Programming for Architects

Version 2019

G. Wurzer and W.E. Lorenz
E259.1 Digital Architecture and Planning

TU WIEN TECHNISCHE UNIVERSITÄT WIEN

Cover Image:
Catania, Teatro Vincenzo Bellini
Photo by Superbizzu, La Sala del Teatro Massimo Bellini

commons.wikimedia.org/wiki/File:Catania-teatro-bellini-interno.jpg

Preface: Programming is a drama.

ROMEO AND JULIET
by William Shakespeare

Juliet ..... Daughter to the Capulet Family
Romeo ..... The Son of the Montague Clan

In a drama, it is normal to begin by introducing the audience to the cast of characters. Because programming is primarily a writing activity, the very same concept is used to give a sense of setting to a program:

ADDNUMBERS by John Doe

a .....the first Number

b.....the second Number

result ....the result of adding a and b

As in a play, the name of the program already hints at its content (or: purpose). Likewise, the names used in a program (here: (**a**), (**b**) and (**result**)) should point at their role in the now beginning chain of events.

Introducing proper names may be difficult at start. It requires a clear concept of what a program should do and in which steps it is going arrive at its solution. Architects often find it difficult to concentrate on the problem their program should solve, when all they want is really "a fancy solution".[1] But understanding the problem, breaking it up into sub-problems which are easily solvable and finally arriving at the solution is what lies at the heart of programming. To teach both knowledge about programming as well as a basic understanding on how to model and structure problems is what we want to do within the lecture "Programming for Architects". In this respect, we will often deviate from this pure "programmer's handbook" and relate to how algorithms we present work, and how they can be understood.

Gabriel P.X. Wurzer and Wolfgang E. Lorenz

Vienna University of Technology

1 Lawson: How Designers Think

Content

Preface: Programming is a drama. 5

Introduction 9

A. The stage: How to use Python in Rhino 11
- A.1. Programming language: Python 11
- A.2. Python for Rhino 12
- A.3. Python Editor 12
- A.4. Obtaining Help 13
- A.5. Debugging 14

B. First Act: The Actors 17
- B.1. Introducing Values, Data Types and Variables 17
- B.2. More on Variables 18
- B.3. Working with Numbers 18
- B.4. Manual Type Conversions 19
- B.5. Intermediate Summary 20

C. Second Act: The Play and its Script 21
- C.1. Python Functions 21
- C.2. Defining Functions, Passing Inputs, Receiving Outputs 22
- C.3. Intermediate Summary 24
- C.4. Test Case: Writing a Function that "Rolls the Dice" 24
- C.5. Intermediate Summary 26
- C.6. Advanced: Nested Function Calls 26

D. Third Act: Alternative Plots 29
- D.1. Simpler Use of Booleans in Conditions 30
- D.2. Summing up this chapter 31
- D.3. Questions 31

E. Fourth Act: The Magic of Words 33
- E.1. Advanced Character Extraction and Slicing 35
- E.2. Summing up this chapter 36
- E.3. Questions 36

F. Intermediate Stage 37
- F.1. Algorithmic Thinking 37
- F.2. Problem-centered strategy 37

G. Fifth Act: We want to draw 39
- G.1. We Need a List (an Array) 39
- G.2. The Case of the Stairs 40
- G.3. Intermediate Stage: Rhinoscriptsyntax 41
- G.4. Continuing with the Case of the Stairs 42

G.5. A Separate Command for a Single Step 43
G.6. Summing up this chapter 44
G.7. Questions 44

H. Sixth Act: We Want to Avoid Writing Unnecessary Code 47
H.1. While Loop (Pre-Test Loop) 47
H.2. For-Loop (Pre-Test Loop) 49
H.3. Do-Loop (Pre-Test or Post-Test Loop) or While True 49
H.4. Intermission: Nice Coding 50
H.5. Summing up this chapter 51
H.6. Questions 51

I. Seventh Act: Make User Interaction Easier 53
I.1. Useful Commands for Converting and Checking 55
I.2. Summing up this chapter 57
I.3. Questions 58

J. Eight Act: Interact with Grasshopper 61
J.1. GHPython component 61
J.2. Questions 63

K. Epilog: Circles in a Polygon 65
K.1. Select curve on layer 65
K.2. Questions 71

Appendix 73

Cheat Sheet: All Programming Constructs at a Glance 75

Cheat Sheet: What to do with every type 76

Cheat Sheet: RhinoScriptSyntax Drawing Functions 77

Introduction

Programming

Programming is more than writing a piece of software but a much broader activity for which we need the following components:

1. The brain, which comes up with the goal of a software (concept).
2. A text editor, in which the concept is translated into a sequence of instructions for a specific programming language (coding or implementation).
3. A program which checks our typed text for correctness and
    a. executes our instructions immediately (interpretation) or
    b. generates an executable file (e.g. program.exe) which contains the instructions in machine-readable form (compilation).

Scripting versus programming

We differentiate between two main groups of programming languages:

1. A scripting language (e.g. Visual Basic for Applications or Python) is usually integrated into a particular application (e.g. Rhino, Cinema4D). It allows the programmer to access all of the application's features for adding new functionality in the form of a "self-written" program (script). Usually applications offering scripting support come with an inbuilt code editor in which scripts can be written and executed. The scripting language 's interpreter will then execute the code line by line, checking for correctness as it moves along.
2. A programming language differs insofar as that the program is written in a stand-alone text editor (called Integrated Development Environment, IDE) which calls up a compiler that translates all instructions into machine-readable form and writes out the result as an executable file (".exe" file on Windows, ".so" or ".o" file on Linux, or a file with a special "executable" flag on a Mac). The program may then be run by clicking on the produced executable file (or have the IDE do so by clicking a "play" button).

A further difference between interpretation and compilation lies in the fact that compiled programs only run on the platform they were compiled for: An ".exe" file can only run on Windows and not on a Mac; one would need to additionally produce an executable file in Mac format for that. In contrast, code written in a scripting language runs everywhere as long as there is an interpreter – which is typically part of the hosting application (e.g. Cinema4D).

Another advantage of scripting is the possibility to use functionality of the hosting application within scripts (see e.g. the "record Macro" functionality in Microsoft Word which records all clicks and keypresses and produces a script containing application instructions). Compiled programs, on the other hand, do not offer any application-specific functionalities per default; for these to

become available, one has to load a so-called library containing instructions for that specific application first.

Given the advantages of scripting languages, most architects stick to that form of coding. In our course, we will do likewise, using Rhino as our application and Python as scripting language.

A. The stage: How to use Python in Rhino

[or what the stage looks like]

A.1. Programming language: Python

Rhino provides several opportunities for extending the application, one of which is **Rhino.Python** (others are C/C#, Grasshopper, RhionoScript, RhinoCommon, openNURBS and RhinoMobile). Being a *scripting language*, Python executes code directly within the Rhino application. The benefit of this approach lies in the fact that no software except Rhino needs to be installed in order to be able to code. On the downside, however, one cannot write independent programs that are distributed without Rhino in this manner.

Python was developed by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands in the late 1980ies (implementation began in December 1989). It was developed as a successor to the ABC language. Major releases were Python 2.0 (released on October 16th, 2000) and Python 3.0 (released on December 3rd, 2008). For more detail see *Wikipedia: History of Python*.[1]

> *Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language.*[2]

We decided to use Python because it is available in Rhino and comes with an editor that allows debugging. Originally we had chosen Visual Basic for Applications (VBA) for AutoCAD as scripting language, but decided to make a radical shift to Rhino/ Python because of the following reasons:

1. In 2007, Microsoft (the company behind VBA) urged all application vendors to switch to their new .NET platform. The intention of Microsoft was no surprise, since VBA had been around for nearly 20 years and the company did not want to invest any more resources into a scripting platform not in its main strategy line. For some years VBA was still supported, however, the switch was unavoidable and would have meant some dramatic didactic changes (e.g. compilation instead of scripting).
2. Rhino (and Grasshopper) got increasingly popular among our students. In our digital design courses of the last couple of years we made the experience that the majority of them was already working with Grasshopper (not scripting language per se but a dataflow language). Since programming (and hence this lecture) is not "only" about learning a programming language but also about the mental ability to formulate a program, switching to whatever language makes no difference once the basic programming constructs are understood.
3. The Python Editor is an integral part of the Rhino application. Code written in the Python Editor can be referred to and used in Grasshopper, which is the only additional plug-in we need (Rhino 6 already comes with Grasshopper, so even that isn't needed).

1 en.wikipedia.org/wiki/History_of_Python (02.03.2018)

2 www.tutorialspoint.com/python/index.htm (02.03.2018). Interpreted means, that the script is executed line by line, which makes the flow more understandable.

A.2. Python for Rhino

We will write our code in Python for Rhino. Hence, Rhino is the only software you'll need. A student license is available and costs around 195€ (excluding sales tax and shipment).[3] Unlike other student versions, commercial use is allowed even after finishing your study. Rhino is also installed in the ARCH LAB of the TU Wien in case you have no computer at hand.[4]

You can start the Python Editor by typing the command EditPythonScript. From there you can open Python files (.py), create new ones or edit existing ones.

A.3. Python Editor

After entering EditPythonScript one can see the editing environment (see *Figure 1*) which is split into several areas:

- The area on the left shows all available libraries (see *Figure 1*, orange area). It includes the <python> standard library, plus libraries called rhinoscriptsyntax, scriptcontext and Rhino. You'll especially need rhinoscriptsyntax, which contains Rhino specific instructions.
- The area on the right shows the script code window in which new scripts can be written, displayed and edited (see *Figure 1*, green area). It has a syntax highlighting feature which displays programming language elements in color so as to make them easier to identify and check (e.g. the command "print" in the example below).
- The top tool bar contains a green "play button". Once clicked it executes the current script, i.e. the code that was written in the script code tab.
- At the bottom you will see the output window in which status codes and error messages will be displayed once the program is running (see *Figure 1*, blue area).

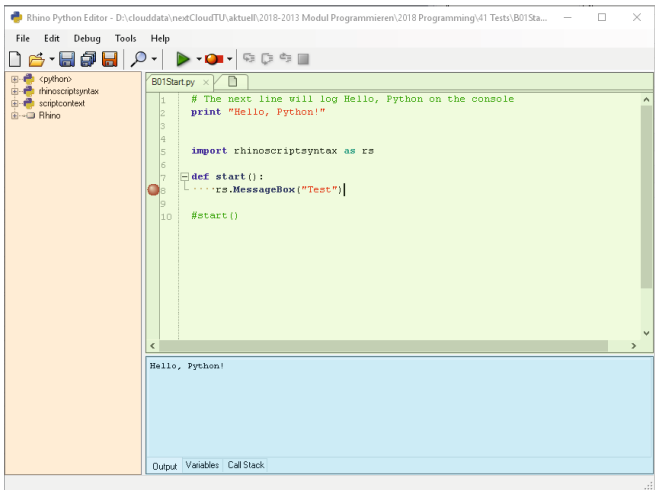


Figure 1: *Rhino Python Editor with its three areas: libraries (orange), script code window (green) and output window (blue).*

3 www.rhino3d.com/de/sales/europe/Austria/all/ (10.03.2018)

4 www.archlab.tuwien.ac.at/raum-pc1.html (12.03.2018)

Let us now enter some basic code in the script tab, without actually knowing what it does:

- Enter EditPythonScript in the command line of Rhino.
- In the script code window enter the following lines:

```
1   # The next line will log Hello, Python on the console
2   print(“Hello, Python!”)
```

- In order to execute the code, choose the “play button” (green arrow) that is available in the top tool bar.
- Examine the output window (blue area in *Figure 1*) which should display the message “Hello, Python!“.

The first program starts with a comment in line 1 (everything after # is ignored). It goes on to the print command in line 2, outputting the text “Hello, Python!”. We can also create our own commands (also see *Second Act: The Play and its Script on page 21*), however this has to be done in two steps:

a. First we have to define what happens when that command is called,
b. then we actually call it:

Program 1: Our first command

```
1   import rhinoscriptsyntax
2   def start():
3       rhinoscriptsyntax.MessageBox(“Test”)
4   start()
```

In line 2, we def-ine a command called “start”, which displays a message box (line 3) which you can quit by hitting OK. However, defining a command does not mean that it is executed. For that, we use start() in line 4.

A.4. Obtaining Help

An important topic for new programmers is the possibility to obtain documentation and help. For example, it is not clear from reading the example code in *Program 1* what MessageBox(“Test”) means. In order to clarify, select the word “MessageBox” and hit F1. You may alternatively use the Help Menu and click ? Python Help. This brings up the help concerning MessageBox, which “*displays a windows message box*” (if not then type in *MessageBox* in the Index menu of *Python Help*).

Searching for help is the daily bread of a programmer. Neglecting to read the documentation is considered to be extremely impolite, and has been punished since the 1970ies by programmer’s proverbs such as RTFM (“Read The Fucking Manual”), GIYF (“Google Is Your Friend”) and so on.

So let's have a closer look on the topic. If one faces a problem when programming or if one does not know what to do, there are several possibilities to get help:

1. Read the Fucking Manual (RTFM).
2. Search the Fucking Web (STFW); Google it up.
3. Not found any help? Go back to step 1.

Especially step 1 is important and should be devoted ample time. You can invoke Help via the menu Help or the F1 key (as is the case for all programs under Windows). Now you can identify three options:

- The Content tab includes information about the rhinoscript package, which gives an overview about Rhino specific functions. They are defined in different modules that can be called within the rhinoscript package. It also contains links to getting started tutorials and documentation on how to open sample scripts.
- The Index tab lists keywords in alphabetical order. This will typically be the first step to solve a particular problem (e.g. look up commands).
- While Index only searches headers, the Search tab includes the whole content. In most cases Search is used, if nothing can be found in Index.

Using help is not a luxury but rather the most basic technique which you will need. In addition, most programming environments come with a feature called IntelliSense or code completion which can help you look up commands: If you type "rhinoscriptsyntax." in the code window, a list of possible commands appears. The desired name can be selected using the arrow keys and inserted using the tab key.

A.5. Debugging

In programming it is often required to run the programs line-by-line in order to check for correctness or simply for understanding what a piece of code does.

