessage(self):
return  self.inputMessage
```

4.3.4 Let an Actor Move: Actor Self

Till now we have described what the environment-manager can do. It's time now to have a look to the actor.

Clearly, describe an actor is in principle an infinite story. Everything you can imagine you can put into an actor.

Here we will start as simple as possible.

We make a general decision: we distinguish here between BODY FUNCTIONS which are – like in the case of real biological systems – mostly AUTOMATIC, and MENTAL FUNCTIONS which are by definition CONSCIOUS actions.

Let us first have a look to the basic ability to *move*.

Given by the world and the body so far the actual actor can do two things: (i) he can change the *direction* and (ii) he can do *a move* or not.

For the change of the direction we propose the following mechanism:

1. Decide whether you will keep your actual direction or whether you will turn 'left' from now or 'right' (by chance).
2. If you have decided the direction you have to decide whether you will move or not.

That's all for now.

Independent of your mental actions there is a body function consuming energy depending from time. To solve the problem of having a common time for the whole environment we introduce an ENVIRONMENT-CLOCK giving time stamps for all.

To enable an environment clock as source for an environment time we have rewritten the class ENVOBJ. Until now we didn't really use this class. Now it's time to do this.

As you can see below the class ENVOBJ contains two properties: a name and a clock.

```
class ENVOBJ:
```

```
def __init__(self,name,clock):

self.name = name
self.clock = clock

def getName(self):
return self.name

def setName(self,name):
self.name = name

def getClock(self):
return self.clock

def setClock(self,clock):
self.clock = clock
```

In the main program 'gdh-win10.py' we have inserted in the beginning a constructor for building an instance of an environment object called 'e1':

```
e1=env.ENVOBJ('env1',0)
```

The new environment object has the name 'e1' and the clock begins counting with '0'. This counting is demonstrated in the log-data from a short run below.

```
Please an x-coordinate (1-7) for the actor 3

Please an y-coordinate (1-7) for the actor 5

Direction =  1
Actual Position =  (3, 5)
No move wanted by actor.
Actual Environment Time =  0
New  Environment Time =  1
Direction =  1
Actual Position =  (3, 5)
No move wanted by actor.
Actual Environment Time =  1
New  Environment Time =  2
...
Shall we finish? (y/n)
```

Therefore we can decrease the amount of energy depending from the environment time by some fixed amount.[5]

The following log shows how the energy level is decreasing until you get a warning that the actor is running out of energy supply.

```
...
Please an x-coordinate (1-7) for the actor 2

Please an y-coordinate (1-7) for the actor 7

Direction =  1
Actual Position =  (2, 7)
No move wanted by actor.
Environment Time = 0
Actor  A1  has new energy level =  2000
New direction of actor  A1  is =  3
actor  A1 wants to move
Actual Environment Time =  0
New  Environment Time =  1

Next Step Wanted (y/n)y
Direction =  3
Actual Position =  (2, 7)
Err =  0
New Position planned =  (2, 7)
Type = 3  has moved to (2, 7)
Environment Time = 1
Actor  A1  has new energy level =  1998
New direction of actor  A1  is =  2
actor  A1 wants to move
Actual Environment Time =  1
New  Environment Time =  2

Next Step Wanted (y/n)y
Direction =  2
Actual Position =  (2, 7)
Err =  0
New Position planned =  (3, 7)
Type = 3  has moved to (3, 7)
Environment Time = 2
```

[5] With more complex systems the amount of consumed energy will depend from several parameters, not only from the time spent.

```
Actor  A1  has new energy level =  1994
New direction of actor  A1  is =  4
actor  A1 wants to move
Actual Environment Time =  2
New  Environment Time =  3

Next Step Wanted (y/n)y
Direction =  4
Actual Position =  (3, 7)
Err =  0
New Position planned =  (2, 7)
Type = 3  has moved to (2, 7)
Environment Time = 3
Actor  A1  has new energy level =  1988
New direction of actor  A1  is =  1
actor  A1 wants to move
Actual Environment Time =  3
New  Environment Time =  4

Next Step Wanted (y/n)y
Direction =  1
Actual Position =  (2, 7)
Err =  0
New Position planned =  (2, 6)
actor = A1  has not moved
Environment Time = 4
Actor  A1  has new energy level =  1980
New direction of actor  A1  is =  2
actor  A1 wants to move
Actual Environment Time =  4
New  Environment Time =  5

