PythonParts for Developers

Created 11/2015

Bert Van Overmeir – SCIA Herk-de-Stad

Contents

Installation

Allplan with Python Integration

Install Allplan with Python support through the installer package provided. After a normal installation has been executed, a few extra additions in relation to PythonParts have been added to Allplan. The respective paths can be found below. We will discuss each one of them more in detail later on the tutorial series.
Firstly, **examples of PythonParts** can be found under:

$$X{:}\backslash ProgramData\backslash Nemetschek\backslash(\dots)\backslash ETC\backslash Examples\backslash PythonParts$$

Please refer to “testing” for more information about the examples.
The **Allplan Python libraries** that we will be using are located under:

$$X{:}\backslash Program\ Files\backslash Allplan\backslash(\dots)\backslash Prg$$

It is important that you remember this pat as we are going to need it further on installation.
Thirdly, **scripts written in Python** can be found under:

$$X{:}\backslash ProgramData\backslash Nemetschek\backslash(\dots)\backslash 2016\backslash ETC\backslash PythonPartsScripts$$

It is important that you do not mover any of the files, nor rename the scripts. In the example files, Allplan refers to certain scripts. Execution will fail upon changing names.

Installing the development kit for Python

To use any of the scripts within Allplan, we do not need to install development kit. If your aim is to use existing PythonParts, you can skip this part and move to “use of PythonParts”.

Visual Studio 2013

PythonParts are best developed in Visual Studio. Although development is, of course, not limited to this tool, we strongly recommend to use visual studio as this tutorial series and the plugins provided will use Visual Studio too.
To install Visual Studio 2013, visit the link below.

$$https://www.visualstudio.com/downloads/download-visual-studio-vs$$

Make sure you pick **Visual Studio 2013 Community Edition with Update 5** as so:

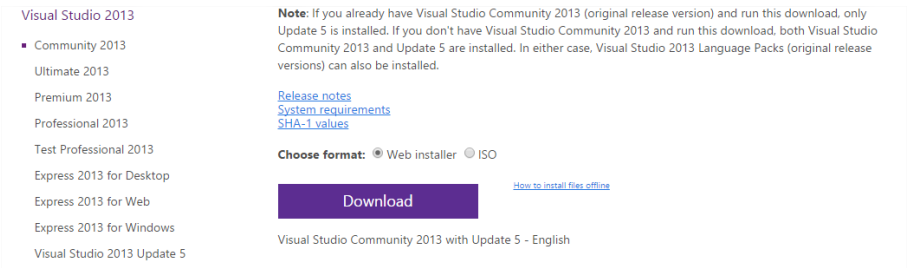

Figure 1: downloading visual studio

Picking another edition of Visual Studio is not wrong, but it is however problematic for installing the Python Tools later on. Download the web installer as we are only going to need a small portion of the full capabilities of Visual Studio. Upon installation at the Optional Features dialogue, it is safe to **untick all the optional features except for the web developer tools**, as they might come in handy for report development in Allplan.

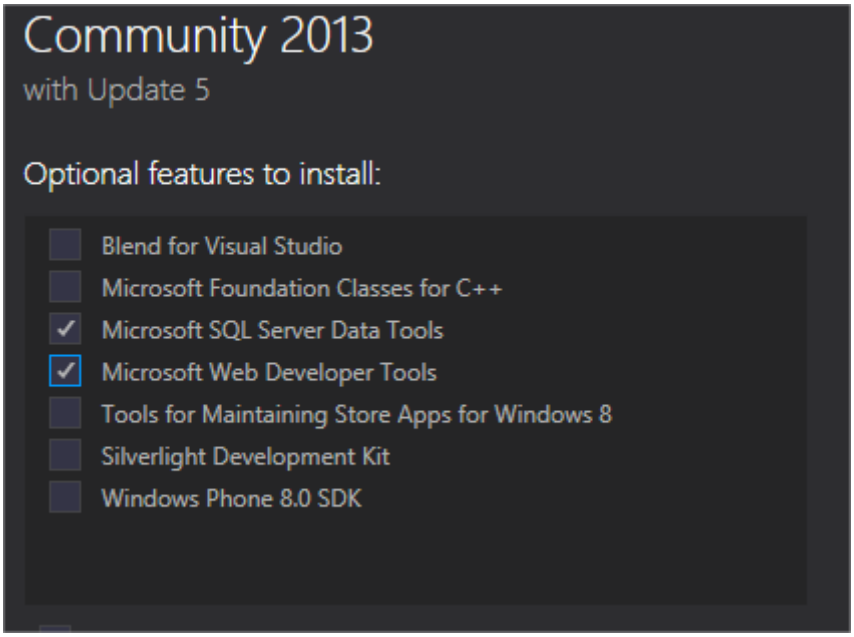

Figure 2: selecting the right installation options in Visual Studio

Now click install. Be aware that installation can take **up to two hours**. So be prepared to do some other work in the meantime. After the installation of Visual Studio has completed, verify if the program starts. You should get the following screen.

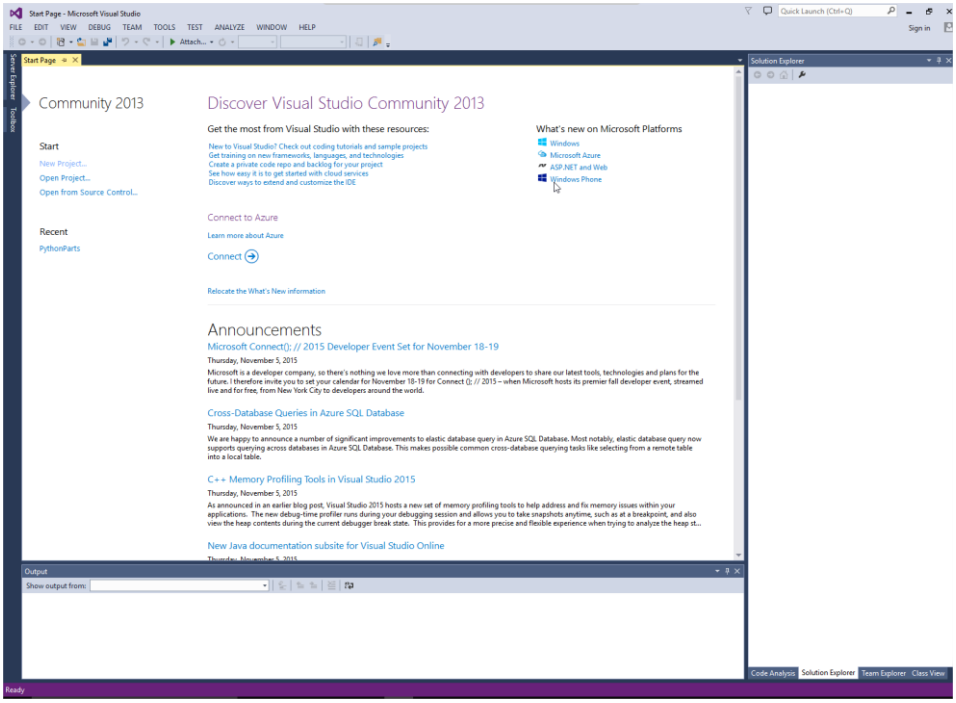

Figure 3: Visual Studio Interface

PythonTools for Visual Studio

The Required Python tools for Visual Studio should have been provided by your local Allplan dealer. If not, please contact them for more information. Alternatively you can search for the python tools via the link below, and search for **version PTVS 2013 BETA (.30811.01)**.

$$https://www.visualstudio.com/en-us/features/python-vs.aspx$$

It is of key importance that you download the correct version of the Python Tools. The wrong version will not work with your Visual Studio installation.