The term debugging originates from the times when computer consisted of vacuum tubes. Such an environment invited bugs to nest (since most of them like it warm), which caused disorders. Therefore, each time one had to check the hardware for bugs before starting a computational process.

In the 1940ies it was U.S. Navy Admiral Grace Hopper, who, for the first time, discovered that a moth was stuck in a relay of the rather largish Mark II computer, preventing it from properly operating. The moth ("bug", see *Figure 2*) was removed and taped to a log book, which is now on display in the Smithsonian National Museum of American History in Washington D.C. Next to the attached moth she wrote:

> *"… First actual case of bug being found."*

Today, the activity of debugging has shifted from hardware defects to the detection of logical errors in a program. In software debugging, the program is executed until an "important line" – e.g. an instruction which has failed in a previous execution. At this position the program is interrupted and the execution can now be inspected line-by-line. Debugging does not correct

syntactical errors, which is already done by the compilation process before the program is executed. Debugging is rather semantic, i.e. one checks the result or the intermediate steps for "correctness". Typical cases are:

- The program crashes, giving the line number of the instruction which caused the crash before it does (most common case). One will thus want to halt execution exactly at this line in order to see what went wrong.
- The program runs until the end, but does not provide the expected results. Most likely one will need to stop at the first line and inspect the whole program line-by-line, double-checking whether any calculations have gone wrong. It might help to have pen and paper nearby in order to first calculate the expected outcome of the program by hand, and then compare that to what the program actually produces.
- The program runs forever (it never comes to an end). This can be caused by several factors – the program may wait for a resource that is currently not available (e.g. a printer is assumed to be connected, which it is not) and keeps retrying; such an infinite repetition is the most common cause of this problem.

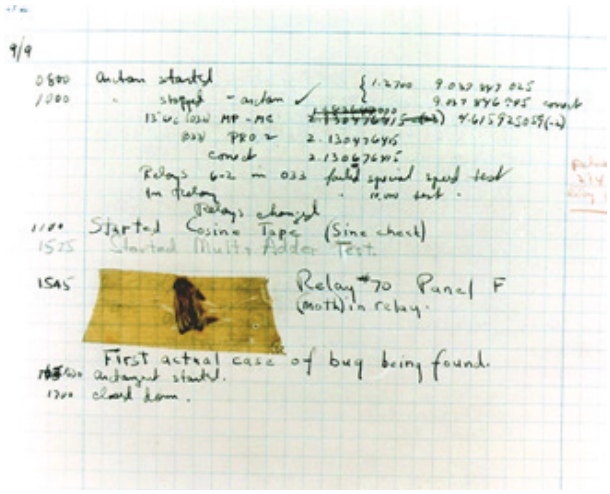


Figure 2: *Grace Hopper found the first actual bug in 1947.*

In practical terms, the usual technique for starting debugging:

- Setting a line where the execution will stop (breakpoint): In our example of *Program 1*, try clicking left of line 3 (the line that reads rhinoscriptsyntax.MessageBox("Test")). A red dot similar to the one in *Figure 3* should appear. If you cannot set a breakpoint then probably the line on which you clicked is empty or contains no executable code (e.g. a line containing only a comment, as to be found in line1 of the example *on page 13*). However, it may also be possible that you did not click left beneath the line number (most common case).

- Stepping through the code line-by-line: start the program using the Play button (green arrow, see Figure 3) in the tool bar and observe how the program stops at the set breakpoint.
- After a breakpoint is reached you can continue by clicking the Play button again. In this case the program will resume until the next breakpoint or until the end of the program if no more breakpoints are set.
- Another option is to continue line by line. There are two options in that case, Step Over and Step Into (see *Figure 3*), which we will now examine in further detail:
  - Hitting Step Over tells the program to continue to the next line of code, if that exists.
  - Step Into tells the program to try jumping "into" the command found at the current line. In our case, this command is rhinoscriptsyntax.MessageBox which is defined in a separate file called 'userinterface.py' which comes with your Rhino program.
- Try pressing Step Into now: The program will open a file 'userinterface.py' and positions the cursor (indicated by a yellow arrow, see again Figure 3) on the first line within MessageBox command.
- There is also the possibility to jump back to after the calling line using Step Out (Figure 3).
- Further observations: The "variables" tab at the bottom shows all momentarily defined values. If you are still in the MessageBox(), you will see multiple entries listed there: Each has a name, a value and a type (int, string, None, etc.). Sometimes these entries also have a [+] sign which allows you to expand their contents – typically when dealing with lists and objects (presented later).
- Finally, you may also hit the Stop button (rightmost button in *Figure 3*) if you want to stop execution immediately, without running a program to its end.

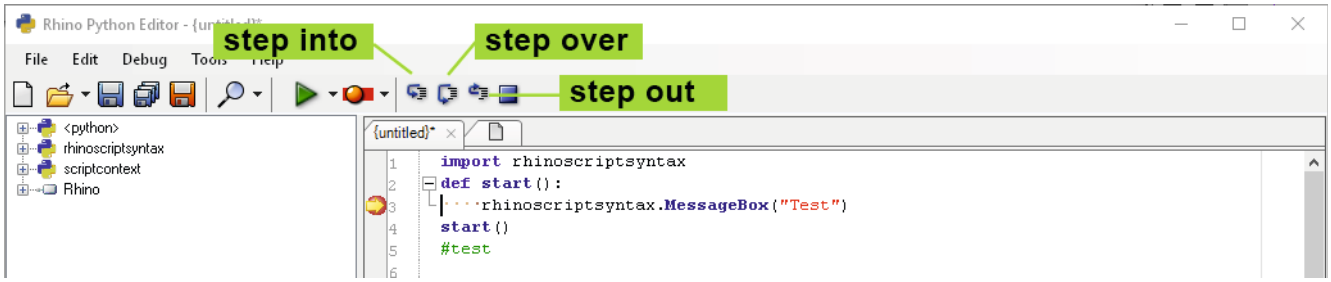


Figure 3: *Debugging in the Python Editor.*

B. First Act: The Actors

[in which data types and variables appear on our mental stage]

ADDNUMBERS begins. After the curtain lifts, there are three actors on the stage: (**a**), (**b**) and (**result**). Taking a closer look at them, we realize that they are all holding a number in their hand (although we do not know exactly which number this is). (**a**) and (**b**) walk over to (**result**) and show it the numbers they are holding in their hands. (**result**) sums these up in its mind and changes the number it holds in its hand to the outcome. The curtain closes. The program ends.

This is a dramatized version of a program that simply adds two numbers. There are two things which must be distinguished in this context:

- The actors (**a**), (**b**) and (**result**) are called variables. Their primary role is to hold values, i.e. numbers, sequences of characters and so on.
- The script that acts on the variables is called the program flow. In our case, there was only one instruction present, which required (**result**) to sum up the values of (**a**) and (**b**).

B.1. Introducing Values, Data Types and Variables

A value is simply a piece of data – e.g. 12, "this is a test" or True. Values have an associated data type:

- Whole numbers (e.g. 12345) have the data type integer.
- Floating-point numbers (e.g. 12.345) are called float. Attention: although in German it is common to use a comma for the decimal mark (12,345), in programming you have to use a decimal point (12.345).
- Sequences of character (e.g. "Abracadabra") are called string. The quotation marks at the begin and end are necessary. You may use single or double quotes ('Abracadabra' is valid as well), but not mixtures ("Abracadabra' and 'Abracadabra" are wrong).
- Logical values (*True* or *False*) are called booleans.

Variables are *named containers* for values. Instead of writing 12.345, you could introduce a variable called (**radius**), set it to 12.345 and refer to it multiple times in your program:

```
Pseudo Code:
    # create a variable called radius and fill it with 12.345
Python:
    radius = 12.345
    ...
    print(radius * radius + radius)
```

In python, declaring variables and filling them is done in one step. Other languages split that into two separate steps (declaring first, filling them next).

B.2. More on Variables

We already learned that variables hold values. Generally spoken they reserve memory locations in order to store these, requiring different size in memory according to their data type. Some programming languages thus require to specify a data type when declaring a variable (strictly-typed languages).

Trying to assign a value of another data type to such a variable will produce an error. Python, on the other hand, is an example of a dynamically-typed language – that means variables can contain values of any type:

Declaration in Python

Description	Code
declare variable "a" and fill it with 1.5 (which is a float value)	`a = 1.5`
now fill the variable with a string	`a = "Abracadabra"`

As shown in the example above, it is perfectly OK to assign values of different data type to the same variable (although we would generally advise against doing so because it is easy to introduce bugs in this way – e.g. by forgetting that (**a**) is a string and thus taking the square root of it later on).

To use a stored value, one simply gives the name of its variable:

```
# display the content of a:
rhinoscriptsyntax.MessageBox(a)
```

A variable can take the value of another variable as well:

```
a = 10
# assign the value of a to the variable b
b = a # by the way, comments can be written after commands as well
```

B.3. Working with Numbers

As mentioned, floating-point numbers are written using a decimal point (e.g. 12.3), integers without (e.g. 12).

Working with numbers is straightforward – you can use the standard operators +, -, / and * for performing simple calculations. *Program 2* demonstrates the concepts we have mentioned so far, plus some new ones, which will be elaborated after the code listing.

Program 2: Simple calculations using doubles and integers

```
1   i = 10                          create a new variable (i)
                                    and assign 10 (integer)
2   i = 1 + 4 * (2 / 10)            assign 1 + 4 * (2 / 10) to the
                                    variable (i) (floating number)
```

In line 1 the variable (**i**) is initialized with the value 10. Obviously, this is an integer. In the next line the variable is automatically changed to a float. This is called automatic type conversion and happens behind the scenes if python can figure out what to do.

B.4. Manual Type Conversions

In seldom cases where python cannot do that automatically, you can convert types manually by calling str(), float() and int().

```
var a = str(3.1415927) # convert float to string => “3.1415927”
var b = float(3)       # convert int to float    => 3.0
var c = int(3.1415)    # convert float to int    => 3
```

The most common case where you will need to type manual conversions is when you print something to the console. Printing generally takes the form

```
print(something [+ something...])
```

where *something* is a string and [...] denotes an optional part where one or more strings are appended (“+”) to the first string. When used on strings, this works just as it should:

```
print(“Hello” + “ “ + “World!”) # result => Hello World!
```

However, when appending a non-string, Python throws an error:

```
print(“PI is ” + 3.14) # unsupported operand type(s) for +: ‘str’ and ‘float’
```

In such a case, one has to manually convert the second part into a string using str(...):

```
print(“PI is ” + str(3.14)) # result => PI is 3.14
```

Another case where an automatic type conversion fails is given in the following example, which queries the type of a variable using the function type():

Program 3: Data type in Python

```
1   i = 10
2   print(“type is: “ + str(type(i)))        prints “type is: int”
3   i = 1 + 4 * (2 / 10)
4   print(“type is: “ + str(type(i)))        prints “type is: float”
```

B.5. Intermediate Summary

We have learned how to use variables. The following questions might help deepen your knowledge:

1. How do you declare a variable in Python?
2. What is the data type of the value “blah”?
3. When do you need to convert values manually?

C. Second Act: The Play and its Script

[in which functions enter our mental stage]

> *"A boy wizard begins training and must battle for his life with the Dark Lord who murdered his parents." (synopsis of J.K. Rowling's "Harry Potter and the Sorcerer's Stone" by Randy Ingermanson)*

From the outside, a play is little more than a title and a short summary as exemplified above. The exact details stated in its script are usually of less concern for an audience, which considers it as a black box through which the main characters move during the course of a story. The actors and the director, on the other hand, need detailed instructions on how to perform their parts. This is stated in the script which – most of the time – the audience has not read (nor are they willing to do so – they are customers after all).

The same is true for programming: Most of the time you will be using others' code without especially caring about how it works, as long as it produces the results you are expecting it to (*black-box thinking*). In order to make this mode of thinking possible, programming languages package pieces of functionality into commands. If you supply them with the necessary inputs, they will produce outputs without requiring you to know how exactly these get computed. The same happened also during you basic math education – did you know how the functions given in *Figure 4* are actually computed?

$$\sin(0) = 0$$
$$\cos(0) = 1$$
$$\sqrt{4} = 2$$
$$\sqrt{\cos(0)} = 1$$
$$\max(1,2) = 2$$

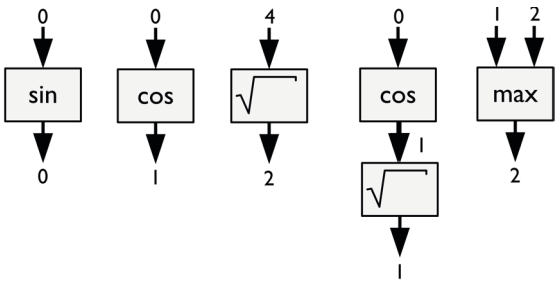


Figure 4: ***Left:*** *Mathematical calculations.* ***Right:*** *Seen as flow chart.*

In this course, we will be exploring both sides – to use and to define commands.

