Creating and Modifying Revit Add-ins Using Python and PyRevit

Dan Boghean, OZ Architecture

Class Description

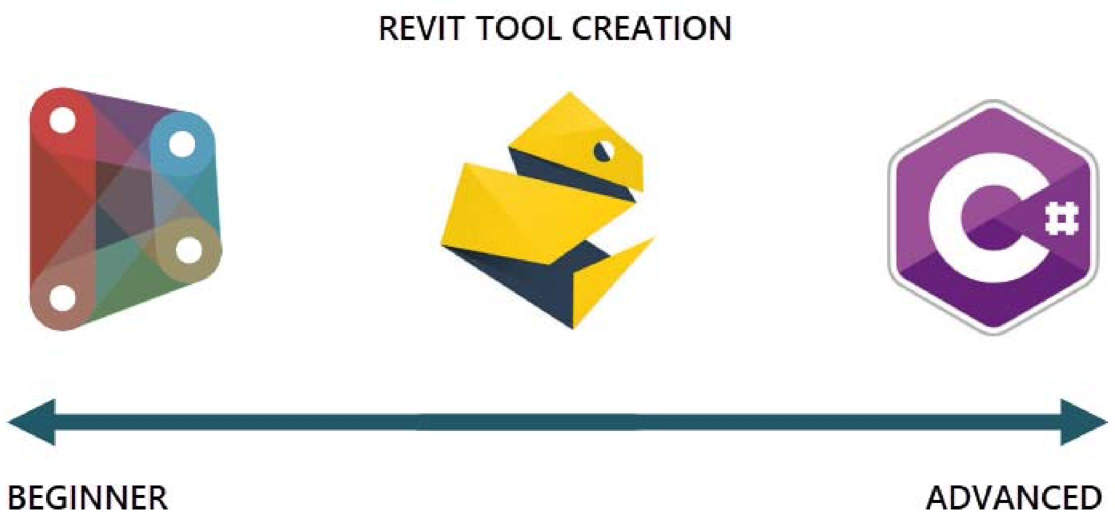
Everyone has downloaded and used a Revit add-in that extended the capabilities of Revit. But how many times have you wished you could modify something about the add-in, or better yet, create your own? Learning C#, however, is daunting and requires extra tools such as Visual Studio to build and compile the code, but what if you could create your own add-ins using nothing more than a text editor?
Python is one of the most popular programming languages because of the ease of use and shallow learning curve. We will learn how to use Python and the capabilities of PyRevit to modify and create add-ins without the hassle of having to learn the complexities of C#. We will also look at the process of converting existing Dynamo scripts to Python allowing you to prototype ideas in Dynamo and ultimately create them as an add-in that can be deployed to users.
The capabilities of using Python to extend the usefulness of Revit are boundless. This class will scratch the surface and allow you to get the framework up and running so that you can start pushing the Revit envelope.

About the Speaker:

Dan Boghean is the BIM Manager for OZ Architecture, an Architecture, Interior Design, and Marketing firm located in Denver, CO. He provides training, support, and implementation of new workflows for the BIM software at the firm. Having previous experience as a registered architect, he is able to bridge the gap between BIM management and architectural practice. Dan is interested in exploring how technology can help transform the AEC industry by creating efficient workflows and removing software barriers.

Programming Methods

Where do PyRevit and Python fit into the spectrum of ways to create new tools in Revit?



What is PyRevit?

PyRevit is an open source add-in for Revit developed by Ehsan Iran-Nejad that contains many incredibly useful tools that extend the capabilities of Revit. Beyond the existing tools that ship with PyRevit, the power of PyRevit is in the ability to create your own add-ins quickly and easily using nothing more than a text editor and a basic knowledge of Python and the Revit API. Today, we'll get our feet wet with how PyRevit works and create our own add-ins!

Installing PyRevit

PyRevit can be installed via an installer from the PyRevit website (https://ein.sh/pyRevit/). There are a few choices on installation. One option is to deploy the install to each user via deployment software such as KACE. This usually requires coordination with IT or knowledge of deployment software and the proper permissions. The other option is a network installation. With this, you install PyRevit in a folder on your network that everyone has access to. All you must do then is to deploy a .addin file to the proper location via a deployment script or log-on script. Depending on the size of your firm, you can even send an email with a .bat script that will automatically copy the .addin file. PyRevit also has a CLI interface to assist with the deployment process, but we will not focus on that in this lab. Just know that there are multiple ways you can install PyRevit and have it work correctly with pros and cons for each method.

Network Installation

At OZ, we currently use the Network location option. This allows for easier management of new/existing tools in near real time. It allows us to easily control what users see and when and add any new tools on the fly without having to re-deploy anything. It also makes the initial installation much easier as all you have to push to the users is a .addin file pointing to the correct DLL on the network .

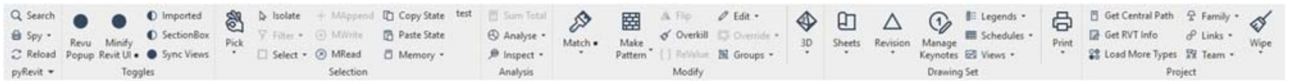
One thing to note, with this Network Location option, we had to add a line to the Revit.exe.config file for each user to allow external dll sources, otherwise Revit will not load it because it thinks it is a threat. If this is an issue for you and your firm, then I would recommend going down the install for each user route.

Existing PyRevit Tools

Finding existing script files

If you get nothing else out of this lab, then I hope it is that PyRevit has an incredibly robust set of tools all available for free as open source. This means you can see the way the script was created and modify it if needed. What a concept!

We're going to focus on a few tools and take a look behind the panels to see how to be able to access existing PyRevit tools, whether you need to modify them or want to figure out how they were made.

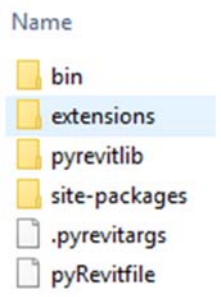


PyRevit allows you to hold down the ALT button and click on any of the existing scripts in the ribbon. This will launch Windows Explorer and open the location of the script file. You can use your favorite text editor to open the script.py file to make modifications or to learn how to implement a feature in a tool you are creating. It really is that simple!

Folder Organization

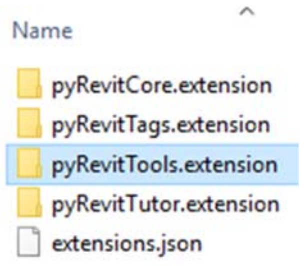
Now that we know how to find the script files from within Revit, let's take a look at how easy it is to create an add-in in the Revit ribbon. The beauty and simplicity of PyRevit allows you to create a tab, panel and button in Revit by simply creating folders.

In the PyRevit base folder, you'll find a list of folders that make PyRevit tick. The one we are interested in is the one labelled "extensions". This is where all of the out of the box PyRevit tools live and also where you can create new tabs, panels, and buttons.


Name

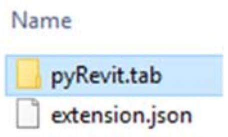
bin
extensions
pyrevitlib
site-packages
.pyrevitargs
pyRevitfile

In the extensions folder, you'll find a few folders with the .extension ending. You can also have your extensions live elsewhere and link to it from PyRevit by going to the settings and adding the location of your custom extensions folder there. On startup, PyRevit will add that location to the list of folders it looks at to load add-ins. We have a OZ.extension folder that lives in a separate location so that we can easily edit it apart from the PyRevit tools. You'll also notice the extension.json file. This is a JSON file that allows you to give specific information about the extension that is loaded into pyRevit. This includes information about the author of the extension and the website, but it also allows you to give access to specific users so that perhaps you have a BIM Manager toolset that is only available to certain people. It also allows you to control the visibility of specific extensions as well.


Name

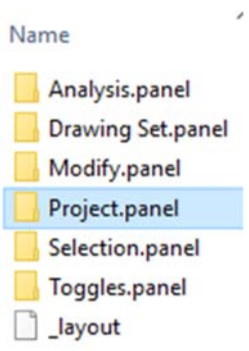
pyRevitCore.extension
pyRevitTags.extension
pyRevitTools.extension
pyRevitTutor.extension
extensions.json

Traversing further into the folder structure reveals the pyRevit.tab folder. This is where you can create separate tabs for your tools. In this case, the tab is created with the same name as the folder on the Revit ribbon, and any subsequent folders will create the add-ins under the specified tab.

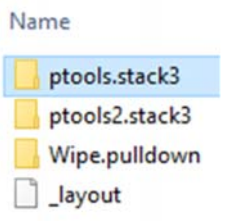

Name

pyRevit.tab
extension.json

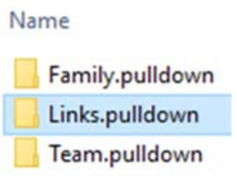
The .tab folder will contain .panel folders. The panel corresponds to a panel in the Revit ribbon and allows you to group add-ins. You can have as many or as little panels as you would like. Here you may notice the _layout file. This is a simple text file with nothing but the name of the folders on each new line. This allows you to specify exactly how the tools will be arranged on the Revit ribbon. Note, it does not have an extension.


Name
Analysis.panel
Drawing Set.panel
Modify.panel
Project.panel
Selection.panel
Toggles.panel
_layout

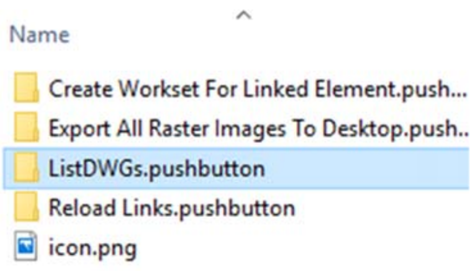
Under each panel, you now have the ability to create different types of organizations for your add-ins. You can create something known as a "stack3" where you have 3 add-ins stacked on top of each other, hence the stack3. You can also do a "pulldown" which allows you to put multiple add-ins under one pulldown button on the ribbon. These are the most common options. For even more options visit the PyRevit documentation here: https://pyrevit.readthedocs.io/en/latest/articles/creatingexts.html


Name
ptools.stack3
ptools2.stack3
Wipe.pulldown
_layout

If we look a bit further in the ptools.stack3 folder, we'll find 3 .pulldown folders. This will stack these pulldowns in a column and provide a dropdown when clicked to allow the user to select an add-in.