Before installing, you should disable the verification of files, this can be done by editing the registry. The file should also be provided within the installation package you got from the Allplan dealer. If this is not the case, do not continue installation before acquiring them.

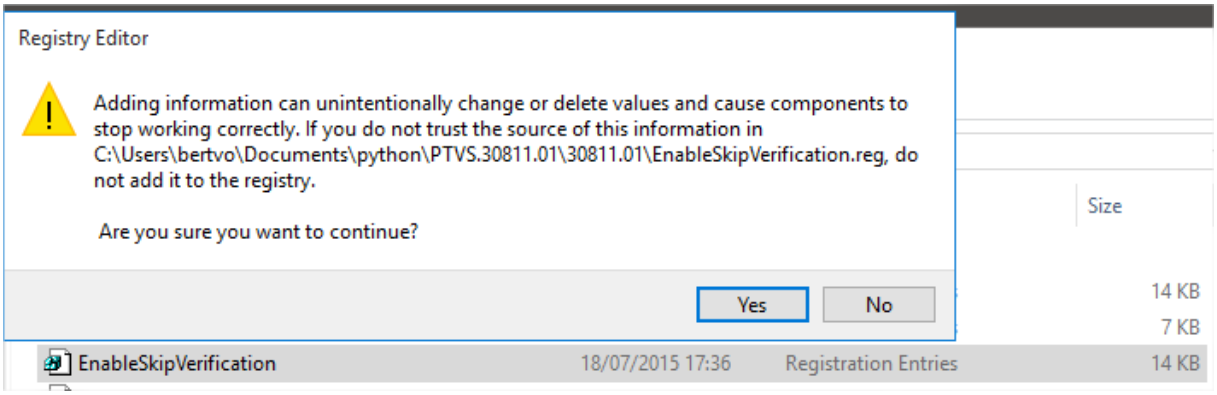

Figure 4: skipping the verification process of the installation, click yes to continue

After adding the entries to the registry, you are free to install the Python Tools for Visual Studio.

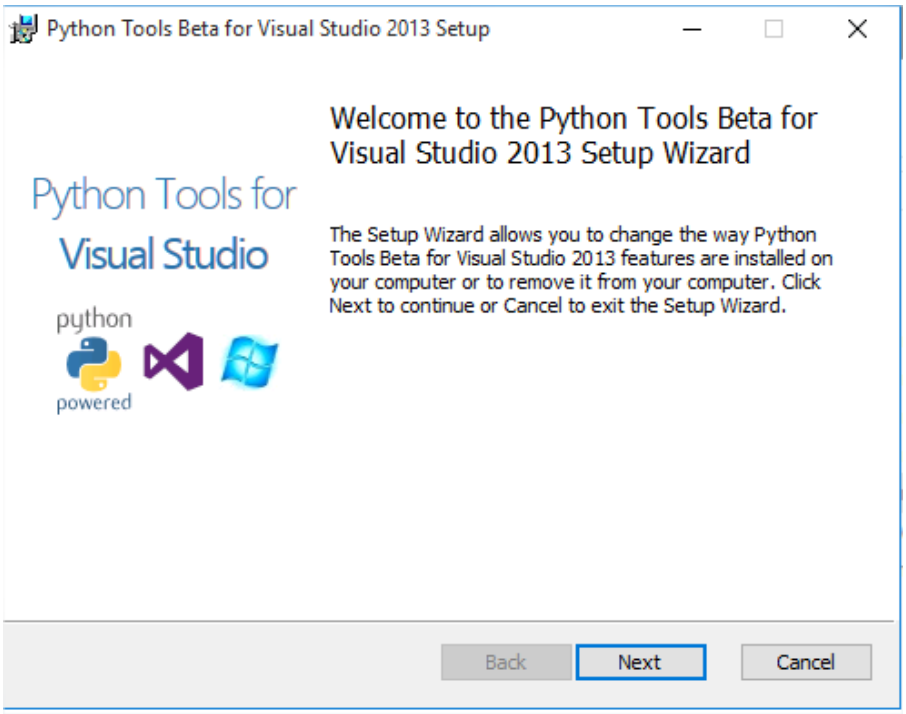

Figure 5: installing AFTER the skip verification has been executed

Installing the Python Interpreter

Upon starting Visual Studio, the Python Tools will be recognized, but you find yourself unable to successfully executing a program as no interpreter has been installed.

To install an interpreter visit:

$$https://www.python.org/$$

Go to downloads, click on windows, scroll down and look for the **python-3.4.3. X64** setup.

After installation, when you start Visual Studio, go to Tools, Options and check if the Python environment is installed.

