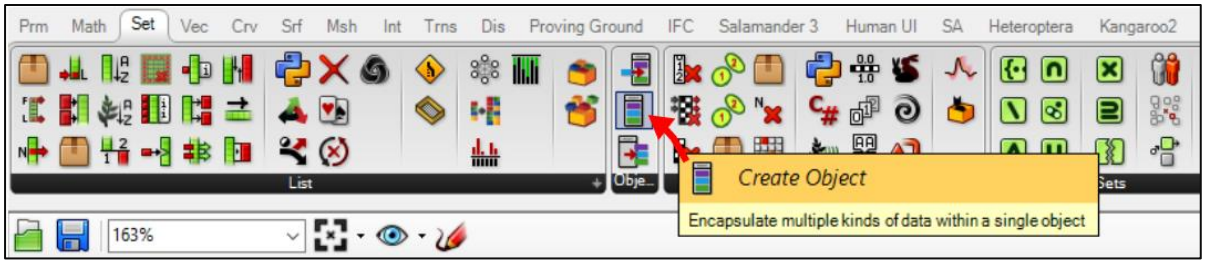
OBJECTIVISM – USER INFORMATION

Objectivism is a Grasshopper plugin that allows users to encapsulate data inside objects as named properties. Objectivism objects have the following capabilities:

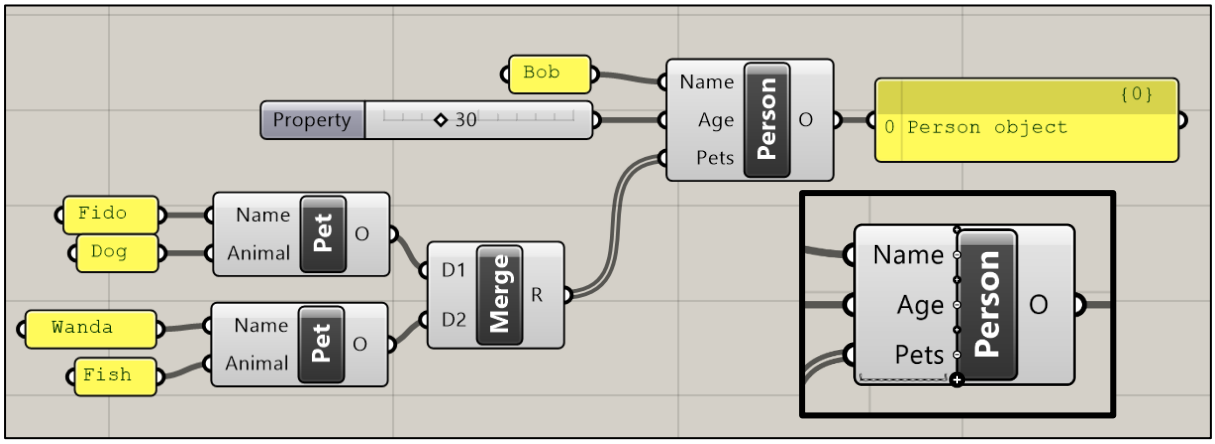
- Any data used in Grasshopper can be put into an Objectivism object, this includes putting an Objectivism object in another.
- Object properties can contain a single item, a list of items of a tree of items, using access modifiers just like the Grasshopper Python/C#/VB scripting components.
- Any geometry within an object will be previewed in the Rhino viewport when preview is active in Grasshopper.
- Objects can be transformed using Grasshopper's transform components. All geometry inside the object is transformed accordingly.
- Objects support serialization, this means objects can be internalised or passed between scripts using the Data Input/Data Output components (note, not all plugins support serialization, data from plugins passed into Objectivism Objects that do not support serialization will not be successfully serialized).

This guide provides basic information on using Objectivism.