Name
Family.pulldown
Links.pulldown
Team.pulldown

Going further down the rabbit hole in the Links.pulldown folder, we find a few different .pushbutton folders. The script files will live in the .pushbutton folders. You can also see the icon.png file. This is the image that gets associated with each part of the ribbon. In this case, since the icon.png resides in the Links.pulldown folder, it will show up next to the pulldown in the ribbon. If no icon.png file is added in the nested folders, the icon will apply to all of the add-ins that are located further down the hierarchy.


Name
Create Workset For Linked Element.push...
Export All Raster Images To Desktop.push..
ListDWGs.pushbutton
Reload Links.pushbutton
icon.png

That is it. That is the organization in PyRevit that allows you to create various options for your ribbon in Revit. This gives you the flexibility to create all kinds of configurations by simply creating folders. I promise this is miles easier than having to do it with C#. And the flexibility of moving tools from one tab, pulldown, or panel to another is amazing. It doesn't get easier than this!

Python Basics

Python Resources

We will not be able to cover the entire Python programming language in this lab. I've provided a few resources below that are a wonderful introduction to Python. However, because Python is one of the most popular programming languages, there are literally hundreds of thousands of videos, articles, and courses on Python on the internet. If you have never used Python before, I would encourage you to check out some of these resources before you dive into the Revit API, I've tried to include a plethora of different learning styles.

Note: The add-ins we will write are going to be executed in IronPython which allows us to use the C# Revit API References, but it is built on Python.

- *Text Tutorials*
  - Real Python – Python Introduction (Text Tutorial)
  - Learn Python the Hard Way (Text Tutorial with Exercises)
  - Google – Python Class (Text Tutorial with Exercises)
  - Python – Python Tutorial (Very in-depth text tutorial)
  - Automate the Boring Stuff (Text Tutorial with Exercises)
- *Video Tutorials*
  - Jeremy Graham – Dynamo for Revit: Python Scripting (Lynda Video Course)
  - Coursera – Programming for Everybody (Getting Started with Python) (Coursera Video Course)
  - Python Tutorial for Beginners – Full Python Programming Course (Youtube Videos)

Revit Python Shell

The beauty of Python is that you can technically write your code in notepad, if you so choose. While this is technically feasible, I wouldn't recommend starting there. If you do want to use a text editor, I highly recommend getting Notepad++. This text editor offers a full-fledged set of features, but also adds color-coding if you save your files with the .py extension. This makes it much easier to identify blocks of code and various functions within your code.

The code we write in the lab will be written in Revit Python Shell (RPS). This is an add-in developed by Daren Thomas that brings the IronPython IDE into Revit and allows us to dynamically test our scripts. I recommend running RPS in the Non-modal shell mode. This means that you can select elements in Revit and work with the GUI without having to close RPS. We'll be using this to run most of the code samples below.

Fire up the Revit Python Shell add-in by going to the Add-ins tab and click on the Non-modal shell. This going to launch a dialog with two panels. The top panel allows you to type python commands as you would in a normal command line interface. You can execute short Revit API commands as well! The bottom panel is more similar to a text editor. It allows you to write entire scripts and run them, making changes easily along the way. One thing to note about the Non-modal shell version. To paste and copy, you have to use the buttons in RPS rather than the keyboard shortcuts. This is because Revit is still technically the active window. Using the shortcuts will start the commands in Revit instead. This is also true for the Undo and Redo functions, and also the delete button. But, I swear that's all the weird quirks, the pros make up for the cons!

Data Types and Structures

Python has a few built-in data types. These include but are not limited to:

- Integers (e.g.-5, 0, 1, 10, 100)
- Floating Point Numbers (e.g. 0.0, 15.37, 100.234)
- Strings (e.g. "Test", "This is a sentence", "a")
- Booleans (e.g. True, False)

Python also has a few built-in data structures:

- Lists (e.g. ["This is a list", 1, 2, "Test"])
- Dictionaries (e.g. {"key": "value", 1: "Test"}
- Tuples (e.g. (0,1) Ordered and Unchangeable)
- Sets (Unordered collection with no duplicate elements)

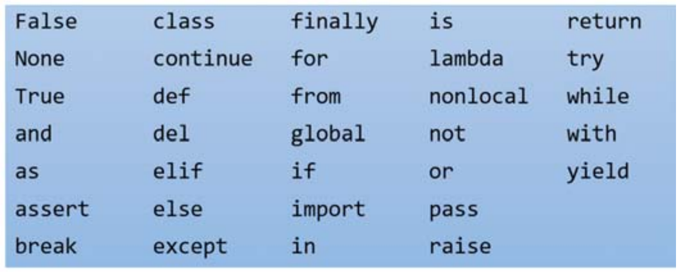
Python offers dynamic variable typing. This means that you don't need to explicitly declare variable types like you would in other programming languages such as C#. So, as you are declaring variables using the data types above, Python automatically figures out what type of input you are feeding into the variable. See the difference between C# and Python below. In the example we are also showing how we can assign data to a variable. Simply start with a name for your variable followed by the = sign and then by the data you want to assign to the variable, just make sure it isn't any of the reserved Python keywords. This allows us to use the variable in any part of our code after it is assigned and gives us the ability to change it in one place and have it affect multiple parts of our code.

C#

```
int int_num = 0
float float_num = 0.0
string sentence = "This is a sentence"
```

Python

```
integer_num = 0
float_num = 0.0
sentence = "This is a sentence"
```



False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Reserved Keywords in Python

Python Logic

As with most any programming language, Python has a few ways to process different logic situations. The most common is Conditionals (If/Else) and Loops (For/While).

Conditionals

Conditionals allow you to test for a condition and process the outcome differently if the test is True or False. In python "if" and "else" are reserved keywords. In this case, they are what sets off the conditional statement. Simply start it by invoking the "if" keyword. After the "if" keyword, you perform the test. This is going to be something that yields a True or False result. In this case, 1 > 0 will yield True in Python. Make sure you end the line with a colon, as this tells Python that everything that is indented after the code block belongs to the if statement.

```
if 1 > 0:
    #Do something here if True
else:
    #Do something here if False
```

Loops

Loops allow you to, well, loop through a set of items in Python. This is usually a list, but it can be anything such as a dictionary, tuple, or any other data structure that allows iteration. Loops are a powerful part of programming since it allows you to do something to a large list of items. You can also introduce different types of conditionals into the for loop to built complexity into the logic. The sky is the limit.

For loops are mainly used to traverse a data structure. They start with the "for" keyword. It is followed by a variable that stores the current iteration allowing you to manipulate it during the current loop. Then it is followed by the "in" keyword and then by the data structure you are iterating through followed by a colon. Again, anything that is indented after the colon becomes part of the for loop. All of the code that is indented gets executed **for** "each element" **in** "the data structure" until the data structure is exhausted. Then the code exits the for loop and continues to execute anything that isn't indented.

```
for item in items:
    #Do something here for each item
```

While loops work similarly to for loops. They allow you to do something as long as a condition is True. An example of this is doing something for a number of times. You create a variable outside of the while loop that starts at, 0, let's say. Then you start the While loop testing to see if the count variable is equal to the maximum number of times you want to loop. Once the condition becomes False, the while loop exits. The important part of a while loop is to ensure you create a condition that eventually makes the condition False, otherwise, the loop will be stuck for infinity and crash your script.

```
count = 0

while count < 10:
    #Do something until count becomes 10

    count += 1 #This will add 1 to count after every loop
    #If this line didn't exist, the while loop would run forever
```

The condition can test multiple things. See some of the different operators below:

Arithmetic Operators	**+, -, *, /, %(mod), **(exponent)**
Comparison Operators	== (equal), !=(not eq), <>, > , >=
Logical (or Relational) Operators	and, or, not
Assignment Operators	=, +=(increment), -+(decrement)
Membership Operators	in, not in
Identity Operators	is, is not

Creating Your Own Add-ins

Hello Built NA Code

Let's start off with the rite of passage exercise in programming, the "Hello World" exercise. We will create an add-in on the ribbon that's whole purpose is to say Hi!

Fire up RPS if it isn't already open so that we can start getting our hands dirty. The first thing we will always need to do is to import the Autodesk classes that we need to run our code. The classes we need for this example is in the Autodesk.Revit.UI namespace.