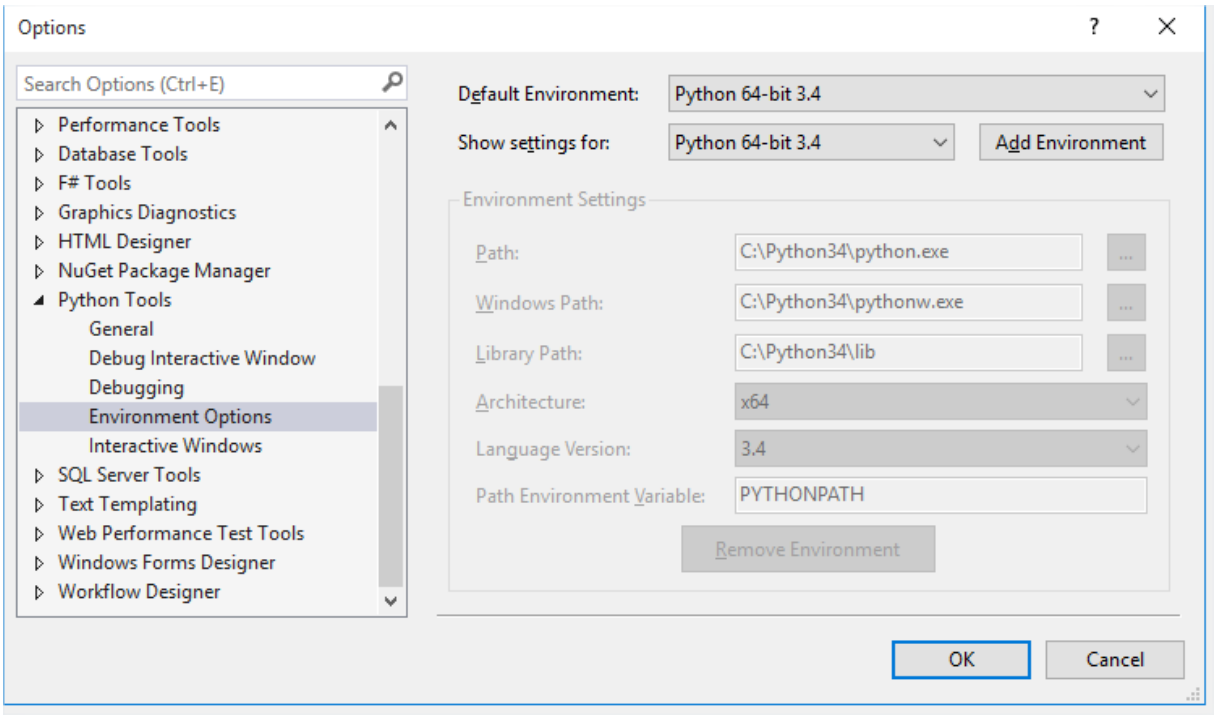

Figure 6: checking if the python environment is installed.

If this is not the case, click on add environment and add it manually providing the paths to the python.exe, python.exe and library (view screenshot). If you did a default install, the paths should resemble the paths in the screenshot.

That's it! You are all set up to start working with Python for Allplan!

Basic editing of PythonParts in Allplan

Opening your first PythonPart

Every PythonPart in Allplan consists of a few files, but in order to open an existing file, only one file is needed. You might have noticed that Allplan Python examples contain the following files in different folders:

$$.py\ files: the\ python\ script, we\ will\ get\ into\ this\ later$$
$$.png\ files: placeholder\ images, obsolete$$
$$.pyp\ files: the\ most\ important\ files\ that\ can\ be\ used\ within\ Allplan$$

PythonParts can be dragged straight into Allplan. By dragging the *.pyp* file in Allplan, a similar interface to SmartParts is shown.

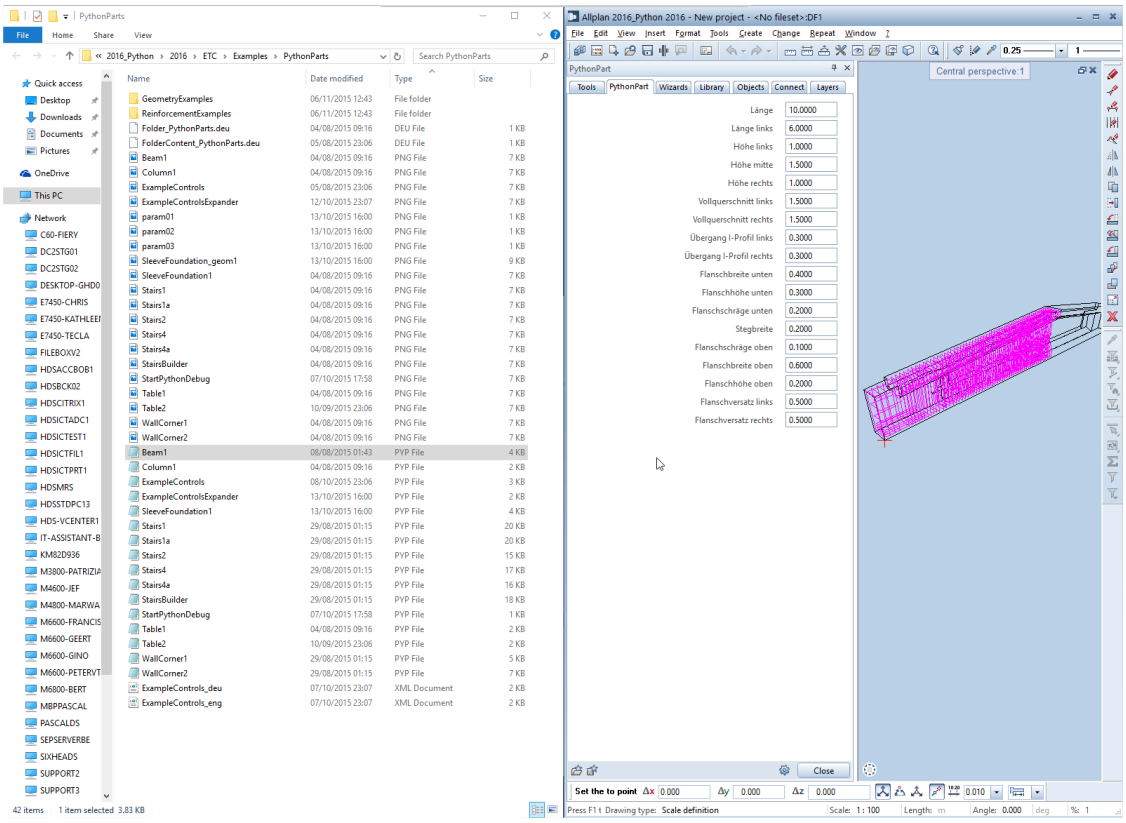

Figure 7: dragging the PythonPart "Beam" into Allplan which shows a few extra options, just like SmartParts.

In current versions (Allplan 2016.1.0) it is only possible to drag the PythonPart in Allplan and edit it **once,** after you close the tab in Allplan, the elements do get generated and become genuine Allplan objects. This way, a fully functional 3D model of reinforcement can be easily made, but it is not possible to easily change special options like in SmartParts, you would just have to use the standard editing tools like “stretch”.

The *.pyp* files do not need to be in the ETC folder in order to be able to work properly. They can be placed anywhere on the computer. It is however important **that the *.py* files reside in the correct folder (X:\ProgramData\Nemetschek\\(…)\2016\ETC\PythonPartsScripts)** as the *.pyp* files reference to them. This should already be the case if you are using the examples provided.

