Basic Introduction to Python for use with BIM

May 8, 2019

1 Introduction to Python 3 in jupyternotebook

This is an introduction to the Python programming language for participants on the Programming with openBIM course as arranged by BIMFag. It is based on learnpythons basic course.

It is not a complete Python course, so for deeper walkthrough of python and the different aspects of the language see more on learnpython or one can get free courses on eg. Udacity, EdX or CodeAcademy.

This tutorial will walk through these chapters:

in addition since this is a openBIM programming course, we will use BIM models as part of the introduction.

At the end of this notebook you will learn how to visualize objects in a BIM model. Like the windows of Grethes hus

2 Hello jupyter, world and you

The jupyter notebook lets us define and edit both text cells and markdown (text) cells. This is used here to introduce the concepts along the way, and also provide code cells where commands could be tested and executed.

We do it in cells, so eg. you could always create a new cell, edit or change.

2.1 Create a new cell, and write what you think about jupyter.

Use the "+" button above to create a new cell under this one.

```
In [25]: ### Change this cell from a Code cell to a Markdown cell, by using the roll down menu a
```

2.2 Let the program tell something to a user

The print statement is a powerfull directive in python and it enables a program to print out a result, eg. "Hello, world". Which prints a data type called a string to the screen.

```
In [20]: # Here we print the string "Hello, world!" to the screen.
         print("Hello, world!")
```

```
Hello, world!
```

We will use the print statement several times in this course together with "string formatting". A typpical way of formatting a string is by appending another string to it, like "Hello, "+"world".

```
In [21]: # Here we append the two strings "Hello, " and "world!" to the screen.
         print("Hello,"+"world!")
```

```
Hello,world!
```

As seen above now we didn't get a space between the two strings. A space could either be part of e.g. the first string, or be a separate string in itself like " ". Lets append that too.

```
In [22]: # Here we append the three strings "Hello,", " " and "world!"
         print("Hello,"+" "+"world!")
```

```
Hello, world!
```

2.3 Let the program ask a user for input and show the result

Since we want to say hello to you too, you could use the "input()" function to get user intput and store that in a variable, and print to screen.

```
In [24]: # Here we get the name from the user and store it in a variable called name, append tha
         x = input("Write your name here: ")
         y = "Hello, "
         print(y+x+"!")
```

```
Write your name here: Ingvild
Hello, Ingvild!
```

3 Variables and types

Python is completely object oriented, and not "statically typed". In other programming languages, that is statically typed, you need to declare the variable and what type of data it holds, before using it. In Python you do not need to declare variables before using them, or declare their type.

This is seen above where we just store the input into x and y that stores the string "Hello, " without first declaring them to hold data of string type first. Every variable in Python is an object, and we will go through some of the types here.

3.1 Numbers

For this course two different number types are relevant, integer and floating point numbers.

```
In [11]: # This is how you define an integer
         myInt = 9
         print(myInt)
```

```
9
```

```
In [13]: # To define a floating point number you could use one of the following approaches
         myFloat1 = 9.0
         print(myFloat1)
         myFloat2 = float(9)
         print(myFloat2)
```

```
9.0
9.0
```

One could aslo "cast" an integer into a float, like we do in the myFloat2=float(9).

- An integer x can be cast into float by using float(x)
- A float y can be cast into an integer by using int(y)
- An integer x and a float y can both be cast into a string by respectively str(x) and str(y).

```
In [3]: # Here we first define an integer, and cast it into a float
        myInt = 9
        myFloat = float(myInt)
        print("My Integer: %s, and my Float: %s"%(myInt,myFloat))
```

```
My Integer: 9, and my Float: 9.0
```

```
In [4]: # Here we cast myInt and myFloat (from above) into strings
        myIntegerString = str(myInt)
        print(myIntegerString)
        myFloatString= str(myFloat)
        print(myFloatString)
```

```
9
9.0
```