In simple terms, a namespace is like a last name. It allows code to provide a unique identifier so that different libraries can use functions that are named the same. In order to access the classes that we need in order to work with the Revit API, we must import them from the Autodesk.Revit.UI namespace. To import it the namespace, you can use the import keyword and then type out the namespace. In this case, rather than accessing the class by typing out the full Autodesk.Revit.UI namespace every time we want to use the class, we import the class directly from the hierarchy of the namespace. This means that we can simply call the TaskDialog() class rather than Autodesk.Revit.UI.TaskDialog() every time we want to use it. We can import the classes in two ways. First you can import all of the classes from a namespace by using the * keyword as such:

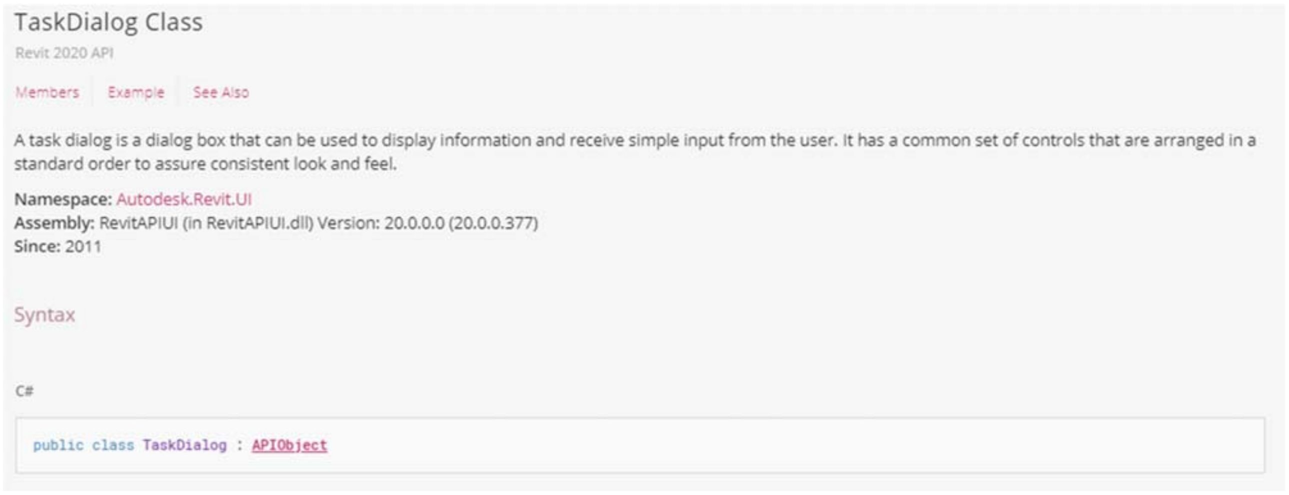
```
from Autodesk.Revit.UI import *
```

This can be fine if you are only working with one namespace, but it is not generally recommended because it brings in all of the classes in the specified namespace. The better way to do this is to identify the classes that we will be using in our code and import those specifically. I tend to come back to this import statement as I need to when I'm creating my code and import the necessary classes as such:

```
from Autodesk.Revit.UI import TaskDialog
```

In this case, we're interested in creating a "Hello World" example using the native Revit Task Dialog, so we will import this class.

One of the first questions I had when I was learning this stuff was, "How do I know what to import?" At first, it's going to be a lot of guessing and researching and prodding. One of the best resources for programming with the Revit API is the API Docs website. Bookmark this right now. It is something that you will use constantly and is invaluable. Let's take a look at how to use this resource. I've already given you the hint that we will use the TaskDialog class, so let's see how we can use it. If I lookup TaskDialog on the API Docs website, this is what comes up:


TaskDialog Class
Revit 2020 API

Members | Example | See Also

A task dialog is a dialog box that can be used to display information and receive simple input from the user. It has a common set of controls that are arranged in a standard order to assure consistent look and feel.

Namespace: Autodesk.Revit.UI
Assembly: RevitAPIUI (in RevitAPIUI.dll) Version: 20.0.0.0 (20.0.0.377)
Since: 2011

Syntax

C#

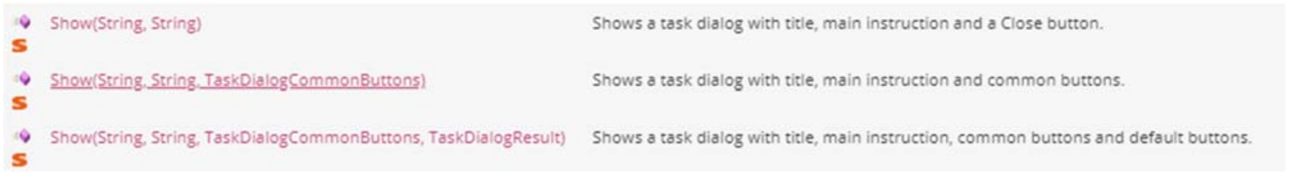
```
public class TaskDialog : APIObject
```

The landing page includes a few things we are interested in -- a short description of the class as well as the Namespace that it is in. At the top you'll see a link labelled "Members". This is where we can find the methods and properties that are part of the class. Think of the Methods as the actions that can be performed using the class (e.g. Show) and the properties as the information that can be accessed or set using the class (e.g. Title).

In this case, we are interested in creating an instance of the TaskDialog class to show our message when the button is clicked. To do this, we want to use the Show() method, what a shocker. In order to access any of the methods or properties we use the dot (.) after the class name and then the method or property we want to use. So in this case we would use TaskDialog.Show(). But, if we put that into our RPS and click the play button, this is what we get:

```
Exception : Microsoft.Scripting.ArgumentTypeException: Show() takes at least 1 argument (0 given)
```

This is because we haven't told the Show() method what to actually show in the TaskDialog. There's a few ways to use the Show() method as shown in the Methods section of the TaskDialog class:



Show(String, String)	Shows a task dialog with title, main instruction and a Close button.
Show(String, String, TaskDialogCommonButtons)	Shows a task dialog with title, main instruction and common buttons.
Show(String, String, TaskDialogCommonButtons, TaskDialogResult)	Shows a task dialog with title, main instruction, common buttons and default buttons.

Each one will do something a little bit different as explained in the short description and needs different arguments. We're going to keep it simple and use the Show(String, String) method. This is going to "Show a task dialog with title, main instruction and a Close button", perfect for what we are trying to achieve!

```
from Autodesk.Revit.UI import TaskDialog

TaskDialog.Show("BILT NA 2019", "Welcome BILT NA!")
```

The last part we want to add is specific to PyRevit. We have the ability to specify the add-ins name and the description the user sees when hovering over the add-in by adding a few simple lines at the top of the code.

The “__title__” variable allows us to specify the name of the add-in that shows up in the ribbon. By default, the name will be the same as the folder that we place the script in, but the “__title__” variable allows you to have more control and even add line breaks if the title gets too long.

The “__doc__” variable allows us to specify the description that will show when hovering over the add-in. Here we can give the user instructions or explain what the add-in does.

These two variables are important because we want to make sure we document our add-ins to make it easy to understand how to use them. There are other variables you can define such as __author__, __helpurl__ (for the F1 shortcut Help), __min_revit_ver__, and __context__ that you can explore. You can find more information about those in the PyRevit docs. Here is what our final Python code looks like:

```
__title__ = "Hello BILT!"
__doc__ = "This add-in shows a TaskDialog that exclaims 'Welcome to BILT NA'"

from Autodesk.Revit.UI import TaskDialog

TaskDialog.Show("BILT NA 2019", "Welcome to BILT NA!")
```

We’ve imported the proper class from the right namespace in the first line. Then we’ve used the TaskDialog class with the Show() method to Show a Revit Task Dialog using the title and main instruction arguments we’ve provided as strings. Now you can click the play button in RPS and you should see a Revit Task Dialog!

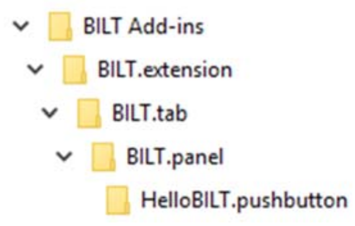
So, after all that, we’ve figured out how to use the API docs website to find the class we need. We can get a short description of the class as well as the Namespace its in in order to make sure we import the correct one. Then we can take a look at the Members which include the Methods and Properties of the class. You can click on any of the Methods or Properties to look at what arguments are expected for each one as well. Like it was previously mentioned, the API Docs website is an amazing resource that will help you find the information you need to code in the Revit API to your heart’s content.

Hello Built NA Add-in

We've got some python code that does what we want now, but this class is about creating add-ins. How do we get this into a button form that you can access from Revit?

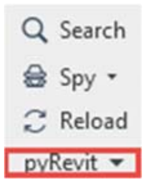
This is where the PyRevit folder structure we looked at earlier comes in. What we need to do first is create a .extension folder that PyRevit can find. We generally want to keep it outside of the PyRevit installation folder because if you decided to update it using GitHub, for example, it is possible that your folder will be removed. But, fear not! PyRevit has the ability to point to extensions wherever they may live by adding the folder path in the settings. We put the extension folder on the network that everyone can access and distribute the .ini file to everyone's computer, but you have various options when it comes to how you want to install PyRevit.

Before we add the folder path to the settings, we first have to make sure it exists. So, let's create a BILT add-ins folder and add a BILT.extension folder inside. Once that folder is created we will create .tab folder so that we tell Revit to create a new tab for our shiny new add-ins. The name of the folder will become the name of the tab. Let's also call this the BILT.tab folder. In the BILT.tab folder, we'll want to create a BILT.panel folder to specify the panel we want the add-in to live under and then add a HelloBILT.pushbutton folder in that for our add-in button. The name of the .pushbutton folder is purposefully different than what we specified in the __title__ variable so that you can see the way that setting the __title__ variable in the script changes the name. Otherwise, if we didn't include the __title__ variable, the add-in would be called "HelloBILT", exactly as it is show in our folder name. This is what the folder structure should look like:

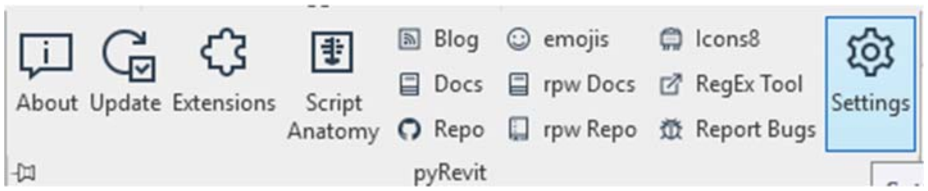


Now that the we've created the folder framework, all that we need to do is save our RPS script in the folder. The naming convention is to save the python script as "script.py" in the necessary folder. You can also add an icon for the add-in by saving it as "icon.png". I recommend looking at Icons8 for some great icons for your add-ins.

The final step is to make sure that PyRevit is pointed to our .extension folder we created. To do this, go to the PyRevit tab, and then click on the PyRevit flyout at the left of the ribbon.