Editing PythonPart parameters

Opening a .pyp file

Every PythonPart consists of at least two files (more is possible as you write more advanced scripts).

$$< name\ of\ PythonPart > .pyp\ located\ wherever\ you\ want$$
$$< name\ of\ Script > .py\ located\ in\ PythonPartsScripts$$

When we open a *.pyp* file in a text editor, we see something that resembles the following:

```
<?XML VERSION="1.0" ENCODING="UTF-8"?>
<ELEMENT>
  <SCRIPT>
    <NAME>COLUMN1.PY</NAME>
    <GEOMETRYEXPAND>0</GEOMETRYEXPAND>
  </SCRIPT>
  <PAGE>
    <NAME>GEOMETRIE</NAME>
    <PARAMETER>
      <NAME>COLUMNWIDTH</NAME>
      <TEXT>BREITE</TEXT>
      <VALUE>600.</VALUE>
      <VALUETYPE>LENGTH</VALUETYPE>
    </PARAMETER>
```

Every *.pyp* file consists of a fairly straightforward structure written in XML and can be easily edited. The *.pyp* file contains all the parameters that will be later on used to generate the menu within Allplan whilst opening a new PythonPart.

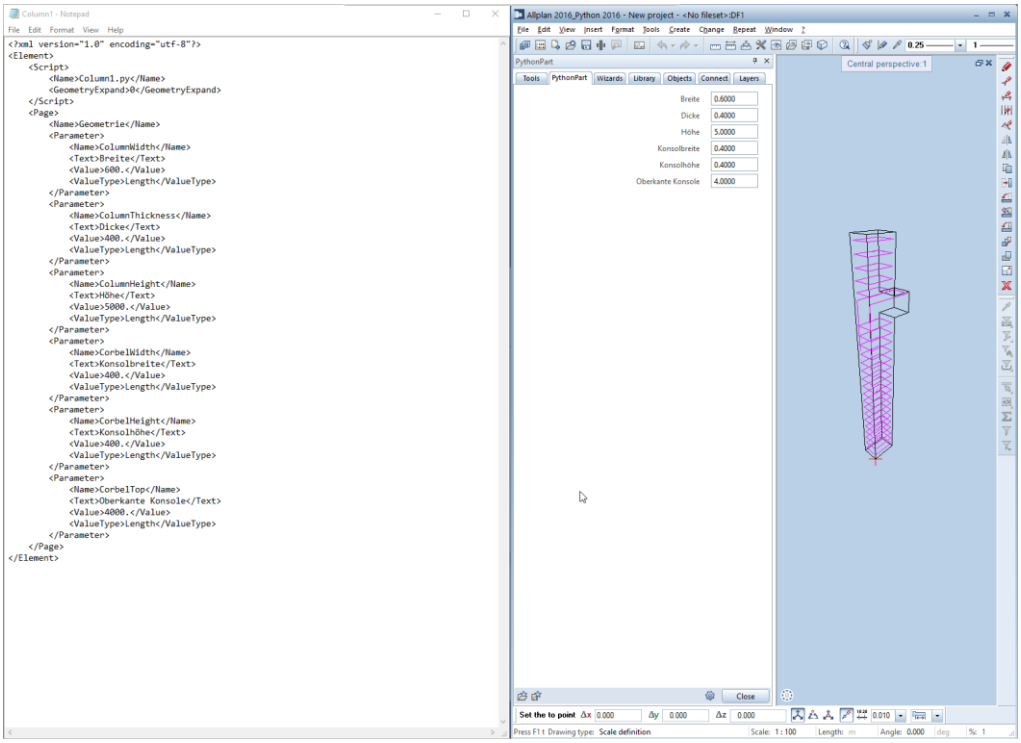

Figure 8: comparison between menu in Allplan and XML structure in notepad of the .pyp file

Editing a .pyp file

Files in .pyp format make use of the XML standard. This means that they can be easily edited. In XML we can make multiple objects and assign values to them. We can also put multiple objects within another object. This is called nesting. Every object (or section) is declared between <object_name> and always ends with </object_name> (notice the forward slash). Every PythonPart in Allplan has the following structure:

Descriptive line

The descriptive line contains the version of XML used and the font format (standard is UTF-8)

Element statement

Every PythonPart in Allplan will (besides the descriptive line) be edited within an <Element> </Element> statement.

Script statement

Within the Element statement, the most important to declare is the adjacent script that is connected to the .pyp file currently in. Allplan will, upon execution of the script, look for this file in the "*PythonpartsScripts*" folder. It is important that the syntax of the name is correct. Nothing will happen once you declare a wrong filename.

```
<SCRIPT>
    <NAME>COLUMN1.PY</NAME>
    <GEOMETRYEXPAND>0</GEOMETRYEXPAND>
  </SCRIPT>
```

The script section contains of two parameters. The first parameter, **name**, refers the the location where the .py file is saved as stated above. Absolute paths, nor relative paths are possible. The location should always be in the destined folder within in Allplan. The second parameter, **GeometryExpand**, is very important. GeometryExpand defines if Allplan should let the Pythonpart react on other objects within its interface. A fine example of this is reinforcement that needs to be placed in a wall around a window. You want the reinforcement to react to the surrounding hole in the wall. Then you will have to set parameter GeometryExpand to 1.
Later on we will see that only setting this value to 1 does not suffice. An extra definition in the script's body needs to be defined.
As a general recommendation: we do recommend that you first develop your .pip files and define all the parameters. Your code in Python has to reference to these parameters. Things go a lot easier when parameters are created before starting on the code. It keeps things structured.

Page statement

Page is a placeholder for the different tabs within an Allplan menu (cfr. to SmartParts). It works the same way as an Element statement. Whenever you want to close the page and start a new one, start a new page and close the previous one with </page>. Every page has two types of fields. A page has a name and a parameter section.

Parameter statement of a page

One parameter section within a page might look like this:

```
<PARAMETER>
  <NAME>COLUMNWIDTH</NAME>
  <TEXT>BREITE</TEXT>
  <VALUE>600.</VALUE>
```

```
    <ValueType>Length</ValueType>
  </Parameter>
```

In this example, the width of a column is defined. The name, text and value is not of utter importance and can be defined by the developer. The **valuetype** on the other hand is very important. The following types are recognized:

VALUETYPE	GENERATED INPUT FIELD IN ALLPLAN
LENGTH	ENTER A DISTANCE FIELD
LAYER	SELECT A SUBLAYER FROM DROP DOWN MENU
PEN	SELECT A PEN THICKNESS FROM DROP DOWN MENU
CHECKBOX	DISPLAYS A CHECKBOX (VALUE 0 OR 1)
COLOR	SELECT A COLOR (ONLY ALLPLAN COLORS)
REINFBARDIAMETER	SELECT A DIAMETER FROM DROP DOWN MENU
INT	ENTER AN INTEGER (ONLY WHOLE NUMBERS)
DOUBLE	ENTER A DOUBLE (COMMA ALLOWED)
EXPANDERSTART / EXPANDEREND	ADD EXPANDERS TO CERTAIN PARAMETERS

Using the expander

The expander allows for showing certain parameters or omitting them in a drop down fashion. The expander code works as follows:

```
<Parameter>
            <Name>Expander1</Name>
                <Text>Expander1</Text>
                <Value>True</Value>
                <ValueType>ExpanderStart</ValueType>
</Parameter>
... (parameters)
<Parameter>
                <ValueType>ExpanderEnd</ValueType>
</Parameter>
```

The expander is firstly defined as a parameter. The value of the expander defines if it should be open upon start or closed. Ending a certain section of the expander works by defining a new parameter with valuetype “ExpanderEnd”.