3.2 Strings

Strings can be define by using single (') or double quoates("). The difference between the two is that using double quotes makes it easy to include apostrophes (whereas these would terminate the string if using single quotes)

```
In [5]: myString = "Don't worry about apostrophes"
        print(myString)
```

```
Don't worry about apostrophes
```

3.3 Exercise 1

The target of this exercise is to create a string, an integer, and a floating point number. The string should be named mystring and should contain the word "hello". The floating point number should be named myfloat and should contain the number 10.0, and the integer should be named myint and should contain the number 20.

```
In [10]: # change this code
         mystring = "change this"
         myfloat = "change this"
         myint = "change this"

         # testing code
         if mystring == "hello":
             print("String: %s" % mystring)
         if isinstance(myfloat, float) and myfloat == 10.0:
             print("Float: %f" % myfloat)
         if isinstance(myint, int) and myint == 20:
             print("Integer: %d" % myint)
```

3.4 Exercise 2

Run the code below first without changing anything, supplying a number to it.
Then, use a cast to correct it so that it outputs correct math on numbers.

```
In [12]: # Here we take and input of a number, and then multiply it 5 times and then prints the
         inputVariable = input("Provide a number: ")
         x = inputVariable * 5
         print(x)
```

```
Provide a number: 6
66666
```

4 Basic String formatting

As seen in the testing code:

```
# testing code
if mystring == "hello":
    print("String: %s" % mystring)
if isinstance(myfloat, float) and myfloat == 10.0:
```

```
    print("Float: %f" % myfloat)
if isinstance(myint, int) and myint == 20:
    print("Integer: %d" % myint)
```

We use some string formatting in the print statements. We also use If statements and conditions which we are covering below.

Here we have used a special string formatting operator, %, that enables us to format the string according to the specified format. %s specifies that we are inputting a variable of type string into the sentence. %f specifies a placeholder for a floating point number and %d for a decimal integer.

In the above example we are adding one value to the string, but we could also add several in one string using the syntax below:

```
"string %s, number %d and floatingpoint %f"%("hello",10,10.0)
```

Feel free to try it out below.

```
In [56]: # Execute this to check multiple variables inserted into a string
         print("string %s, number %d and floatingpoint %f"%("hello",10,10.0))
```

```
string hello, number 10 and floatingpoint 10.000000
```

4.1 String functions

In addition to this there are several other string formatting options that you could do. Strings are objects that have several predefined functions. Feel free to add some code cells below here with some of the example code snippets from w3schools on strings

```
In [58]: # Example 1 -- eg. convert a string to lower letters
         a = "HElLo!"
         print(a.lower())
         # covert to captial letters
         print(a.upper())
```

```
hello!
HELLO!
```

```
In [59]: # Example 2 -- eg. split a string into two.
         a = "Hello, world!"
         #split at a specific character
         var = a.split(",")
         print(var)
         #split whitespace at beginning or end
         print(var[1].split())
```

```
['Hello', ' world!']
['world!']
```

4.2 Exercise 3

Build a script in the cell below that:

1) Takes in a string that you input
2) Make sure it is a string
3) Store it in a variable
4) Print out the variable
5) Then grab only the first and last characters in the name
6) Store it in a new variable
7) Print out the new variable

see w3schools on strings for help.

```
In [ ]: ### Exercise 3 answer here:
```

5 Collection types (Arrays)

There are four collection data types in the Python programming language:

- List is a collection which is ordered and changeable. Allows duplicate members.
- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- Set is a collection which is unordered and unindexed. No duplicate members.
- Dictionary is a collection which is unordered, changeable and indexed. No duplicate members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security (ref w3schools)

Below we will walk through Lists and Tuples. Find more info on sets and dictionaries on eg. w3scools above.