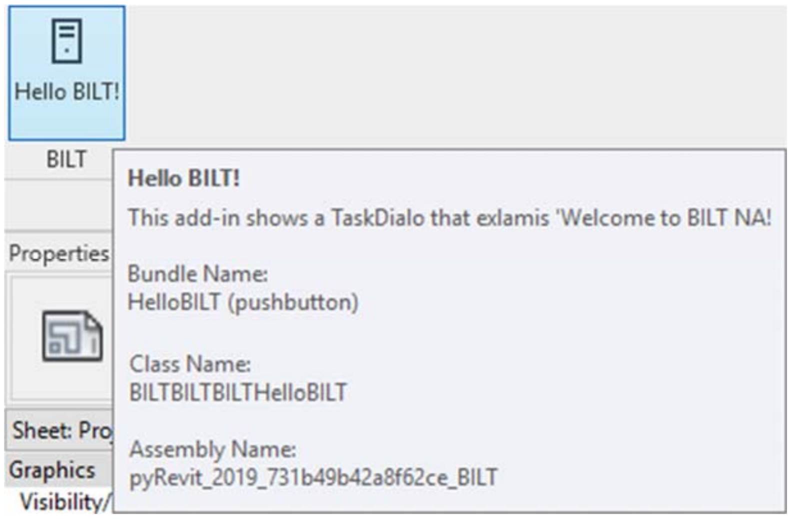


And then click the Settings button.



In the settings, you can add additional extensions by clicking the Add Folder button. When you add a folder here, PyRevit will look for the .extension folders and then follow the folder structure to create the necessary items on the Revit ribbon. You should only have to do this once, as any new add-ins added to this folder will get loaded as part of the PyRevit Loading process. Let's go ahead and add the BILT Add-ins folder we created that houses the .extension folder. Now you have two options. You can simply save the settings, or you can save and reload. Another awesome feature of PyRevit is the ability to reload your add-ins on the fly. This means that if you make any changes such as adding or removing tabs, panels, or add-ins in the ribbon, you can simply click the reload button without having to close Revit and re-opening it to see the changes. This allows for quick and pain-free development with on-the-fly changes!

Let's go ahead and click the Save and Reload button so that the ribbon is reloaded and our add-in shows up. Once PyRevit finishes the reload process, you should see the BILT tab we create, as well as the BILT panel, and the Hello BILT! Add-in we created.



Hello BILT!
This add-in shows a TaskDialo that exlamis 'Welcome to BILT NA!

Bundle Name:
HelloBILT (pushbutton)

Class Name:
BILTBILTBILTHelloBILT

Assembly Name:
pyRevit_2019_731b49b42a8f62ce_BILT

Congratulations! You've created your first add-in using Python and PyRevit! That wasn't so bad, right?

Revit API and Python Examples

Now that we've got our first add-in under our belts, we're going to focus on some scripts that help us access the Revit API via Revit. We'll look at some foundational items such as:

- Working with selections in Revit
- Collecting Elements using the Filtered Element Collector
- Reading and Setting parameters
- Using the PyRevit output to show data to the user
- Creating Elements using the Revit API
- Moving Elements using the Revit API

Combining these within your scripts will allow you to create a plethora of variations that will give you programmatic control over Revit in a few easy steps! Once we've written and tested the scripts in the Revit Python Shell, it's as simple as setting up the folder structure we previously looked at, saving your code as script.py in the proper folder and adding an icon, and you'll be a Revit add-ins ninja in no time!

Importing the Common Language Runtime

Before we start digging into how to work with selections in Revit, I want to touch on something I glossed over in the Hello Built NA code we wrote earlier. One thing I skipped was importing the Common Language Runtime (CLR) and adding the proper references that allows IronPython to be able to reference the classes in the Revit API. This is because PyRevit technically does this for us already in the initialization of the script. I would be remiss if I don't show that part of the script so that you get the whole picture. Again, this is technically not needed in your scripts and they will run just fine in PyRevit without the extra bit of code, but I want to show it for completeness.
At the beginning of every script, before we import the classes from the Autodesk.Revit.DB namespace, we need to first tell IronPython to add the RevitAPI Reference so that we can import the namespace from it. But before we can do that even, we have to make sure we import the Common Language Runtime (CLR). The Common Language Runtime, as Wikipedia puts it, is "the virtual machine component of Microsoft's .NET framework that manages the execution of .NET programs." IronPython does all the heavy lifting for us and allows us to work in the CLR in order to access the C# libraries we need to play with the Revit API. So, after all that, it all boils down to the fact that we need to import the CLR before we do anything. Once the CLR is imported, we can use the AddReference() method to make sure that the RevitAPI DLL is imported into our script environment which will give us access to the Revit API namespaces. This is what the code looks like:

```
import clr

clr.AddReference('RevitAPI')
clr.AddReference('RevitAPIUI')
```

Again, PyRevit takes care of this for us, so we don't need to add it every time, but you will see it in all of the dataset scripts to make sure that the understanding is there. Phew! That's enough of that. Let's get back to the fun stuff.

Working with selections in Revit

We've built our first add-in whose sole purpose was to show a TaskDialog. As fun as that was, it wasn't very useful. So, how do we build on our momentum? Let's figure out how we can get the selection that a user selects in Revit to allow us to work with it programmatically.

The first thing we will do is make sure to import the proper namespaces and classes. In this case, I know that we are going to be using classes from the Autodesk.Revit.DB namespace but also from the Autodesk.Revit.UI namespace where the Selection class we will be using resides, so let's go ahead and import the classes from those namespaces.

```
from Autodesk.Revit.DB import *
from Autodesk.Revit.UI import *
```

I mentioned that you typically want to import the classes you are working with rather than everything in the namespace using the * keyword, but when I first start a script, I import it all and then come back and clean it up at the end. This is how *I* do it and it has served me well so far.

Now we want to define the UI Document and the Document since we will be working with both. You'll probably memorize that the Document can be defined as __revit__.ActiveUIDocument.Document since this is something that is used in almost every single script you will write. The UI Document may be used less seldomly depending on whether or not you need to access the UI of Revit. In this case, we're going to be using both, so let's define them as such:

```
uidoc = __revit__.ActiveUIDocument
doc = __revit__.ActiveUIDocument.Document
```

Note, since the Document is derived from the UI Document, we could also have defined the doc variable as such: doc = uidoc.Document

There's a few different ways we get selections with the Revit API. They include but are not limited to the Active Selection when the add-in is run, the Pick Object(s) method, and the Pick By Rectangle method. These are the ones we'll look at in this lab.

The Active Selection Method

We can access the current selection in Revit by using the Selection property of the UI Document. From there, we can consult the API Docs website to see that we can use the GetElementIds() method to create a list of all of the selected element ids and assign it to the selectionIds variable which will become a list that will store all of the element ids of the elements the user has selected in Revit when the add-in is run.

```
selectionIds = uidoc.Selection.GetElementIds()
```

Now we have a list we can iterate through and perform an action on the individual elements. This is done using the for loop we looked at earlier. Since the GetElementIds() method spits out element ids and that is what our selectionIds list contains, we need to convert them to elements in order to access any of their information. To do this, we can use the GetElement() method of the Document which takes an ElementId as an input.

```
for id in selectionIds:

    element = doc.GetElement(id)
```

Once we have the element in the element variable, we can gather information from it by accessing the properties we need. In this case, we are going to print the Family Name in a task dialog.

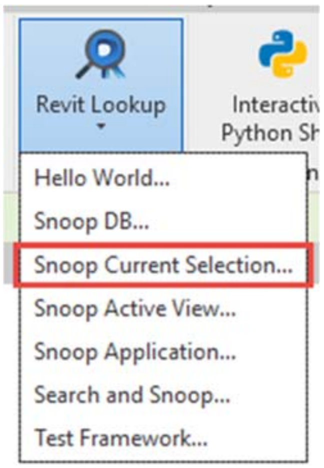
```
    TaskDialog.Show("BILT NA 2019", element.Symbol.Family.Name)
```

So how did I know that the Family Name was accessed by element.Symbol.Family.Name?

Revit Lookup Add-in

This is a good place to talk about a useful add-in called Revit Lookup. This was developed by Jeremy Tammik, a legend in the Revit API coding community. This add-in allows you to "snoop" specific objects in Revit and dig into the underlying element content in the Revit databse.

Let's look at the element we want to access the family name of. If we select the element in Revit and click on the Revit Lookup add-in under the Add-ins tab, you will see a dropdown of options. In this case, we are interested in the "Snoop Current Selection…" option.



This will launch a dialog with a bunch of information about the element. This ties directly to the Revit API properties and methods we previously looked at on the API Docs website. On the left, you'll see you'll see the current element type and name, and on the right, you'll see all the methods and properties of the element. If the fields are bold, that means you can double click on them to access even more information. This is akin to using the "." in our code that digs further into the properties or elements in the Revit API.

In our example, in order to access the family name of the element, we first need to access the Revit Type, or what is known as the Symbol in the Revit API world. Double clicking on the Symbol field brings up another dialog with information about the FamilySymbol. This is equivalent to us using the "element.Symbol" code. Once we have the Family Symbol (Revit Type) of the Family Instance, we need to access the Family to get the Family Name. Double clicking on the Family field, will yield yet another dialog with information about the Family. This is equivalent to the "element.Symbol.Family" piece of the code. Now that we are in the Family, we need to find the last piece of information we need which is the Name of the family. In this case, you'll see that it is not bold, meaning that it is a property that we can get and use as needed. This is equivalent to our final version of the code "element.Symbol.Family.Name". This piece of the code allows us to take the selected element, access the Symbol (Revit Type), get the Family associated with that Symbol, and then access the Name parameter of the Family and add that to our Task Dialog that we present to the user.

Here's what the final for loop code looks like:

```
for id in selectionIds:

    element = doc.GetElement(id)

    TaskDialog.Show("BILT NA 2019", element.Symbol.Family.Name)
```

Note, because the TaskDialog is part of the for loop, Revit will show us a Task Dialog for every element that is selected, so be sure not select too many elements, or you will likely be spending a lot of time clicking the OK button to close them out.