Adding an image

Adding an image is as of today not yet implemented (Allplan 2016.1.0)

Working with reinforcement

After closing the final page on your *.pip* file with a </page> statement, you can add reinforcement when needed. Reinforcement sections let you define placements for bars with certain parameters. Be warned that declaring a reinforcement shape does not automatically create an entry in a menu of the PythonPart to enable and disabling it. A parameter type checkbox should be created and should enable and disable the reinforcement shape in the script. Remember that the *.pyp* script absolutely has no intelligence. Any connection between scripts MUST be defined in the python script. One exception to this rule is the use of referencing to already existing parameters. One example is shown below where, in the menu, a “ConcreteCover_reference” parameter is shown. In the reinforcement object it is possible to refer to this parameter through the *.pyp* file alone.

```
<REINFORCEMENT>
    <SHAPENAME>USER SELECTABLE PART OF REINFORCEMENT</SHAPENAME>
    <ID>0</ID>
    <DIAMETER>8</DIAMETER>
    <DISTANCE>100</DISTANCE>
    <CONCRETECOVERSHAPE>CONCRETECOVER_REFERENCE</CONCRETECOVERSHAPE>
    <PLACEMENTCOVERLEFT>CONCRETECOVER_REFERENCE </PLACEMENTCOVERLEFT>
    <PLACEMENTCOVERRIGHT>CONCRETECOVER_REFERENCE </PLACEMENTCOVERRIGHT>
    <CONDITION>SHAPE1</CONDITION>
  </REINFORCEMENT>
```

VALUETYPE	DESCRIPTION
SHAPENAME	NAME OF THE PLACEMENT
ID	ID NUMBER
DIAMETER	DIAMETER OF THE BARS
DISTANCE	DISTANCE BETWEEN THE BARS IN THE PLACEMENT
CONCRETECOVERSHAPE	COVER OF THE BARS IN RELATION TO THE SHAPE IN PI
CONCRETECOVERLEFT	END COVER LEFT OF BARS
CONCRETECOVERRIGHT	END COVER RIGHT OF BARS
CONDITION	UNKNOWN PARAMETER (CONDITION=SHAPENAME)

Externally editing parameters for Allplan

Although it is possible to change the parameters inside Allplan through the menu that shows when dragging them inside the program, a much more interesting approach would be to externally access and edit the files.

An example could be a website where users could define their own preferences for a beam. On the website, a plain XML file is generated with PYP code. This file can now easily be dragged in Allplan (providing the script is functional) without the user having anything more to do than extracting the file from the website and opening the finished beam in Allplan.

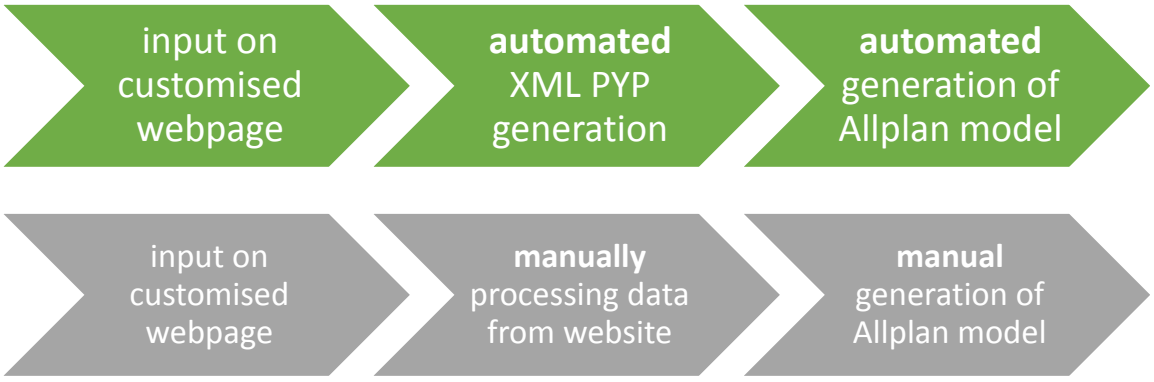

Figure 9: comparing the old and new workflow of Allplan with PythonParts

Currently, a test version of this XML PYP editor/generator is under development @SCIA Herk-De-Stad.

Advanced Editing of PythonParts in Visual Studio

Setting up the workspace

In order to use Python in Visual Studio (which we already covered in chapter one) and more precise, use the libraries provided by Nemetschek for development, we need to set up the Visual studio environment.

Allplan libraries

The Allplan libraries which we need for development can be found in the PRG folder in the Program Files. Libraries for Python always have *.pyd* file extension. Remember the path to the files.

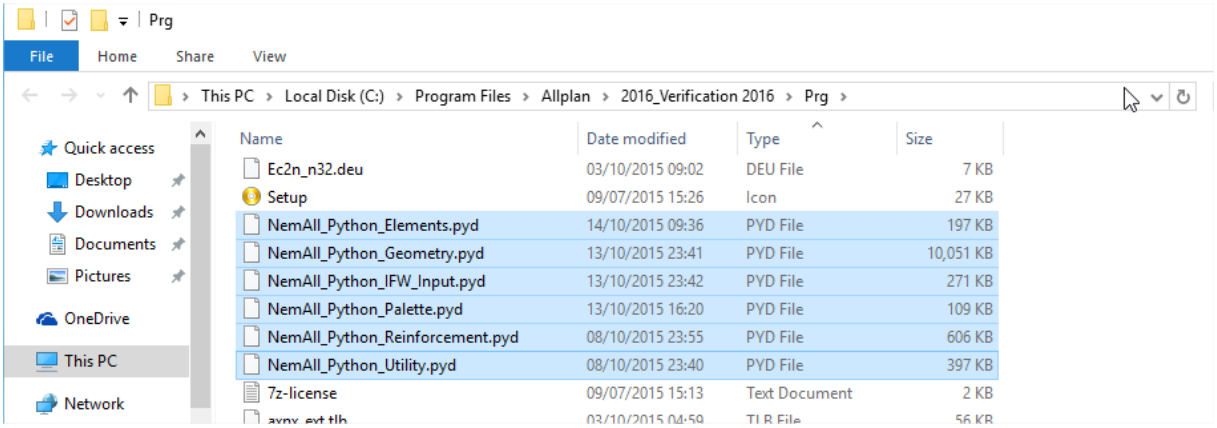

Figure 10: locating the Allplan PYD libraries

LIBRARIES	DESCRIPTION
ELEMENTS	?
GEOMETRY	CONTAINS ALL GEOMETRY FUNCTIONS AND OBJECTS
IFW_INPUT	?
PALETTE	?
REINFORCEMENT	CONTAINS ALL REINFORCEMENT FUNCTIONS AND OBJ
UTILITY	GENERAL UTILITIES

Python Project in Visual Studio

You can either create a new Python project manually in Visual Studio, or use the template provided by your Allplan reseller.