5.1 List

A list is a collection which is ordered and changeable. They can contain any type of variable, and they can contain as many variables as you wish. In Python lists are written with square brackets.

```
In [13]: myList = ["apple", 1, "cherry", 10]
         print(myList)
```

```
['apple', 1, 'cherry', 10]
```

You access the list items by referring to the index number. Remember that the index starts at 0.

```
In [14]: myList = [1,2,3,4,5]
         print(myList[1])
```

```
2
```

5.1.1 Lists are changable

Lists are changable, so you could eg. add or remove from it.

```
In [27]: # Adding to a list my append()
         myList = [1,2,3]
         myList.append(3) # adds the value 3 to the list
         print(myList)

         # Removing from a list by remove()
         myList.remove(2) # removes the value 2 from the list
         print(myList)

         # Removing from a list by pop()
         myList.pop() # removes the last item in the list
         print(myList)
         myList.pop(0) # removes the first item in the list
         print(myList)
```

```
[1, 2, 3, 3]
[1, 3, 3]
[1, 3]
[3]
```

5.2 Tuple

A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets. When working with IFC and the ifcopenshell, we often work with tuples. Tuples are written with brackets.

```
In [28]: myTuple = ("apple", 1, "cherry", 10)
         print(myTuple)
```

```
('apple', 1, 'cherry', 10)
```

You access the tuple items, similar as for lists, by referring to the index number. Remember that the index starts at 0.

```
In [29]: myTuple = (1,2,3,4,5)
         print(myTuple[1])
```

```
2
```

5.2.1 Tuples are unchangable

Tuples are unchangable, but you could cast it into a list and work with it as a list.

```
In [31]: myTuple = (1,2,3,4,5)
         myList = list(myTuple) # Cast tuple into a list
         myList.pop(2)
         myTuple = tuple(myList) # Cast list into a tuple
         print(myTuple)
```

```
(1, 2, 4, 5)
```

5.3 Check the number of elements in list or tuple

It is often needed to check how many elements that are in a list or a tuple.

```
In [34]: # Check how many elements there are in myList (defined above)
         print(len(myList))

         # Check how many elements there are in myTuple (defined above)
         print(len(myTuple))
```

```
3
3
```

5.4 Exercise 4 - how many IfcWall elements are in Grethes Hus model?

In the excercise below we want you to use what you have learned above to find how many elements of type IfcWall are in the Grethes Hus Modell.

We have provided some code to get the file and to query out all elements of type IfcWall and stored it in a variable called *walls*.

Print out to the screen the number of walls there are in the variable **walls**.

```
In [1]: # We import ifcopenshell and open the Grethes Hus Model and store it in a variable calle
        import ifcopenshell
        file = ifcopenshell.open("../models/Grethes_hus_bok_2.ifc")

        #We query the file for its number of IfcWall type elements
        walls = file.by_type("IfcWall")

        """ToDo: Change the zero below to list out the number of walls in the file."""
        numberOfWalls = 0 # store the number of IfcWall types in this variable.
        print(numberOfWalls)
```

```
0
```

6 Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

These conditions can be used in several ways, most commonly in "if statements" and loops. (ref. w3schools)

We'll walk through use of if statements and the most common loops here and use it to explore some elements from the Grethes Hus Model.

The syntax for a If statement that checks if a variable **a** Equals a variable **b** is as follows:

```
a = 0
b = 2
if a == b:
    print("they are eaqual")
```

Notice the **indentation**. Python relies on indentation, using whitespace, to define scope in the code. Other programming languages often use curly-brackets for this purpose. Here the print is within the if statement.

