DIVING DEEPER:
A BEGINNERS LOOK AT PYTHON IN DYNAMO

Sol Amour

Associate

19th June, 2018

Join the conversation #AUCity #AU2018

AUTODESK. UNIVERSITY

@solamour

sol.amour@designtech.io

www.designtech.io/

SOL AMOUR
ASSOCIATE

- Previously worked in Architecture, Landscaping, Industrial Design, and the Construction Sectors
- Joined designtech in 2016
- Specialise in Computational Workflows, Automation and Education
- Work alongside some leading organisations including Heatherwick, Landsec, Arup, Mott MacDonald, Natwest and Nissan
- Passionate about leveraging technology to do laborious and repetitive tasks in order to allow designers to have fun again

About designtech

At designtech we create custom workflows and software to enable businesses to develop their designs and manage their data while cutting down on costly inefficiencies and errors.

INTRODUCTION
IS THIS CLASS FOR ME?

New Horizons Beckon

Do you know how to use nodal Dynamo?

Leveling Up

Do you want to take Dynamo to the next level?

Journey into the Unknown

Are you ready to take the first step into a scripting language?

Growing more Arms

Is there functionality missing in out-of-the-box Dynamo that you want to tap into?

Lets Try and Keep it Simple

Are technical explanations a bit over your head?

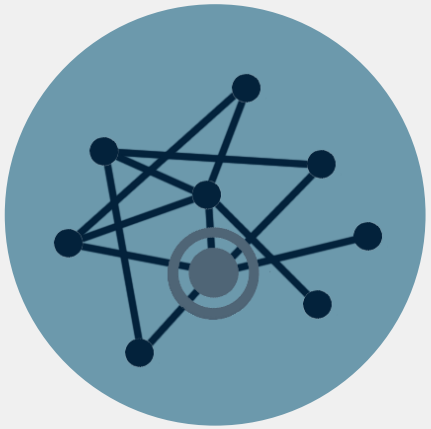
Personal Growth

Are you simply curious?

INTRODUCTION
KEY LEARNING OBJECTIVES

1.

Explore fundamentals such as Syntax, Looping, Object Types and Casting



2.

Understand how Python interacts with Dynamo and External Resources

3.

Discover the limitations of the Python node and how to handle Errors

4.

Learn how to access the Revit API for functionality beyond out-of-the-box Dynamo

What is Python?

Monty Python

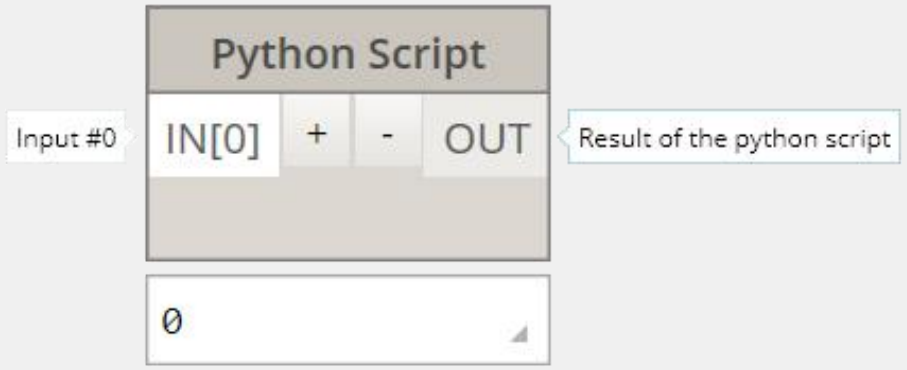
Legible scripting language

Open Source

How does Dynamo interact with Python?

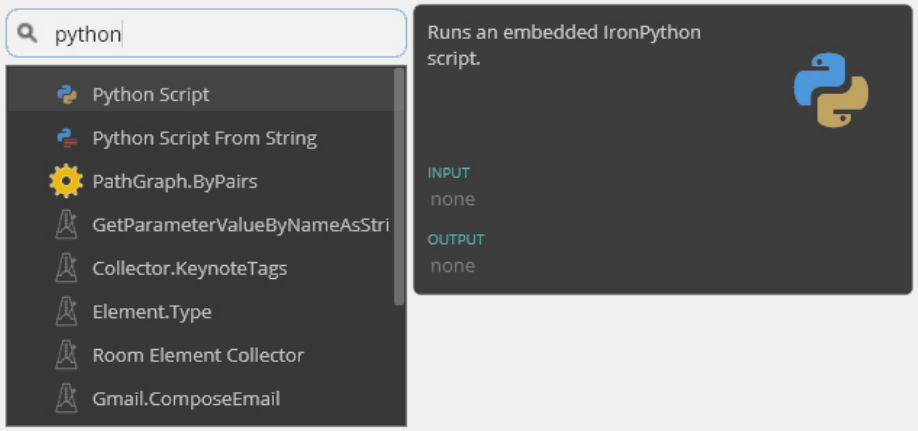
IronPython

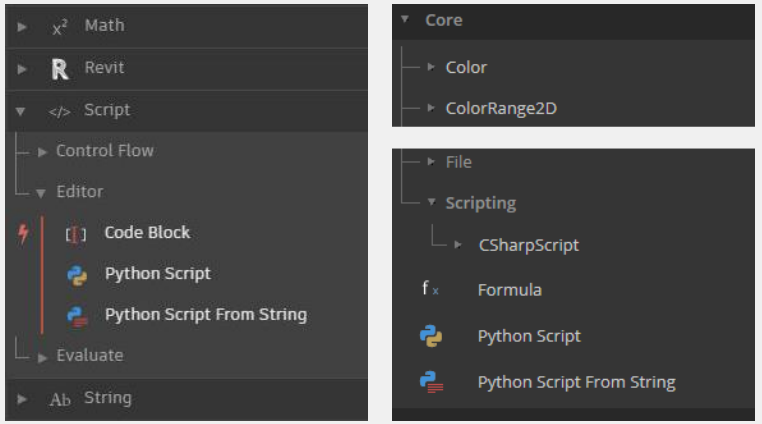
Microsoft .NET framework
Common Language Runtime (CLR)



Python Node

Basic Editor / Integrated Developer
Environment (IDE)

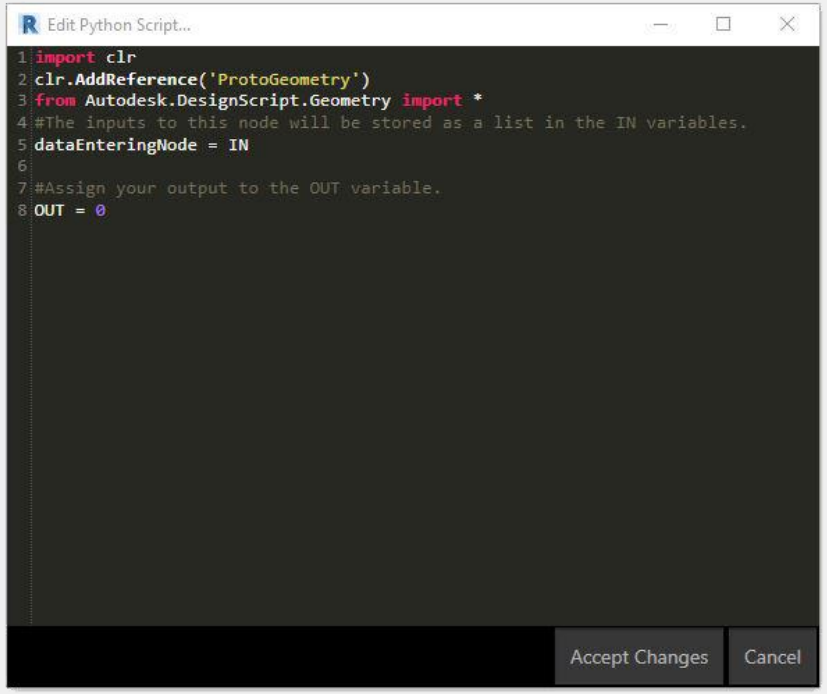




Where can I find Python in Dynamo?