When using the template, open the *.pyproj* file in a text editor and change the paths of the <reference include="…"> and all other paths to the appropriate path in the library. Open the template afterwards, Visual Studio will start with the necessary libraries already included.

When starting from scratch, create a new Visual Studio project and select Python Application from the list. Afterwards you need to add the libraries yourself. Therefore click right on the references in the right pane and select “add reference”. Now you can also browse for the libraries. With shift-click you can select multiple libraries at the same time.

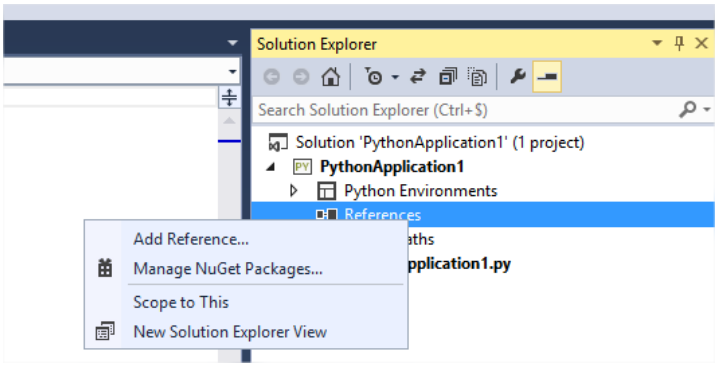

Figure 11: adding a new library to the project

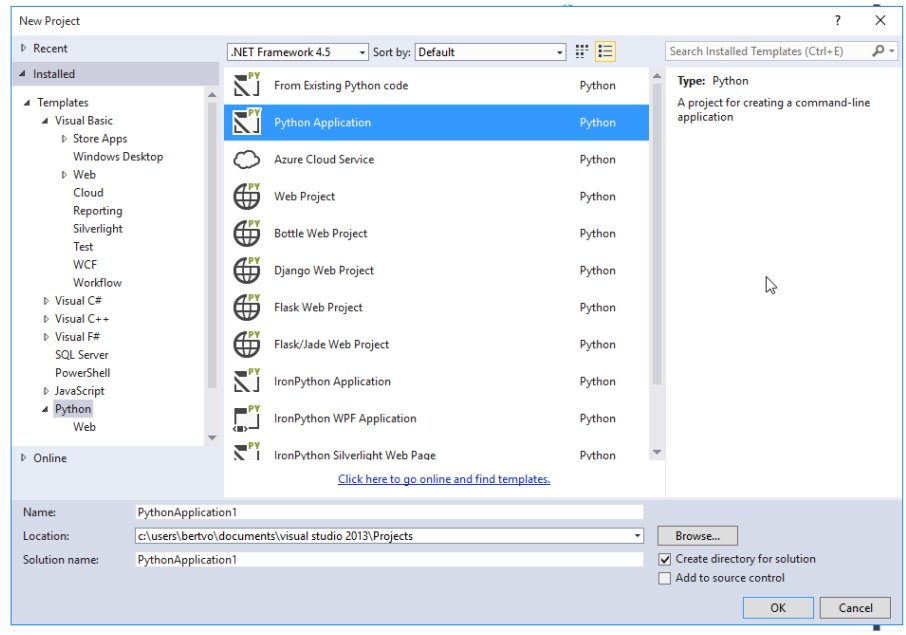

Figure 12: selecting Python application from the Python tab

When going to project > properties, you can select the interpreter to be used. When no interpreter is available, please refer to the installation chapter. An interpreter is the vital component that will execute and compile your code and check for errors. You can also define a startup application. In most cases you are only going to have one script in a project. So the name of that script should already be in the list.

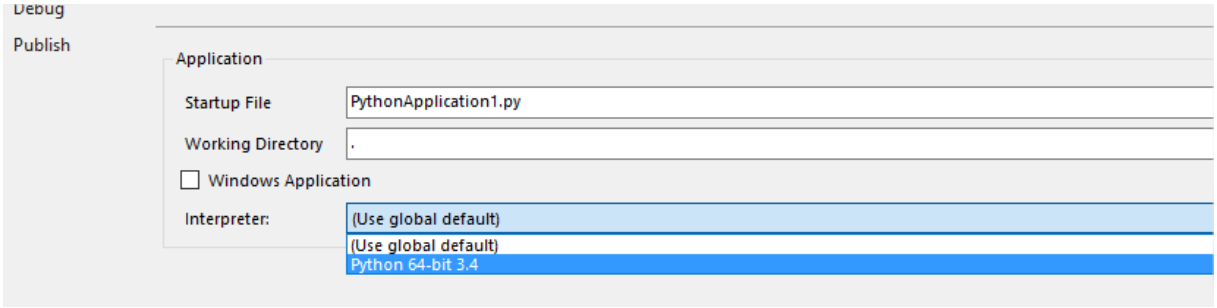

Figure 13: selecting an interpreter in the properties pane

Double clicking on the script in the right pane should open the script and show an absolutely empty screen with a cursor

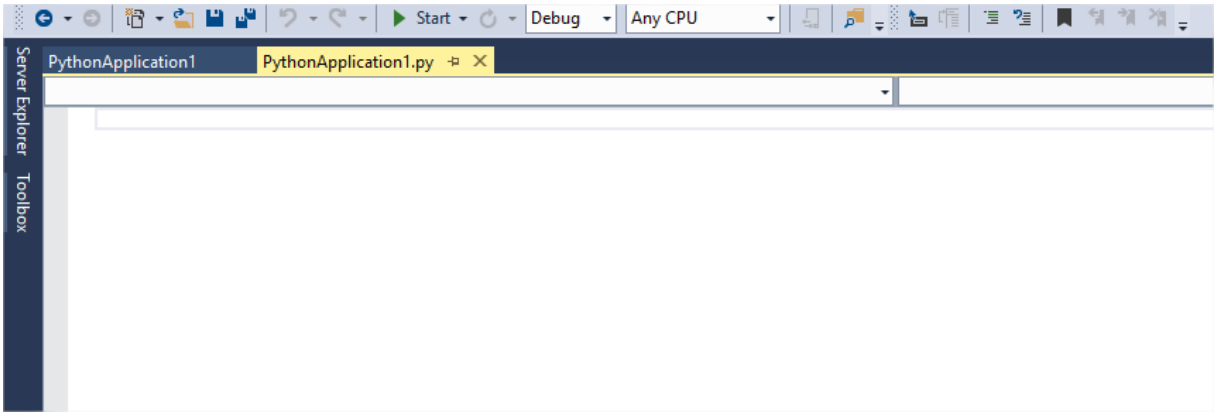

Figure 14: the empty script

In order to test if everything is working, type <print (“hello”)> (without <>). You should get an output window showing you hello after clicking on the green little arrow in the top screen.