Lets use an if statement to check if there are more than zero walls in the Grethes Hus Model.

```
In [47]: ## Here we refer back to the variable "numberOfWalls" as defined above, and check if it
         if numberOfWalls > 0:
             print("You probably did something right above.")
```

6.1 elif and else

We could also add to the If statement, by using an **Elif** statement. This will allow you to add another condition to check. At the end, we could also do an **Else** statement. This will be executed if no of the previous conditions where met. The syntax for this is:

```
if <condition1>:
    # do something based on condition1
elif <condition2>:
    # Do something based on condition2
else:
    # Do something if non of the previouse contitions where met.
```

You are free to at as many elif statements as you wish, but you could only have one else statement.

```
In [48]: if numberOfWalls > 0:
             print("You probably did something right above.")
         elif numberOfWalls == 0:
             print("You did probably do something wrong above")
         else:
             print("How could this happen...?")
```

```
You did probably do something wrong above
```

6.2 Check if an element exist in a list or tuple

Often times one would like to check if a particular value is in a list or a tuple. This is easy with python.

```
In [33]: # Check if a value is in a list
         myList = ["BIMFag", "banana", "BIM"]
         if "BIMFag" in myList:
             print("I found BIMFag in myList!")
         myTuple = ["BIMFag", "banana", "BIM"]
         if "BIMFag" in myTuple:
             print("I found BIMFag in myTuple!")
```

```
I found BIMFag in myList!
I found BIMFag in myTuple!
```

7 While Loops

Conditions and if statements are usefull also in while loops. While loops are executing as long as a condition is met. The syntax for while loops are:

```
x = 1
while x<10: # Checks if x is less than 10
    # do something eg.
    print(x) # since x is bigger than zero this will print for as long as x is less than 10
    x = x + 1 ## this statement is adding 1 to x for each iteration of the loop.
```

Keep in mind that if the condition on which the while loop is checking on is never met, it will execute indefinately. so whitout the `Python x= x+1` statement above, this would have just printed out 1 indefinately.

Notice the **indentation**. As mentioned above Python relies on indentation, using whitespace, to define what is within the while loop. Here the `print(x)` and `x=x+1`statement is indented, so to show that it is inside the while loop.

```
In [54]: # Lets see it in action. Print out x as long as its less than the number of walls
         x = 1
         while x < numberOfWalls:
             print(x)
             x = x +1
         else:
             print("x is: %s and numberOfWalls is: %s"%(x,numberOfWalls))
```

```
x is: 1 and numberOfWalls is: 0
```

7.1 Loops, break and else

Notice that the loop exits when the condition is no longer met. Notice also that loops could also have an else statement, similar to if statements. If the loop condition are no longer met, then the else part is executed.

If we want to exit the while loop early one could use the special **break** statement. The while loop could have a condition that always will be true, like `True`but then have a check further in, and use `break`to exit the loop based on a condition in the condition of the respective check.

```
In [50]: x = 1
         ## Loop while True --> True is always True...
         while True:
             print(x)
             # check if x is bigger or eaqual to 10
             if x >= 10:
                 break # if condition met, break out of the while loop.
             x = x+1 # if the condition of the if statement was false, this get executed in the
```

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

8 For Loops

A *for* loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

The syntax for for loops in python is:

```
for element in listOfElements:
    # Do something with each element eg. print it out.
    print(element)
```

Notice the **indentation** here as well. It define what is within the for loop. Here the print statement is indented, so that will be done every iteration of the elements in the list.

```
In [42]: myTuple = (1,2,3,4,5,6)

         for element in myTuple:
             print(element)
```

```
1
2
3
4
5
6
```

8.1 Iterating over a number of IfcWindow objects and access attributes

Below we'll use a for loop to loop through all IfcWindow elements and print out some of it's direct attributes. See the relevant attributes for IfcWindow (ifc2x3 version) here

We'll be getting its **Name** attribute. Since the IfcWindow elements are objects in Python we could access these using the "." operator.