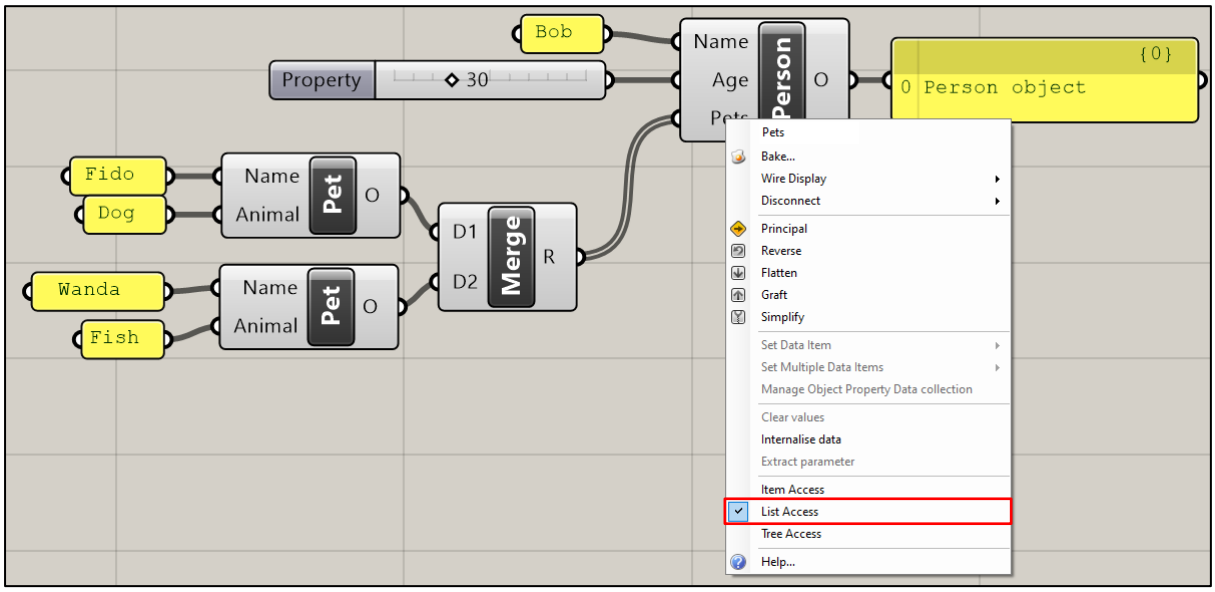
CREATING AN OBJECT



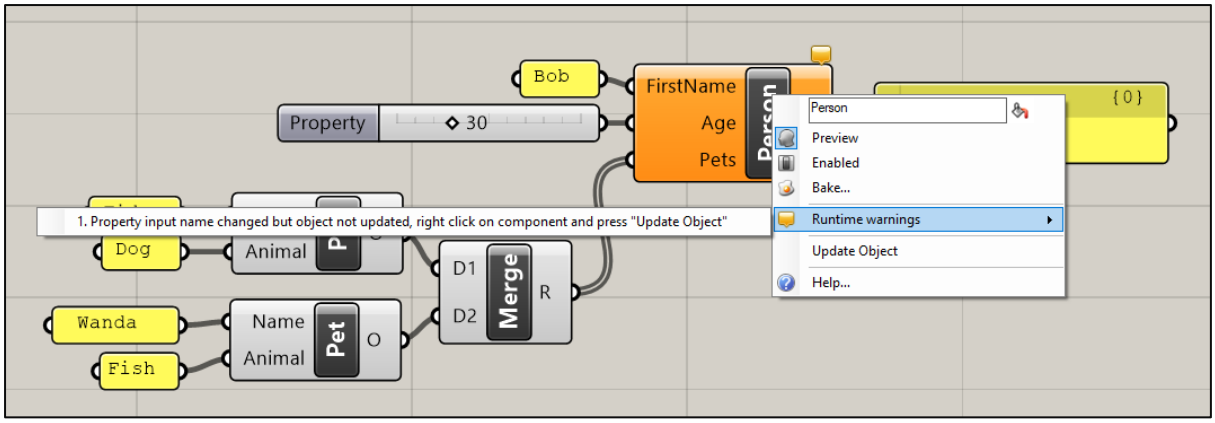
Use the Create Object component. This is found in the Sets tab on Grasshopper.



To create an object, add properties using Grasshoppers zoomable user interface (ZUI). Properties are renamed changing the property parameter's nickname. To change the objects type name (in the above example both "Pet" and "Person" types are created), simply change the components nickname.