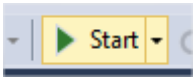

Figure 15: run application

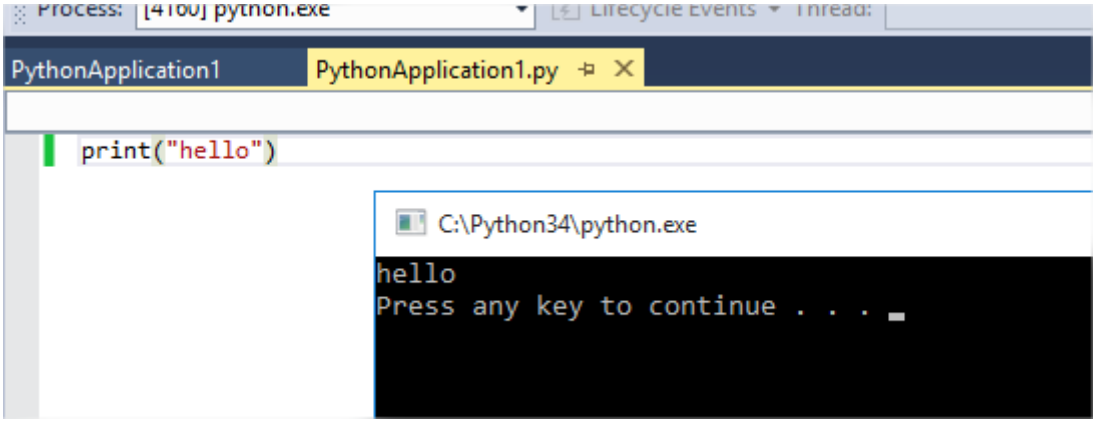

Figure 16: application running and well - Visual Studio has been set up correctly

Analyzing an example script

Generating PythonDoc

To help you get going, it might be useful to generate documentation of all the definitions in the Allplan libraries. This can be done through the test projects included in your installation.

When using the project template, these will already be in the list with python scripts. When you created a project from scratch, you will have to drag the file from the Examples/PythonParts folder into your project’s solution explorer. It will be added to the project and automatically use your imported libraries.

Import the “createHTMLDocumentation.py” file in your project.
A little bit down in the script you will see the following:

$$DRIVE\_LETTR = 'X:\\'$$

Change X to the drive on which you installed Allplan and run the script. Documentation is now written to *ETC/PythonPartsScripts/Docu*. The documentation consists of html files that are easy to read.

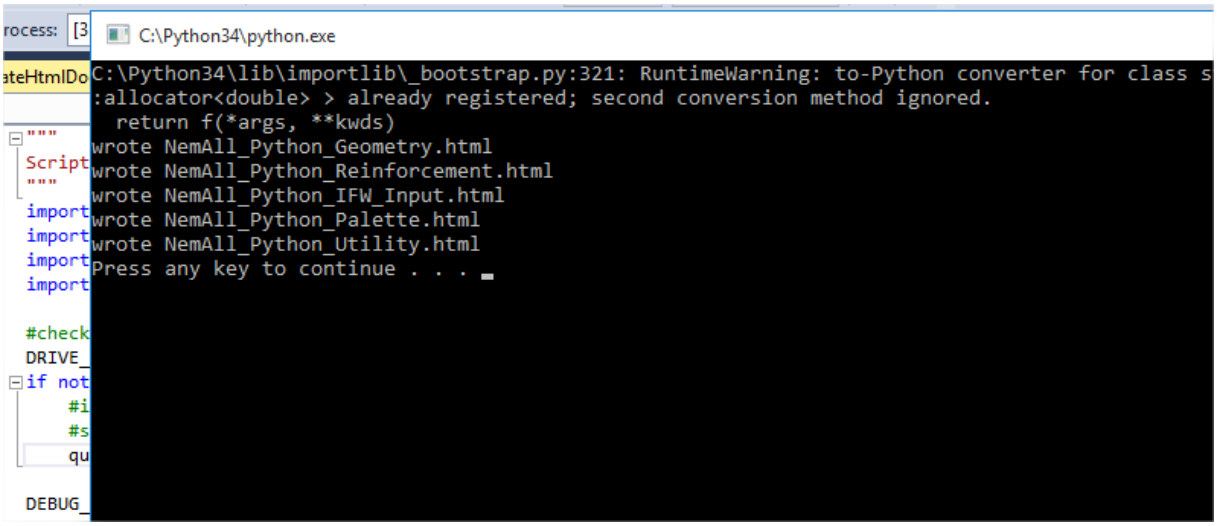

Figure 17: successfully generated pythonDoc

Before we start

In this tutorial series, we do expect some basic knowledge of programming. If you are unfamiliar with how Python works or coding in general, we refer to the following tutorial on YouTube in English. In order to fully understand the next paragraphs, you should be familiar with concepts like: definitions, classes, inheritance, objects, instances and general python syntax.

$$https://www.youtube.com/watch?v=N4mEzFDjqtA$$

Basic PythonPart structure

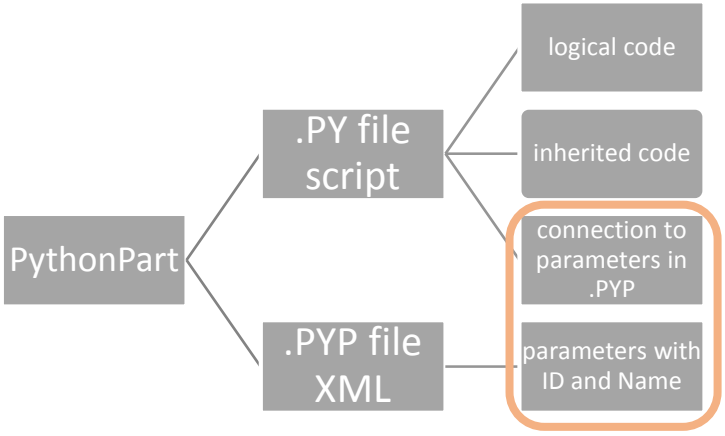

Figure 18: the basic guideline scheme every PythonPart needs to follow

A PythonPart mainly consists of two files, the PYP and the PY file. As discussed above, the PYP file contains parameters for menu creation and reference in XML format. The PYP file can also be used as an indirect modifier of the PythonPart through a basic XML editor. More complex changes and adaptations to Objects in Allplan (E.g. reacting to existing object in the Allplan object space) need to be done with the use of real Python Scripting. This is where the PYP file comes in.