Now to add it to the Revit ribbon as an add-in. Since we did all the hard work of setting up the extension, tab and panel in the previous add-in, all we have to do is create a .pushbutton folder to get it to show up! We can create the Active Selection Family Name.pushbutton folder in order to get this to be a pushbutton in our Revit ribbon. Once the folder is created, the script.py file saved in the proper location, and the icon.png file added, we can click the Reload button on the PyRevit tab and see our add-in come to life!

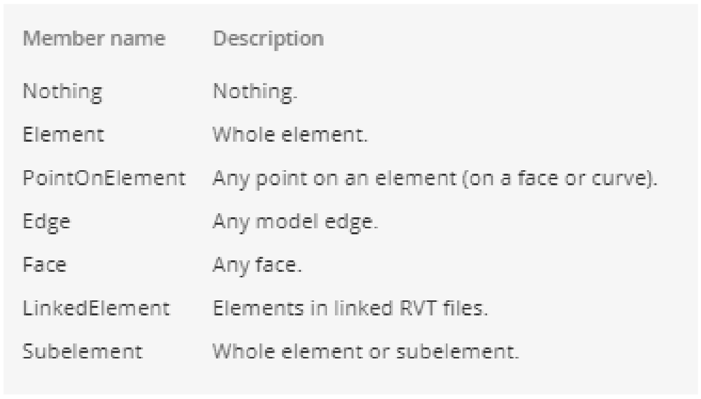
Note: If you put the icon in the .panel folder, all of the add-ins in the panel folder will use that icon unless you add a icon in the child folder to overwrite it.

The Pick Object(s) Method

So that's all fine and dandy if your users select the elements before running the add-in, but what about if you want to give them the option to select the elements after they run it instead?

That is where the PickObject() and PickObjects() methods come into play. As you can probably surmise, the difference between the two is that one allows the user to pick a singular object, while the other allows the user to pick multiple objects.

The PickObject() method requires one argument which is an enumeration of ObjectType. If we consult the API Docs website and search for ObjectType, we are told that the ObjectType is an enumeration. An enumeration is a restricted set of values that a user can access. In this case, it's a restirect set of selection methods we can perform in Revit:



Member name	Description
Nothing	Nothing.
Element	Whole element.
PointOnElement	Any point on an element (on a face or curve).
Edge	Any model edge.
Face	Any face.
LinkedElement	Elements in linked RVT files.
Subelement	Whole element or subelement.

We are interested in the element, so to use this enumeration it would look like this:
ObjectType.Element

In order to use this enumeration, however, we need to make sure we bring it into our code by importing it from the Autodesk.Revit.UI.Selection namespace as such, otherwise we will get an error that ObjectType is not defined:

```
from Autodesk.Revit.UI.Selection import ObjectType
```

If we look at the PickObject() method, we can see that it returns a Reference instead of an element or element id that we have worked with before. Luckily, our doc.GetElement() method we used previously still works with references, so we can use this to get the element object we need in order to access the family name property.

```
public Reference PickObject(
	ObjectType objectType
)
```

We stored the Reference in the "selectionReference" variable so that we can work with it down the line. And then used the doc.GetElement(selectionReference) method to store the element in the selectionElement variable. Finally, we can use the same properties we used previously to access the Family Name.
Since the PickObject() method only returns one element, it is not in a list, and thus, we will not need to iterate over it using the for loop. Here is the final code:

```
from Autodesk.Revit.DB import *
from Autodesk.Revit.UI import *
from Autodesk.Revit.UI.Selection import ObjectType

uidoc = __revit__.ActiveUIDocument
doc = uidoc.Document

selectionReference = uidoc.Selection.PickObject(ObjectType.Element)

selectionElement = doc.GetElement(selectionReference)

TaskDialog.Show("BILT NA 2019", selectionElement.Symbol.Family.Name)
```

The PickObjects() method allows the user to pick multiple elements after launching the add-in and then click the Finish button not he ribbon when the selection is finished.The PickObjects() Method is actually more similar to the Active Selection method, as it returns a list that needs to be iterated over using a for loop. The only difference here between the Active Selection code is that we changed the uidoc.Selection.GetElementIds() method to uidoc.Selection.PickObjects(ObjectType.Element) method. Here is the final code for that:

```
from Autodesk.Revit.DB import *
from Autodesk.Revit.UI import *
from Autodesk.Revit.UI.Selection import ObjectType

uidoc = __revit__.ActiveUIDocument
doc = uidoc.Document

selectionIds = uidoc.Selection.PickObjects(ObjectType.Element)

for id in selectionIds:

    element = doc.GetElement(id)

    TaskDialog.Show("BILT NA 2019", element.Symbol.Family.Name)
```

The Pick By Rectangle Method

The PickElementsByRectangle() method allows the user to draw a selection box around the elements they want to select rather than picking the elements individually. The code is exactly the same as the Active Selection and Pick Objects methods with only the selectionIds variable changing as seen here:

```
from Autodesk.Revit.DB import *
from Autodesk.Revit.UI import *

uidoc = __revit__.ActiveUIDocument
doc = uidoc.Document

selectionIds = uidoc.Selection.PickElementsByRectangle()

for id in selectionIds:

    element = doc.GetElement(id)

    TaskDialog.Show("BILT NA 2019", element.Symbol.Family.Name)
```

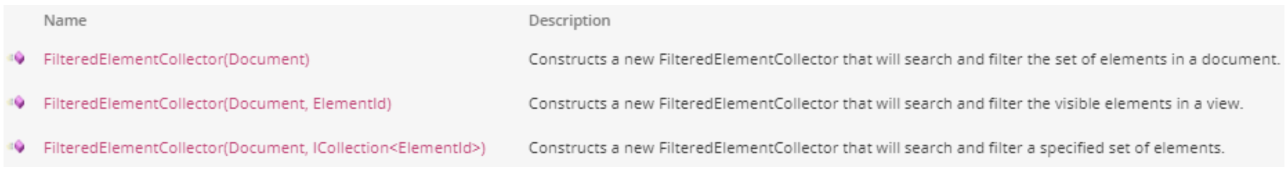
Now that we have three different selection method add-ins, we can create a pulldown option on the Revit ribbon that encompasses all of them. Let's create a .pulldown folder under the BILT.panel folder, and then place all of the .pushbutton folders we've created in it. Then all we have to do is reload under the PyRevit tab and we should see the change!

Collecting Elements using the Filtered Element Collector

Up to this point, we've figured out how to create an add-in and work with user selections, but what if we want to work with lots of elements of a specific category for example, how do we get those elements programmatically?

This is where the FilteredElementCollector() class comes in! According to the documentation, the FilteredElementCollector() class "is used to search, filter and iterate through a set of elements." We can apply specific filters that will run at a high-level on the elements and return the set of elements we want in an efficient way.

The FilteredElementCollector has a few different constructors (Different ways to initialize the class).



Name	Description
FilteredElementCollector(Document)	Constructs a new FilteredElementCollector that will search and filter the set of elements in a document.
FilteredElementCollector(Document, ElementId)	Constructs a new FilteredElementCollector that will search and filter the visible elements in a view.
FilteredElementCollector(Document, ICollection<ElementId>)	Constructs a new FilteredElementCollector that will search and filter a specified set of elements.

The one we are interested in at the moment is the first one that takes a Document as the argument. Let's instantiate the FilteredElementCollector class by passing in the doc variable we have been using to specify the current document like so:

```python
from Autodesk.Revit.DB import *
from Autodesk.Revit.UI import *

uidoc = __revit__.ActiveUIDocument
doc = uidoc.Document

fec = FilteredElementCollector(doc)
```

If we run this code and print the fec variable, we will get this:

```
<Autodesk.Revit.DB.FilteredElementCollector object at 0x0000000000001355 [Autodesk.Revit.DB.FilteredElementCollector]>
```

The FilteredElementCollector object is an iterable list, which means we can run through it using the for loop. If we use a for loop to run through the FilteredElementCollector and print the results, this is what we get:

```
Exception : Autodesk.Revit.Exceptions.InvalidOperationException: The collector does not have a filter applied.
   at Autodesk.Revit.DB.FilteredElementCollector.GetFilteredElementCollectorIterator()
   at IronPython.Runtime.Operations.PythonOps.GetEnumeratorFromEnumerable(IEnumerable enumerable)
   at System.Dynamic.UpdateDelegates.UpdateAndExecute1[T0,TRet](CallSite site, T0 arg0)
   at Microsoft.Scripting.Interpreter.DynamicInstruction`2.Run(InterpretedFrame frame)
   at Microsoft.Scripting.Interpreter.Interpreter.Run(InterpretedFrame frame)
   at Microsoft.Scripting.Interpreter.LightLambda.Run2[T0,T1,TRet](T0 arg0, T1 arg1)
   at IronPython.Compiler.PythonScriptCode.RunWorker(CodeContext ctx)
   at Microsoft.Scripting.Hosting.ScriptSource.Execute(ScriptScope scope)
   at Microsoft.Scripting.Hosting.ScriptSource.ExecuteAndWrap(ScriptScope scope, ObjectHandle& exception)
```

This is because all we have done is created an instance of the FilteredElementCollector() class and assigned it to the current Document we are working in. In order for the FilteredelementCollector to collect any elements we have to apply a filter to it.

So how do we use the filters to gather the elements we're interested in? Two of the most common ways is by using the OfClass() method or the OfCategory() method. The OfClass() method can typically be used with most of the classes you see in the API Docs list of classes. This allows you to gather elements of that class such as Walls and Views as well as more broad things like FamilyInstances. The OfCategory() class requires a BuiltInCategory enumeration. Remember that an Enumeration is simply a restricted list of items. In this case, it allows you to specify a specific category within Revit such as Walls, Plumbing Fixtures, or Mechanical Equipment. These enumerations can be found on the API Docs website by searching BuiltInCategory and finding the Category you are looking for. They will typically start with OST_ which stands for "Object STyle". Not a particularly important piece of information, but something to note and might be good for office Revit trivia!