Next Step Wanted (y/n)y
Direction =  2
Actual Position =  (2, 7)
Err =  0
New Position planned =  (3, 7)
Type = 3  has moved to (3, 7)
Environment Time = 5
Actor  A1  has new energy level =  1970
New direction of actor  A1  is =  4
actor  A1 wants to move
Actual Environment Time =  5
New  Environment Time =  6
```

```
Next Step Wanted (y/n)y
Direction =  4
Actual Position =  (3, 7)
Err =  0
New Position planned =  (2, 7)
Type = 3  has moved to (2, 7)
Environment Time = 6
Actor  A1  has new energy level =  1958
New direction of actor  A1  is =  3
actor  A1 wants to move
Actual Environment Time =  6
New  Environment Time =  7
...

Next Step Wanted (y/n)y
Direction =  2
Actual Position =  (2, 7)
Err =  0
New Position planned =  (3, 7)
Type = 3  has moved to (3, 7)
Environment Time = 45
Actor  A1  has new energy level =  -70
New direction of actor  A1  is =  4
actor  A1 wants to move
Actual Environment Time =  45
New  Environment Time =  46
ATTENTION: Actor  A1  has no more Energy!!!
...
Next Step Wanted (y/n)y

You want to close the window? (y/n)y
```

4.3.5 The Final Main Procedure

Here you can see the final main() procedure for the first milestone.[6].

```
# -*- coding: utf-8 -*-
# gdh-win10.py
# author: Gerd Doeben-Henisch
# Trying a first environment
# Using book from John Zelle (2002) and his simple graphics.py library
```

[6] There will be a direct download on the web-page too

```
def main():
import graphics as grph        #Graphics onbjects from Zelle
import environment as env      #Functions to generate grids
import numpy as np             #numerical laibrary numpy
import acctor as acc           #Acctor related functions

##
## SET UP THE ENVIRONMENT
##

e1=env.ENVOBJ('env1',0)

xmax = 700
ymax = 700

win = grph.GraphWin('ENVIRONMENT 1', xmax, ymax)
win.setBackground('HoneyDew')


first = 1    #First element of the list
last = 7     # Last element of the list used
distance = 100   #distance of two lines in thr grid
maxXcoord = 7  #size of the data array in the background
maxYcoord = 7
nobj = 20    #objects: percentage of the array space !
nfood = 1    #food: percentage of the array space !

#generating the lines of the grid

env.grid(win,xmax, ymax, first, last, distance)

# Fill the grid with spaces and objects
gr2=env.fillgridobj2(win,first, last, maxXcoord,maxYcoord,nobj)

# Fill the grid with food
gr2=env.fillgridfood(win,first, last, maxXcoord,maxYcoord,nfood,gr2)

# Introducing an actor for the event-loop

time=e1.getClock()  #Setting the time of 'birth'
a=input("Please an x-coordinate (1-7) for the actor ")
b=input("Please an y-coordinate (1-7) for the actor ")
x=int(a)
```

```
y=int(b)
dir=1
color = 'red'
label="A1"
energy=2000
energyrate=2

a1=acc.ACTOBJ(label,time,x,y,dir,color, label,energy,energyrate)

a1.setPos(x,y)
outputMessage='noMove'  #This will allow a move without interaction with
# the actor
a1.setOutputMessage(outputMessage)

env.introduceActor(x,y,dir,label,energy,distance,win)

##
## START AN EVENT LOOP FOR THE ENVIRONMENT
##
## ASK ACTOR BEHAVIOR FUNCTIONS WHAT TO DO
## CHECK WETHER ACTIONS ARE POSSIBLE
## MODIFY THE ENVIRONMENTR IF NECESSARY
## REPEAT AS LONG AS ACTORS ARE ALIVE
##

# condition for while

FINISH=100

while FINISH >0:

# Look to the actor class for new responses

dir=a1.getDir()
print('Direction = ', dir)
p=a1.getPos()
print('Actual Position = ',p)
xold=p[0]
yold=p[1]

if a1.getOutputMessage() == 'yesMove':

# If there are responses then compute the next possible move
```

```
Err, xnew, ynew = env.newPosition(dir,xold,yold)
print('Err = ', Err)
print('New Position planned = ', (xnew,ynew))

#Check whether this new position  is possible

if gr2[xnew-1,ynew-1] != 0:
#Telling the actor that the planned new position is occupied
a1.setInputMessage("NoMove")
print('actor =',label,' has not moved')
else:
type=3 #actor
#First change position in data array
env.moveArray(type,xold,yold,xnew,ynew,gr2)
#Tell the actor his new position
a1.setPos(xnew,ynew)
#Next change position in window
env.introduceActor(xnew,ynew,dir,label,energy,distance,win)
env.deleteActor(p,win)
else:
print('No move wanted by actor.')

