Tutorial: creating a Grasshopper component with the Python GHPY compiler

python, ghpy, compiler, gha

Giulio Piacentino piac McNeel Nov '16

GhPython in Rhino WIP ships with an all-Rhino Python compiler that can be used to create .Net compiled **assemblies containing compiled instructions**. Unlike other methods that are based on embedding a Python code string in a C# component, this speeds up code execution and makes reverse engineering more difficult.

Nov 2016
1 / 60
Nov 2016
Aug 2019

There are two options for the process of making a component, **#1** is extremely simplified and **#2** is more advanced. The component is able to pick up names, descriptions and help from the original Grasshopper component, and turn it into an executable *dll* file.

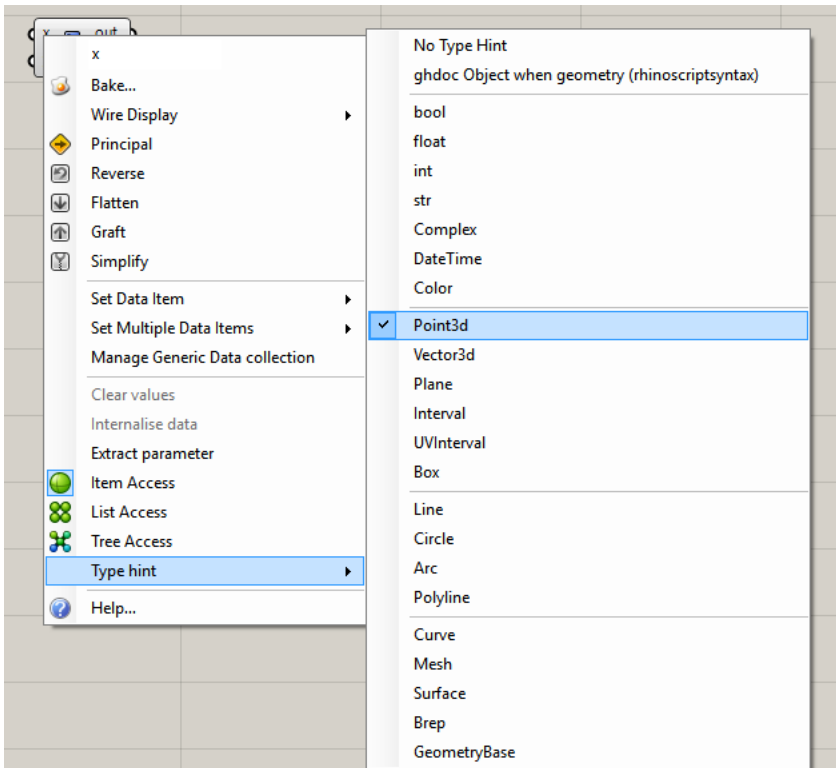
To perform this operation, the compiler makes heavy use of the facilities provided by *clr.CompileModules()*, which is a IronPython module devoted to creation of compiled assemblies, and is also what *pyc.py* in *IronPython* uses to produce standalone executable files.

Note: this is for Rhino WIP **#6.0.16313.01411** and greater!

1. Compiling a *single* component in an assembly with the wizard

We will start by compiling a single component with the all-automated wizard.

1.1. For this sample, we will start by placing a GhPython component on the canvas, and by setting the first hint to Point3d. The compiler will write code that is dependent on **type hints**.



1.1.1. Right-click the "x" input to show this context menu.

1.2. Then we can **remove an input** for this sample. We do not need it. The compiler will write code that is dependent on the **amount of inputs** and outputs. We also remove the **“out” output**.

1.2.1. You need to zoom in to see the "-" (minus) control.

1.3. Then we can rename the input to “P” for this sample. This will have an influence on the way we write code. We and the compiler will have to write code that is dependent on the **naming of inputs** and outputs.

1.3.1. Right-click to see this context menu.

1.4. Next, we will write the code that forms our logic for this component. We can provide the user with a good amount of information by placing **docstrings** at the beginning of our module, and then we can write code that references our variables. Here is the full text written in the image below.

```
"""Draws a graphic symbol parallel to world XYZ showing orientations.
    Inputs:
        P: The point where the symbol should be drawn
    Output:
        X: The symbol, drawn as three lines"""

__author__ = "piac"

import rhinoscriptsyntax as rs
from Rhino.Geometry import Point3d as p3

if P:
    X = []

    line_EW = rs.AddLine(P - p3(10,0,0), P + p3(10,0,0))
    X.append(line_EW)

    line_NS =rs.AddLine(P - p3(0,10,0), P + p3(0,10,0))
    X.append(line_NS)

    line_UD =rs.AddLine(P - p3(0,0,10), P + p3(0,0,10))
    X.append(line_UD)
```