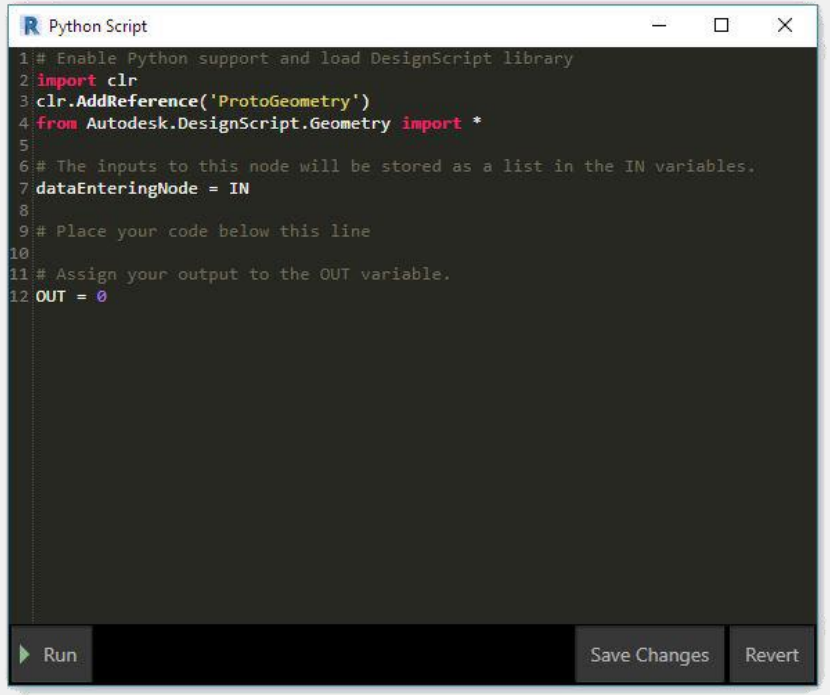
Special node called 'Python Script' or 'Python Script From String'
Accessible through the Search features or Located in the Library:

- [1.3.3] Core -> Scripting
- [2.0.1] Script -> Editor



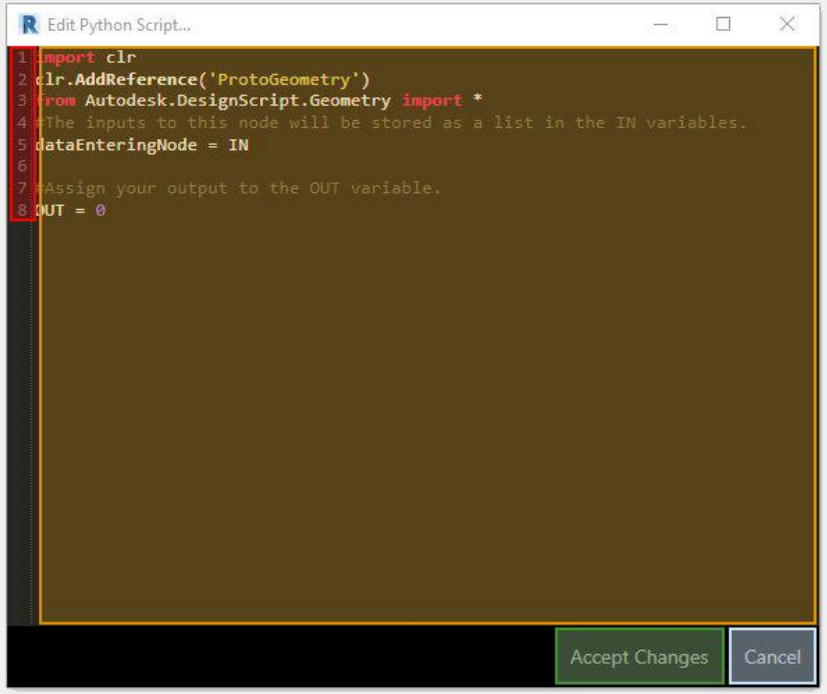
Dynamo 1.3.3

Sits on top of Dynamo
Cannot use Dynamo with Python node window open
Will not run the graph with Python node window open



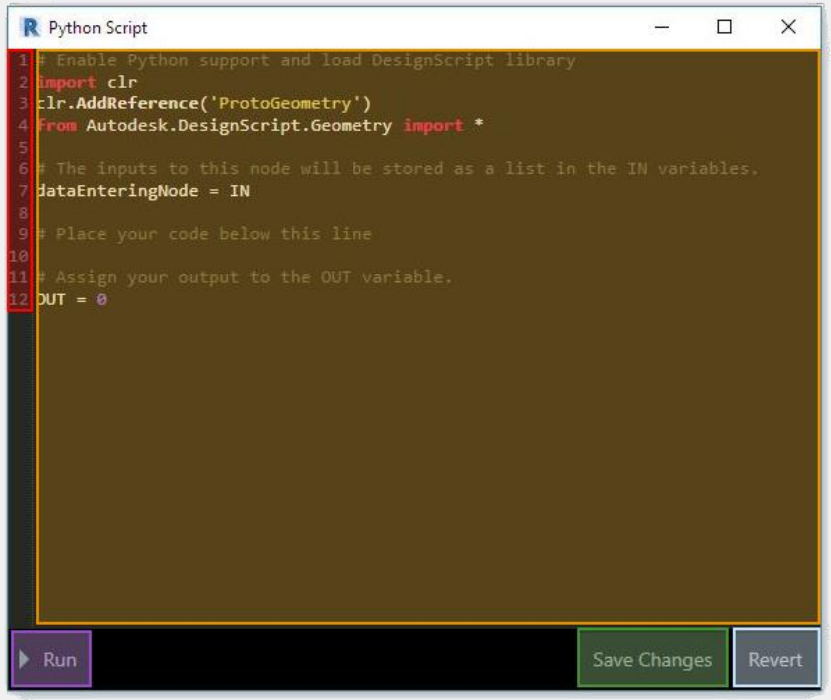
Dynamo 2.0.1

Sits on top of Dynamo
Can use Dynamo while Python node window is open
Can have multiple Python node windows open
Can run the graph with Python node window open



Dynamo 1.3.3

[1] Code line number
[2] Development window
[3] Save and accept changes
[4] Cancel without saving changes