# Get the actual environment time

t2=e1.getClock()
print('Environment Time =', t2)

#Call the actor

acc.behaviorAc(e1,a1)


# The actor will compute i t's response and will update the class

#counting FINISH down for to stop after finitely many steps
FINISH=FINISH-1

envClock = e1.getClock()
print('Actual Environment Time = ',envClock)
envClock= envClock+1
e1.setClock(envClock)
print('New  Environment Time = ',envClock)

energy=a1.getEnergy()
```

```
if energy <0:
print('ATTENTION: Actor ',label,' has no more Energy!!!')

answer = input("Next Step Wanted (y/n)")
if answer == 'y':
FINISH = FINISH  #Repeat the Loop

else:
FINISH = -1    #Finish

answer = input("You want to close the window? (y/n)")

win.close()
main()
```

4.3.6 Final Helper Functions for the Environment

Here you can see the final version for Milestone 1 of the helper functions used in the main() procedure.

```
# -*- coding: utf-8 -*-
# environment.py
# author: Gerd Doeben-Henisch
# Collection of functions to support simple enviroments
# Using book from John Zelle (2002) and his simple graphics.py library

class ENVOBJ:

def __init__(self,name,clock):

self.name = name
self.clock = clock

def getName(self):
return self.name

def setName(self,name):
self.name = name

def getClock(self):
return self.clock

def setClock(self,clock):
self.clock = clock
```

```
def grid(win,xmax,ymax,first,last,distance):

import graphics as grph
import numpy as np

lx=list(range(first,last+1))
linex = grph.Line(grph.Point(0,0), grph.Point(0,ymax-1))

for i in range(first,last):
lx[i]=linex.clone()
lx[i].move(i*distance,0)
lx[i].draw(win)
print(i)

ly=list(range(first,last+1))
liney = grph.Line(grph.Point(0,0), grph.Point(xmax-1,0))

for i in range(first,last):
ly[i]=liney.clone()
ly[i].move(0,i*distance)
ly[i].draw(win)
print(i)


def fillgridobj(win,first, last, xmax,ymax):
import graphics as grph
import numpy as np

# Generate a matrix with a random distribution of '0' and '1'.
gr2=np.random.randint(2, size=(last,last))

print('MAP OF ARRAY \n')
print(str(gr2))
print('ATTENTION: Columns represent Rows on screen!\n')

for y in range(0,7):
for x in range(0,7):
if gr2[x,y] == 1:
print(x+1,y+1)
rect1 = grph.Rectangle(grph.Point(x*100,y*100), grph.Point((x+1)*100,(y+1)*100))
rect1.setFill('black')
```

```
rect1.draw(win)

return gr2

def fillgridobj2(win,first, last, maxXcoord,maxYcoord,nobj):

#nobj gives the percentage how many  objects shall be inserted into the matrix

import graphics as grph
import numpy as np

#Compute number of wanted objects by percentage

number=int(((maxXcoord*maxYcoord)/100)*nobj)
if number <1:
number=1

gr2=np.zeros((maxXcoord,maxYcoord),int)

# Produce randomly the coordinates of the wanted objects

for i in range(1,number):
p=np.random.randint(first,last+1,size=2) #coordinates for the fillgridobj
print(p)

gr2[p[0]-1,p[1]-1]=1    # Set a marker in the data array
x=p[0]-1
y=p[1]-1

#translating the fillgridob coordinates into the graphics coordinates

rect1 = grph.Rectangle(grph.Point(x*100,y*100), grph.Point((x+1)*100,(y+1)*100))
rect1.setFill('black')
rect1.draw(win)

print('MAP OF ARRAY \n')
print(str(gr2))
print('ATTENTION: Columns represent Rows on screen!\n')

return gr2

def fillgridfood(win,first,last, maxXcoord,maxYcoord,nfood,gr2):