C.1. Python Functions

In Python, commands are called functions. We distinguish between their definition (i.e. the "script") and its use:

- Defining a function means writing our code underneath a def, indented with either spaces or tabs:

```
def name ( [input] ):
  ...code...
```

name defines how the function is called. It is followed by round brackets in which you optionally write inputs that the function needs (more on this later, at the moment one can simply leave that away):

```
def name ():
    ...code...
```

After the round brackets follows a double point ":", then comes your code. A complete example would be:

```
def MyFunction ():
    print("Hello World")
```

- If you want to run the function we have just defined, you need issue a function call by writing the name of the function followed by round brackets, e.g.

```
MyFunction()
```

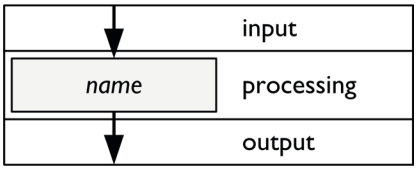
This tells Python to (1.) look whether it can find a function definition for *MyFunction* and (2.) run all the commands contained therein. Because of point (1.), you will always need to define the function *before* calling it:

```
def MyFunction ():
    print("Hello World")
MyFunction()
```

C.2. Defining Functions, Passing Inputs, Receiving Outputs

The general form of a function definition in Python is:

```
def name ( [input] ):
    ...code...
    [return output]
```



A function is a black box that processes any given input and may produce one value as output. For example, remember sin(x) which processes the supplied number "x" and returns you the sinus of (**x**) as output. This output can then be saved to a variable or printed to the console:

```
import math
a = math.sin(13)    # call function sin with 13 as input => outputs 0.22
```

Input and output are both optional (also refer to the example presented further below):

- You define inputs by writing one or more names (separated by a comma) into the round brackets of a `def`. Through these names you can later access the passed values (e.g. (**x**) in the case of sin()).
- One output can be defined by using `return` followed by a value. Python will take this value and return it to the calling code, which can afterwards save it e.g. into a variable:

```
def math.sin(x):
   ...code to compute the sinus of x...
   return ...result of the calculation...

def replace(needle, haystack, replacement):
   ...code to replace string “needle” in “haystack” with “replacement”...
   return ...string with applied replacements...
```

In order to call a function which expects inputs, one needs to put values or variables (separated by a comma) in the round brackets of a function call. The `returned` output can, finally, be printed or saved to a variable:

```
import math
print(math.sin(13))                    # input: 13
                                       # output: 0.22 (printed to console)
a = replace(“bra”, “Abracadabra”, “X“) # inputs: “bra”, “Abracadabra” and “X”
                                       # outputs: “AXcadaX” (saved to a)
```

There are two caveats in that context: Failing to give a function the inputs it needs will result in an error:

```
import math
math.sin() # sin() takes exactly 1 argument (but none is given)
```

On the other hand, assigning the output of a function that does not `return` a result produces “None” (Python’s way of saying that the output is “nothing”):

```
b = MyFunction() # MyFunction does not produce output, b therefore is “None”
```

Textual programming languages allow you to return exactly zero or one output – unlike in graphical languages (e.g. Grasshopper, Dynamo), which let you have any number of outputs. Python however has a syntax which allows you to return multiple values and assign them to variables in the following fashion:

```
def MultipleOutputs():
  return (1, 2, 3)        # pack 1, 2, 3 into a tuple and return it
c,d,e = MultipleOutputs() # unpack the tuple into variables c, d and e
```

Technically, returning multiple values uses a trick: The output values 1, 2 and 3 are packed into a tuple which is returned and unpacked into three variables. More on tuples will be presented later.

C.3. Intermediate Summary

We have had a look at simple calculations from both the outside (*using* functions) and inside (def*ining* functions). We have also looked into Functions with and without output. Here is a program which you can use to test and deepen your knowledge:

Program 4: What does this program do?

```
1   def Sum(a, b)          input two numbers
2       c = a + b          set c to a + b
3       return c           output either float or integer
                           (depends on the two input types)
```

C.4. Test Case: Writing a Function that "Rolls the Dice"

Until now, the presented functions were highly artificial, concerning their scope and purpose. We now wish to come to a more practical application of the gained knowledge, in form of a function that can produce a number between 1 and 6:

Program 5: Random values

```
1   import random                load library called random
2   def RollTheDice():           RollTheDice will take no input
3       a = random.random()      calls function random (of
                                 library random) without input,
                                 receive result in (a)
4       return a                 set the result of function
                                 RollTheDice to (a)
5   RollTheDice()
```

In line 1, we load the *random* library which contains functions for working on random numbers. It contains the function *random* (written as random.random()) that returns a floating-point number between 0 (inclusive) and 1 (exclusive). This is impractical for a function called RollTheDice(), which rather should produce something in the range 1 to 6. Another point for due criticism is that the result of *RollTheDice* is never saved into a variable, which would actually make sense for subsequent calculations.

A second attempt. We will now be using another function called random.randint() which expects two inputs – a lower bound and an upper bound. The changed program will produce something in the

needed range [1... 6]. Furthermore, we will save the output of *RollTheDice* into a variable (**b**) which could be used later on in the program:

Program 6: Random values

```
1   import random
2   def RollTheDice():
3       return random.randint(1, 6)      return number between 1 and 6
4   a = RollTheDice()                    store output in variable (a)
5   print(a)                             use the stored outut
```

A closer look at the program reveals that we have not used an intermediate variable inside *RollTheDice*; rather, we had that function return the result of random.randint immediately in line 3. In line 4, we have stored the output of *RollTheDice* in a variable (**a**), which we used as input for print in line 5.

The library random contains many useful functions, e.g.

```
random.random()                      returns a random float x, 0.0
                                     <= x < 1.0 (exclusive)
random.uniform(1, 6)                 returns a random float x, 1.0
                                     <= x < 6.0 (exclusive)
random.randint(1, 6)                 returns an integer between 1 and
                                     6 (start and end inclusive)
random.randrange(0, 7, 2)            returns an integer as even
                                     number between 0 and 7
```

Instead of writing random.*name of function* all the time, one may also import *names* of all functions in that library (works also for other libraries):

```
from random import *                 import names of all
                                     functions in "random"
a = randint(1, 6)                    now we do not need to write random.
                                     randint but only randint
```

Another extension would be to pass in the number of sides of the dice as an input. Using that functionality, one can simulate dice with any number of sides:

Program 7: Random values

```
1   from random import *
2   def RollTheDice(sides):          receive the number of sides as input
3       return randint(1, sides)     pass number of sides to randint
4   print(RollTheDice(6))            print the output
```

C.5. Intermediate Summary

We have had a look at a real program named *RollTheDice* which we used to compute a random number for a dice with a specified number of sides. Here are some points to deepen your knowledge:

1. Write the previously mentioned formula **Int((Upper Bound-Lower Bound+1)* Rnd+Lower Bound)** as a function using random.random() in Python. The returned value should be between 1 (inclusive) and 6 (inclusive).
2. Is there a difference between defining an input for a function and introducing a variable in Python? Elaborate!

C.6. Advanced: Nested Function Calls

Until now we have always used one function definition and one function call. However, you can (and often will) call a function from within another function you have created:

```
1   def RollTheDice(6):
2      ...as before...
3   def RollTheDiceTwice():
4      return (RollTheDice(6),RollTheDice(6))
5   first, second = RollTheDiceTwice()
```

Functions are a black box for functionality, as was said before. Often this functionality will be useful for many situations – and thus will be used multiple times throughout your code. This style of "write once, use multiple times", is also good for fixing bugs – if there happens to be one in our function *RollTheDice*, then we need to only look into that function. If, on the contrary, we had duplicated the code of *RollTheDice*, then we would have to look through every one of its occurrences. Programmers thus come up with an important design principle that is called: "Don't Repeat Yourself" (DRY) – if the same code appears multiple times in your source, make it into a function. More on design principles of good software are to be found in the excellent book "*The Pragmatic Programmer. From Journeyman to Master*" by Andrew Hunt, David Thomas and Ward Cunningham (1999).

D. Third Act: Alternative Plots

[in which logical values enter our mental stage]

Zeitgeist has produced theater plays in which *the audience*, rather than the director, controls the ongoing of events. At a certain point in the play, you might be asked: What should the actor do next, kiss the girl or tell her that he wants a divorce? "Divorce, Divorce", I hear you scream, but wait: The important thing is that you have the option, right?

Options in computer terms always evaluate to `True` or `False` – which is called *Boolean* (or `bool` in Python). In *Program 8* we show how to define a new variable which is set to *True* (Python is case sensitive, therefore you really have to write *True* and not *true*) and then perform some basic evaluations using "**>**", "**<**", the logical `and`, logical `or` and the negation `not`:

Program 8: Working with Booleans

```
1   def BooleanEvaluations():
2       b = True                          True
3       b = 10 < 20                       True
4       b = 10 > 20                       False
5       b = 10 < 20 and 10 > 20           False
6       b = 10 < 20 or 10 > 20            True
7       b = not 10 < 20 or 10 > 20        False
8   BooleanEvaluations()
```

Among the comparisons that result in a `bool` are: "greater-than" **>**, "less-than" **<**, "greater-than or equal to" >=, "less-than or equal to" <=, "equality" **==**, "inequality" **!=** as well as the logical `and`, `or` and `not`, which require some more thought. `And` is *True* when both sides of the equations evaluate to *True*. `Or` is *True* if one of both sides is *True*. The negation `not` negates whatever comes after it, i.e. the expression `not True` would evaluate to *False*.

Booleans are used for having `conditions` in the code, i.e. if some condition is *True*, perform a set of operations, else, perform another set. The general form for a condition is:

```
if condition:
    block of code to execute if condition is True
else:
    block of code to execute if condition is False
```

The **else** part of the conditions can be left out if nothing is to be done else, giving you the stripped-down version of **if** ... (**Then**, indicated by "**:**" and indentation of the following line) ... (**End if**, indicated by unindentation of the following line).

A practical example using our previously-defined *RollTheDice* function shows that conditionals can be nested:

Program 9: Nested Conditionals

```
1   def RollTheDice(sides):                  RollTheDice produces a number
        ...                                  between 1 and sides
2   result = RollTheDice(3)                  roll three-sided die, store result
3   if result == 1:                          compare result to 1
4       print("the die shows one")           if result is 1, print "one"
5   else:                                    in all other cases,
6      if result == 2:                       compare result to 2
7        print("the die shows two")          if result is 2, print "two"
8      else:                                 in all other cases,
9        print("the die shows three")        print "three"
```

Python's required indentation makes sure that your code is easy to read even when nesting multiple ifs (or whiles, fors and so on – which will be presented later).

A word of advice on using spaces or tabs in that context: Don't mix both! Decide for one of either and then stick to that choice. Why? If you mix spaces and tabs and open your Python code in another code editor outside of Rhino, your statements will likely become mis-aligned:

```
if RollTheDice(2) == 1:
  print "one" # this line was indented using two spaces
else:
        print "two" # this line was indented using a tab
```

This is because editors convert tabs into a number of spaces internally, however, the number of spaces they use differs. Had you written your code using either of both, then the lines would align perfectly.

A further note: The Rhino Python Editor automatically converts tabs to spaces per default (see setting [x] Convert tabs to spaces under Tools > Options > Text Editor), in order to prevent this error from happening.

D.1. Simpler Use of Booleans in Conditions

For this part, let us compare two code snippets which use a function that checks whether this computer is connected to the web (defined elsewhere) which returns a bool:

```
# snippet 1

haveNetwork = functionThatChecksWhetherThisComputerIsConnectedToTheWeb()
if haveNetwork == True:
  print “online”
else:
  print “offline”

# snippet 2

haveNetwork = functionThatChecksWhetherThisComputerIsConnectedToTheWeb()
if haveNetwork:
  print “online”
else:
  print “offline”
```

The only difference between the two snippets is the part "..if **haveNetwork == True**". Technically this is nothing more than an equality comparison between *haveNetwork* and *True*.

- If *haveNetwork* is *True*, the result of this comparison is also *True*.
- If *haveNetwork* is *False*, the result is *False*.

"Hmmm", I hear you say, "so basically this is the same as when you use only *haveNetwork*". Exactly. This is why snippet 2 is written without "**== True**".

D.2. Summing up this chapter

Booleans are used predominantly in if statements. An if checks if a supplied condition evaluates to *True*, then executes an enclosed block of code. Optionally, you might define a second block of code that is executed when the condition is *False* (else).

In the course of the chapter, we also mentioned that you should not mix spaces and tabs in indentation.

D.3. Questions

1. Compare a command with indentation to one without. Assume that there are three nested if-statements (i.e. an if in an if in an if) present, now speculate what would happen if you press the play button.
2. Initialize three variables w1, w2 and w3 with true. Then change w2 to the result of (20 < (10 - 2)) and w3 by combining w1 and w2 by using the logical and.
3. Write the code for calling a message box and print w3 to the output window.
4. Write a function max(a, b) which returns a if a > b, and b in every other case.

E. Fourth Act: The Magic of Words

[in which strings enter our mental stage]

"Was it a car or a cat I saw" is a *palindromic* sentence which you can read from both sides and get the same result if you disregard punctuaction: "*was I tac a ro rac a t i saW*". Even though programmers are not generally known for playing with words, they do manipulate large amounts of text – think e.g. of Google search.

The data type for text is called string. Such values are given in between double or single quotes, i.e. "this is a string value" or 'this is a string value'. A string offers several unique features which distinguish it from other data types:

- A string consists of a *sequence of characters*: The first character is character number 0, the second has the index 1 and so forth.
- The number of characters corresponds to the length of the string. The last character is thus always located at index (length - 1), because we start counting at 0.

The function len(s) calculates the length of a string, where (**s**) is a string whose length should be computed. The result of len() can be stored in a variable:

Program 10: Length of a string

```
1  import rhinoscriptsyntax                  import RhinoScript library
2  def CalcLength():
3      s = "Abracadabra"                     initialize (s) with a string value
4      l = Len(s)                            calculate the length of s and give
                                             it back to a variable, called (l)
5      rhinoscriptsyntax.MessageBox(l)       display result in a message box
6  CalcLength()                              run command CalcLength()
```

One can also access a specific character in a string using the notation *string*[*index*]. Since counting starts at zero, the 11 characters of the string "Abracadabra" would be accessible as follows:

Program 11: Accessing characters of a string

```
1  s = "Abracadabra"
2  print(s[0])                               "A"
3  print(s[1])                               "b"
4  ...                                       ...
5  print(s[10])                              "a"
6  print(s[11])                              index out of range: 11
```

We might also get a part of a string by using the notation *string*[*from*:*to*] where *from* is the starting index (inclusive) and *to* is the end index (exclusive) of the string to extract (this is called slicing):

Program 12: Substrings

```
1   s = "Abracadabra"
2   print(s[0:4])                    "Abra"
3   print(s[1:10])                   "bracadabr"
```

If *to* is greater than the length of the string or if it is omitted, the substring is extracted until the last character. If the *from* index is omitted, the substring is extracted from the first character on:

Program 13: Substrings again

```
1   s = "Abracadabra"
2   print(s[0:10000000000000000])    "Abracadabra"
3   print(s[7:])                     "abra"
4   print(s[:4])                     "Abra"
```

Beside extracting a single character or a substring one can also *search* for a specific string in another string. That is done with the command *find* – e.g. *haystack*.find(*needle*), where *haystack* is the string to search in and *needle* is the string to search for. Notice that you append find directly to a string you are searching in, e.g. "Abracadabra".find("cad"). The function returns -1 if *needle* cannot be found in *haystack*, or the index at which the first character of *needle* is to be found in *haystack*:

Program 14: Find string in another string

```
1   import rhinoscriptsyntax
2   def FindStringInString():
3       s = "Abracadabra"            the string to be searched in
4       a = "cad"                    the search term
5       b = s.find(a)
6       rhinoscriptsyntax.MessageBox(b)   the result is 4
7   FindStringInString()
```

As already mentioned when explaining manual type conversions, the "+" operator joins two or more strings together: "Rhino" + "Script" results in "RhinoScript". This can happen multiple times, i.e. "you " + "could " + "do " + "this " + "more " + "than " + "once."