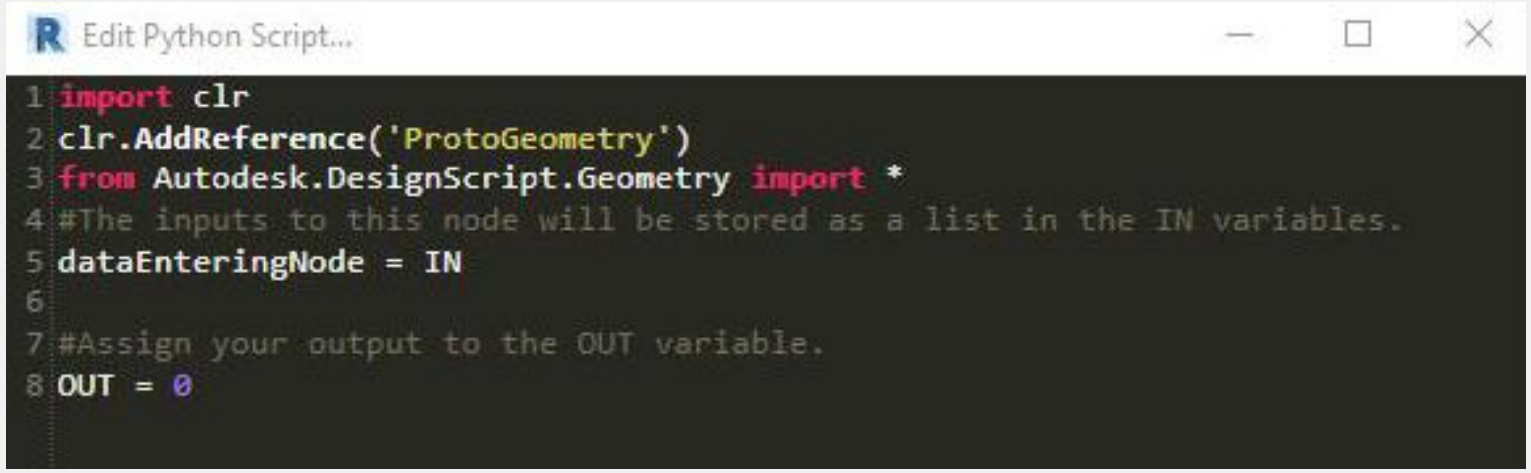


Dynamo 2.0.1

[1] Code line number
[2] Development window
[3] Save and accept changes
[4] Revert to previous save state
[5] Run python script

First Contact

[Explaining the out-of-the-box Boilerplate Code defaults]



[1] Imported Module: Common Language Runtime (CLR) module | Provides access to .NET framework
[2] Added Reference: ProtoGeometry module added so we can use Dynamo Geometry
[3] Demarcating loaded classes: From the ProtoGeometry module importing all of the Geometry classes using asterisk (*) to stipulate that we want all of them.
[4] Commented code: Uses the Hashtag (#) to tell the Node to ignore this line.
[5] Variable tethered to Input Ports: A legacy line of code that is typically superseded by IN[0] | IN[1] et al.
[6] Blank line : Does nothing | Increases legibility
[7] Commented code: Uses the Hashtag (#) to tell the Node to ignore this line.
[8] Variable tethered to Output Port: Singular output allowed from Python node | Can attached a list of objects to OUT port

Was that a little daunting?

...intention is that by the end of this Lab it won't be anymore! So lets have some fun ☺

Fundamentals

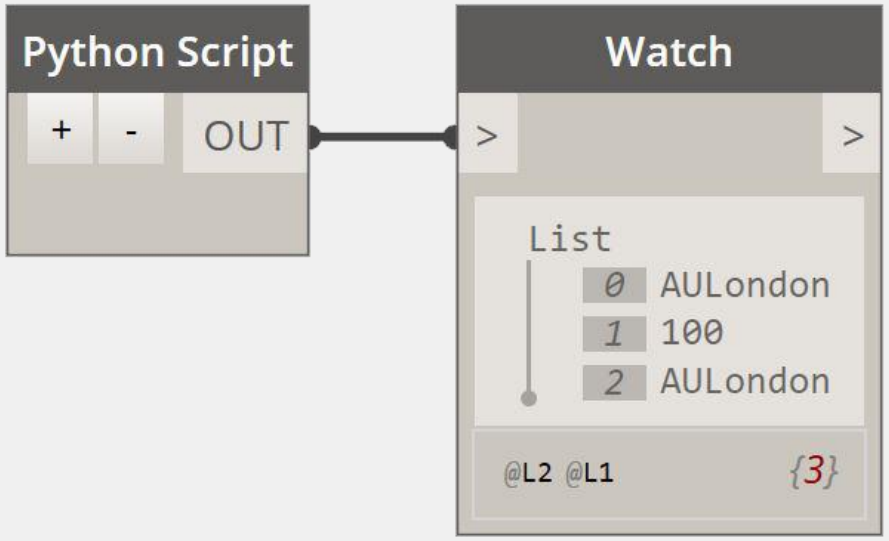
[Exploring the Building Blocks of Python]

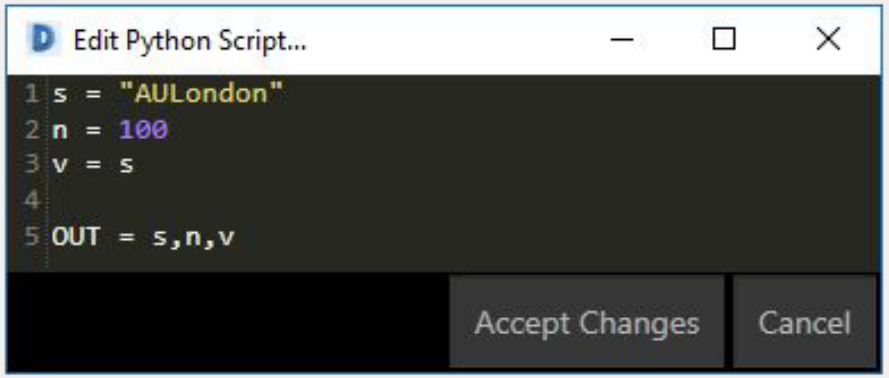
CONCEPT
VARIABLES

Variable (Variable) – You name stuff and shout (call) to it to come over when you need it:

```
s = 'AULondon'
n = 100
v = s
```

Assign using the 'variableName = item' syntax
Variables are used to store information in memory to then be referenced and manipulated in Python
They allow us to label data with a descriptive name to make sense of our code.
They act like containers that hold information, callable by their name
They provide clarity when others read your code (If you are diligent at naming!)
You can assign a variable to another variable





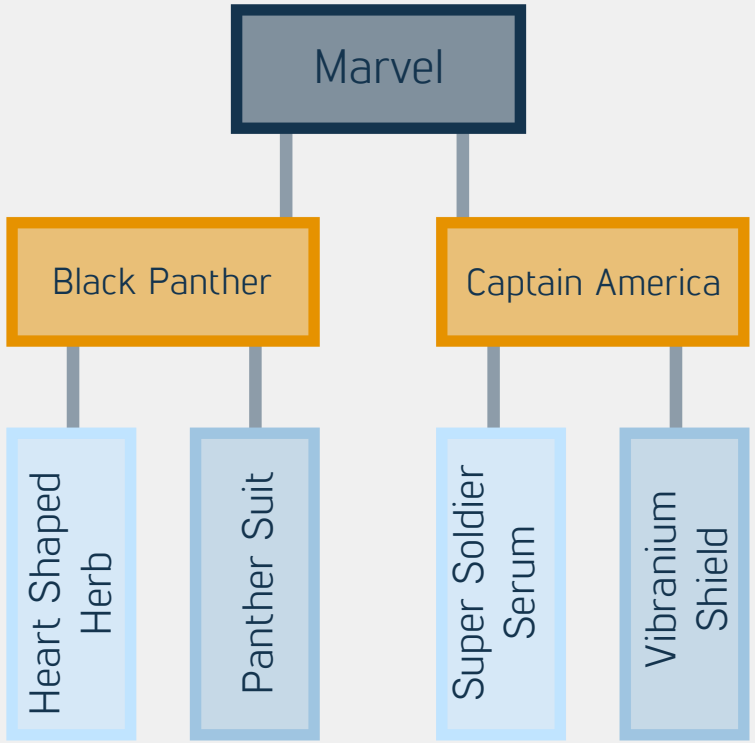
CONCEPT
NAMESPACE

A **namespace** is an abstract container to collect a logical grouping of unique names (identifiers), where that name is identifier only with it's namespace. This is best illustrated with an example:

Marvel (**Namespace**)

- Captain America (**Classes**)
  - Super-Soldier Serum (**Methods**)
  - Vibranium Shield (**Properties**)

Namespaces are hierarchical.
They are a way to implement scope (Define boundaries)
Can be understood as a 'Parent : Child' relationship



OBJECT TYPES

NUMBERS

Int (Integer) – Whole Number:

int = 10

Float (Double) – Decimal placed (floating point) number:

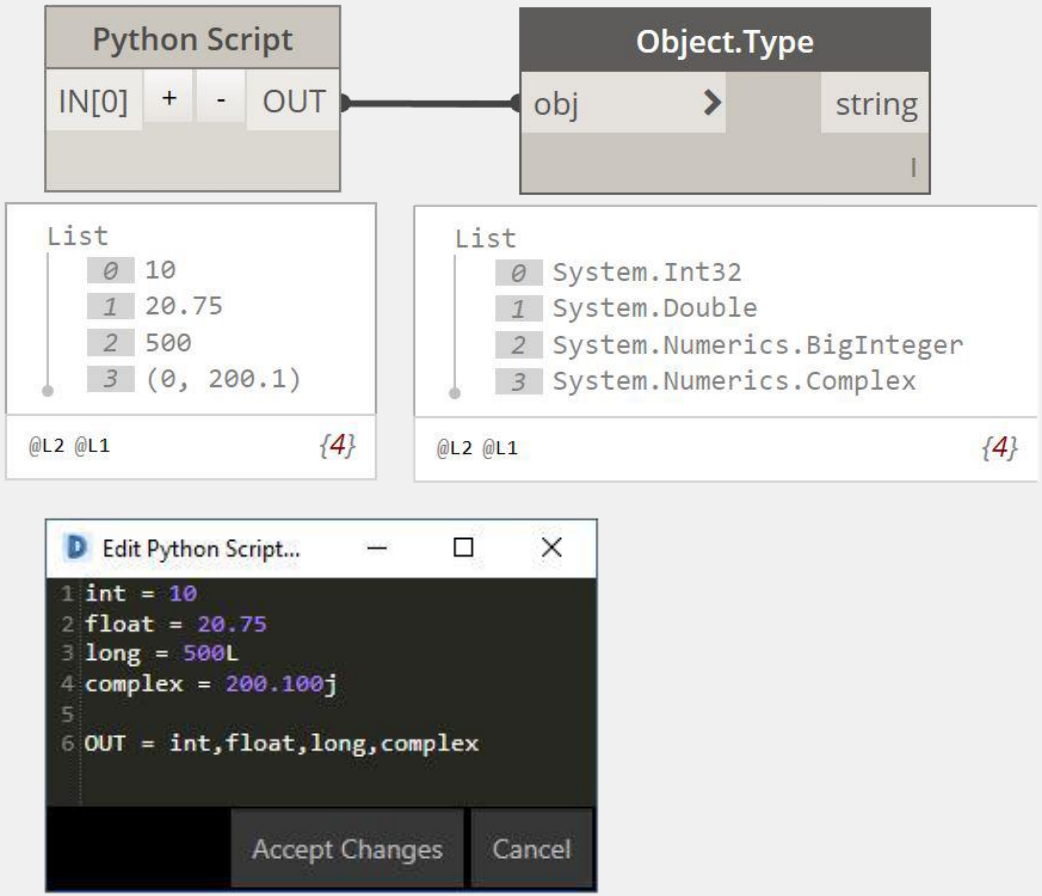
float = 20.75

Long (Long Integer) – Can be Octal / Hexadecimal:

long = 500L

Complex (Real + Imaginary part) – Imaginary written with a J-suffix:

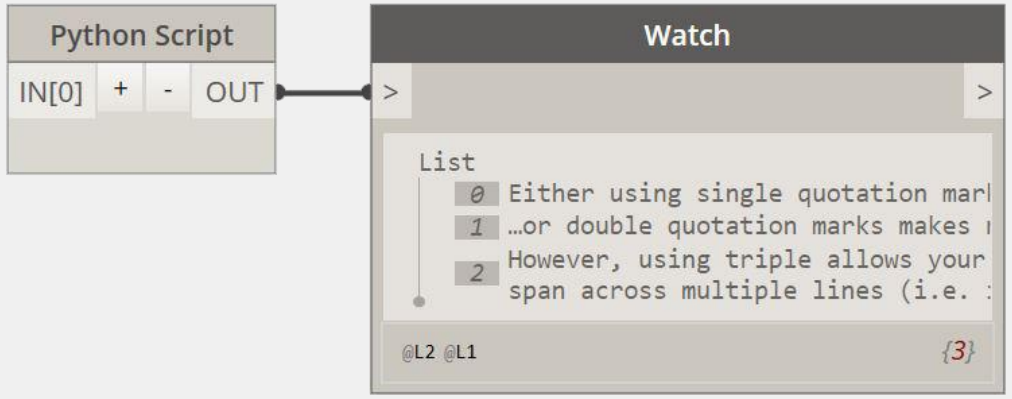
complex = 200.100j

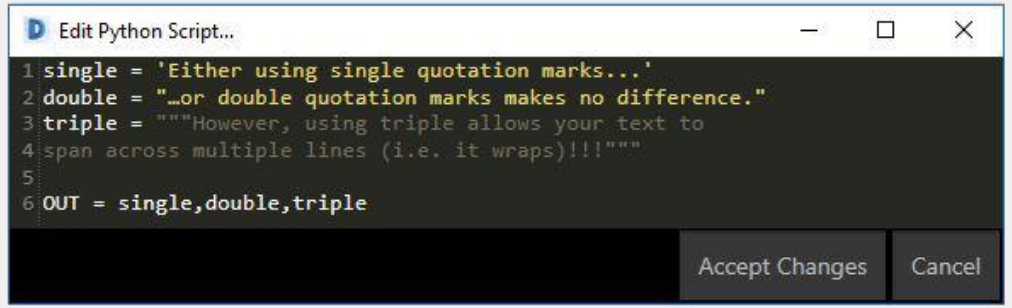


OBJECT TYPES
STRINGS

String (String) – Python for 'Text'. Enclose inside of quotation marks:

```
single = 'Either apostrophes...'
double = "...or quotation marks makes no difference."
triple = """However, using triple quotation marks allows your
          text to span across multiple lines ( i.e. it wraps )!!!"""
```