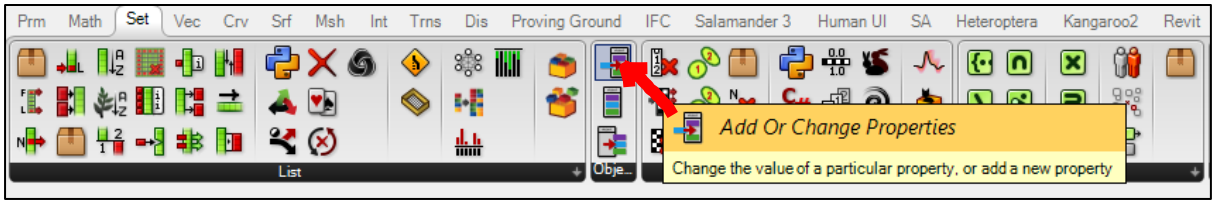
By right clicking on a property parameter the access level can be changed, here list access is used on the "Pets" property so a person can have multiple pets.



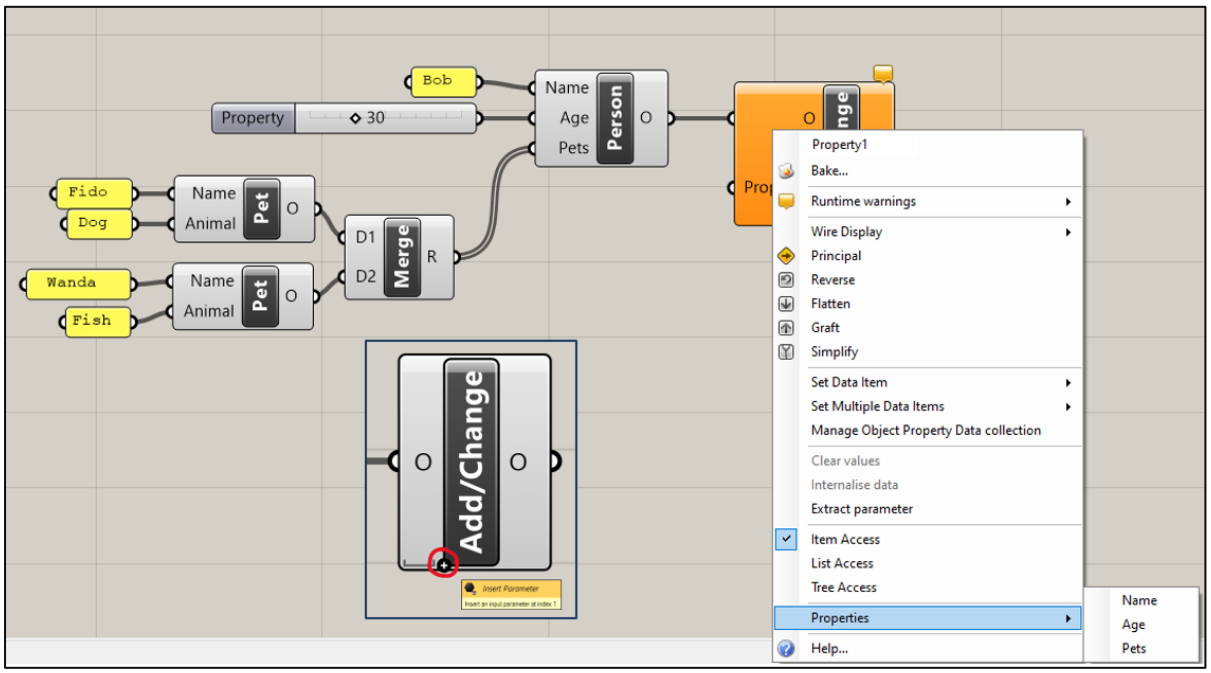
Note: once an object is created, if you change a property name or the objects type name (here “Name” is changed to “FirstName”), you will get a warning. This is because when a name is changed, it does not trigger the component to recompute and send the object with the new name(s) to downstream components. Instead, the warning is triggered informing you the object is not up to date. To update the object, you need to trigger the component to recompute, this can be done in several ways, including:

- Right click on the component and press “Update Object” (see image above).
- Re-wire any of the inputs to the component.
- Trigger a solution re-compute on any object upstream from the component (e.g. in this case adjust the slider connected to age)

MODIFYING AN OBJECT

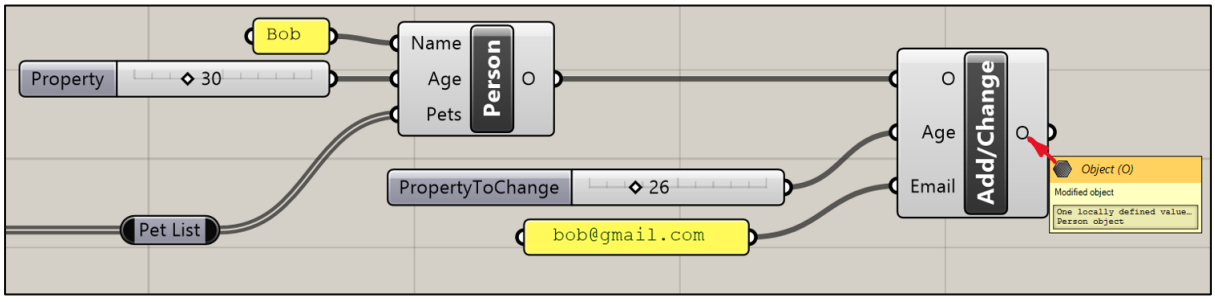


The “Add Or Change Properties” allows you to add properties to an object or change existing properties.



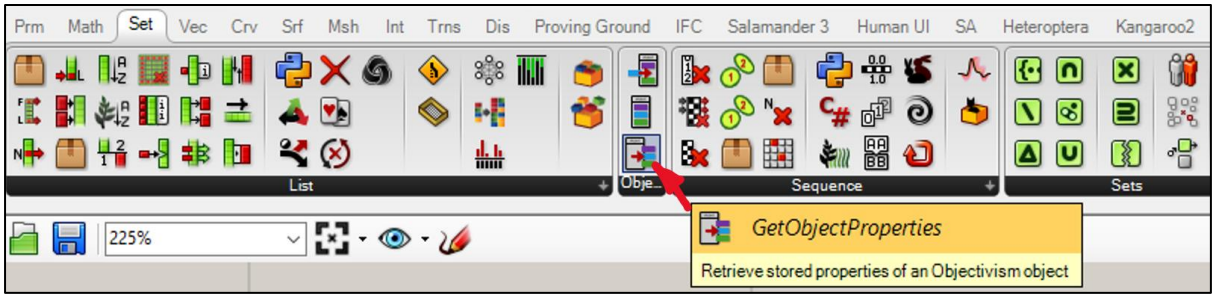
Use the ZUI to add property parameters to the component. These automatically fill with the names of existing properties. Note all property parameters default to item access, regardless of the access in the source property (This is because when multiple objects are connected to this component, they could all have a property with the same name, but different access).

Right click on a property parameter to select existing properties in the object to change. Or change the nickname to type an existing property name (to change that property) or new property name (to add that property)