#n gives the numer of food inserted into the matrix
```

```
import graphics as grph
import numpy as np

number=int(((maxXcoord*maxYcoord)/100)*nfood)
if number <1:
number=1

for i in range(1,number+1):
p=np.random.randint(first,last+1,size=2) #coordinates for the fillgridobj
print(p)
gr2[p[0]-1,p[1]-1]=2
#translating the fillgridob coordinates into the graphics coordinates
rect1 = grph.Rectangle( grph.Point( (p[0]-1)*100,(p[1]-1)*100 ),grph.Point( p[0]*100
rect1.setFill('green')
rect1.draw(win)

print('OBSTACLES with FOOD \n')

# This conversion shows a matrix whose columns correspond to the
# rows of the original matrix !

gr2s=str(gr2)
print(gr2s)
print('anders \n')

#This conversion shows rows after the conversion which correspnd
#to the columns in the original array !

gprint(gr2,last)

return gr2

def gprint(a,lasty):

#This conversion shows rows after the conversion which correspnd
#to the columns in the original array !

import numpy as np

s=""
for i in range(0,a.size):
s=s+str(a.item(i))
```

```
print(s)

for j in range(0,lasty):
print('\n ', s[(j*lasty):(j*lasty)+7])

#introduce actor in the window
def introduceActor(x,y,dir,label,energy,distance,win):

# Convert array coordinates (x,y) into window coordinates (xwin,ywin)
import numpy as np
import graphics as grph

xwin = ((x-1)*distance)+(distance/2)
ywin = ((y-1)*distance)+(distance/2)

#gnerate a red circle
center = grph.Point(xwin,ywin)
circ = grph.Circle(center, distance/2)
circ.setFill('red')
circ.draw(win)

#gnerate a name as a lable
name = grph.Text(center, "A1")
name.draw(win)

#Delete an actor on a certain position in the window

def deleteActor(p,win):

# Convert array coordinates (x,y) into window coordinates (xwin,ywin)
import numpy as np
import graphics as grph

x=p[0]-1
y=p[1]-1

#translating the fillgridob coordinates into the graphics coordinates

rect1 = grph.Rectangle(grph.Point(x*100,y*100), grph.Point((x+1)*100,(y+1)*100))
rect1.setFill('HoneyDew')
rect1.draw(win)

#Move to new position in data array
```

```
def moveArray(type,xold,yold,xnew,ynew,gr2):
import numpy as np
gr2[xold-1,yold-1]=0
gr2[xnew-1,ynew-1]=type
print('Type =',type,' has moved to',(xnew,ynew))

# Compute a new position starting with an old position and a direction

def newPosition(dir,xold,yold):

xnew=xold
ynew=yold
Err=0

if dir == 1:
if yold-1 < 1:
ynew = yold
else:
ynew=yold-1
elif dir == 2:
if xold+1 > 7:
xnew = xold
else:
xnew=xold+1
elif dir == 3:
if yold+1 > 7:
ynew = yold
else:
ynew=yold+1
elif dir == 4:
if xold-1 < 1:
xnew = xold
else:
xnew=xold-1
else:
print('ERROR : Wrong Direction ! ')
Err=-1

return Err, xnew, ynew
```

4.3.7 Final Helpfer Funtions for Actors

Here you will find the final helper Functions for Milestone 1 for the actor(s).

```
# -*- coding: utf-8 -*-
# actor.py
# author: Gerd Doeben-Henisch
# Collection of functions to support simple actors



class ACTOBJ:
#The actor object has more propeties as a simple environment object.

def __init__(self,name,time,x,y,dir,color, label,energy,energyrate):
self.name = name
self.time = time
self.position = (x,y)
self.dir = dir
self.color = 'red'
self.label = label
self.energy = energy
self.energyrate = energyrate

def getName(self):
return self.name

def setName(self,name):
#The time will be derived  from the environment time
self.name = name

def getTime(self):
return self.time

def setTime(self,time):
#The time will be derived  from the environment time
self.time = time

def getPos(self):
# This is the position in the data array with simple (x,y) coordinates
return self.position

def setPos(self,x,y):
# This is the position in the data array with simple (x,y) coordinates
# The array counts from 0 ... n-1
# The user counts from 1 ... n
self.position = (x,y)
```