As with any other object in Python, we can stack these together by using the "." operator. Here is an example of the OfClass() method:

```
from Autodesk.Revit.DB import *
from Autodesk.Revit.UI import *

uidoc = __revit__.ActiveUIDocument
doc = uidoc.Document

fec = FilteredElementCollector(doc).OfClass(Wall)
```

Using a for loop and iterating through the fec list, we now get this:

```
fec = FilteredElementCollector(doc).OfClass(Wall)

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

```
<Autodesk.Revit.DB.Wall object at 0x0000000000001358 [Autodesk.Revit.DB.Wall]>
<Autodesk.Revit.DB.Wall object at 0x0000000000001359 [Autodesk.Revit.DB.Wall]>
```

This is because I only have two walls created in my Revit Project. This will return a list of only the Walls that are placed in your project.

Here is an example of the OfCategory() method:

```
fec = FilteredElementCollector(doc).OfCategory(BuiltInCategory.OST_PlumbingFixtures)

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

Note, in order to use the Enumeration, you'll have to start with BuiltInCategory followed by a "." and then by the category you are interested in gathering from the BuiltInCategory Enumeration on API Docs. Another good way to find the BuiltInCategory Enumeration of a family is to use the Revit Lookup Tool and look under the Category Parameter.

If we look at the output, you'll see this:

```
<Autodesk.Revit.DB.FamilySymbol object at 0x000000000000135A [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x000000000000135B [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x000000000000135C [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x000000000000135D [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x000000000000135E [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x000000000000135F [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x0000000000001360 [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x0000000000001361 [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x0000000000001362 [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x0000000000001363 [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x0000000000001364 [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x0000000000001365 [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x0000000000001366 [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x0000000000001367 [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x0000000000001368 [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilySymbol object at 0x0000000000001369 [Autodesk.Revit.DB.FamilySymbol]>
<Autodesk.Revit.DB.FamilyInstance object at 0x000000000000136A [Autodesk.Revit.DB.FamilyInstance]>
<Autodesk.Revit.DB.FamilyInstance object at 0x000000000000136B [Autodesk.Revit.DB.FamilyInstance]>
```

If you remember, in Revit API terms, the FamilySymbol is equivalent to the Revit Type. So in this case, we are getting all of the Revit Types of the Plumbing Fixture category as well as the two family instances I have placed in my project.

To specify one or other, you can add the WhereElementIsElementType() method or the WhereElementIsNotElementType() method. The former will give you only the Revit Types while the latter will give you only the Revit Instances. Here is an example:

```
fec = FilteredElementCollector(doc).OfCategory(BuiltInCategory.OST_PlumbingFixtures).WhereElementIsNotElementType()
```

This is getting really long. We can break it up by adding a separate variable as such:

```
fec = FilteredElementCollector(doc)
plumbingFixtures = fec.OfCategory(BuiltInCategory.OST_PlumbingFixtures)
instances = plumbingFixtures.WhereElementIsNotElementType()
```

This will help you clean up your code a bit. Python's recommendation is to keep line lengths to 79 characters or less and this is just one way to achieve that. And then this is what the output is once we iterate over the list and print each element, only the instances we have placed in the project:

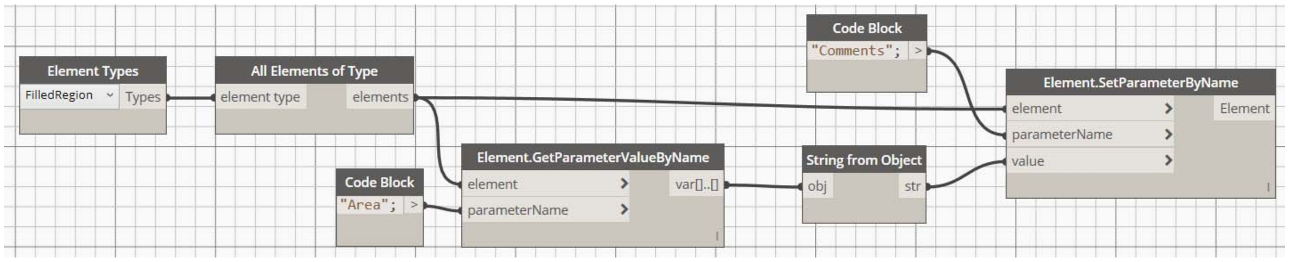
```
<Autodesk.Revit.DB.FamilyInstance object at 0x000000000000136C [Autodesk.Revit.DB.FamilyInstance]>
<Autodesk.Revit.DB.FamilyInstance object at 0x000000000000136D [Autodesk.Revit.DB.FamilyInstance]>
```

So, there you have it. Now you can use the FilteredElementCollector to gather a list of specific elements to work with programmatically. You can also combine multiple categories or add other filters that we won't cover in this lab but would be something to explore.

Reading and Setting Parameters

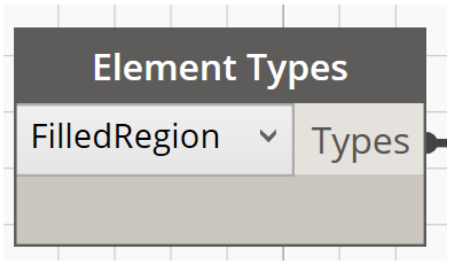
We now know how to work with selected elements, how to gather all elements of a specific category. Look how far we've come!

Let's keep the momentum going. Rather than going straight to the code for this one, however, I figured it might be useful to take a look at how we can think about converting an existing Dynamo script to Python. This is going to be a very easy script, but we'll walk through it so that you can apply the same framework to more complex scripts.



This script finds all of the filled regions in the project, gets the "Area" parameter and then writes it to the "Comments" parameter (since we can't schedule Area for filled regions). So, how do we transfer this to something we can do in Python?

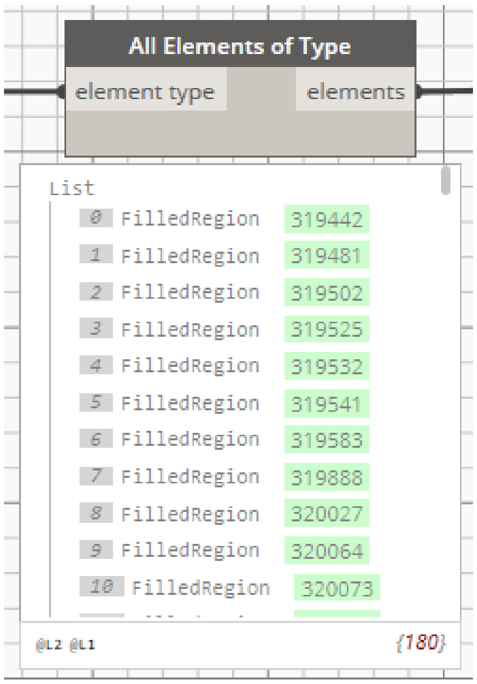
Well, the first part is easy, we've already done it! The Element Types node in Dynamo is similar to the FilteredElementCollector and the OfClass() method we used previously. In this case we'll use the OfClass() method with the FilledRegion class. This gathers all of the filled regions in the project. That wasn't hard, was it? Any time you see the Element Types node, think FilteredElementCollector in Python with the OfClass() method. If you see the Categories node in Dynamo, think of the FilteredElementCollector with the OfCategory() method.



To do this in Python:

```
fec = FilteredElementCollector(doc)
filledRegions = fec.OfClass(FilledRegion)
instances = filledRegions.WhereElementIsNotElementType()
```

Looking at the All Elements of Type output, we can see that it iterates through all of the filled regions in the project and prints out result. We've also done this already using a for loop to iterate through the list of filled regions and printing out the result.



```
List
0 FilledRegion 319442
1 FilledRegion 319481
2 FilledRegion 319502
3 FilledRegion 319525
4 FilledRegion 319532
5 FilledRegion 319541
6 FilledRegion 319583
7 FilledRegion 319888
8 FilledRegion 320027
9 FilledRegion 320064
10 FilledRegion 320073
@L2 @L1                    {180}
```

To do this in Python:

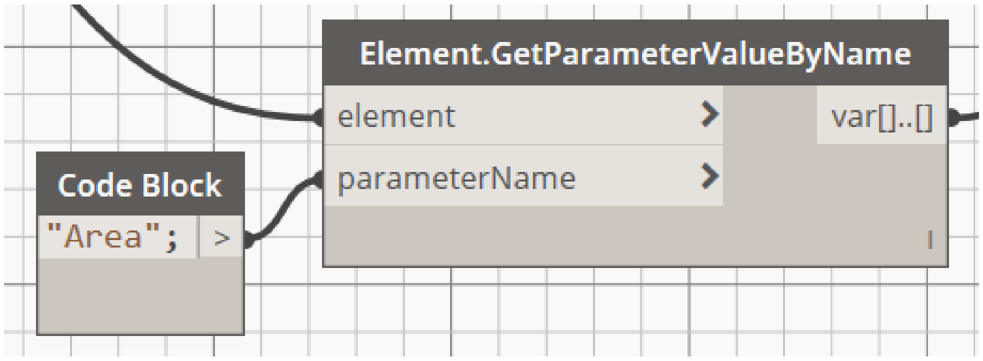
```
for instance in instances
    print(instance)
```

If we run our code so far, the output looks exactly like the output in Dynamo:

```
2240-Toilet-ADA-Commercial-Flush Valve_PLM_NH_OZ
2240-Toilet-Commercial-Tank_PLM_NH_OZ
2240-Toilet-ADA-Commercial-Flush Valve_PLM_NH_OZ
2240-Toilet-Commercial-Tank_PLM_NH_OZ
```

Great! Now we know how to get all the elements in a category and then iterate through them which is similar to the Categories Node and the All Elements of Category nodes. This should just be review so far. We did actually learn something after all!

Now we're getting to the fun stuff.