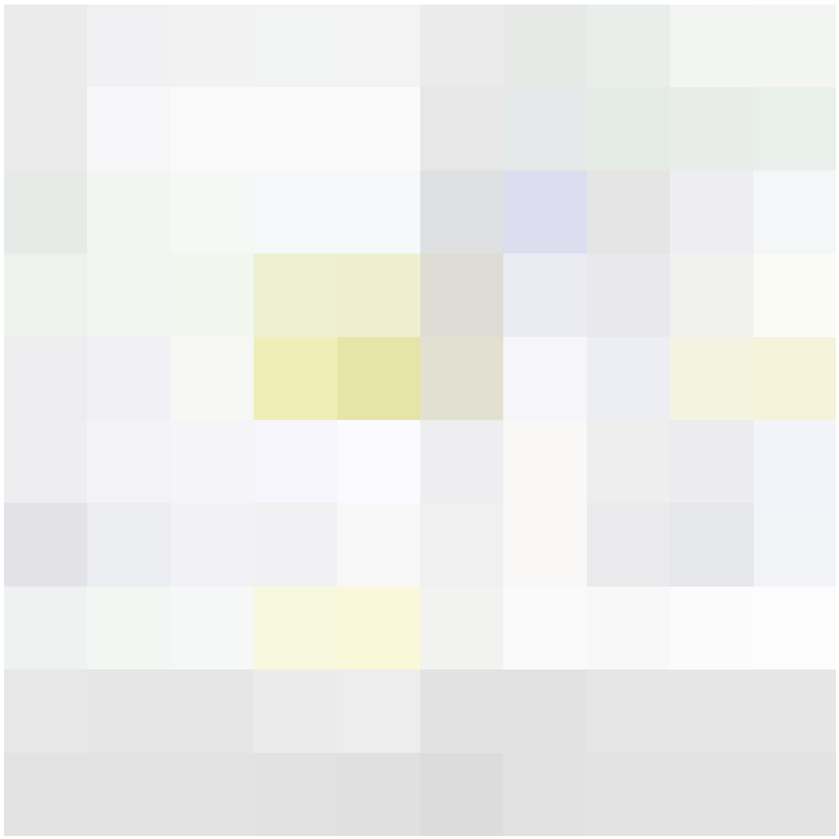
You can also download the readily-made file. **compiling.gh** (2.7 KB)

1.4.1. Right-click to see this context menu.

1.5. The code works in procedural mode. We switch to SDK mode.

1.5.1. Click the jigsaw puzzle icon and switch to GH_Component SDK Mode.

1.6. A few adjustments are necessary to make our code robust. The automatic conversion, though, will work out-of-the-box most of the times or will give a good idea about a starting point. Remember to declare more component _def_s, rather than nesting all your functions inside the RunScript function. This will improve runtime behavior.

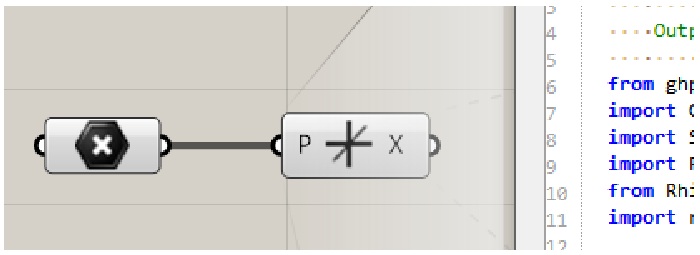


1.6.1. Perform these changes in the editor.

1.7. We draw this icon in an image processor.

We should draw a 24x24 pixel icon and save it, as best practice, as PNG. We use a transparent background and some good antialiasing if possible.

1.8. We apply the image. Drag-and-drop of the image on top of the component will work.



```
    ····Outp
    ·······
    from ghp
    import (
    import S
    import R
    from Rhi
    import r
```

1.8.1. This is the appearance of the component after applying the image.

1.9. Compile…
If the option is grayed out, you need to remove the “out” output. Printing should not have effects in compiled components.

1.9.1. Let the "magic" begin.

1.10. We still need to set a few details. This form should pop-up. Particularly important is **the Guid**. If we use the same Guid from an old version of our add-on, we will replace its behavior once the file is swapped and the new assembly is loaded. However, we need to delete the old version of the add-on first. Grasshopper uses Guids to uniquely identify component types.