Program 15: Join two strings together

```
1   def JoinStrings():
2       a = “Rhino”                    first string
3       b = “Script”                   second string
4       c = a + b                      join together
5       print c
6   JoinStrings()
```

Manual conversions to string are necessary whenever a command expects a string but gets something different. One typical example is the print() function which was presented earlier:

```
print(“l: “ + len(“abc”)) # unsupported operand type(s) for +: ‘str’ and ‘int’

print(“l: “ + str(len(“abc”)) # l: 3
```

Two strings can be compared by the double equals sign “==”. If both sides are equal the result is *True*, else it is *False*.

Program 16: Compare two strings

```
1   import rhinoscriptsyntax as rs     import library and
                                       give it a name
2   def CompareStrings():
3       rs.MessageBox(“Rhino” == “Rhino”)    the result is True
4       rs.MessageBox(“Rhino” == “Script”)   the result is False
5   CompareStrings()
```

In the above program, notice that we have renamed library rhinoscriptsyntax into *rs* during import (line 1). Subsequent calls can use the much shorter form rs.function instead of rhinoscriptsyntax.function. Another option for shortening our code (which we have already encountered previously) would be to use from rhinoscriptsyntax import *, which makes all functions in rhinoscriptsyntax available without having to prepend the library name. In case you use multiple libraries, the latter approach is discouraged, as you might import from two libraries which have the same name for a function.

E.1. Advanced Character Extraction and Slicing

We have so far gotten to know *string*[*index*] for extracting a character and *string*[*from*:*to*] for getting a substring (slice) of a string. You may also specify negative values for index *from* and *to*, as shown in the following example:

Program 17: Character extraction and slicing using negative indices

```
1   s = “Abracadabra”
2   print(s[-1])          get the last character -> “a”
3   print(s[-2:])         get last 2 characters -> ”ra”
4   print(s[:-2])         get everything but the last
                          2 characters -> “Abracadab”
```

In fact, the slicing operator can also take a third part called *step*, which lets you extract every n-th character from a string. If this *step* parameter is negative, then Python will walk through the string from the right to the left in steps of *step* characters at a time. The following example clarifies these advanced uses once more and is used to reverse the string mentioned in the beginning of this part:

Program 18: Slicing using a specified *step* size

```
1   s = “Was it a car or a cat I saw”
2   print(s[::2])         extract every 2nd character
                          -> “Wsi  a ractIsw”
3   print(s[::-1])        get every first character from right
                          to left
                          -> ”was I tac a ro rac a ti saW”
```

If you have not remembered every command that was presented herein, do not despair. A cheat sheet is given in the Appendix, showing what you can do with every type (see *page 72*). You can also use the help menu to find more commands on strings.

E.2. Summing up this chapter

We have encountered the String type. There is a variety of functions to be performed with strings, some of which are given in the appendix. Most of the time, though, you will want to concatenate strings using the + operator. Sometimes you will need to do a manual type conversion using str().

E.3. Questions

1. Find the position of the colon in the string s = “Mein Name ist: Killroy” and save it as a variable called indexOfDoublePoint. Use the position of the colon to extract the name and store it in a variable theNameIs. Join the string “Your name is: “ with the NameIs and display the result in a message box.

F. Intermediate Stage

[or two sides of the same coin]

F.1. Algorithmic Thinking

To think algorithmically means *describing a process*. This does not necessarily have to do with computers but can also relate to the real world:

> *"Making a toasted cheese. ... a cheese toast is a simple slice of bread topped with melted cheese. To make it, the oven is your best bet. ...*
>
> *... Toast the bread on one side and remove from the oven. Top the untested side with slices of your favorite kind of cheese, or a shredded variety. Return the bread to the oven so the bread continues toasting on the top side, melting the cheese at the same time. Remove from the oven when the cheese is bubbly and the toast is brown."*[1]

These instructions address a real person. When programming you will address a computer instead. The term algorithm comes from mathematics. It was coined by Muḥammad ibn Mūsā al-Khwārizmī, who, around 820, wrote the Arabic treatise on mathematics "Al-kitāb al-mukhtaṣar fī ḥisāb al-ğabr wa'l-muqābala" ("The Compendious Book on Calculation by Completion and Balancing") . It contains calculation rules and – which is of much importance for us – the description of the number 0.[2]

F.2. Problem-centered strategy

Algorithmic thinking requires one to think of the problem first (*problem-centered* strategy):

- A large problem is split into several smaller parts that are independently solvable[3].
- The act of finding a solution is decomposed into a step-by-step process.

Another difference between finding a solution in classical and computational design is that the latter tries to find one (best) solution for a specific (well-defined) problem rather than a huge set of possible variations for an (often elusive) problem domain.

Lawson states in his book "How Designers Think" (2006) that architects are rather solution-focused. In an experiment under "laboratory conditions" two different groups had to solve a given problem. One group comprised postgraduate science students and the other final year students of architecture. They had to arrange certain wooden blocks on a ground plan (similar to Tetris). However, they did not know which combination is right or wrong. The only way to find out, whether the arrangement

1 www.wikihow.com/Make-Toast (13.03.2018)

2 The word zero derives from the Arabic sifr "cipher"; the translation of Sanskrit sunya-m means "empty place, desert, naught"

3 A subsequent merge of each individual solution into an overall solution generally requires an own synthesis step, see Christopher Alexander's "Note on the Synthesis of Form" (1964)

was a valid solution or not, was to enter it in a computer. The feedback was then either yes or no, depending on the saved rules. Lawson writes about the students:

> *"… The two groups showed quite consistent and quite strikingly different strategies. Although this problem is simple compared with most real design problems there are still over 6000 possible answers. Clearly the immediate task facing the subjects was how to narrow this number down and search for a good solution. The scientists adopted a technique of trying out a series of designs which used as many different blocks and combinations of blocks as possible as quickly as possible. Thus they tried to maximise the information available to them about the allowed combinations. If they could discover the rule governing which combination of blocks were allowed then they could search for an arrangement which could optimize the required colour around the design. By contrast, the architects selected their blocks to achieve the appropriate coloured perimeter. If this proved to be an acceptable combination, then the next most favourable coloured block combination would be substituted and so on until an acceptable solution was discovered. The essential difference between the two strategies is that while the scientists focused their attention on understanding the underlying rules, the architects were obsessed with achieving the desired result. Thus, we might describe the scientist as having a problem-focused strategy, and the architect as having a solution-focused strategy."*[4]

This difference is exactly what complicates communication between architects and computer scientists. On the one side, the architect expresses his/her wishes with the words "it would be great if we could do it so and so, then we could also, ...". On the other side, the computer scientist thinks about how to realize all wishes, which may already cause depressions. It is not possible to work together without understanding how the other side is thinking. On the one extreme there is an introvert programmer (nerd) and on the other an aloof architect (diva). Therefore, throughout the lecture we think problem-oriented and break down the whole problem into smaller parts before working on a solution. This will, for sure, also help in daily life.

4 Bryan Lawson, "How Designers Think" (2006) page 43f

G. Fifth Act: We want to draw

[in which lists enter our mental stage]

When using drawing functions, Rhino expects the user to supply three-dimensional coordinates. So far, we know integers and floats, each having one value; so how can one specify a variable that has three entries – e.g. (1.6, 2.5, 3.8)?

G.1. We Need a List (an Array)

This is the point where **lists** (and **arrays**) enter the stage. An array is an ordered list of values of the same type, e.g. three booleans, four integers and so on, while a list is an ordered list of items which can be of any type. Python only knows lists (which is more flexibly than arrays). *Program 19* gives a listing on how to define a list in Python:

Program 19: Defining Coordinates as a List

```
1   def definingACoordinate():
2       pos = []                        define an (empty) list
3       pos.append(1.6)                 append the first element
4       pos.append(2.5)                 append a second element
5       pos.append(3.8)                 append a third element
6   definingACoordinate()
```

When a list enters the stage (see line 2 in *Program 19*) it contains nothing; no item, no value. In order to add an item, one has to use the function **append()**. This function does exactly what it says: it appends (add) an item with a value to the end of the array. The result is a list of certain items (in our case 3, see *Program 19*). An alternative way is to specify a list giving some initial values separated by comma:

```
1   def definingACoordinate():
2       pos = [1.6,2.5,3.8]
```

If you have troubles in comprehending the concept of lists, the following analogy will help: As soon as a play has ended, the curtain drops. When the applause has reached an adequate volume, the curtain lifts again and you see all actors assembled in a line. "Look", you might say, "the third one from the left, I found that she played really well". Picking out the third actor (item) from the left was what we would do programmatically by using actors.

Hint! In Python, the first index of an array is always zero-based (starting with the index 0).

But how about picking the third actor from the right? First we have to find the length of the list, that means the total number of items. This is achieved by the **len()** function, which we already know from strings:

```
1   def findLenOfArray():
2       pos = [1.6,2.5,3.8]
3       numEntries = len(pos)
4       print numEntries
5   findLenOfArray()
```

```
get number of entries of the list
result will be 3
```

A list can store any data type. Lists can even contain other lists, e.g. [[1,2],[3,4]].

G.2. The Case of Stairs

As a first full-fledged program, we want to develop a stair algorithm that produces a staircase in 2D, given the following parameters:

- StartingPoint – the starting point of the polyline should be located at (0,0,0);
- StepHeight – the height of a single step should be 18cm;
- StepDepth – the depth of a single step should be 32cm;
- NumberOfSteps – the number of single steps should be 2 for now.

Now comes the analysis phase. A step starts at a certain position in 3D space, initially this would be StartingPoint (point0 in *Figure 5*). From this position, we draw a line of length StepHeight (h) vertically upwards. This defines the second point of the staircase (point1). Then we draw a horizontal line of length StepDepth (d) to complete the first step. We already reached the third point (point2). The whole process is repeated from the current position to get the second step (fourth and fifth point; see *Figure 5*).

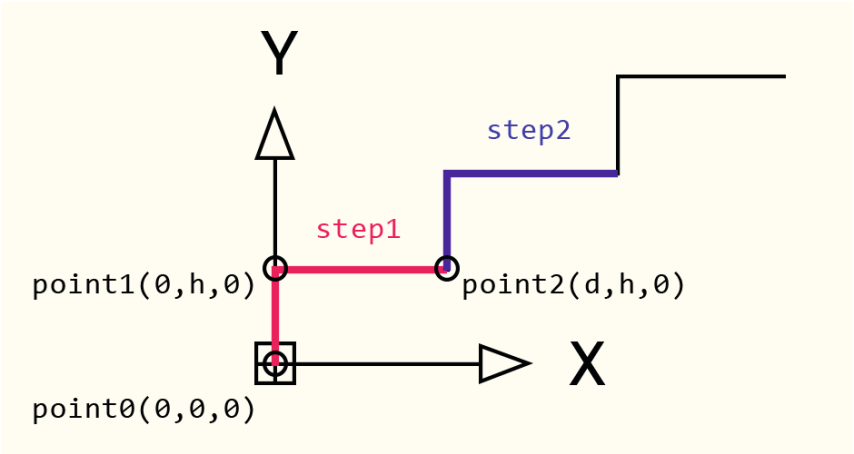


Figure 5: *Output of Program 22.*

The Pseudo Code (a description of the operating principles of an algorithm) runs like this:

Program 20: Pseudo Code of the program staircase

```
# define x, y and z as coordinates of a (changing) point and
    initialize x, y and z as starting point (0,0,0);
# initialize a list “point” with x, y and z of the starting point

# define a list “points” that saves all points sequentially
# add the first point of the staircase to the list “points”

# change y-coordinate for 1st point of first step and add point to points
# change x-coordinate for 2nd point of first step and add point to points

# draw polyline
```

G.3. Intermediate Stage: Rhinoscriptsyntax

Since you want to draw in Rhino, you will need the RhinoScriptSyntax methods library. This allows the user to access Rhino’s geometries and commands. You’ll find the RhinoScriptSyntax in the left window of the Python Editor (see *Figure 6*, left). If you choose a function, additional information will appear in the bottom gray area (the output window, see *Figure 6*, right).