Feel free to experiment with other attributes. You could eg. try with **OverallHeight and OverallWidth**.

```
In [2]: # We import ifcopenshell and open the Grethes Hus Model and store it in a variable calle
        import ifcopenshell
        file = ifcopenshell.open("../models/Grethes_hus_bok_2.ifc")

        #We query the file for its number of IfcWindow type elements
        windows = file.by_type("IfcWindow")

        """By uncommenting (removing the '#') the line below you see the contents of the windows
        #print(windows)

        # Here is the for loop and its syntax. Notice the indentation.
        for window in windows:
            print("The window elements name is: "+window.Name)
```

```
The window elements name is: M_Fixed:_1400x2200mm:348960
The window elements name is: M_Fixed:_1400x2200mm:350239
The window elements name is: M_Fixed:_1400x2200mm:350479
The window elements name is: M_Fixed:_1400x2200mm:350484
The window elements name is: M_Fixed:_1400x2200mm:350489
The window elements name is: M_Fixed:_1400x2200mm:350494
The window elements name is: M_Fixed:_1400x2200mm:350499
The window elements name is: M_Fixed:_1400x2200mm:350504
The window elements name is: M_Fixed:_1400x2200mm:350509
The window elements name is: M_Fixed:_1400x2200mm:350514
The window elements name is: M_Fixed:_1400x2200mm:350519
The window elements name is: M_Fixed:_1400x2200mm:350524
The window elements name is: M_Fixed:_800x750mm:351890
The window elements name is: M_Fixed:_1100x700mm:353017
The window elements name is: M_Fixed:_800x700mm:354254
The window elements name is: M_Fixed:_800x750mm:355941
The window elements name is: M_Fixed:_1100x700mm:355942
```

```
The window elements name is: M_Fixed:_800x700mm:355943
The window elements name is: M_Fixed:_800x1000mm:356683
The window elements name is: M_Fixed:_800x1000mm:357125
The window elements name is: M_Fixed:_1400x1000mm:357137
The window elements name is: M_Fixed:_800x2000mm:357354
The window elements name is: M_Window-Casement-Double:_1400x1000mm:385488
The window elements name is: M_Window-Casement-Double:_1400x1000mm:385889
The window elements name is: M_Window-Casement-Double:_1400x1000mm:385919
The window elements name is: M_Window-Casement-Double:_1400x1000mm:385922
```

9 Functions

Functions are a useful way to divide your code into blocks of code, that only runs when they are called. It is also a good way to share a small snippet of code with others.

A function is defined by the special defstatement. Like the example below.

9.1 Defining a function

```
In [61]: def myDefinedFunction():
             print("This is a print statement within my function")
```

Notice that the function uses the same indendation style as for If statements and loops as walked through above. A function could contain all the concepts we have walked through above as well.

Notice however that noting happended above. That is why we havent called the function. It is however stored, so that we can call it below. ## Calling / Using a function

```
In [62]: myDefinedFunction()
```

```
This is a print statement within my function
```

9.2 Input variables to functions

We could also build functions that takes in input variables that could be used in the function. Lets define a function that takes in a string variable and prints it out.

```
In [3]: # A function definition that takes in a parameter called "string"
        def myDefinedFunction2(string):
            print("My variable is %s"%string)

        # A call to myDefinedFunction2 with the string variable "Hello again!"
        myDefinedFunction2("Hello again!")
```

```
My variable is Hello again!
```

```
In [4]: # A new call to myDefinedFunction2 with the string "and again and again"
        myDefinedFunction2("and again and again")
```

```
My variable is and again and again
```

9.3 Input variables and assumed type

What would happen if we call myDefinedFunction2 with a number, instead of a string? Try it below.

```
In [ ]: string = 10
        """Try calling myDefinedFunction2 with the variable define above"""
```

9.4 Exercise 5 - ensure that a function works as intended

You can pass as many variables to a function as you'd like, just separate them with a comma like we do with var1 and var 2 below. They could be whatever object you'd like.