OBJECT TYPES

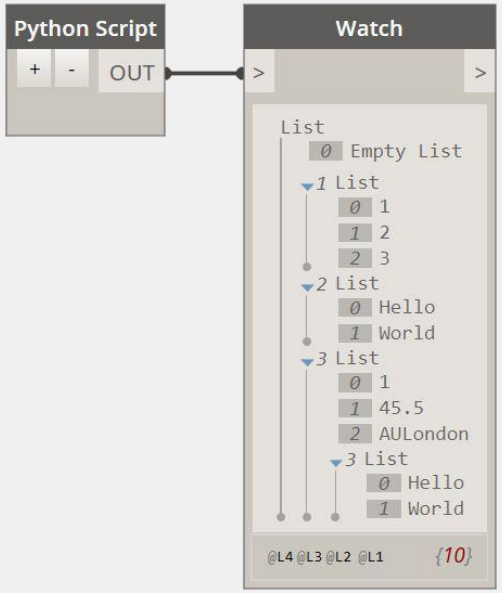
LISTS

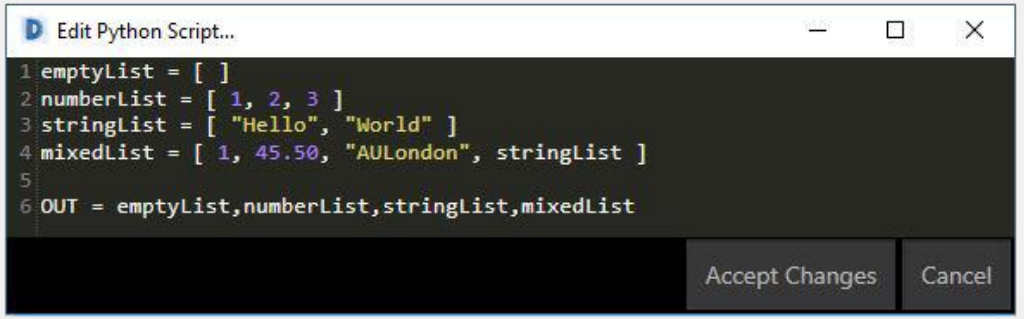
List (List) – A changeable (mutable) ordered container of elements:

```
emptyList = [ ]
numberList = [ 1, 2, 3 ]
stringList = [ 'Hello', 'World' ]
mixedList = [ 1, 45.50, "AULondon", stringList ]
```

Lists are declared by a named variable (var) followed by equals (=) and square brackets ([ ])

Note: The way I am naming my variables is called Camel Case, where the first word is all lowercase and all subsequent words have their first letter capitalised. This is one of the preferred naming conventions discussed in the Python Style Guide.





OBJECT TYPES

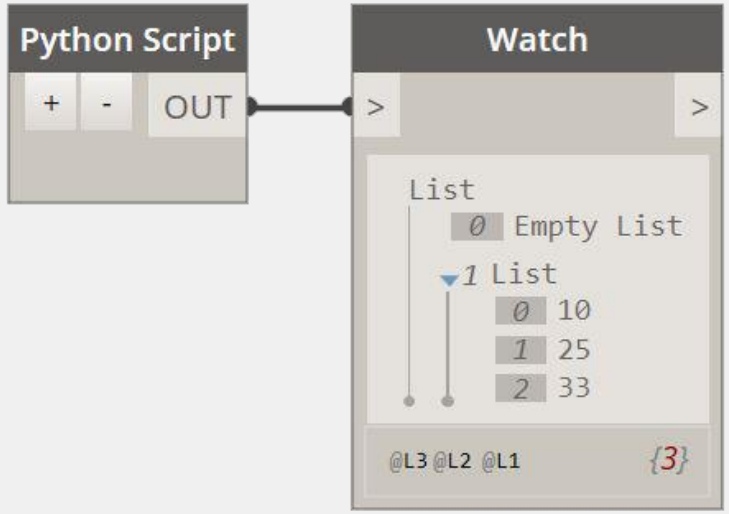
TUPLE

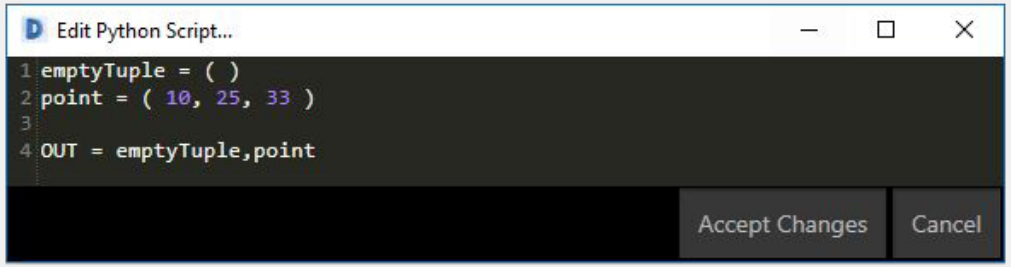
Tuple (*) – A collection of objects separated by a comma:

```
emptyTuple = ( )
point = ( 10, 25, 33 )
```

Tuples are declared by a named variable (var) followed by equals (=) and parenthesis (( ))
Tuples are immutable (Non-changeable) collections
Tuples are finite ordered Lists (They have a set length)
Tuples will appear as Lists in Dynamo

* There is no direct equivalent to a Tuple in Dynamo. However, you can consider the Constructors of Creators as Tuples (i.e Point.ByCartesianCoordinates(x, y, z))





OBJECT TYPES
DICTIONARY

Dictionary (Dictionary.ByKeysValues*) – A collection of 'key : value' paired objects:

emptyDictionary = { }

myDictionary = { "Room 01" : 100, "Room 02" : 125 }

Dictionaries are declared by a named variable (var) followed by equals (=) and braces ({ })

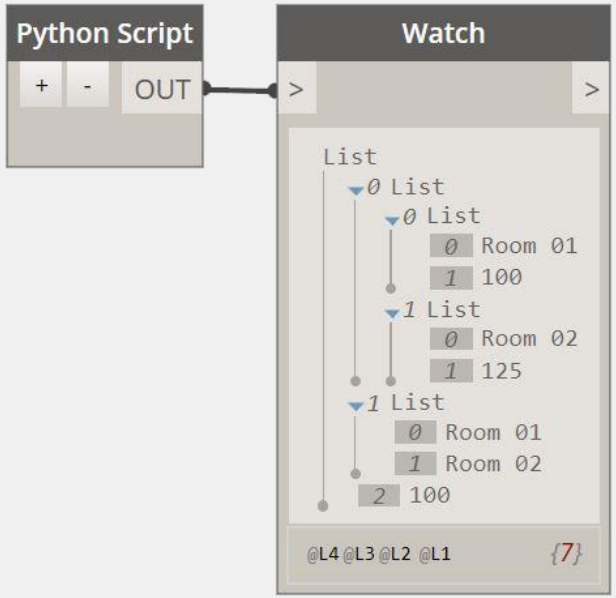
Dictionaries are unordered lists. They will 'shuffle'. You get correct values by calling their respective 'keys'

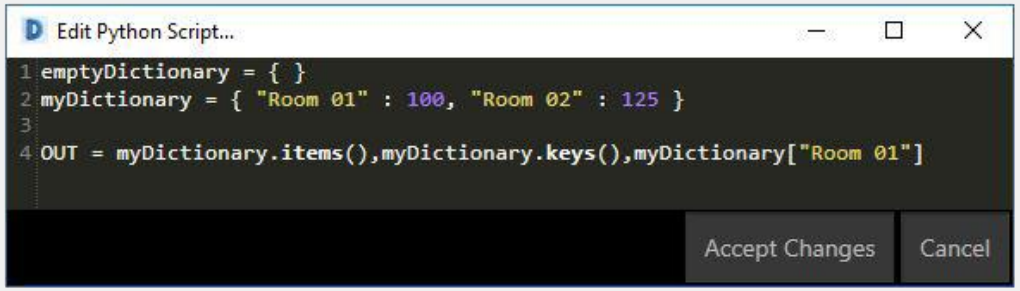
Values are only returned when you query the 'key' index

myDictionary['Room 01']

Get the items (Keys and Values) of a Dictionary by calling

'myDictionary.items( )'