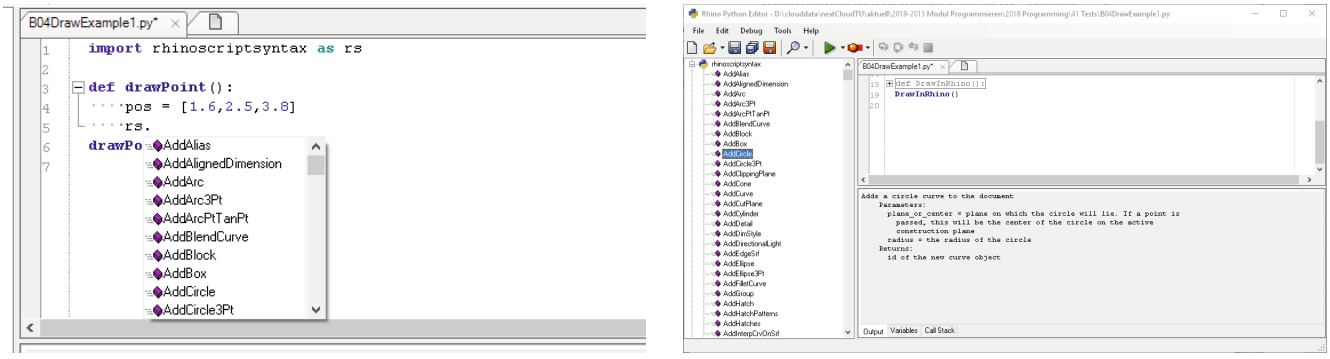


Figure 6: *Left: Contents of RhinoScriptSyntax.* ***Right:*** *Additional information about the command AddCircle().*

Throughout this course, we are going to use RhinoScriptSyntax, one example of which is given in *Program 21*:

Program 21: Drawing a circle

```
1   import rhinoscriptsyntax as rs          Import the library and rename the
                                            namespace to rs
2   def DrawInRhino():
3       pos = [1.6,2.5,3.8]                 define a (Point) list
4       rs.AddCircle(pos, 10)               adds a circle located at
                                            pos, with radius of 10
5   DrawInRhino()
```

The **AddCircle** Function takes two parameters, a list of numbers (float or integer) defining the center of the circle and a number giving its radius. For further information press F1 and search for "AddCircle". This brings up the help file on "AddCircle" with more elaborate description. As a shortcut, we have also compiled a list of popular drawing commands as appendix (see *page 72*).

G.4. Continuing with the Case of Stairs

We are now going to use the **AddPolyline()** function to draw a polyline. AddPolyline takes a list of 3D points as input (or more precisely a list of lists representing coordinates of the points). Note that there must be at least two points in the list; if the list contains less than four points, then the first point and last point must be different.[1]

Program 22: First version of the step program

```
1   import rhinoscriptsyntax as rs
2   def StepByStep():
3       #define x, y and z as coordinates of a (changing) point and
            initialize x, y and z as starting point (0,0,0);
4       x = 0                               set start position (x-coordinate)
5       y = 0                               set start position (y-coordinate)
6       z = 0                               set start position (z-coordinate)
7       #initialize a list "point" with x, y and z of the starting point
8       point = [x,y,z]                     write start position to a list
                                            with 3 positions (x-coor, y-coor,
                                            z-coor) representing a single point
9       #define a list "points" that saves all points sequentially
10      points = []                         list for all points
11      #add the first point of the staircase to the array "points"
12      points.append(point)                append start point to list "points"
13      #step 1                             1st step
```

1 Compare help menu of Python

Program 22: First version of the step program

```
14      #change y-coordinate for 1st point of 1st step and add point to points
15      y = y + 18                      add 18(cm) to y-coordinate
16      point = [x,y,z]                 1st point of 1st step
17      points.append(point)            append 2nd point to list “points”
18      #change x-coordinate for 2nd point of 1st step and add point to points
19      x = x + 32                      add 32(cm) to x-coordinate
20      point = [x,y,z]                 2nd point of 1st step (=1st
                                        point of 2nd step)
21      points.append(point)
22      #step 2
23      y = y + 18
24      point = [x,y,z]                 2nd step vertical point
25      points.append(point)
26      x = x + 32
27      point = [x,y,z]                 2nd step horizontal point
28      points.append(point)
29      #draw polyline
30      rs.AddPolyline(points)          finally draw: list is
                                        passed to AddPolyline
```

Referring to the finished program (*Program 22*, also see *Figure 5*), there are several oddities which require attention:

- The starting position is fixed to (0,0,0). The StepHeight and StepDepth are also fixed. Clearly, one should be able to pass these values into the command rather than defining it in the code (e.g. in the form StepByStep(StepHeight, StepDepth)).
- We notice that the code for the 1st step is the same as for the 2nd step.

You might have guessed it – these points smell like a proper cause to introduce something new that will fix all these problems.

G.5. A Separate Command for a Single Step

Programmers hate code duplication, therefore coining the **DRY Principle** ("Don't Repeat Yourself"). **Code repetition** is error-prone and tedious to work through when changing software, the key to proper coding lies in trying to avoid it at all costs. A mechanism for coping with such repetitions was already given, namely the use of functions. If we could write a function that draws a single step and use that multiple times, we could eliminate a lot of code (see *Program 23*):

Program 23: Second version of the step program

```
1   import rhinoscriptsyntax as rs
2   def SingleStep(x,y,z,StepDepth,                new command for a single step
        StepHeight,points):
3       y = y + StepHeight                         add StepHeight to y-coordinate
4       point = [x,y,z]                            1st point of nth step
5       points.append(point)                       append 1st point to list "points"
6       x = x + StepDepth                          add StepDepth to x-coordinate
7       point = [x,y,z]                            2nd point of nth step
8       points.append(point)                       append 2nd point to list "points"
9       return (x,y,z,points)                      return tuple of 4 elements:
                                                   x,y,z,steps
10  def StepByStep():
11      StepDepth = 32
12      StepHeight = 18
13      x = 0                                      set start position (x-coordinate)
14      y = 0                                      set start position (y-coordinate)
15      z = 0                                      set start position (z-coordinate)
16      point = [x,y,z]                            write start position to a list
                                                   with 3 positions (x-coor, y-coor,
                                                   z-coor) representing a single point
17      points = []                                list for all points
18      points.append(point)
19      #step 1                                    1st step
20      x,y,z,points = SingleStep(x,               get back new values of x,y,z,steps
            y,z,StepDepth,StepHeight,
            points)
21      #step 2
22      x,y,points = SingleStep(x,y,
            z,StepDepth,StepHeight,
            points)
23      rs.AddPolyline(points)                     finally draw: list is
                                                   passed to AddPolyline
24  StepByStep()                                   call command
```

We start off by giving our new function three coordinates (x,y, and z) that denote the current position in space. Adding StepHeight to y gives a next point, which we add to a list of generated points (see lines 3-5 in *Program 23*). Then, we add StepDepth to "x", giving a further point, which we add to the

points list (see lines 6-8). We return the current position x,y and z as well as the points list. Calling this function multiple times fills the points list (see lines 20-22). Note that the very first point must be manually appended to the points list before using our new function (see line 18).

G.6. Summing up this chapter

We have seen that there are variables that can take sequences of values, i.e. **lists (Python)**. While an array (e.g. used in VBA) is an ordered list of values of the same data type, a list is an ordered list of items which can be of any data type. Each value (i.e. each item or entry) can be accessed and changed individually. Lists and arrays are needed to store coordinates, a circumstance we often encounter when working with drawing functions.

G.7. Questions

1. What is a list?
2. Using whatever help you get, figure out
    a) what a multi-dimensional list is
    b) how you can define a three-dimensional list with 3x3x3 entries.
3. Define a list with three entries: "Rhino", "Python" and "rules". Save the third character of the second entry in a new variable.

H. Sixth Act: We Want to Avoid Writing Unnecessary Code

[in which loops enter the stage]

Coming back to the criticism over the repeated step code in our staircase program (see *Program 22*), we need something new that can repeat code (i.e. the same lines are executed multiple times).

In the previous chapter we described a command that draws a staircase by repeating the part of a single step. In our particular case we called the command SingleStep(x,y,z,StepDepth, StepHeight,points) twice. But how about 1,000 stairs, I ask? This would again produce repetition in the form of all these calls to SingleStep(...). No, there has to be some other construct that can really solve this problem. This construct is called a **Loop**. A simple loop is to count up:

Program 24: Pseudo Code of a loop

```
# define variable x and initialize it with 1
# repeat unless a termination condition is met
    # increase the value of x by 1
# end of loop
```

In programming we have different forms of loops. They differ in the position of the condition. With some types of loops the condition is asked at the very beginning. Only if it is true the block inside is entered. In all other cases the program continues after the end of the loop. We call loops with termination condition at the very beginning pre-test loop. The loop checks the condition before the block is executed. Loops that check the condition after the block are called post-test loop. In this case the block is passed through at least once.

H.1. While Loop (Pre-Test Loop)

The part "*unless a termination condition is met*" from the Pseudo Code example equals a logical expression (*True* or *False*). This looks quite similar to an **If**, you might say. This is completely right – a **While** is like an if-condition, only that the if-condition executes the enclosed block of code one time if the condition is *True*. On the contrary, the while-loop executes the block over and over again, as long as the condition is *True*. "x < value" would be a common condition (in our example x < 10). That means, the loop will be repeated until x = value+1 (in our example x=11). I.e. the block inside the while loop is repeated as long as the condition is fulfilled, i.e. *True*.

Program 25: While loop – count-up example

```
1   def loop():
2       x = 1
3       while x < 10:
4           x = x + 1                    increase x by 1
```

Hint! The loop continues as long as the condition remains *True*. That means: as long as (**x**) is smaller than 10, the counter (**x**) has to be increased by 1. All commands of the while block are executed until the condition is not met any more. So, if you forget to increase the counter (**x**) the loop will continue infinite number of times. This is called endless loop (**pitfall!**). Without counting up, the condition is always *True* since the variable (**x**) always equals 1 and fulfills the condition x < 10.

The general form of a while loop runs:

```
While condition:
    ... block of code to execute if condition is True ...
```

With this knowledge in mind, the reimplementation of our staircase program can be conducted (see *Program 22*). The new version will be cleaner, shorter and easier to read.

First, we have to think about, which parts can/should be repeated. From *Program 23* we already know which parts belong to a single step:

Program 26: Single step of staircase program

```
1   y = y + 18                  add StepHeight to y-coordinate
2   point = [x,y,z]             1st point of nth step
3   points.append(point)        append 1st point to list “points”
4   x = x + 32                  add StepDepth to x-coordinate
5   point = [x,y,z]             2nd point of nth step
6   points.append(point)        append 2nd point to list “points”
```

For more flexibility we have to rewrite the code:

Program 27: Pseudo code of loop

```
# initialize a variable “step” with the value 0

while step < NumberOfSteps:
    # add a single step (its points) while increasing
      height and depth of coordinates

# draw polyline
```

The final version of the program uses the while-loop, repeating the step drawing process for as many times the user wishes. Loops and commands (functions) can help in keeping code free of repetitive parts (this was referred to as the **DRY principle**).

Program 28: Third version of the step program

```
1   import rhinoscriptsyntax as rs
2   def GenNStairs(NumberOfSteps):
3       x = 0
4       y = 0
5       z = 0
6       point = [x,y,z]
7       points = []
8       points.append(point)
9       step = 0                                  initialize a variable
                                                  "step" with the value 0
10      while step < NumberOfSteps:
11          #add a single step (its points) while
                increasing h & depth of coors
12          y = y + 18
13          point = [x,y,z]
14          points.append(point)
15          x = x + 32
16          point = [x,y,z]
17          points.append(point)
18          step = step + 1                       don't forget to
                                                  increase step by 1!
19      rs.AddPolyline(points)                    draw polyline
20  GenNStairs(10)
```

H.2. For-Loop (Pre-Test Loop)

Another kind of pre-test loop is the so called For-loop. In contrast to the While-loop, counting-up is done automatically (see line 2):

```
1   def ForLoop():
2       for i in range(1, 9):
3           print i
4   ForLoop()
```

The term "*in range(1,9)*" means that the counter (**i**) starts with the number 1, increases its value by 1 and repeats the loop until i = 9. The condition returns false as soon as 9 is reached. Inside the for-block the value of the counter (**i**) is printed in the output window. When you run the command, you'll

recognize that i = 8 is the last printed value for (**i**). This is because as soon as i = 9 the condition is no more valid and the loop stops (the program switches to the first line after the end of the loop).

H.3. Do-Loop (Pre-Test or Post-Test Loop) or While True

In some program languages there is a do-while loop. Python has no such construct; however, there is a workaround in Python to achieve the same.

In contrast to a pre-test loop, a post-test loop is executed at least once. Only at the very end of the loop the condition is tested. Even if the condition is not met from the very beginning, the block inside the loop is nevertheless executed. This is the big difference between a pre-test and post-test loop. At the end the condition is checked. If it holds *True*, the loop runs another time.

In practice post-test loops are seldom used. An example of their usage is the repetition of instructions in case of an error: first, some commands are executed; then one or more retries are started, to see if the error was only temporary. If the error still occurs after n-retries (defined as termination condition), the loop is finally terminated.

There is no do-while loop in Python. However, there is a similar construct that does the same as a while loop:

Program 29: Do-while in Python

```
1   def Condition():
2       i = 0
3       while True:
4           i = i + 1
5           print i
6           if i < 2:
7               break
8       print “finished”
9   Condition()
```

At the beginning of the while-loop the condition is in any case *True*. Therefore the following block of commands are executed at least once. Only in line 6 the actual condition is tested (as an if-condition). If the condition is not met, the loop finishes (using the command break in line 7) and the command continues at line 8.

H.4. Intermission: Nice Coding

At this point we want to repeat our advice about the correct indentation, which is essential for being able to write Python code. Other programming languages are not that strict, however, in any case, correct indentation makes reading the code much easier. This especially holds true if one revisits code after a longer period of time or has to read code produced by someone else.

Rules for nice coding:

- Insert an empty line before and after every function.
- Indent after a block starts. A block includes functions, conditions and loops. It is best to use a tab stop. Always use the same indentation (2 spaces or tab stop or ...).
- Comments can be helpful. Insert them before coding.
- Names of variables and commands should speak for themselves. This saves time for documentation.
- Split long calculations into shorter ones. No one will understands a*a+b*29+calculate(a,b,c).

Most of these rules are just recommendations and should make the code easier to read.

H.5. Summing up this chapter

The urge to produce a full-blown program has led us to the code for GenNStairs which uses a while-loop (see *Program 28*). A loop repeats a block of code several times until a certain condition is met.