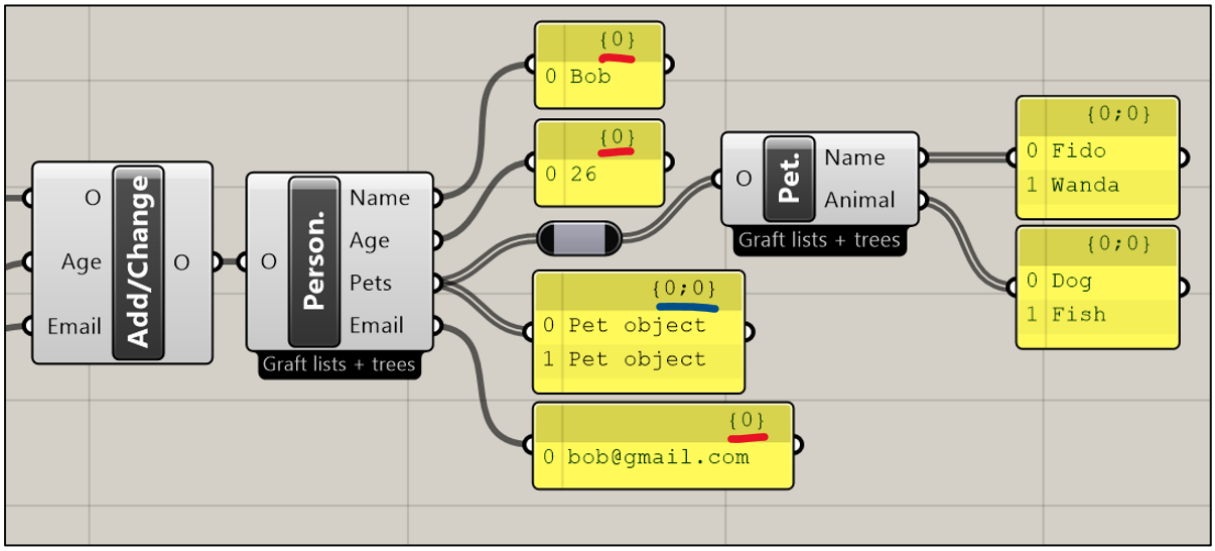


In the above image “Bob” has his age changed (Lucky him!) and is given an email address.