```
def setDir(self,dir):
# There are four basic directions as {1,2,3,4}
self.dir = dir

def getDir(self):
# There are four basic directions as {1,2,3,4}
return self.dir

def getColor(self):
# All actors have the color red (for now; perhaps more later)
return self.color

def setColor(self,incolor):
# All actors have the color red (for now; perhaps more later)
self.color = incolor

def setLabel(self,label):
#This should be a short name different from others; a numer
# is enough ...
self.label = label

def getLabel(self):
#This should be a short name different from others; a number
# is enough ...
return self.label

def setEnergy(self,energy):
# Show actual energy level
self.energy = energy

def getEnergy(self):
# Show actual energy level
return self.energy

def setEnergyRate(self,energyrate):
# Show actual energy level
self.energyrate = energyrate

def getEnergyRate(self):
# Show actual energy level
return self.energyrate


def setInputMessage(self,inputMessage):
```

```
#Some message from the environment
self.inputMessage=inputMessage

def getInputMessage(self):
return  self.inputMessage

def setOutputMessage(self,outputMessage):
#Some message to the environment
self.outputMessage=outputMessage

def getOutputMessage(self):
return  self.outputMessage


def behaviorAc(e,a):

import numpy as np
import environment as env
import acctor as acc

#BODY PART
# Compute time since last call and change energy level

name = a.getName()
t1 = a.getTime()
t2 = e.getClock()
erate = a.getEnergyRate()
ener = a.getEnergy()

dt = t2-t1
energynew = ener - (erate * dt)
a.setEnergy(energynew)

print('Actor ',name,' has new energy level = ',energynew)

#MENTAL PART
# Determine the new direction

dirspace = 4
dirnew = np.random.randint(1,dirspace+1)
a.setDir(dirnew)
print('New direction of actor ',name, ' is = ',dirnew)

a.setOutputMessage('yesMove')
```

```
print('actor ',name, 'wants to move')
```

5 Close Up

This text describes the experiment which has started Tuesday, 10.Oct.2017 about 14:00 and ended Saturday 14.Oct.2017 about 14:00h. At the beginning I had no idea about python, only some 'pounds of disgust' about such an 'un-mathematical language'. But because the arguments to use it now and really have become so 'dense' that I started this experiment.

The final point – perhaps – has been that we decided Tuesday Morning to use some kind of software for our music performance in Hamburg at 9.Nov.2017 and that this perhaps could be done with python... The first Milestone was planned for this week to check a simple actor-environment scenario with graphical output.

As one can see now this has been solved.

I learned a lot about python but – as i learned too – this is only a small fragment of those things which are possible with python.

I have to thank all the wonderful people who put t heir work with python in the internet, first of all those who have programmed all the wonderful tools which enable you to work with python in a productive way. I am really enthusiastic about the package WinPython which includes the integrated development environment 'spyder'. This is really great: not too overloaded, but with everything you need. Thanks to those who did this!

The other point are the many sources in the internet. If I had a problem I immediately found good answers and examples in the internet, the book of John Zeller with his graphic library, the fantastic manuals of the python language and numpy, different videos, many forums. Without all this free stuff open science would nearly be impossible.

My work with python will continue (and will be shown here!).

One line of development will be the usage for our Philosophy-in-Concert experiments where we are experimenting with new ways of composing and performing.

The other line will be the development of more experiments around the emerging-mind lab (emerging-mind.org). Part of this is the integration with

the ubuntu platform and the robot operating system (ros), although with tensorflow.

The only reason that I still have windows and I am using windows is that I am working with a music program called 'ableton live' combined with 'Max4live'. Because I could not find any substitution for this in the ubuntu world I am in need for windows (there are some programs trying to mimic ableton live and Max4live but they are far, far weaker).

And there is a third line of development in the area of social, economical, and political simulations. Instead of wasting time with computer games which cost you money but do not give you any kind of new knowledge it could be a good approach to built up learning environments for the real world with intelligent machines as your helpers.

Hope to see you again on this platform ... or elsewhere :-)

References

[Zel02] John M. Zelle. *Python Programming: An Introduction to Computer Science*. Published by John Zelle, 100 Wartburg Blvd, Waverly, IA 50677, USA, 1 edition, 2002. URL: http://www.leetupload.com/database/Misc/Papers/Python