1.10.1. We need to write down or select the Category, the Subcategory, give the component a Name and a Nickname.

1.11. You will be prompted a **save location**. The default is also the automatically-watched folder of Grasshopper. Saving here will cause the assembly to be immediately loaded.

1.12. If we did not modify the defaults, the assembly will be placed in the Libraries folder, and automatically loaded into the Grasshopper ribbon. We created our first GHPY file!

A few notes **on this name**:

- Axes is the name of the assembly
- There is an underscore, and then the assembly ID follows
  You can remove the assembly ID, but this is the full name of the module that IronPython will load: `Axes_7af034972e964ca4b11ac1b9209267f7`. This long name ensures that no other add-on or module mistakenly uses your same name. You can rename this part to anything that pleases you. However, this ID will always be the ID by which Grasshopper identifies your add-on. You should write it down or you can discover it again later.

1.12.1. This is the usual Grasshopper Libraries folder, with our file.

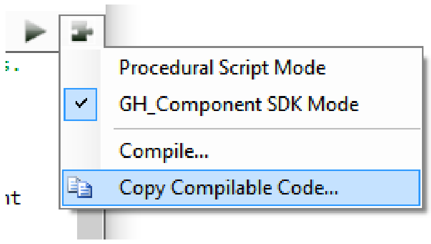
Ready downloads for part #1

- compiling.gh (2.7 KB) Component in procedural mode.
- compiling-sdk.gh (2.7 KB) Component in Grasshopper SDK mode.

2. Advanced compiling: *multiple* components, projects with modules, updating

Sometimes, there is a need of a setup that is more advanced than one component, as in #1 on this page.
We will now cover the creation of an update to the assembly above, making sure that it loads the previous component, and we will add a new “battery” as well, that uses functionality from another private module. This should cover a larger ground of user cases.

2.1. Using the code from the **component above**, we choose the other option, copy compilable code.



2.1.1.

2.2. We get the same panel as in **1.10**, but this time it will copy the result to the clipboard.

2.2.1.

2.3. We get the same panel as in **1.10**, but this time it will copy the result to the clipboard.

2.3.1.

2.4. We paste the result from the clipboard into a new _EditPythonScript form. We can remove docstrings, because they are no longer needed. Their content was copied by the wizard into the Grasshopper SDK instantiation code. We **save this file** as `Axes.py`, in a new, empty folder.

2.4.1. The entire code divided into areas.

2.5. We also **create two new** Python files: `Main.py` and `Axes_helper.py`.

The content of `main.py` is then:

```
import clr

clr.CompileModules("Axes.p2.7.5.0.ghpy", "Axes.py", "Axes_helper.py")
```

2.6. The content of `Axes_helper.py` is then:

```
import rhinoscriptsyntax as rs

class SphereArtist(object):

    def __init__(self, radius):
        self.radius = radius


    def Draw(self, location):
        """Retuns new spheres"""
        return rs.AddSphere(location, self.radius)
```

2.7. We then follow the instructions from 2.1 to 2.3, with this code:

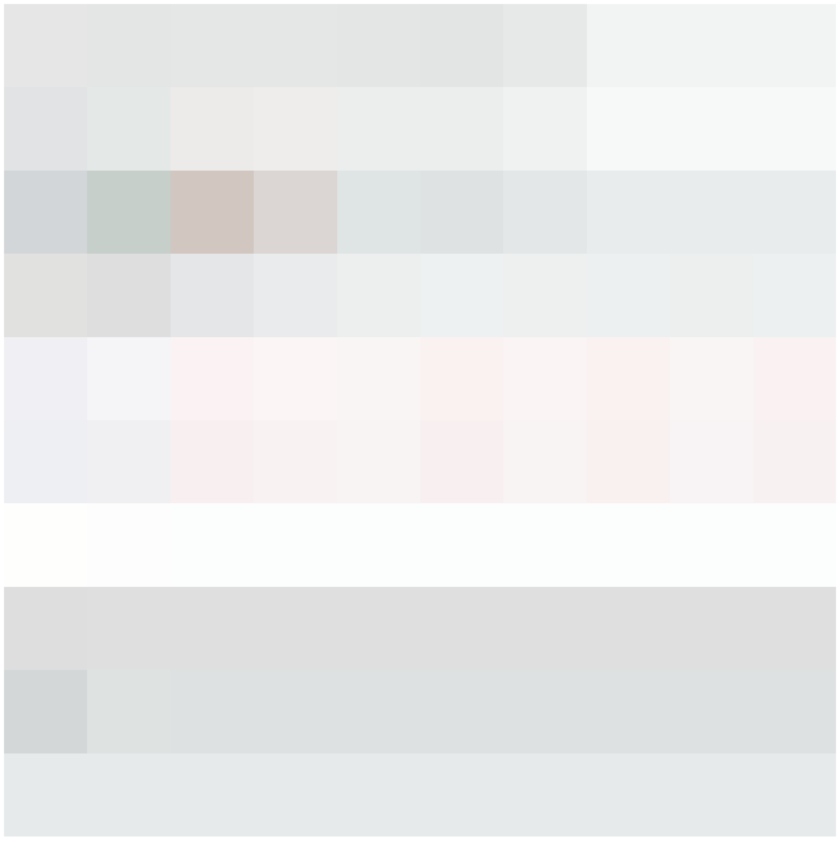
```
import Axes_helper

class MyComponent(component):
    def __init__(self):

        self.artist = Axes_helper.SphereArtist(5)


    def RunScript(self, P):

        return self.artist.Draw(P)
```