The Element.GetParameterValueByName gets the value of the specified parameter from each element in the list that is provided. Since we want to get the parameter of each element in the list, we need to perform this action in the for loop we've created. In order to get a parameter value in Python, we first need to get the parameter. To do this, we use the Element.LookupParameter() method. This allows us to provide a string and Revit will try to find a parameter that matches what we input. This works fine most of the time, but be advised that if there are multiple parameters that match the input, you'll only get the first one that Revit matches. Another way to get a parameter is to use the BuiltInParameter enumeration. You can specify the correct enumeration for the parameter you're looking for and use the Element.get_Parameter() method instead. For simplicity, we're going to use the LookupParameter() method. This is what the Python looks like:

```
for instance in instances:
    copyParameter = instance.LookupParameter("Area")
```

We've swapped the print statement with the LookupParameter method to get the Revit Parameter we're interested in. We're not done yet, though. If you were to print the parameter variable, you would get something like this:

```
<Autodesk.Revit.DB.Parameter object at 0x00000000000001C2 [Autodesk.Revit.DB.Parameter]>
```

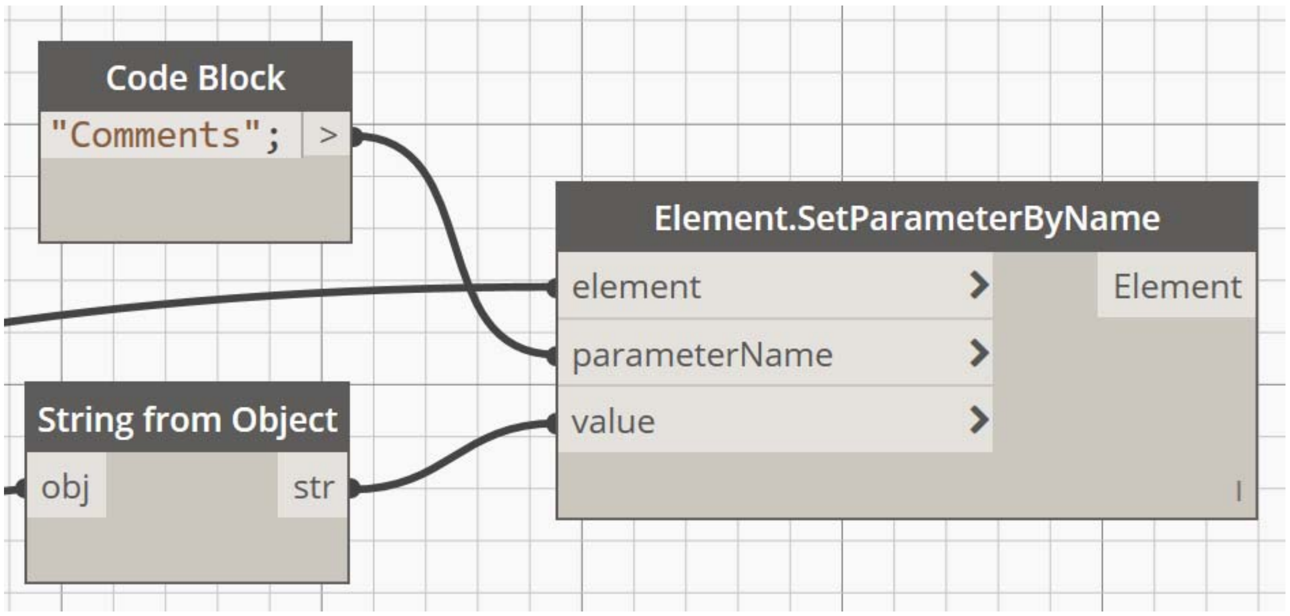
That's not what we want. We want the actual value of the parameter. To access the Value, you first need to know what the StorageType of the parameter is. You can find this out by printing the Element.StorageType property. In this case, we find that the Area parameter is a Double, or Number in Revit terms. In order to get the value of a parameter that is a Double StorageType, we can use the AsDouble() method. For other storage types such as integers, stromgs, and element ids, you can use AsInteger(), AsString(), and AsElementId() methods. There's also the

AsValueString() if you want to get units with your string parameter as well (e.g. Area with the SF suffix).

We now have the value of the parameter! This corresponds to the Element.GetParameterValueByName node in Dynamo. You are taking an element, finding the parameter by specifying a name, and then getting the value. This is what it looks like in Python:

```
for instance in instances:
    copyParameter = instance.LookupParameter("Area")
    copyParameterValue = copyParameter.AsDouble()
```

One last piece left to the puzzle:



If we look at the Element.SetParameterByName node, you can see that it looks at the element input, it finds the parameter specified, and then sets that parameter to the value specified.

We can follow the same logic with Python. Again, since we're already getting the element by using the for loop, we've already done the first part of the element input. Next we need to get the parameter, which we've done previously by using the LookupParameter() method. We're going to create another parameter called pasteParameter and use the LookupParameter() method to find the Comments parameter as such:

```
pasteParameter = instance.LookupParameter("Comments")
```

Next, we need to set the value of the parameter rather than get the value as we did previously. Conveniently, to set a parameter value, we can use the Set() method and put the value we retrieved previously as the argument, easy peasy.

And this is what our for loop will look like in the end. For each element (instance) in the list of instances we first find the parameter we want to :

```
for instance in instances:
    copyParameter = instance.LookupParameter("Area")
    copyParameterValue = copyParameter.AsDouble()

    pasteParameter = instance.LookupParameter("Comments")
    pasteParameter.Set(str(copyParameterValue))
```

But there’s one thing missing. Any time that we make any changes to elements in Revit, we need to create a Transaction to work in. If you were to run the above code, you will get yelled at. A Transaction will create an undo event in Revit and make the necessary changes to the database. To do this, we have to create a variable and assign it a Transaction. The Transaction class takes two arguments. The first is the document and the second is the name of the transaction that will show up in the undo list. Once the Transaction class is instantiated, we have to start it using the Start() method. Then we will be able to run our code that changes the Revit database and finally call the Commit() method of the transaction to finalize the changes as such:

```
transaction = Transaction(doc, "Create Element")
transaction.Start()

#Code goes here

transaction.Commit()
```

Now we can add our code within the Start() and Commit() methods.

So, after we setup the Transaction, this is what it all looks like:

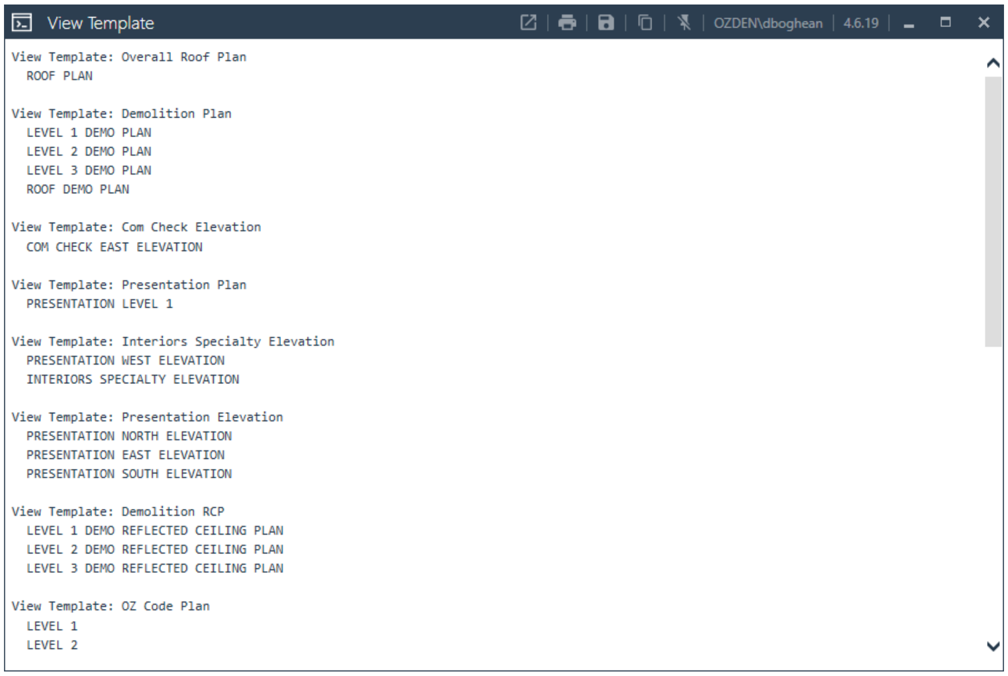
```
from Autodesk.Revit.DB import *

fec = FilteredElementCollector(doc)
filledRegions = fec.OfClass(FilledRegion)
instances = filledRegions.WhereElementIsNotElementType()

transcation = Transaction(doc, "Read and Set Parameters")
transaction.Start()

for instance in instances:
    copyParameter = instance.LookupParameter("Area")
    copyParameterValue = copyParameter.AsDouble()

    pasteParameter = instance.LookupParameter("Comments")
    pasteParameter.Set(str(copyParameterValue))

transaction.Commit()
```

That wasn't so bad, was it? The important thing to remember is that it is possible to break down the individual Dynamo nodes into logic. Once you figure out the logic, it's easy to translate it across to Python.