Within the PY script file, a connection is laid with the parameters defined in the PYP parameter file. If this connection has not been executed properly, or some objects are not referenced to, nothing will happen upon executing the script in Allplan.
Furthermore, the PY script file "inherits" a few definitions. Upon dragging the file into Allplan, the program will start looking for these definitions and try to execute them. The output should ALWAYS be the same. There are three types of main definitions in Allplan script language. They will be discussed later on.

Defining inherited methods

Although, Python is unable to inherit certain properties from classes (in comparison to Java for example), inheritance in Python can be seen as the knowledge that the following three Python Definitions will always be accessed by the Allplan Interpreter. It is therefore important to define them properly.

DEFINITION NAME	USAGE ALLPLAN	ASSOCIATED PARAMETER	OUTPUT
CREATE_ELEMENT(...):	ELEMENT GEOMETRY	NONE	GEOMETRY
MOVE_HANDLE(...):	ELEMENT HANDLES	NONE	GEOMETRY, HANDLES
EXPAND_CREATE_ELEMENT(...):	EXPANDED GEOMETRY REACTS TO OBJECTS	GEOMETRYEXPAND (0/1)	BOOLEAN CHECK, POINT, GEOMETRY

Figure 19: different definitions that should be used in Python for Allplan

"Expand Create Element" makes use of the GeometryExpand parameter. If this parameter is set to zero, the definition does not need to be executed and defined.
A basic Python script structure for a beam would be like below.

```
(IMPORT STATEMENTS OMITTED)

DEF CREATE_ELEMENT(BUILD_ELE, DOC):

  ELEMENT = BEAM(DOC)  //REFERENCE TO THE BEAM CLASS
 ( ...)
  RETURN ELEMENT.CREATE(BUILD_ELE) //RETURNS GEOMETRY


DEF MOVE_HANDLE(BUILD_ELE, HANDLE, HANDLE_PROP, INPUT_PNT, DOC):

 (  ...)
  RETURN ELEMENT.CREATE(BUILD_ELE) //RETURNS GEOMETRY

DEF EXPAND_CREATE_ELEMENT(BUILD_ELE, EXPAND_UTIL, REF_PNT, VIEW_PROJ, DOC):

  ELEMENT = BEAM(DOC) //REFERENCE TO THE BEAM CLASS
 ( ...)
 RETURN (TRUE, MODEL_PNT, ELEMENT.CREATE(BUILD_ELE, BEAM_LENGTH, BEAM_HEIGHT))

CLASS BEAM() :
DEF __INIT__(SELF, DOC): //INITIALIZE THE BEAM > THE REFERENCED DOC IS ALWAYS PASSED THROUGH!
(...)
```

We can see that the following structure is maintained: Object data gets created in in the Object class (below), this data gets handled to Allplan through the basic definitions. If only defining a simple beam, *create_element* and *move_handle* are enough. When creating reinforcements that react to

the environment, expand_create element is necessary. (*create_element* is obsolete as one of the two gets selected because of the *expandGeometry* parameter)
When handling reinforcement, this should also be passed through the *create_element* method together with the other geometry. Allplan will handle the recognition of different elements.

Referring to parameters from the python script

Assigning parameters from within the Python script is fairly easy once you have created the correct parameters in the PYP file. An example will clarify this.
When I define a parameter with *<name>Beam_Width</name>* and <value>200</value> I can refer to this parameter in the Python script as follows:

```
CLASS BEAM() :

(…)

DEF CREATE_GEOMETRY(SELF, BUILD_ELE): //CREATED A DEFINITION FOR SIMPLIFICATION, NOT NECESSARY
    BEAM_WIDTH = BUILD_ELE.BEAMWIDTH.VALUE
    (…)

    COLUMN = ALLPLANGEO.POLYHEDRON3D.CREATECUBOID(BEAM_WIDTH, (…) )
    (…)
```

As you can see, the assignment is fairly easy. In this case, *Beam_width* in the Python script will be 200, and this value can, afterwards, be used to create the column with another function (in this case *createCuboid*).
All parameters, in order to use them properly, need to be referred to. The only parameters that are handled by Allplan are *ExpanderStart, ExpanderEnd* and *GeometryExpand.*

Import statements and python philosophy

Every script, as described above, should start with the import statements and the definitions that Allplan will use. In these import statements, it is of course possible to import your own scripts. In case you want to use an external interface to change some object properties, or you want to write an import application for Allplan, Python can be easily used to create a few python scripts that handle these. A major advantage of using Python is data conversion. It is now possible to create your own data converters. You handle the conversion, you control the data loss and conservation, and you are in charge. This is contrasting to how import was handled before in Allplan.

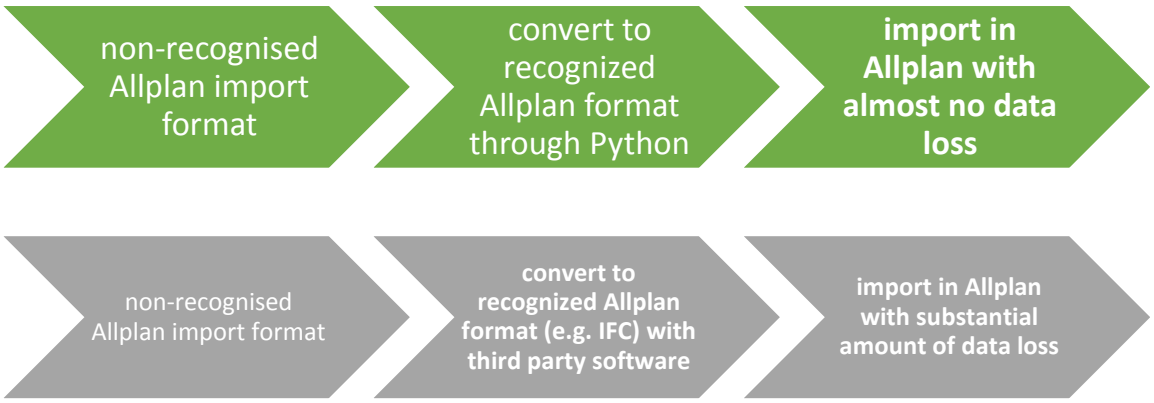

Figure 20: advantages of Python in the Allplan import Workflow

Simple column script example

We will discuss the creation of a simple column in Allplan. The column has a little reinforcement as an example and is adjustable.
In the Allplan PythonPart example scripts, a column1.py script can be found. This will be used as a first basic example. Besides the basic imports from the *Allpan NemAll* libraries, this simple class also uses predefined scripts for easier object handling. We will keep the description and explanation limited to the functions used in the column script.

TODO