H.6. Questions

1. Write a program with a while loop. The variable (**x**) should be an empty string at the beginning. In the while block add continually the string "a" to the variable (**x**). Print the resulting string in the output window.
2. We learned about three different loops (while, for-next and do-while).
   a) What are loops used for?
   b) Write an example for each of them.
3. What is the difference between pre-test and post-test loops? When are they used?
4. Write down an if-elseif-else condition and explain how the individual parts work.
5. "Bugfixing": The following code has several errors and minor flaws. Can you spot and fix all of them?

```
def example():
mid = [0 To 2]
counter == 0
while counter < 10
    rhinoscriptsyntax.AddCircle(mid,3)
example()
```

I. Seventh Act: Make User Interaction Easier

[in which we supply parameters using the User Interface]

Graphical User Interface

Instead of writing *GenNStairs(10)* in *Program 28*, we want to present a user interface where parameters can be entered. In order to create a Graphical User Interface (GUI) you may rely on a UI creation tool like **SharpDevelop** (Version 4.4), which uses the Microsoft Windows Forms.[1,2] SharpDevelop is an integrated development environment that also supports Python. The workaround to create your own GUI is as follows:

- Open SharpDevelop (after you have installed it)
- Click "File" ... "new" ... "File"
- Select "Python" from the categories area (left) and choose "Form" from templates (right)
- Press "create" button
- At the bottom you'll see that the "source" view is activated; choose "design" instead
- Select Tools on the left side and choose Windows Forms (see *Figure 7*)
- Now you can use all items under Windows Forms (e.g. a button); simply use drag and drop
- Properties of each element can easily be changed at any time (see right in *Figure 7*)
- Finally, save the file as ".py"-file

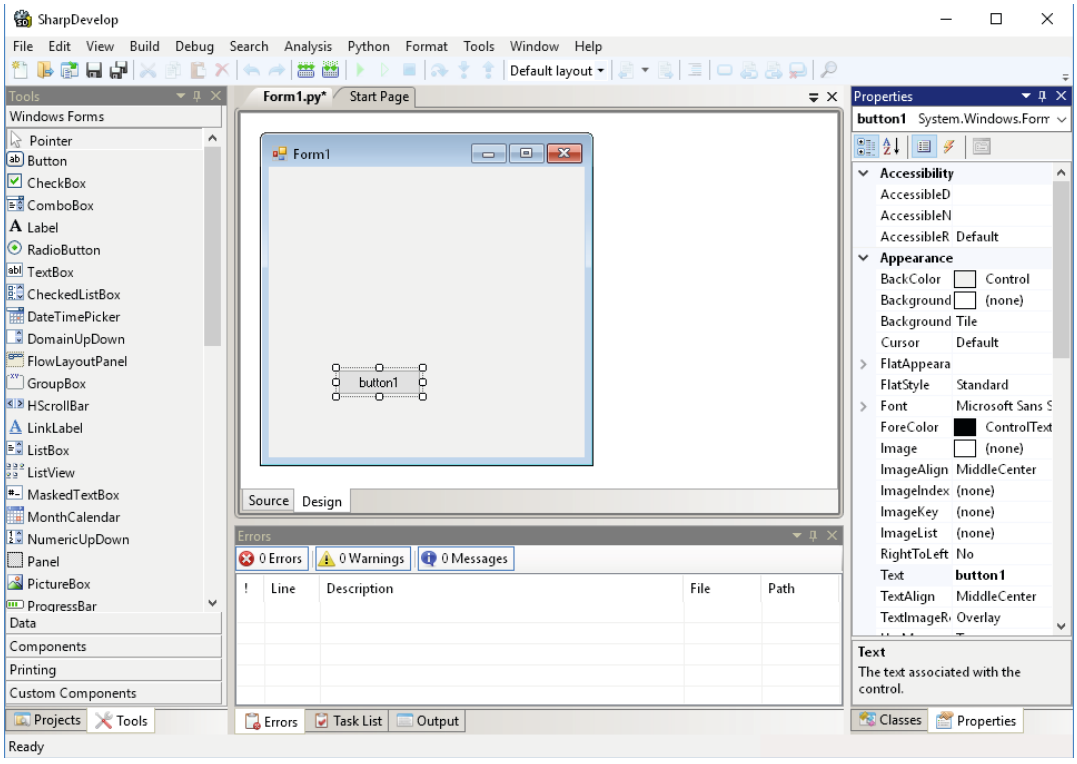


Figure 7: *SharpDevelop 4.4;* ***Left:*** *items of Windows Form;* ***Middle:*** *design field;* ***Right:*** *properties.*

1 www.icsharpcode.net/opensource/sd/Default.aspx

2 discourse.mcneel.com/t/can-winforms-be-used-with-python-to-make-this/17167/9

Look at the first lines of the new ".py" file (in SharpDevelop change to "source view" or compare *Program 30*). As you can see, the new file refers to certain modules (.NET assemblies). This includes the System module (line 1) and the Microsoft Windows Forms module (line 2). In line 3 and 4 you import all elements that are therein available. Finally, in line 5 you create your own class. A class describes a program code template (a blueprint) for creating objects. It describes properties and methods (behavior) of objects. In our case, we initialize our components (our previously inserted button) and define some properties, like its position (line 11).

Program 30: Import Windows Forms

```
1   import System.Drawing
2   import System.Windows.Forms                    import module
                                                   Windows Forms
3   from System.Drawing import *
4   from System.Windows.Forms import *             import all elements
5   class MainForm(Form):                          create your own class
6       def __init__(self):                        "self" refers to the
                                                   current instance
7           self.InitializeComponent()             initialize your components
8       def InitializeComponent(self):
9           self._button1 = System.Windows.
                Forms.Button()
10          self.SuspendLayout()
11          self._button1.Location = System.       configure the element
                Drawing.Point(136, 133)            button1, e.g. its position
```

We will now build up a form using three components: Labels, TextBoxes and Buttons. Open SharpDevelop click "New" and select "Form template" from category "Python". Rename the class to StepUserInterface: choose "source" view and change class Form1(Form) to class StepUserInterface(Form). Now, change the "source" view to the "design" view. On the left side open Windows Forms in the Tools bar. Hover over A Label, and drag and drop it to the form (you can also click the symbol then change to the Form and click again). Do the same with ab| TextBox and the ab button.

Your form should contain three parts: a Label, a TextBox and a Button. Click on the Form (anywhere in the gray area will do) and look at the property editor (on the right). As example you can change the title of the form window by changing the text under the tab Appearance (e.g. "Enter Parameters"). Once finished, go click on the Label and set its Name under the tab Design to NumberOfSteps (again in the property editor). Set the Text (tab Appearance) of the Label to "Number of steps". Click on the TextBox. Set its Name to Steps. Click on the Button and set its Text under the tab Appearance to OK. The Form should look similar to *Figure 8*:

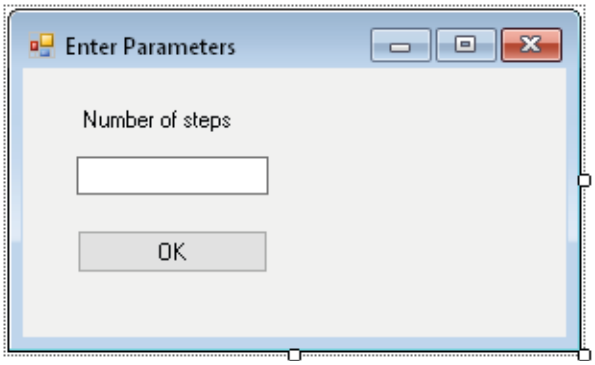


Figure 8: *Our first UserForm*

Then double-click the Button, this should bring up the Editor ("source" view) with the command **def Button1Click()**. Note that you can return anytime to the form by clicking the "design" view.

Button1Click() is an event handler. It is being called whenever the user clicks on the OK Button. Inside Button1Click() comes all commands you want to have executed. For example, we might write:

```
def Button1Click(self, sender, e):
    self.Close()                        remember, "self" refers to the
                                        current instance; this just
                                        closes the Form on click
```

Now, we save the new project (e.g. as DrawStepsGUI.py). You may be asked to save the Resource Files as well (DrawStepsGUI.resx). Switch to the PythonEditor in Rhino. Open the project in which you want to use the new GUI (in our case this is *Program 28*). In here we (e.g. at the end of the staircase program) have to import the new GUI File in order to use it (see line 1 in *Program 31*).

Program 31: Use GUI in your project

```
1   import DrawStepsGUI                 import Form defined in SharpDevelop
2   if __name__ == "__main__":          start when play button is pressed
3       form = DrawStepsGUI.            initialize the variable to the
            StepUserInterface()         function StepUserInterface of the GUI
4       form.ShowDialog()               show dialog
5       numSteps = int(form.            get number of steps from TextBox
            _Steps.Text)                and convert it to an integer
6       GenNStairs(numSteps)            start command
```

Line 2 is an if-condition, which is true as soon as the play button is pressed (as soon as the program is started). In the next line we initialize a new variable with the class StepUserInterface in DrawStepsGUI. Finally, in line 4 the command ShowDialog of the Form StepUserInterface is called. When pressing the green play button from the Rhino Python Editor the new GUI will appear. Once you press OK, the form window will hide itself (remember self.close() from before), and execution will finish. This brings

us back to line 5 where we read whatever the user has inserted into the TextBox "_Steps". Since this is a string, we convert the value to an integer. Finally, we call the command from *Program 28* (line 6)

Forms are Objects. They have different variables and Functions for working with the window you see. Try typing "DrawStepsGUI.StepUserInterface." inside the Rhino Python Editor (handler). This should bring up the list of things available for calling.

I.1. Useful Commands for Converting and Checking

Remember that we had to convert the insert of TextBox to an integer. But what if the user types "blah"? This cannot be converted into an Integer. Consequently, we have to check the input for correctness. The following program illustrates some string functions that come handy. They may be useful when fields of a GUI should only allow certain inputs:

Program 32: Strings

```
1   def checkInput():
2       s = "123,3"
3       if s.isnumeric():                          check if string contains
                                                   only numbers
4           s = int(s)                             if yes, you can convert
                                                   "s" to an integer
5       else:
6           s = s.replace(",",".")                 in any case change "," to "."
7           if s.count(".") == 1:                  check if string contains
                                                   exactly 1 point
8               onlyChars = s.replace(".","")      remove point
9               if onlyChars.isnumeric():          check if it's now a number
10                  s = float(s)                   if yes, the string is a float
11              else:
12                  print "try again!"             in all other cases it
                                                   is not a number
13  checkInput()
```

The function isnumeric() on line 3 returns *True* if the passed **string** contains only numeric characters. In this case, the input is a whole number and we can immediately convert the **string** into an **int** (line 4).

Hint! Programming means to use the English notation for numbers (e.g. 0.3 and not 0,3). This is important when dealing with floating-point numbers.

In our command we want to consider country-specific writing styles. I.e. the separation mark between the integer and the decimal part of a floating-point number can be either a point or a comma. Consequently, we, at first, use the function replace() in order to change any comma to a point (see line 6). Thereafter we search for exactly one point in (**s**), which is an indication for a floating-point number. Since the string can nevertheless contain none numeric characters we eliminate the point

and check if the remaining string (onlyChars) isnumeric() (line 9 in *Program 32*). If "yes", the original string is a floating-point number and (**s**) can be converted into a float (line 10). In all other cases (**s**) is a string.

The important point why we do this is to now get the number of steps (a string) from the previously defined form. This must be checked e.g. if it is empty or if it is a number. Then it is converted to an integer. Finally, we may call the stair algorithm in order to do useful work.

We now turn back to our staircase program where we have added our GUI (see *Program 28* and *Program 31*). After we call the Form, we add the checkInput program (see *Program 32*):

- In order to obtain the value of the TextBox, we use "form._Steps.Text".
- If the user has not typed anything into the TextBox, we ask him to enter something.
- If we find (after many checks) that the value is an integer, we call the steps program, passing the converted integer value as number of steps.

Program 33: Handler that calls the steps program

```
1   import DrawStepsGUI
2   if __name__ == "__main__":
3       form = DrawStepsGUI.StepUserInterface()
4       form.ShowDialog()
5       if form.Button1Click:
6           s = form._Steps.Text                get text that is typed
                                                in the element "_Steps"
7           if len(s) == 0:                     check if string-
                                                length is 0
8               rs.MessageBox("Enter a value")  if True then show message
9           else:                               in all other cases
10              if not(s.isnumeric() or         check if the string
                    isFloat(s)):                is not a number
11                  rs.MessageBox("Enter num")  if True then show message
12              else:                           in all other cases
13                  s = s.replace(",",".")      replace comma with point
14                  s = int(float(s))           then convert to integer
15                  GenNStairs(s)               start program
```

You might recognize that no function called "isFloat" exists (line 10). In other words, this is a new function (see *Program 34*). Basically, the new function replaces any comma with a point (regardless how many there are). In case there is exactly one point in the string it might be a floating-point number (see *Program 34* line 3). Next, we remove the point and look if it is now a number (line 4 and line 5 in *Program 34*). This is the final indication that (**value**) is a floating-point number.

Program 34: Function isFloat

```
1   def isFloat(value):
2       value = value.replace(“,”,”.”)          replace comma with point,
                                                even if there is none
3       if value.count(“.”) == 1:               if there is exactly
                                                one point, it might
                                                be a number
4           onlyChars = value.replace(“.”,””)   remove point
5           if onlyChars.isnumeric():           check if it’s now a number
6               return True
```

Hint! The order of your functions in the program is important. I.e. each function you call has to be defined previously in the code.

I.2. Summing up this chapter

Filling a program with parameters in a graphical way has lead us to dive into user interfaces, or (as called in Windows) Forms. A Form is an object onto which you can draw controls (e.g. TextBoxes, Labels, Buttons). These controls are available by prepending the form’s name (a TextBox “Steps” of a Form “StepUserInterface” would be, after calling the Dialog, accessible via “._Steps”). We have also seen handlers in the form of a click handler for the “OK” Button. A handler is a code inside the form that usually does some error-checking on the form’s controls and then goes on to call some program. There are many handlers available (but beyond scope of this lecture), for example for handling keypresses, selection of a list etc. We encourage the reader to look inside these when the basic knowledge has settled.

I.3. Questions

1. Write a GUI with a TextBox, a Label, a Button and a fourth element of choice. Check for valid inputs, e.g. max characters of the TextBox, no number in the TextBox or the like.
2. Search for an alternative possibility to check for numbers (hint: error checking).