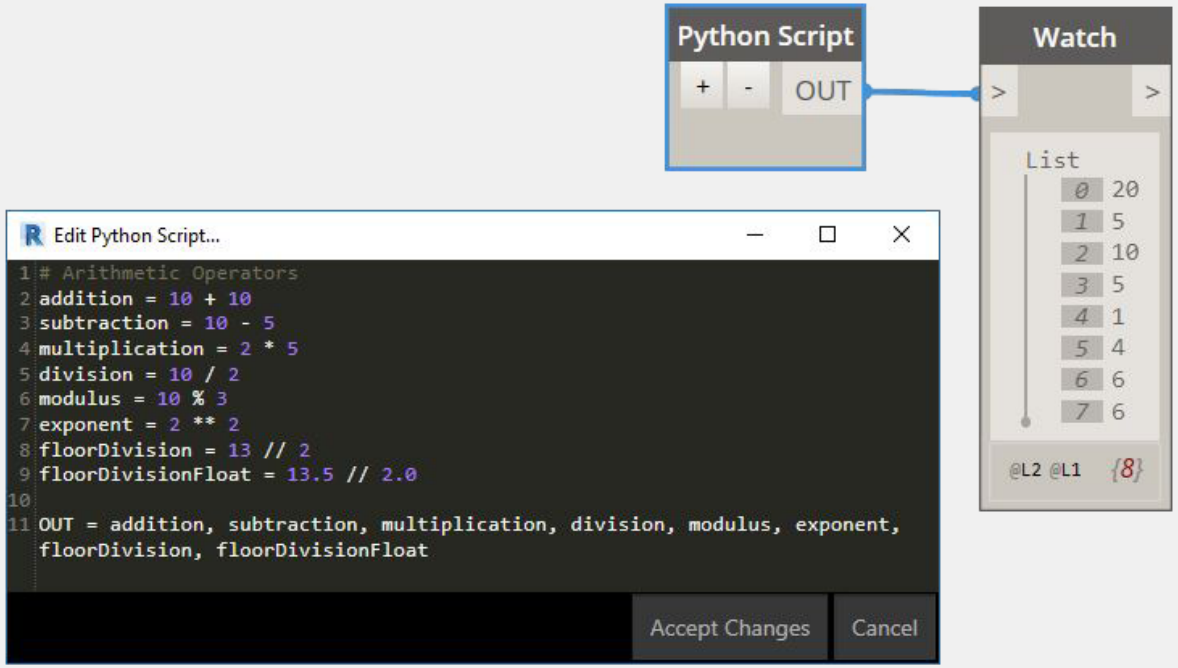
* Dictionary nodes available beginning in Dynamo 2.0

OPERATORS
ARITHMETIC

In Python we have Operators which manipulate a value / variable. Arithmetic covers the mathematical operations and are as follows syntactically:

```
addition = 10 + 10
subtraction = 10 – 5
multiplication = 2 * 5
division = 10 / 2
modulus = 10 % 3
exponent = 2 ** 2
floorDivision = 13 // 2
floorDivisionFloat = 13.5 // 2.0
```

Read more about Operators here.

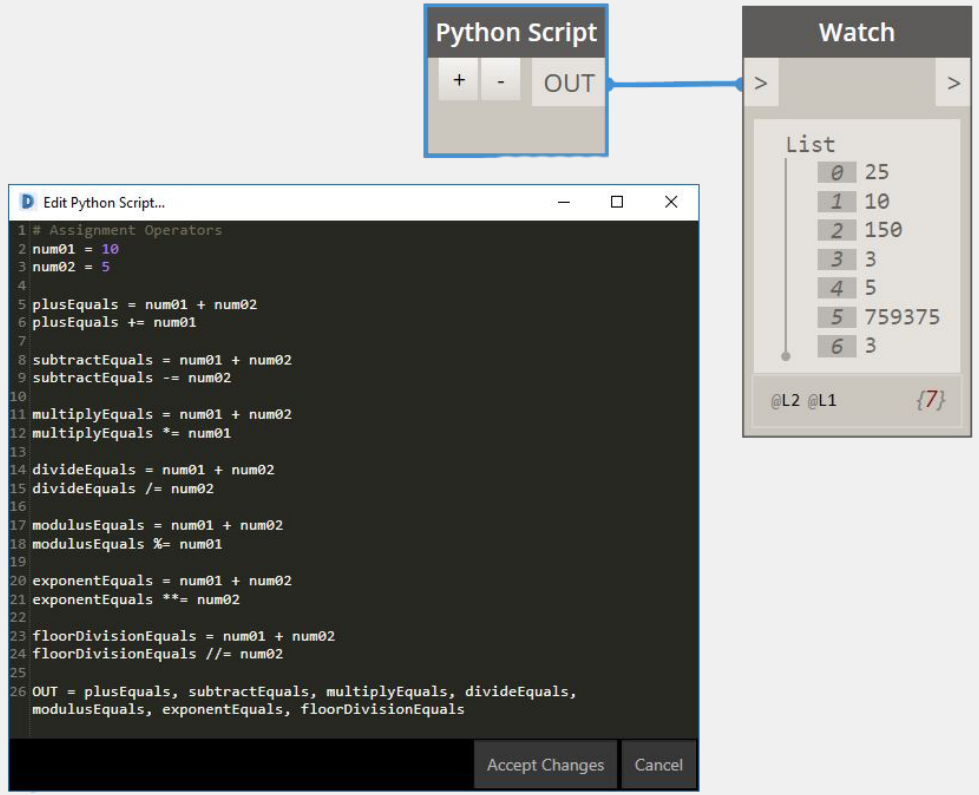


OPERATORS
ASSIGNMENT

Assignment Operators allows us to assign (In various ways) information to variables. They are allow short-hand data manipulations and are as follows syntactically:

- Equals (=): Assigns values from right side to left side
- Add And (+=): Adds right value from left and assigns result to left operand
- Subtract And (-=): Subtracts right value from left and assigns to left operand
- Multiply And (*=): Multiplies right value from left and assigns to left operand
- Divide And (/=): Divides right value from left and assigns to left operand
- Modulus And (%=): Takes modulus using two operands and assigns to left operand
- Exponent And (**=): Performs exponential (power) calculation on operators and assigns to left operand
- Floor Division And (//=): Performs floor division on operators and assigns to left operand

Read more about Operators here.



OPERATORS

MEMBERSHIP & BOOLEAN

Python has two membership operators: Whether or not something is inside a container (list).