Using the PyRevit output to show data to the user

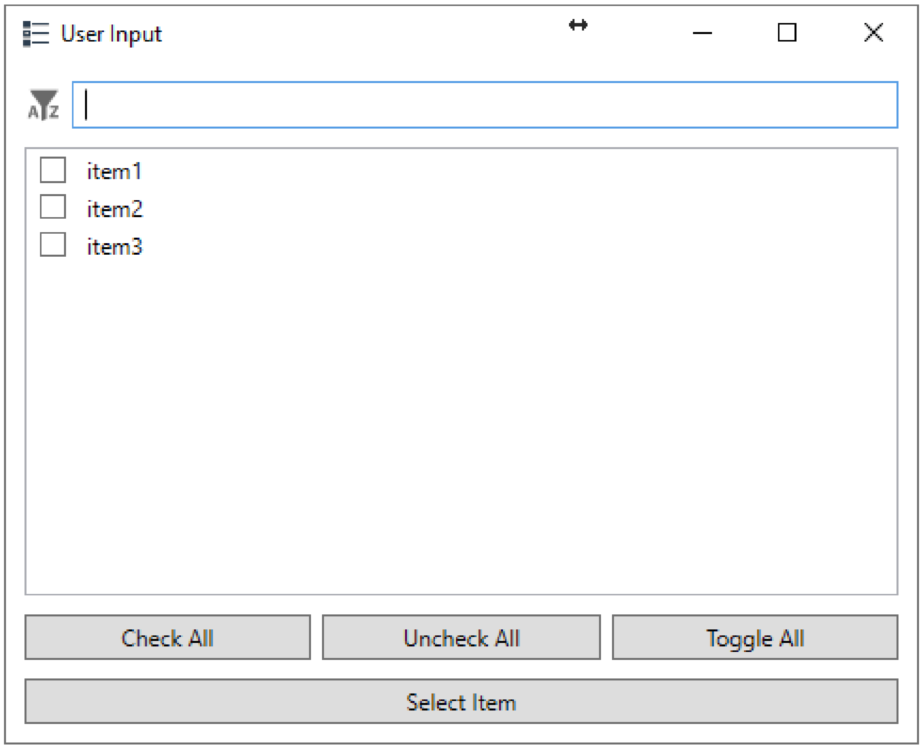
Another piece add to your arsenal, and that is how to show information to the user. We've already touch on creating TaskDialogs, but those are mainly used for alerts.

The most basic way to output data back to the user is similar to what we've been doing all along, the print() method. In RPS, we've used the print() method to print variables to the output window for debugging and testing. We can also use the print() method in our scripts and PyRevit will output the results of the print() method to an extra window that the user can see. This is a very simple, quick, and easy way to push information back to the user. This is what that looks like:



```
View Template: Overall Roof Plan
  ROOF PLAN

View Template: Demolition Plan
  LEVEL 1 DEMO PLAN
  LEVEL 2 DEMO PLAN
  LEVEL 3 DEMO PLAN
  ROOF DEMO PLAN

View Template: Com Check Elevation
  COM CHECK EAST ELEVATION

View Template: Presentation Plan
  PRESENTATION LEVEL 1

View Template: Interiors Specialty Elevation
  PRESENTATION WEST ELEVATION
  INTERIORS SPECIALTY ELEVATION

View Template: Presentation Elevation
  PRESENTATION NORTH ELEVATION
  PRESENTATION EAST ELEVATION
  PRESENTATION SOUTH ELEVATION

View Template: Demolition RCP
  LEVEL 1 DEMO REFLECTED CEILING PLAN
  LEVEL 2 DEMO REFLECTED CEILING PLAN
  LEVEL 3 DEMO REFLECTED CEILING PLAN

View Template: OZ Code Plan
  LEVEL 1
  LEVEL 2
```

PyRevit also has a lot of builtin forms you can use to request input from the user. One example is the select from list form:



In order to use any of the pyrevit forms you first have to import them by including this import statement at the top of your script:

```
from pyrevit import forms
```

Then you can use the following code to get the “Select From List” form. You can assign the form to a variable and then iterate through the variable to get the elements that were selected by the user:

```
items = ['item1', 'item2', 'item3']
selectedItems = forms.SelectFromList.show(items, button_name='Select Item', multiselect=True)
```

And that’s how simple it is. You don’t have to deal with creating a windows form and figuring out how to place all the buttons, set the colors, or any of the confusing code that goes behind creating forms.

Check out this link for lots of other types of forms and output that comes with PyRevit:

https://pyrevit.readthedocs.io/en/latest/articles/outputfeatures.html

Some of them are not documented, but you can probably find an existing PyRevit add-in which you can look under the hood and reverse engineer. The beauty of open source!

Creating Elements

How do we create family instances using the Revit API? It’s not as bad as you think.

First, we’ll need to setup our initial variables such as our doc and our uidoc as we usually do. But in this case, we also want to make sure we capture the ActiveView property of the document as we will need this to specify which view to create the Family Instance in:

```
from Autodesk.Revit.DB import *
from Autodesk.Revit.UI import *

uidoc = __revit__.ActiveUIDocument
doc = uidoc.Document
activeView = doc.ActiveView
```

Once we have done that, let’s take a look at the method we will use to create the new family instance and determine what we arguments it needs. There are several ways to create new family instances, but we will be using the method that requires a location, a symbol (Revit Type), and a structuralType.

```
public FamilyInstance NewFamilyInstance(
	XYZ location,
	FamilySymbol symbol,
	StructuralType structuralType
)
```

We can easily specify the location by using the XYZ() method. We will need to gather the Revit Type using some code. The structuralType in this case is an enumeration and we are going to set it to StructuralType.NonStructural for our plumbing fixture. We need to make sure we add this to the top of the code: from Autodesk.Revit.DB.Structure import StructuralType
So, the last thing we need to do is get the Revit Type we want.

To do this, we first need to create a FilteredElementCollector of the Revit Category we want, in this case, Plumbing Fixtures. Make sure to use the WhereElementIsElementType() method to only select the Revit Types.

```
fec = FilteredElementCollector(doc)
plumbingFixtures = fec.OfCategory(BuiltInCategory.OST_PlumbingFixtures)
types = plumbingFixtures.WhereElementIsElementType()

for type in types:
    typeName = Element.Name.GetValue(type)
    typeTest = '19" Round'

    if typeName == typeTest:
        familySymbol = type
        break
```

Then we can iterate through the types and get the type name using the Element.Name.GetValue() method. Typically we might be able to use the type.Name property but there is a weird IronPython casting issue that doesn't allow us to use that property so we have to make sure we use the other way instead. Don't worry about understanding this at this point, just know that I would first try the type.Name property to see if it worked before reverting to the Element.Name.GetValue(type) method.

We then create a variable called typeTest and set it to the name of the Revit Type we want to use. This will allow us to use the if statement to test if the typeName variable is equal to the typeTest variable we created. If this is true, then we assign the type variable to the familySymbol variable so that we can use it further in our code. Finally, you see there is a "break". This means that the loop will end when the test is true in our if statement because we have found what we are looking for and don't need to waste resources continuing to loop through to the end of the list.

Now we have the Revit type, we can create our family instance. However, any time that you need to edit the Revit project in any way using the Revit API, you have to encompass the changes within a Transaction.

```
transaction = Transaction(doc, "Create Element")
transaction.Start()

#Code goes here

transaction.Commit()
```

Then we can add the NewFamilyInstance code within the Start() and Commit() methods.

```
 doc.Create.NewFamilyInstance(XYZ(0,0,0), familySymbol, StructuralType.NonStructural)
```

We specify the location by using the XYZ() method and giving it a 0,0,0 argument to place it a 0,0,0 within the project. Then we give it the familySymbol parameter we gathered using the FilteredElementCollector. Finally, we set the StructuralType to NonStructural using the enumeration.

This is a good place to introduce the try/except blocks in Python. Try/Except blocks allow us to catch exceptions that might occur. In this case, we are creating a new instance with the familySymbol we had defined previously be iterating through our FilteredElementCollector. However, since we are not instituting a check to see if the type we are looking for is found, there is a chance that our familySymbol parameter never gets created.This is where the Try/Except block comes in. If an error occurs within this block of code, it skips to the except block and executes whatever is there and continues to read the rest of the code. In this case, we are calling the exit() method from sys in order to exit the script after a Task Dialog is shown because we don't want to keep the code running. You'll also see the transaction.RollBack() method meaning that any changes attempted to be made in the try block will be undone.

```
transaction = Transaction(doc, "Create Element")
transaction.Start()

try:
    doc.Create.NewFamilyInstance(XYZ(0,0,0), familySymbol, StructuralType.NonStructural)

except Exception as e:
    TaskDialog.Show("Error", str(e))

    transaction.RollBack()
    sys.exit(0)

transaction.Commit()
```

Note, in order to exit the script, we used the sys.exit(0) method. To use this, we first need to make sure that we add “import sys” at the top of our code.

When you run the code, as long as the Revit Type exists, you will see a new family instance at the 0,0,0 point in Revit of the specified type!

Moving Elements

Now that we’ve created a family instance, let’s manipulate it by moving it.

To move elements we need to use the MoveElements() method from the ElementTransformUtils class in the Revit API. Note, there is also a MoveElement() method for a singular element, but we will go over the MoveElements() method to show you how to create a list of elements to move.

```
public static void MoveElements(
        Document document,
        ICollection<ElementId> elementsToMove,
        XYZ translation
)
```

The MoveElements() method requires a document, an ICollection List of ElementId and an XYZ translation.

An ICollection list is a special C# list type that we need to create using IronPython. To do this, we first have to make sure we import the correct class at the top of our code as such:

```
from Autodesk.Revit.DB import *
from Autodesk.Revit.UI import *
from System.Collections.Generic import List
```

This will allow us to use the List class to create a ICollection List of Element Ids. We will use the Active Selection method we went over earlier to select the elements we want to move and assign their ids to the selectionIds variable. Then we create an empty list that will contain element ids by using the List[ElementId]() constructor and assigning it to the elementList variable. Finally, we will iterate over the selected ids using a for loop and then add the id of the instance to the element id list we created by using the Add() method of the list. This will create a list of Element Ids we can use with the MoveElements() method

```
selectionIds = uidoc.Selection.GetElementIds()

elementList = List[ElementId]()

for id in selectionIds:
    elementList.Add(id)
```

Now that we have all of the arguments we need, we can call the MoveElements() method. Remember that since we are modifying the Revit project, we will need to encompass this within a Revit Transaction:

```
transaction = Transaction(doc, "Move Element")
transaction.Start()

ElementTransformUtils.MoveElements(doc, elementList, XYZ(0, 3, 0))

transaction.Commit()
```

This will move the selected elements 3’ in the Y direction (XYZ(0,3,0)).

Resources

Python Revit API Examples:

Revit Python Shell Scripts
Proving Ground API Examples

Useful Add-ins:

Revit Lookup Tool
Revit Python Shell

Useful References:

API Docs
Archi-lab.net
Autodesk Developer Guide