It will again create a large class with more boilerplate code: the wizard takes care of every need of the Grasshopper SDK. We then add the result to `axes.py`.

_

2.8. In total, **we have two components, and one assembly information class**. They need to have different names, otherwise, the Python interpreter will override one declaration with the other. All component declarations need to be in the file which is the second argument to the

###This is the folder to be compiled at this point:
compiling_display.zip (7.9 KB)



2.8.1. Compile by running this.

We run the `main.py` file, and we get a new `axes.p2.7.5.0.ghpy` file. This contains our entire code, in compiled instructions. HURRAY!

2.8.2. Our compiled file.

–

2.9. To have Grasshopper load this, we can copy this file to the Grasshopper Components Folder. Open Grasshopper, then go to **File -> Special Folders -> Components Folder**. Move the file there. Immediately, Grasshopper will load this file and we will be ready to use our new components.

2.9.1. Done!

Axes.p2.7.5.0.ghpy (34 KB)

Some few last words of caution:

- Remember to always save your source code in the original form! It is really difficult to get it back once compiled.
- This is new functionality in Rhino WIP. It can change and you might need to re-compile.

Happy coding!

Compliling Components using GHPython and PyDev

Add ghPython inputs dynamically from script?

Compiling Python scripts into a plugin

Question on different compilers

Documentation strings not used in GH_Component SDK mode

24 more

Mostapha **Nov '16**

@piac , thank you for the post. Is there a solution to automate the whole process for several component or python files? I'm still not clear about the strcuture of the files to compile several files. For instance, I have 51 files for butterfly . What would be the best approach?

Giulio Piacentino piac McNeel **Nov '16**

Hi @Mostapha !

First of all, this method does not replace any previous working method, so you can keep on using importing from folders, having all code in a single component, etc, for as long as you want.

You can compile as many files as you want into the library. There can be a Python script that forms the list for you, to then feed into CompileModules(). That is all you should need. Does it help?

Mostapha **Nov '16**

Hi @piac , It does. I think my question is how to put the script together and if I can access the functionalities that you showed within the script. Currently we have a component for ladybug that automates the process of creating userobjects from a GHPython component from inside Grasshopper without having to deal with any form, and I was thinking to devlope a similar component to compile them. Can you share a code snippet on how to get this started?

Giulio Piacentino piac McNeel **Nov '16**

OK; I see where you are aiming now. Let me have a look and (try to) write something useful.

Paul Poinet **Nov '16**

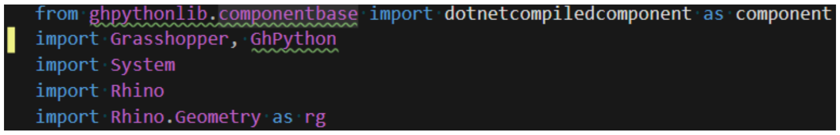
Hi @piac , thanks a lot for the post! it is really useful.

Just wondering, do you think if it is possible to compile directly within Visual Studio?

I think I am missing the GhPython and ghpythonlib assemblies. Since it is shipped directly within Grasshopper in WIP, I am not really sure where to find those. Any ideas?

Best

Paul



```
from ghpythonlib.componentbase import dotnetcompiledcomponent as component
import Grasshopper, GhPython
import System
import Rhino
import Rhino.Geometry as rg
```

Willem Derks Willem **Nov '16**

Hi Giulio,

I'm going off on a tangent now, but can you elaborate on the possibilities to compile standalone IronPython scripts this way.
For a while now I've been pondering how I can gain speed and get some form of compiling to protect the code from prying eyes (at least get some level of hiding the source), without delving into C# or the like.