RETRIEVING OBJECT PROPERTIES

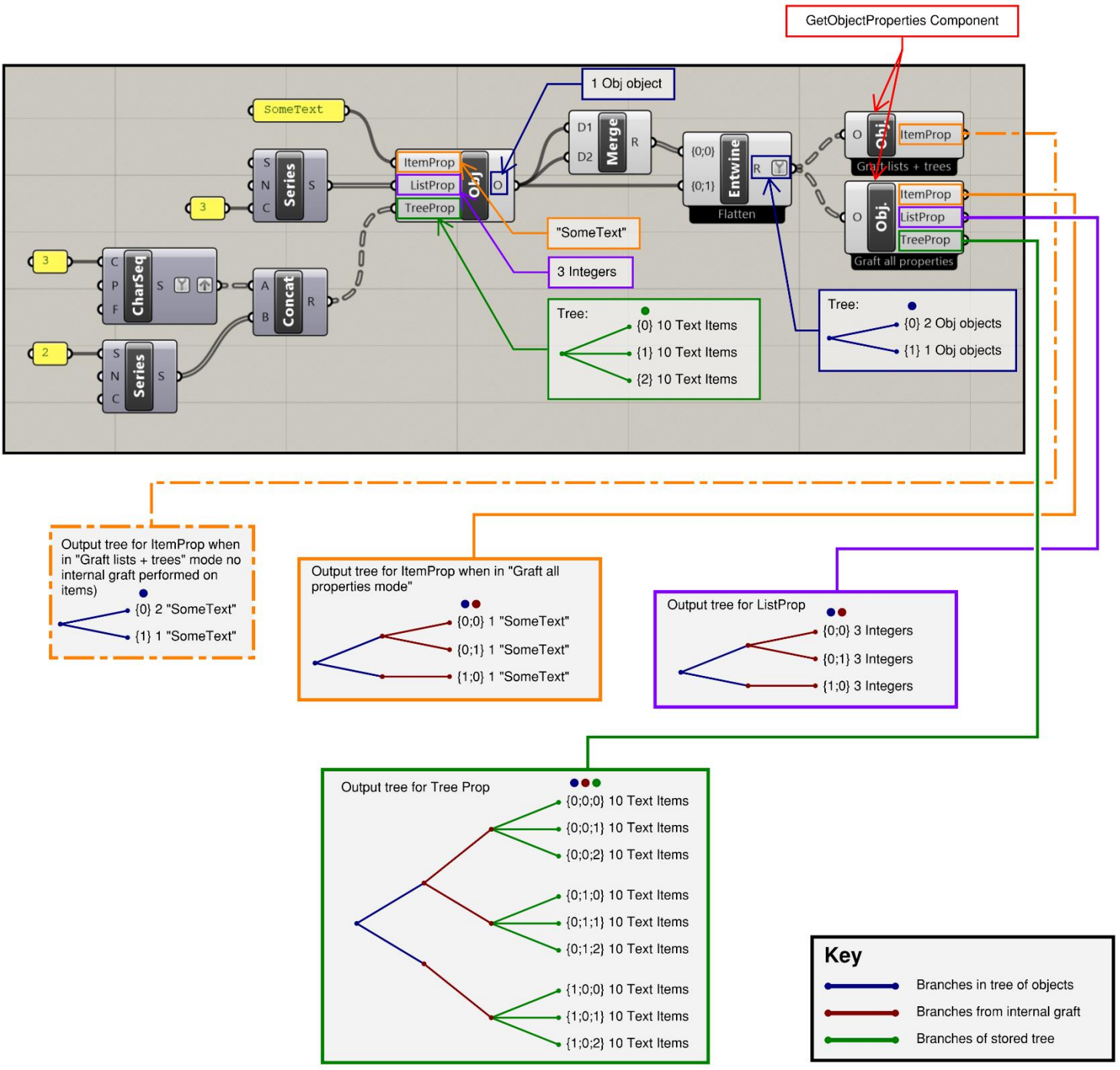


The “Get Object Properties” component is used to get properties from a component.



Like with the Add Or Change Properties component, properties can be added using the ZUI and are automatically filled in, or selected by right clicking on a parameter and going to properties. Alternatively, you can right click on the component and hit “Full Explode” to get every property. Note in the above example the component is in the “Graft list + trees” mode, and the “Pets” property (a list) has a different path to the other properties. See the next section: “Objects whose properties have multiple access modes” for an in-depth explanation of the behaviour of trees and lists in objects.

OBJECTS WHOSE PROPERTIES HAVE MULTIPLE ACCESS MODES



The retrieval of object properties using the "GetObjectProperties" component becomes complex when objects have list access and/or tree access properties. The example above illustrates these complexities and shows how Objectivism deals with them.

An example object is created that has an item, a list, and a tree property. A simple tree is then formed containing duplicates of this object.

For retrieving list properties: the object tree must be internally grafted by the component so that each retrieved list is in a different branch.

For retrieving item properties: in "Graft all properties" mode, the same internal graft is performed, and the output tree has the same structure as the output tree for any list properties; in "Graft lists + trees" mode the internal graft is not performed, and the output tree for item properties matches the input tree of objects.

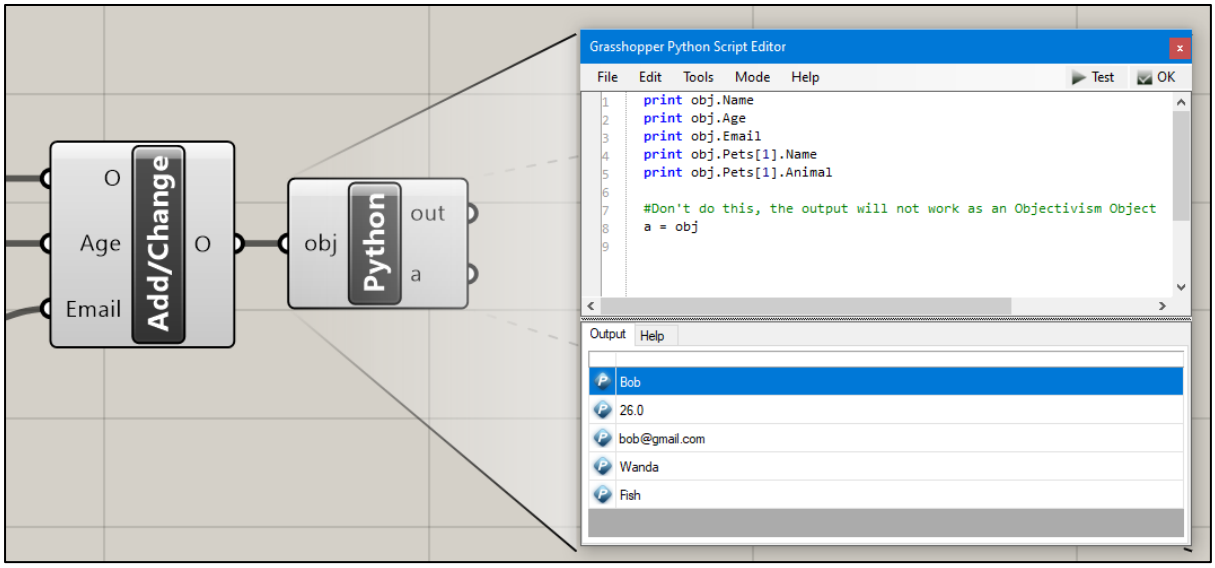
For retrieving tree properties: the internal graft is performed. The stored trees are added to the end of the branches generated by the internal graft.