J. Eight Act: Interact with Grasshopper

[in which we want to open the communication between Grasshopper and Python]

The prerequisite for communication between Grasshopper and Python programs is to install the open source component GhPython. GhPython is a scripting component which allows to use the rhinoscriptsyntax (which is exactly what we need). How to install:

- Download th component e.g. from food4rhino.[1]
- Save the file (ghpython2.gha) in the Component Folder of Grasshopper: in Grasshopper, choose File/Special Folder/Component Folder.
- Right click on the file, open properties and make sure that it is not blocked.
- Close Grasshopper and restart Rhino.

Now, you can use GhPython. You'll find the component under Math/Script (see *Program 9*).

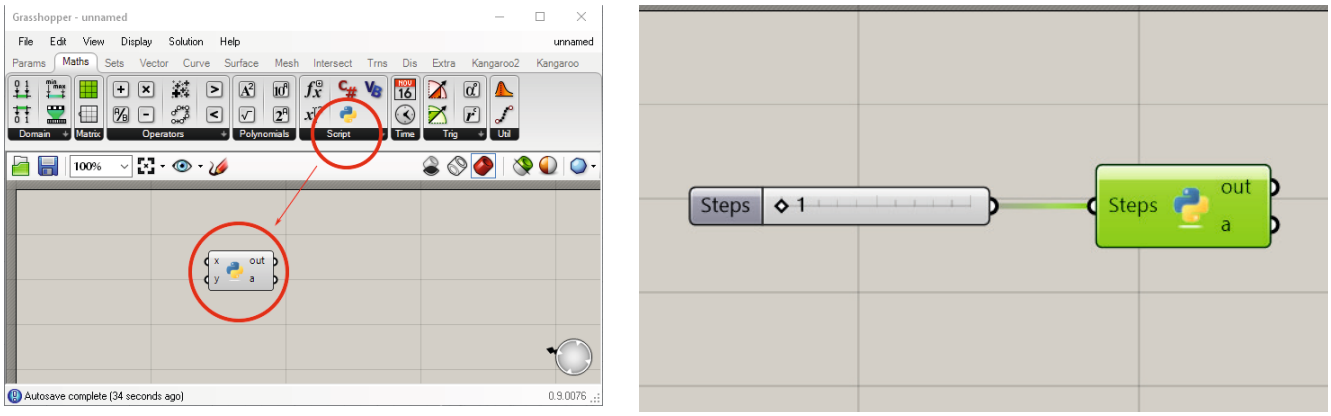


Figure 9: ***Left:*** *GHPython in Grasshopper;* ***Right:*** *Insert a slider and connect it to Steps parameter.*

J.1. GHPython component

Similar to other Grasshopper components you can manipulate the two input parameters (**x**) and (**y**) (you can rename, add or remove them). On the other side there is again an output parameter (out) for execution information (output and error streams) and one for a result parameter (**a**). The number of output parameters can be changed as well. For the moment we will remove the y-input and rename the x-input to Steps (right click on (**x**) and change name at the top). Add a slider for Steps (integer between 1 and 20) and connect it with the input parameter Steps (see *Figure 9* right).

Now we want to call our staircase program inside the component. Double click on the component will open the script window. In there we have to import our staircase program. How to do this? First we import the sys module (see *Program 35* line 1). We need to do this in order to add a sys-folder which contains our staircase program (see line 2). Finally, on line 4 we import the staircase.py program (compare *Program 28*).

1 http://www.food4rhino.com/app/ghpython (30.03.2018)

Program 35: GHPython call .py program

```
1   import sys
2   ghFile = str('C:\\..\\41 Tests')
3   sys.path.append(ghFile)                  add folder of staircase program
4   import staircase                         import the program (staircase.py)
5   reload(staircase)                        if you change something in the
                                             .py file, update it in GH
6   a = C08StaircaseGH.StepByStep(Steps)     call the command in .py and put
                                             it into the output parameter (a)
```

In the staircase.py file we have to comment out the call of the command on the last line and to return the polyline (in order to "get it into our hands"):

```
return rs.AddPolyline(points)
# GenNStairs(10)
```

Finally, we have to call the function in the GHPython component and equate the result to the output parameter (**a**) (see *Program 35* line 6). Only if we hand over the result to (**a**) Rhino draws the final polyline.

But, what if we want to draw the polyline directly in Grasshopper? In this case we return a list of point-coordinates (instead of a polyline). In Grasshopper we then turn this list into a list of points, since polyline asks for vertices. You just have to change the last line of our .py-program (see line of *Program 36*):

Program 36: GHPython call .py program

```
1   import rhinoscriptsyntax as rs
2   def SingleStep(x,y,z,StepDepth,StepHeight,points):
3       y = y + StepHeight
4       point = [x,y,z]
5       points.append(point)
6       x = x + StepDepth
7       point = [x,y,z]
8       points.append(point)
9       return (x,y,points)
10  def StepByStep(NumberOfSteps):
11      StepDepth = 32
12      StepHeight = 18
13      x = 0
14      y = 0
15      z = 0
16      point = [x,y,z]
```

Program 36: GHPython call .py program

```
17      points = []
18      points.append(point)                          add point (AddPoint)
19      step = 0
20      while step < NumberOfSteps:
21          x,y,points = SingleStep(x,y,z,StepDepth,
                StepHeight,points)
22          step = step + 1
23      return points                                 return list
                                                      of points
```

In Grasshopper, connect the output (the list of point-coordinates) to a point component. This will turn the points-coordinates into a list of points. Finally, connect the new list of points with a polyline component (see *Figure 10*):

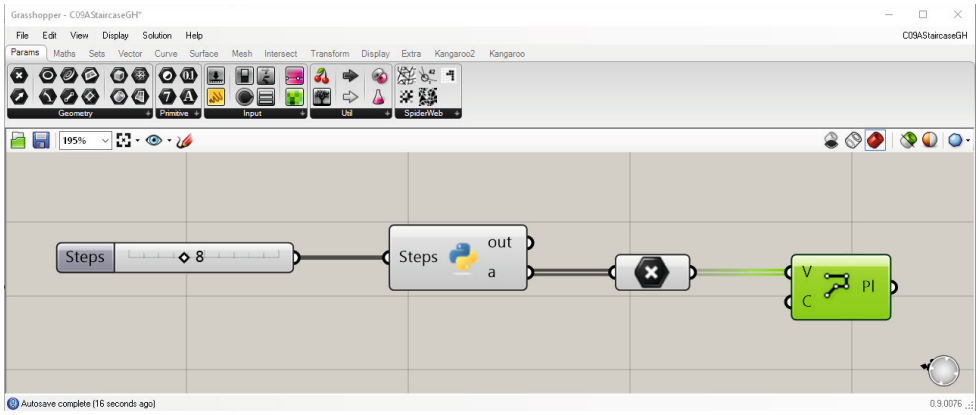


Figure 10: *Draw polyline with Grasshopper component.*

J.2. Questions

1. Change the staircase program and the GHPython component so that you can insert the StepHeight and StepDepth directly in Grasshopper (similar to Steps).

K. Epilog: Circles in a Polygon

[in which we want to close with a final program]

First we start with a description of the new program: We want to draw (n) non-overlapping circles within one or more predefined closed polylines. For a better distinction, these polylines are on a specific layer (let's name the layer PlanningArea; red lines in *Figure 11*). In addition, the circles have to avoid the inside of closed polylines on another layer (let's call this layer ExistingBuildings; yellow lines in *Figure 11*). A possible outcome is given in *Figure 11*:

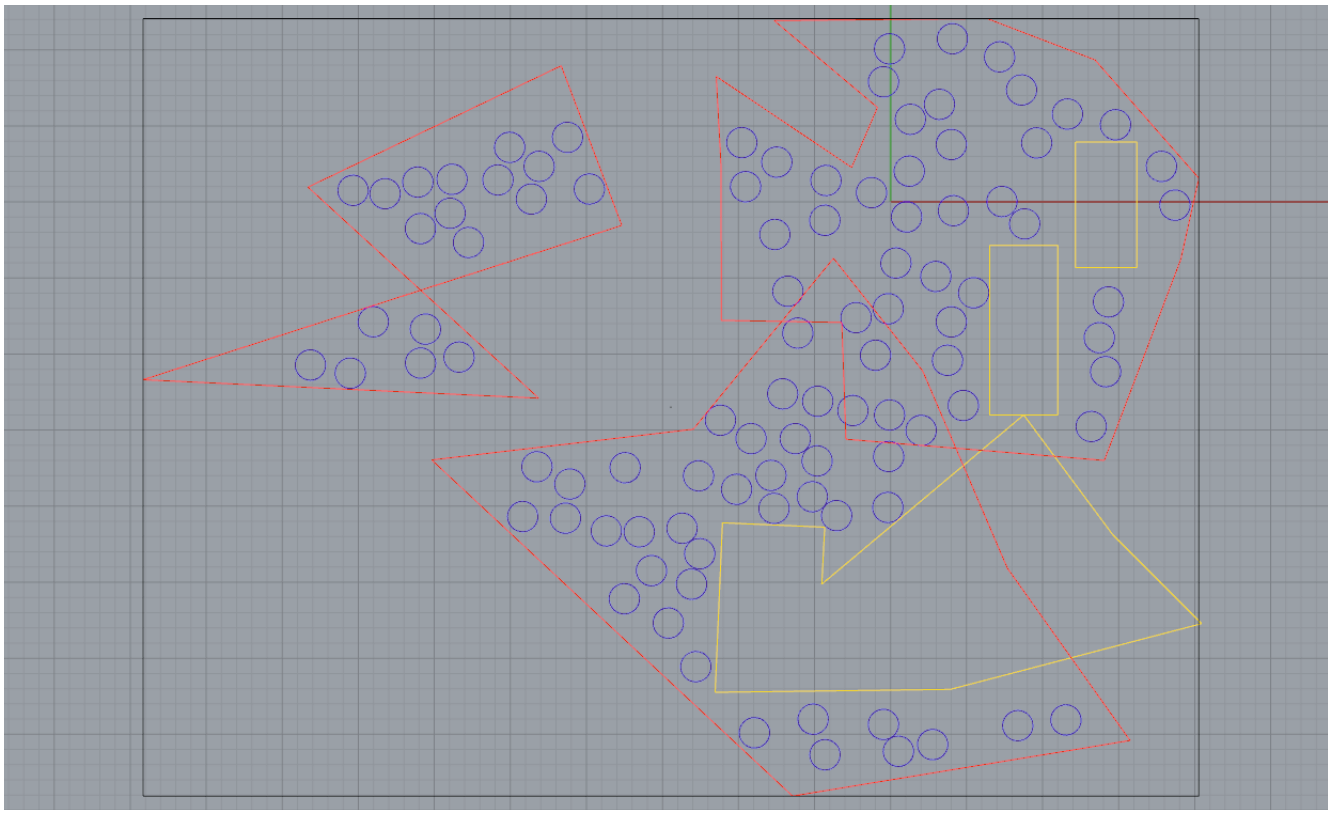


Figure 11: *The Circle program.*

K.1. Select curve on layer

Since we want to draw inside predefined polylines on a layer named PlanningArea, we have to write a function that gives back all polylines on a given layer.

Program 37: Get elements on certain layer

```
1   import Rhino                          import module Rhino
2   def ObjOnLayer(layername):
3       rhobjs = Rhino.RhinoDoc.          use function FindByLayer() of
            ActiveDoc.Objects.            module Rhino (which does what you
            FindByLayer(layername)        need): all objects on layer are
                                          stored in a variable (rhobjs)
4       if not rhobjs:                    check if (rhobjs) is empty
```

Program 37: Get elements on certain layer

```
5           Rhino.Commands.Result.Cancel    if yes, then cancel the command
6       else:                               in all other cases
7           listOfObj = []                  initialize an empty list
8           for obj in rhobjs:              for all objects in (rhobjs)
9               listOfObj.append(obj)       append to (listOfObj)
10          return (listOfObj)              hand over (listOfObj)to the caller
11  def Main():
12      objs =ObjOnLayer("PlanningArea")    save the result in a variable
13  Main()                                  start program
```

If you look at line 9, you may recognize that all Rhino objects are added to the list, no matter if it is a (closed) polyline or not. This may be improved by yourself. (Homework)

In the next step we want to get the bounding box of all elements on layer "PlanningArea":

Program 38: Get bounding box of elements in a list

```
1   import rhinoscriptsyntax as rs          import module
                                            RhinoScriptSyntax
1   def GetBoundingBox(objs):
2       ptsBoundingBox = rs.BoundingBox(objs)   use function BoundingBox()
                                                of module RhinoScriptSyntax
3       cornerA = ptsBoundingBox[1]         left bottom point of box
4       cornerC = ptsBoundingBox[3]         right top point of box
5   return cornerA, cornerC                 return two corners of
                                            the bounding box
```

In our main program, we call the command GetBoundingBox() and hand over the two returned corner points to a command called DrawCircles():

Program 39: Get elements on certain layer

```
1   def Main():
2       objs = ObjOnLayer("PlanningArea")
3       minPt,maxPt = GetBoundingBox(objs)      save corner A+C of the
                                                bounding box in 2 variables
4       DrawCircles(minPt,maxPt)                call command DrawCircles
5   Main()
```

Now we want to draw n-circles of radius (r) within the bounding box:

Program 40: draw Circles within bounding box

```
1   import random as rnd                          import module with
                                                  random functions
2   def GetRndPoint(minPt,maxPt):
3       x = rnd.uniform(minPt[0],maxPt[0])        get random number between
                                                  min x-val and max x-val
4       y = rnd.uniform(minPt[1],maxPt[1])        get random number between
                                                  min y-val and max y-val
5       pt = [x,y,0]
6       return pt                                 return the point within
                                                  bounding box
7   def DrawCircles(minPt,maxPt):
8       for i in range(0, n):                     repeat n-times (i is a counter)
9           pt = GetRndPoint(minPt,maxPt)         get random point within
                                                  bounding box
10          rs.AddCircle(pt,r)                    draw circle
```

Program 40 results in overlapping circles within a bounding-box. However, we want none-overlapping circles! How can we achieve this? We can calculate the distance between a (randomly) generated midpoint to every other midpoint of already existing circles (which means we have to store all midpoints of previously drawn circles in a list); (see *Program 41*). We only draw a circle (and add it to the list of midpoints) if no distance is smaller than two times the radius (r). But this opens up a new problem. What if we want to draw n-circles, but there is only space for less (e.g. n-1)? You are right when you say that we simply need a second abort criterion (e.g. another counter inside the for-loop):