Am I correct to assume that there might be a way to apply this type of compiling to rhinopython scripts as well?
Maybe copile modules as dll's?

Thanks
-Willem

🔗 What class/method does Grasshopper use when picking objects from rhino as command opti…

Giulio Piacentino **piac** McNeel **Nov '16**

Hi @PaulPoinet

> PaulPoinet:
>
> *do you think if it is possible to compile directly within Visual Studio?*

If you have exactly the same version of IronPython as the one that ships with Rhino, I think it should be possible.

> PaulPoinet:
>
> *GhPython and ghpythonlib assemblies. […] I am not really sure where to find those*

- **GhPython** is the Grasshopper .gha file, a .Net assembly. It is located at

```
%programfiles%\Rhino WIP\Plug-ins\Grasshopper\Components\
```

- **ghpythonlib** is a Python package (a directory containing modules) that is located right beside the rhinoscriptsyntax. That is,

```
%appdata%\McNeel\Rhinoceros\6.0\Plug-ins\IronPython (814d908a-e25c-493d-97e9-
ee3861957f49)\settings\lib
```

Giulio Piacentino piac McNeel Nov '16

Hi Willem

> Willem:
>
> *can you elaborate on the possibilities to compile standalone IronPython scripts this way.*

As it was stated earlier, it is possible to use clr.CompileModules() to compile any module. You will have to solve a series of technical hurdles that the GHPY solution provides you with, but it should be possible. The biggest issues are locating the .dll, and providing a way to differentiate assemblies for different IronPython language versions.

If you are considering _EditPythonScript scripts for this, there *is* the RhinoScript "compiler". It is listed here . It takes all your code and puts it into a giant string, that will be later run by Rhino. This approach offers less protection and there is no need to "recompile" for different IronPython languages.

There isn't a system that works like this one, for standalone scripts, AFAIK. Sorry.

Paul Poinet Nov '16

Thanks for the locations. I should have dig a bit more…
It works well from VS 🙂

Miriam Nov '16

Yay 🙂 all working! Many thanks for setting this up for us Giulio!

Just as a matter of feedback. The workflow does it all for us, and since this is so accessible now, it would be nice to comment what the classes that assemble and the functionality that defines a component do. There may be some people that never compiled in C# or VB before that would like to understand what those are doing.

I will try to call PyDev modules now.

Yay again 🙂

Miriam Nov '16

> Compliling Components using GHPython and PyDev
>
> *PyDev for compilation: Tick! [image] Thank you!*

Giulio Piacentino piac McNeel Jan '17

Hi @Mostapha

I made changes to the compiling SDK and the last set of addition (ability to call from code the same compiler tool that is used in the wizard) is in the Rhino WIP of last week – 6.0.17010.13211.

I am attaching here a definition that shows how that works.
Please let me know if this is useful.

compiling-sdk.gh (14.0 KB)

Issues building ghpython

Mostapha **Jan '17**

Hi @piac , This is very helpful. Thanks!

Matthew Breau **Jan '17**

A word of advice, in case anyone else is having the problem I had earlier…

I was getting a trace error, something along the lines of:
Void Microsoft.Scripting.Utils.ExceptionUtils.RemoveData(System.Exception, System.Object)

Installing the latest IronPython fixed this issue

Giulio Piacentino **piac** McNeel **Jan '17**

Hi @Matthew_Breau , in general there should be no need to install IronPython separately. You were doing that for other reasons, right?

Matthew Breau **Jan '17**

I found that trying to switch from procedural mode to the GH Component SDK Mode would cause the error I mentioned above. Installing the latest IronPython eliminated the error.

Giulio Piacentino **piac** McNeel **Jan '17**

There are general problems when using independent IronPython installs. They should be avoided, or the exactly matching version should be installed, when possible. This is a Rhino-wide limitation, not a specific one of Grasshopper, GhPython or the EditPythonScript editor.

Fabio_S **Feb '17**

Hi Giulio thank for this post. I have an issue after compiling. It display an error message "Illegal character in path. Try copying the source and compiling directly" (see pic). Copying from where to where exactly? Cheers F.

.

Giulio Piacentino **piac** McNeel **Feb '17**

@Fabio_S , you can just click on "Copy Compilable Code" (Part 2 of the tutorial above).
Did you put a space in the module name? Or other special characters?