Below we want to create a function that takes in two variables, var1 and var2. If they are both strings we want to just concatenate them together and print them out. If they are not strings we only want to add them to a list and print

A couple of hints

- you could check if a variable `a` is a string by using `isinstance(a,str)`
- If you want two conditions to both be True in order to pass, you could use the `and`operator

The correct output:

```
Hello, World!
['hello', 4]
```

```
In [69]: def myFunc3(var1,var2):
             """Fill in code to comlete the challenge"""

         """Don't change the code below"""
         myFunc3("Hello, ","World!")
         myFunc3("hello",4)
```

```
Hello, World!
['hello', 4]
```

9.5 Lists and tuples as input variables

We could also add lists and tuples as input variables to functions.

```
In [6]: def myFunc4(myList):
            for elem in myList:
                print(elem)
```

```
In [72]: # Define a list
         aList = [1,2,3]
         # Send it as input variable to myFunc4
         myFunc4(aList)
         # Define a tuple
         aTuple = (4,5,6)
         # Send it as input variable to myFunc4
         myFunc4(aTuple)
```

```
1
2
3
4
5
6
```

9.5.1 What will happend if you pass a string instead of a list?

Try below to call the function above passing in a string value like eg. `myFunc4("your name")`
What do you think will happen?

```
In [9]: ### Do the test here ###
```

9.6 Have functions return values

A good way to use functions is to have them perform som logic on input variables and have them return the result. For this you use the `return` statement.

```
In [15]: # defines a function that takes in a list of items as parameter,
         # loops through it and adds all integers together, and then returns the resulting value
         def myFunc5(myList):
             tmp = 0 # Defined outside the for loop to be able to return it outside the scope of
             for elem in myList:
                 if isinstance(elem,int):
                     tmp = tmp +elem
             return tmp
         aList = ["string",10,10,30,4, "stuff"]
         print(myFunc5(aList))
```

```
54
```

9.7 Exercise 6 - Names and Descriptions of Window objects.

Many times one would get an ifc model where one want to change or restructure information in the file. That is tediouse work on big models, so one may want to script it. In addition it would be a good code block to have, regardless of object type.

In IFC the name and description is something all objects *inherrit* from IfcRoot, and cosequently they all have the following attributes (ref. Ifc documentation):

- GlobalId : Assignment of a globally unique identifier within the entire software world.
- OwnerHistory : Assignment of the information about the current ownership of that object, including owning actor, >application, local identification and information captured about the recent changes of the object, NOTE: only the >last modification in stored.
- Name : Optional name for use by the participating software systems or users. For some subtypes of IfcRoot the >insertion of the Name attribute may be required. This would be enforced by a where rule.
- Description : Optional description, provided for exchanging informative comments.

So, it might be a good idea to package code into functions. That way we could share it and reuse the code regardless of which element we want to change the name and description on.

The first challenge is to create a function that takes in a IfcWindow (or another Ifc object), get the Name and store that in the Description property of the object. Then, add an input variable that *if set* will be given as a Name to the window or it will store a blank string as the Name.

The second challenge is to write code to print out the names and descriptions (before and after) to confirm your result.

** Some hints:** * Python functions could have input variables that have default variables eg. def func (var1,var2 =""):. * Look above to find code that enables you to import the ifcopenshell code library, that have code that enables you to work with teh IFC file. * Look above, or eg. in academy.ifcopenshell.org, to find code in the ifcopenshell library that could store a reference to an IFC model in a variable and another function that lets you query that file for all IfcWindow objects you might want to find in that model. * You can set the `Name`variable of a window object that is store in a variable called "window" by “‘window.Name = "the name you want to set"“”

```
In [17]: ### Add your code here ###
```

10 Classes and Objects

We have already worked alot with objects, some window objects, some string objects, and even some list objects. So, what is objects and where do they come from? Objects are an encapsulation of variables and functions into a single entity. Objects get their variables and functions from classes. Classes are essentially a template to create your objects.

When functions are code blocks that could be shared and reused, classes and objects are even bigger blocks of reusable code. For the IfcWindow example above, it has a class that defines what the variables and functions the window objects of that class has. The documentation of what capabilities the IfcWindow class should have is defined by the Ifc documentation for IfcWindoweg. forIfcWindow version Ifc2x3. You could also have documentation of how that class is defined/implemented in the particular programming language.