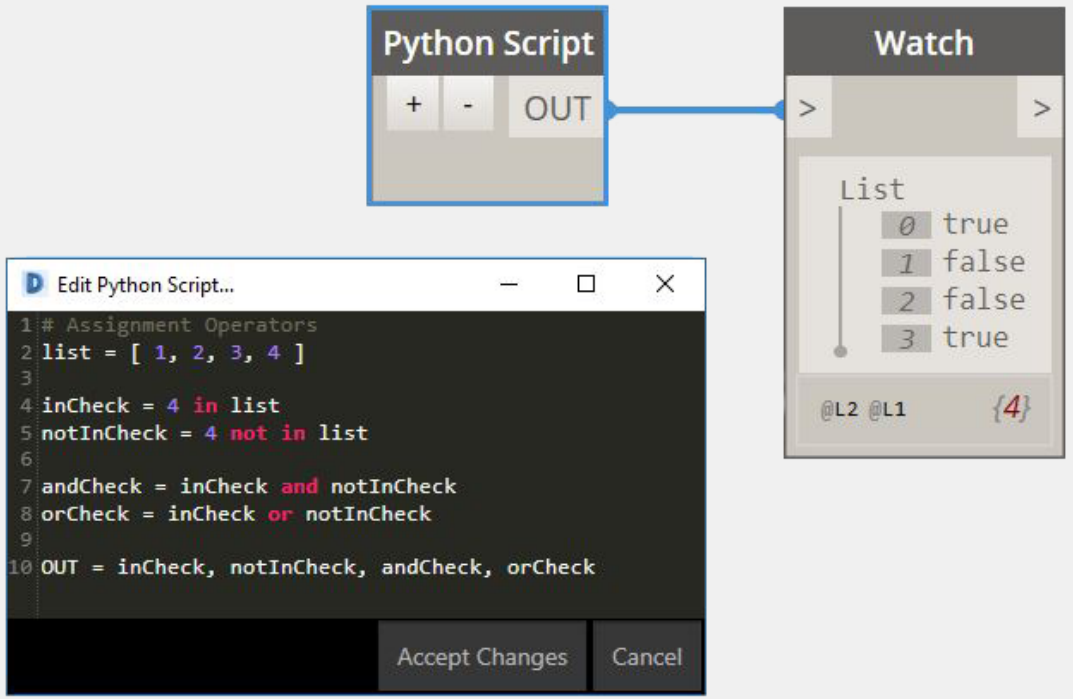
in = Evaluates to true if it finds a thing (variable) inside the container, false if it does not.

not in = Evaluates to true if it does not find a thing (variable) inside the container, false if it does.

Boolean operators will check if a list of values are True or False and will evaluate as follows:

and = Evaluates to true if all elements are True, false if it does not.

or = Evaluates to true if any element is True, False if it does not.



Read more about Operators here.

OPERATORS
COMPARISON

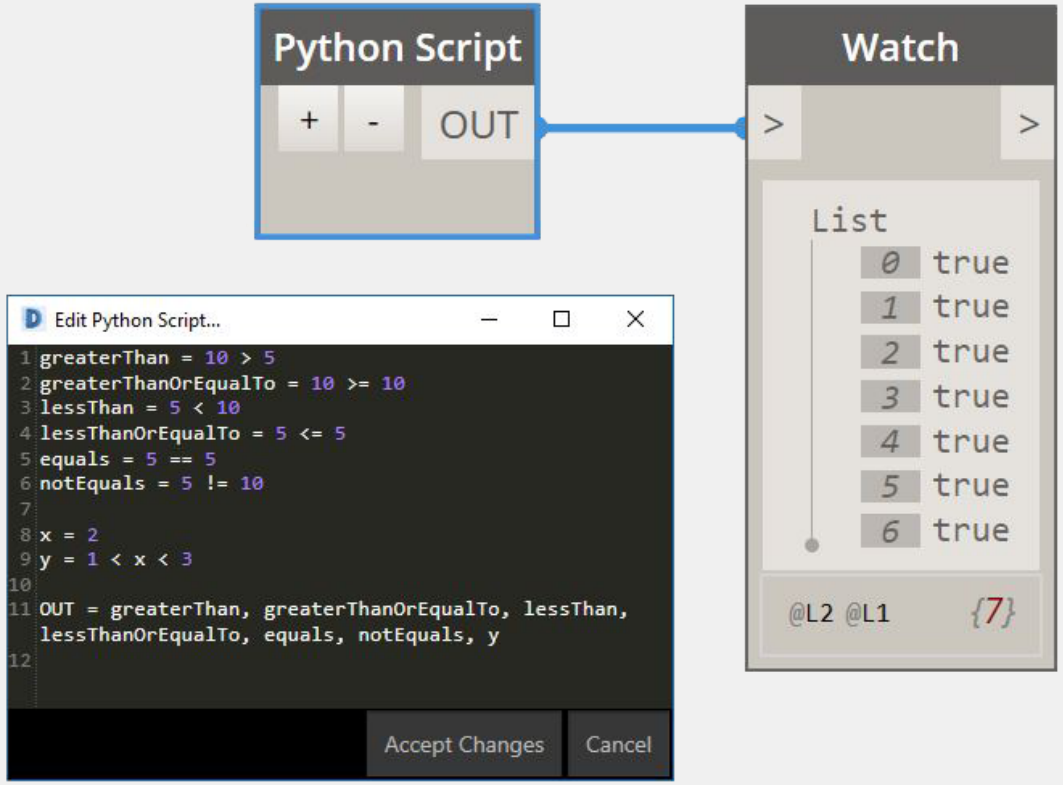
Comparison Operators allows us to check relationships between things (variables) and result in either a True or a False. Syntactically they are:

```
greaterThan = 10 > 5
greaterThanOrEqualTo = 10 >= 10
lessThan = 5 < 10
lessThanOrEqualTo = 5 <= 5
equals = 5 == 5
notEquals = 5 != 10
```

Note: You can chain comparison operators.

```
x = 2
y = 1 < x < 3
```

Read more about Operators here.



SYNTAX
RESERVED KEYWORDS

Some words in Python are reserved – which means you cannot use them as variable names. They each have special interactions inside the Python node

Keywords are Case Sensitive
A null in Dynamo is '**None**' in Python (A nothing)

We cover some of these later in the Lab

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	

SYNTAX

INDENTATION AND CLARITY

Python uses indentation instead of bracketing to define blocks of code. This gives a natural level of clarity and legibility:

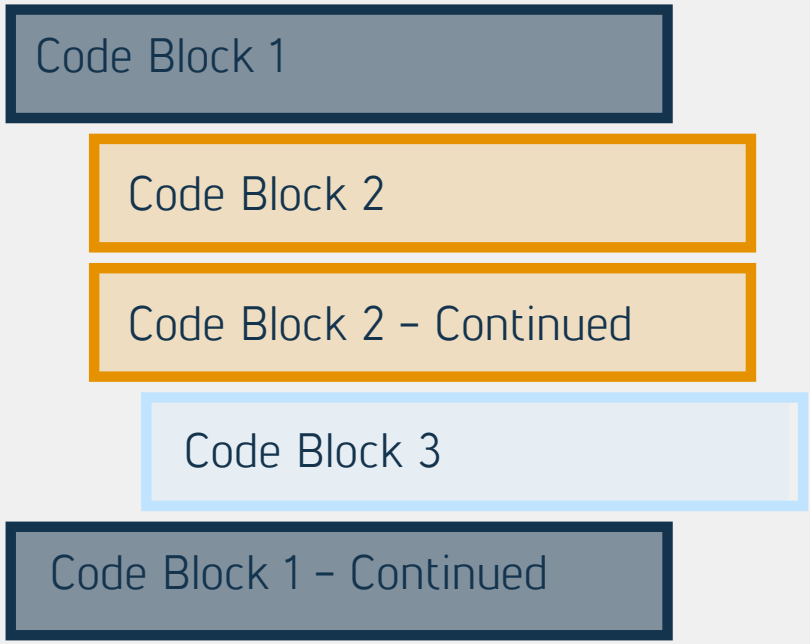
- Indentation is either through spaces (Typically 4x)
- or tabs

Indentation is required in Python – A style choice in most other languages

Indented code blocks will fully execute before 'stepping back' to the next line's unindent

Python has a Style Guide called Pep-8 that sets out best practise – readability is prized in Python

Choose one indentation style and stick with it for consistency



Read more about the Python Style guide here.