WORKING WITH OBJECTS IN SCRIPTING COMPONENTS

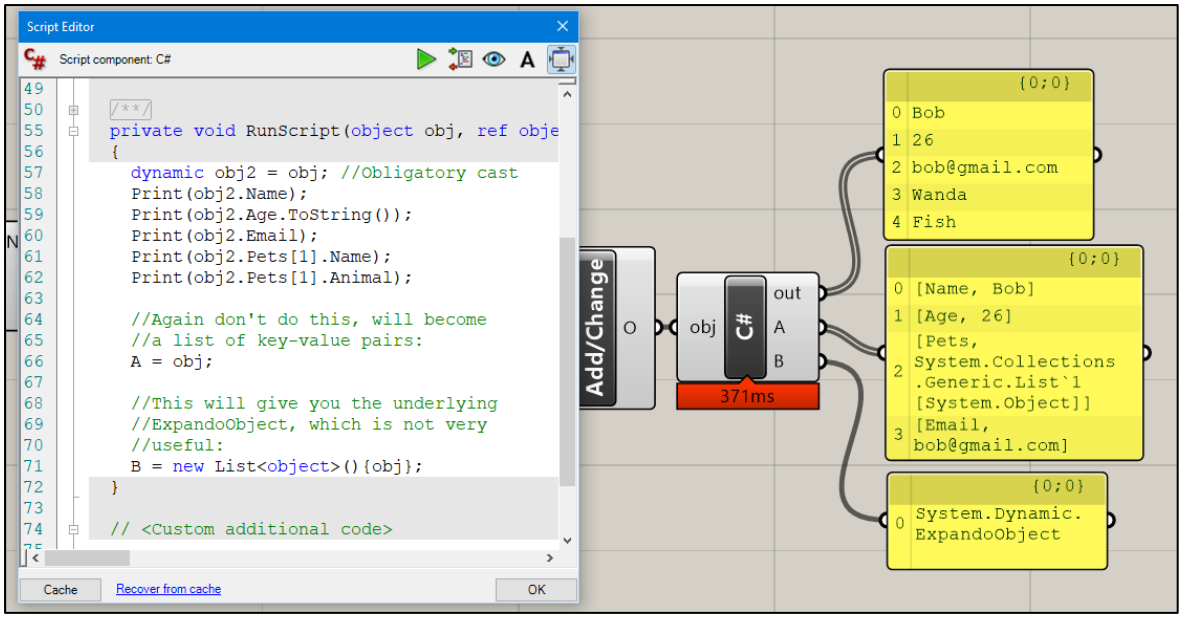
Objectivism is really designed to allow use of objects (or at least encapsulated data) in Grasshopper without coding. Therefore it is not highly recommended to be

When passed into a scripting component, Objectivism creates a dynamic version of the object for use in the scripting component. Internally Objectivism is implemented using dictionaries, and stores the IGH_Goo wrapped version of objects. Doing this creates a much more streamlined scripting experience.

It is not possible to pass an objectivism object out of a scripting component and then reuse as an objectivism object (e.g. pass it into the Get Object Properties component), primarily this is because the base type of the dynamic object (ExpandoObject) implements IEnumerable, so Grasshopper automatically converts it to a list on the way out (unless it is in a list).



Personally I recommend using Python with the objects, not C#. In python, it just “works”, and there is autofill on properties.



In C# the object must be casted to “dynamic” to work in C#. However sub-objects do not need this cast. Also there is no autofill on properties.