A very basic class would look something like this:

```
class MyWindowClass:
    nameVariable = "Window"

    def namePrint(self):
        print("My Name is "+self.nameVariable)
```

10.1 Defining a Class

Lets define a Window Class

```
In [29]: ## Defines a Window Class
         class MyWindowClass:
             nameVariable = "Window"

             def namePrint(self):
                 print("My Name is "+self.nameVariable)
```

10.2 Creating objects from class definitions

When you have defined a class, you could create several objects based on it.

```
In [30]: ## Defines two different objects based on the MyWindowClass
         window1 = MyWindowClass()
         window2 = MyWindowClass()
```

10.3 Accessing Object variables and functions

Now the window1 and window2 objects have both a nameVariable and namePrint function each. As we have seen above, we access its variables (attributes) by the "." operator.

```
In [32]: # Get the nameVariable of window1
         window1.nameVariable
```

```
Out[32]: 'Window'
```

10.3.1 MiniExercise: Do the same for window2 below

```
In [ ]: ### Get the the name variable of window2
```

10.4 Exercise 7 - Setting object variables

Both windows have the same name. Use what you know from above to set the name of window1 and window2 to something proper.

```
In [ ]: ### Solve Exercise 7 here:
```

10.5 Exercise 8 - Accessing object functions

In the MyWindowClass definition we have a function to print out the name. Show below how this function can be accessed on both window1 and window2.

```
In [ ]: ### Solve Exercise 8 here:
```

11 Modules and Packages

In programming, a module is a piece of software that has a specific functionality. For example, when building a ping pong game, one module could be responsible for the game logic, and another module could be responsible for drawing the game on the screen. Each module is a different file, which can be edited separately.

Modules in Python are simply Python files with a .py extension. The name of the module will be the name of the file. A Python module can have a set of functions, classes or variables defined and implemented.

You might have seen the `ifc_viewer.py` file in the folders? That is a module, and particularly a module that enables viewing of models in jupyter notebooks. So, how do one use that?

11.1 Using Modules in other programs

Modules could be used in other programs by using the special `import`statement. Eg. to import all of it use `import ifc_viewer`would do the trick.

However, the module may define several classes, variables and functions. What if we only wanted to import the `ifc_viewer` class in the `ifc_viewer`module? Then we do like bewlow.

```
In [36]: # Import the ifc_viewer class of ifc_viewer module
```

11.2 Using Packages and Modules in another program

Another package we already have used is the **ifcopenshell**. Packages are namespaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.

Each package in Python is a directory which MUST contain a special file called **init**.py. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported (ref. learnpython.org).

We have already imported ifcopenshell. But we could also import specific parts of it. Lets combine it to visualize the windows of the Grethes Hus model.

```
In [35]: import ifcopenshell
         import ifcopenshell.geom
         from ifc_viewer import ifc_viewer
         # Storing the model in a file variable, and giving the path to the file as input.
         file = ifcopenshell.open("../models/Grethes_hus_bok_2.ifc")

         # Storing all windows of the file by ising the by_type function of the file class
         windows = file.by_type("IfcWindow")

         # Setting the geometry settings.
         s = ifcopenshell.geom.settings()
         s.set(s.USE_PYTHON_OPENCASCADE, True)

         # Instansiationg a viewer object from the ifc_viewer class
         viewer = ifc_viewer()

         # Running through all window elements and create a shape and add it to the viewer for d
         for window in windows:
```

```
            shape = ifcopenshell.geom.create_shape(s, window)
            viewer.DisplayShape(window, shape.geometry, shape.styles)

        viewer.Display()
```

```
Renderer(background='white', camera=PerspectiveCamera(children=(DirectionalLight(intensity=0.5,
```

```
HTML(value='No element selected')
```