Program 41: Check distance between new midpoint and existing ones

```
1   import rhinoscriptsyntax as rs
2   import math                                   import math functions
3   def getDist(x1,y1,x2,y2):                     calculate dist. between 2 points
4       dist = math.hypot((x1-x2),(y1-y2))        Pythagoras: calculate hypotenuse
5       return dist                               return the result (=distance)
6   def CheckOverlapping(pt,midPts):              check if circles overlap
7       isOverlapping = False                     initialize a variable with false
8       if len(midPts) > 0:                       check if at least one
                                                  midpoint exists in list
9           for midPt in midPts:                  if yes, go through each
                                                  element (midpoint) in list
10              dist = getDist(pt[0],pt[1],       get distance between
                    midPt[0],midPt[1])            generated point (pt) and
                                                  current midpoint in list
11              if dist < (2 * r):                check if distance is smaller
                                                  than two times the radius
```

Program 41: Check distance between new midpoint and existing ones

```
12                  isOverlapping = True        if yes, the circle of the
                                                generated midpoint overlaps
                                                an existing circle
13      return isOverlapping                    return the result (true or
                                                false for overlapping)
14  def DrawCircles(minPt,maxPt):
15      midPts = []                             initialize an empty list
                                                to store all midpoints
16      for i in range(0, n):                   loop n-times (for n circles)
17          j = 0                               initialize a counter j with 0
18          while j < m:                        loop m-times (for m tries
                                                to draw a single circle)
19              j = j + 1                       don't forget to increase j!
20              pt=GetRndPoint(minPt,maxPt)     get a random (mid)point
21              if CheckOverlapping(pt,         check if the circle with
                    midPts) == False:           (mid)point overlaps with
                                                an existing circle
22                  rs.AddCircle(pt,r)          if not, then draw circle
23                  midPts.append(pt)           append new midpoint to list
24                  j = m                       we don't need a further try, so
                                                equate counter to max tries (m)
```

Still we have a distribution of circle within the bounding box, but not within the given polylines on layer PlanningArea. So we have to insert another abort criterion (before line 22 of *Program 41*):

Program 42: check if midpoint is in PlanningArea

```
1   import rhinoscriptsyntax as rs
2   def CheckInArea(pt,layerName):
3       result = False                          at the very beginning we
                                                say that the point is
                                                outside the PlanningArea
4       objs = ObjOnLayer(layerName)            we already know this (see
                                                Program 37 line 12)
5       for obj in objs:                        we check for every object
                                                on layer PlanningArea
6           curve = obj.Geometry                convert obj into curve
7           if rs.PointInPlanarClosedCurve      check if (pt) is within an
                (pt,curve) == True:             object on layer PlanningArea
8               result = True                   if yes, result is true
9       return result                           hand over the result to caller
10  def DrawCircles(minPt,maxPt):
11  ...                                         see line 15-20 of Program 41
12      if CheckOverlapping(pt,
            midPts) == False:
13          if CheckInArea(pt,                  call function CheckInArea
                "PlanningArea") == True:
```

Program 42: check if midpoint is in PlanningArea

```
14              rs.AddCircle(pt,r)
15              midPts.append(pt)
16              j = m
```

And what about the addition that the circles have to avoid the inside of closed polylines on another layer (called ExistingBuildings). Now you will recognize the advantage of programming with commands (procedural). You just have to insert another query after line 12 of *Program 42*. This time we ask if the point is not inside an object on the layer ExistingBuildings:

```
1   def DrawCircles(minPt,maxPt):
2   ...
3       if CheckOverlapping(pt,midPts) == False:
4           if CheckInArea(pt,"PlanningArea") == True:
5               if CheckInArea(pt,"ExistingBuildings") == True:
6                   rs.AddCircle(pt,r)
7                   midPts.append(pt)
8                   j = m
```

Now you might say: we are finished. But have a look at *Figure 12*. You see that some circles are overlapping the border lines.

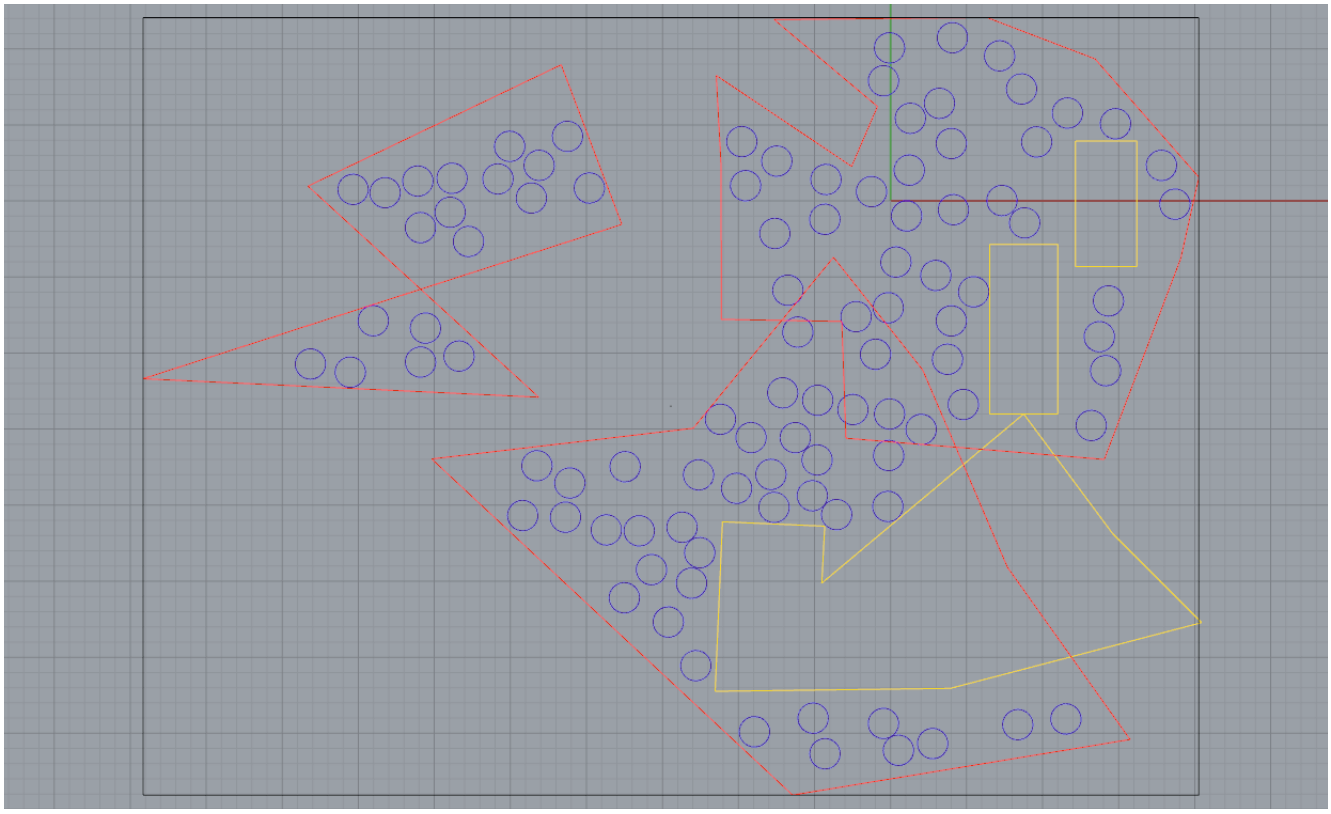


Figure 12: *The Circle program so far.*

So let's insert a last check for intersection with border lines of the planning area and the existing buildings:

Program 43: check for intersection

```
1   import rhinoscriptsyntax as rs
2   def CheckIntersection(tempCircle,
        layerName):
3       objs = ObjOnLayer(layerName)
4       for obj in objs:
5           curve = obj.Geometry                    convert circle
                                                    into curve
6           intersects = rs.CurveCurveIntersection  we use a function of
                (curve,tempCircle)                  rhinoscriptsyntax to
                                                    check intersection
7           if not intersects == None:              "not none" means that
                                                    both objects intersect
8               return True                         therefore return True
9   def DrawCircles(minPt,maxPt):
10  ...
11      tempCircle = rs.AddCircle(pt,r)             store circle in variable
12      if not CheckIntersection(tempCircle,        checks intersection
            "PlanningArea") == True and not         with objects on layer
            CheckIntersection(tempCircle,           PlanningArea and on
            "ExistingBuildings") == True:           layer ExistingBuildings
13          midPts.append(pt)                       if no intersection
                                                    exists, delete circle
14          j = m                                   we don't need a further
                                                    try, so equate counter
                                                    to max tries (m)
15      else:                                       in all other cases
16          Rhino.RhinoDoc.ActiveDoc.Objects.       delete circle (because
                Delete(tempCircle, True)            it intersects a
                                                    border line)
```

You can now call the file within a GHPython component in Grasshopper and add sliders for the three input parameters (homework):

- r = 1 (radius of circles)
- n = 30 (number of circles in best case, if all tries to set a circle are successful)
- m = 30 (number of tries to set a single circle; exclusion criteria are: overlapping with other circles, placement outside objects on layer PlanningArea, placement inside objects on layer ExistingBuildings and intersection with border lines)

K.2. Questions

1. Think about what happens if no object is on the layer PlanningArea and ExistingBuildings. At which line of code would you insert such a query?
2. Test the code and look if there are other cases that produce an error.

Appendix

Some code snippets that might be interesting:[1,2]

Program 44: add layer to project

```
1   import Rhino
2   def AddLayer(layerName, col):
3       layerIndex = Rhino.RhinoDoc.ActiveDoc.
            Layers.Find(layerName, True)          find layer with
                                                  proposed name
4       if layerIndex>=0:                         if index
5           return Rhino.Commands.Result.Cancel   then cancel command
6       layerIndex = Rhino.RhinoDoc.ActiveDoc.
            Layers. Add(layerName, col)           try to add a layer
                                                  with proposed name
7       if layerIndex<0:                          if it didn't work
                                                  (e.g. some character
                                                  is not allowed)
8           return Rhino.Commands.Result.Failure  return failure
9       return Rhino.Commands.Result.Success      if everything worked,
                                                  return success
10  AddLayer("Name", System.Drawing.Color.Blue)   call command and
                                                  passe new name of
                                                  layer and its color
```

Program 45: delete objects on a layer

```
1   import scriptcontext as rc
2   def DeleteObjectsOnLayer(layerName):
3       rhobjs = Rhino.RhinoDoc.ActiveDoc.
            Objects.FindByLayer(layerName)        store all objects on layer
                                                  in a variable (rhobjs)
4       if not rhobjs:                            check if no object found
5           Rhino.Commands.Result.Cancel          if yes, cancel
6       else:
7           for obj in rhobjs:                    for all objects
                                                  in (rhobjs):
8               Rhino.RhinoDoc.ActiveDoc.         delete object
                    Objects.Delete(obj, True)
9           rc.doc.Views.Redraw()                 redraw view
10          return Rhino.Commands.Result.Success  return that everything
                                                  was successful
11  DeleteObjectsOnLayer("layerName")             call function that deletes
                                                  layer with layerName
```

1 see code snippets on add layer: developer.rhino3d.com/samples/rhinocommon/add-layer/ (31.03.2018)

2 see code snippets on delete objects on layer developer.rhino3d.com/samples/rhinocommon/select-objects-on-layer/ (31.03.2018)

Cheat Sheet: All Programming Constructs at a Glance

	Python
Commands Function	*name* (*input*): *block of code*

Conditions	**If** *condition*:
	    block of code to execute if condition is True
	Else:
	    block of code to execute if condition is False

Loops	**While** *condition*: *block of code to execute if condition is True*

Cheat Sheet: What to do with every type

	Sub, Function or Operator	Description
Integer	x + y, x – y, x / y, x * y math.sin(x), math.cos(x) math.sqrt(x) round(x) int(x)	For calculations Trigonometric functions Square Root Round a Double to Integer Cut off a Double (no decimals)
Float	same as Integer, additionally: random.random()	 get a number between [0..1[
Boolean	x > y, x < y, x == y, x <> y not x, x and y, x or y	Comparison operators Logical operators
String	+ len(s) s[posStart, posEnd] s.find(searchword) s.isnumeric() float(s), int(s), bool(s) s.lower(), s.upper s1 == s2	Concatenation operator Length of a String Extracts characters from the string in the range from posStart to posEnd Returns position of searchword in s, or -1 if not found Returns true for strings containing only integers Converts a String Returns lower and upper case Returns True if s1 equals s2, else False
List	len(a) a .append(value)	gives the length of a list appends a value to a list (adds an element at the end of the list and hands over the value)

Don't forget to import the correct libraries first, e.g. "math" for mathematical operations or "random" for random numbers.

Cheat Sheet: RhinoScriptSyntax Drawing Functions

	Description	Parameter(s)	Note
AddPoint	Adds point object to the document	point = [x,y,z]: location of midpoint;	
AddCircle	Adds a circle curve to the document	plane on which the circle will lie or center of the circle; radius of the circle;	
AddLine	Adds a line curve	start and end point;	
AddPolyline	Adds a polyline curve	list of 3D points;	N is at least 2 (if N<4 then start and end must differ)
AddBox	Adds a box shaped polysurface	3D points that define the corners of the box;	N = 8; Points need to be in counter-clockwise order starting with the bottom rectangle of the box
AddSphere	Add a spherical surface	plane on which the sphere will lie or center point of the sphere; radius of the sphere;	If a plane is input, the origin of the plane will be the center of the sphere
AddCone	Adds a cone shaped polysurface	plane with an apex at the origin and normal along the plane’s z-axis or 3D origin point of the cone and 3D height point;	If base is a plane, height is a numeric value
AddCylinder	Adds a cylinder-shaped polysurface	base plane or 3D base point and 3D height point; radius;	If base is a plane, height is a numeric value
AddText	Adds a text string	the text to display; a 3-D point or the plane on which the text will lie;	The origin of the plane will be the origin point of the text