SYNTAX

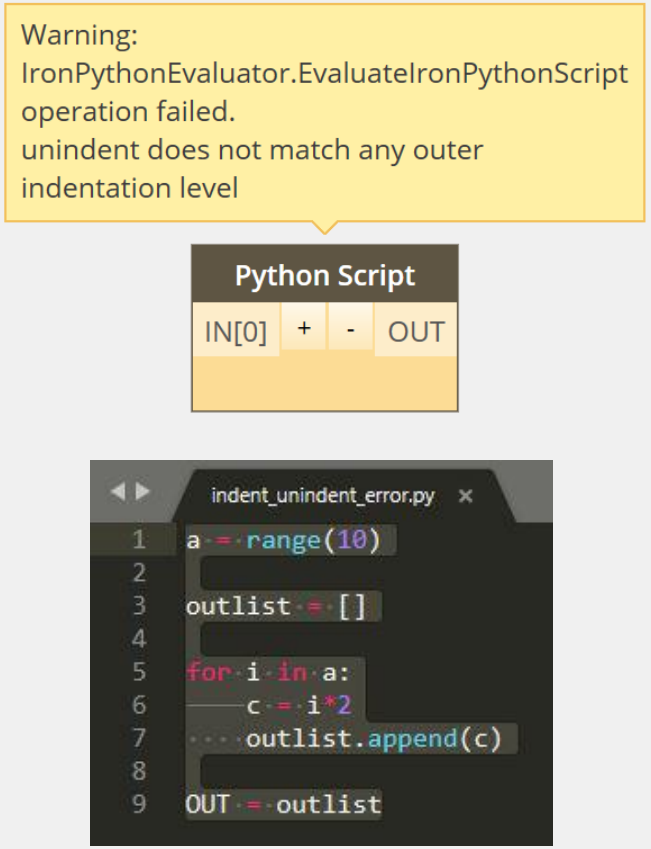
MIXED INDENTATION ERRORS

The Python node cannot switch between indentation styles inside a single block of code (Such as a 'For Loop') or it will fail and present the error shown.

Typically this error will refer to unindent (The opposite of indent) where the evaluator (The thing that reads and executes your script) cannot make sense of what is written and in this case you could have mixed indentation styles.

If you open up your Python code in an editor (Such as Sublime*) you can see the variance in between indentation styles – the dots refer to spaces and the lines refer to tabs.

* Sublime is used to showcase the example.



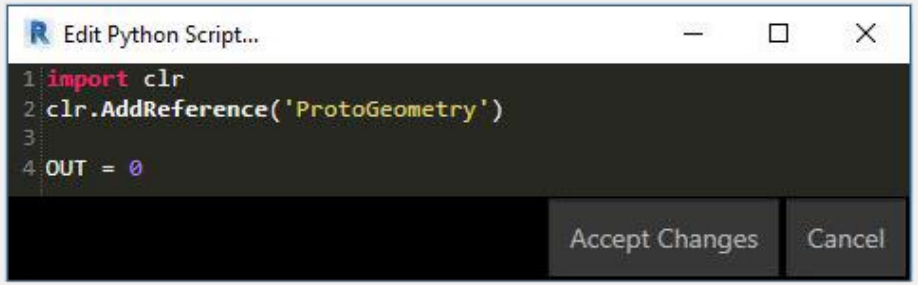
Importing / References

[Leveraging other peoples work to get places faster]

A HEAD START
ADDING A REFERENCE

In Python we have to add references to external content (Modules) to access their functionality. A reference is like a box of Lego – you know where the Lego is, but you haven't yet unpacked the pieces – e.g. If we wish to use Dynamo Geometry we need to add in the 'ProtoGeometry' dll*:

clr.AddReference('nameOfModule')



The module is simply an invitation – It hasn't turned up to the party yet
Certain modules don't need an invitation have free access (Don't need to have their reference added – called 'Built-in Modules')
As with all of Python, capitalisation matters
Only default CLR modules and Dynamo .dll's can be added in this way – We have to point towards the location other .dll's if we wish to use them**

* A dll is a Dynamic Link Library – a file that contains instructions which other programs can use to do certain things. In other words – we can re-use content. Hurray for not re-inventing the wheel!
** We do this by using the sys.path method, explained in the next slide.

USING EXTERNAL LIBRARIES

SYS.PATH

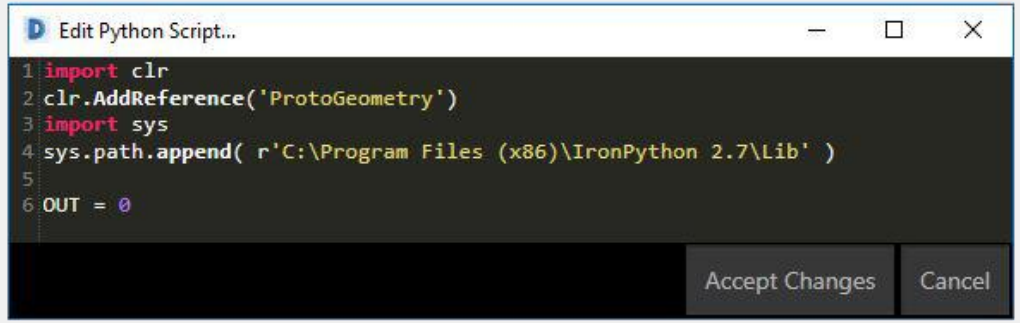
IronPython can access a whole bunch of modules that come with the IronPython Library as bundled up with Dynamo. They are located inside the 'C:\Program Files (x86)\IronPython 2.7\Lib' path. We need to tell Python to look here for the modules. To do this, we need to append said path to the sys module as follows:

```
import sys
sys.path.append( r'C:\Program Files (x86)\IronPython 2.7\Lib' )
```

For non built-in modules, the Python Interpreter will search for the imported module in a list of directories which have been added to (appended) to 'sys.path'

We preface the path with an 'r' (Raw string) that allows us to avoid the use of Escape Characters* in our directory path.

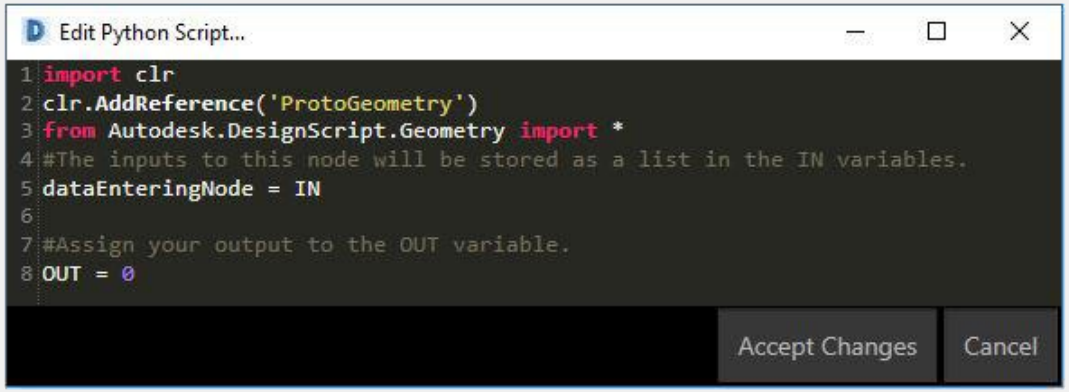
* An Escape Character will be ignored by Python and has special interactions. Find out more here



TOOLING UP
IMPORTING

After referencing a module, we need to Import that module to use it's functionality – You are getting the lego out of the box now and can start using it to make things. If we wish to use contents of the Dynamo Geometry (ProtoGeometry) module, we need to import as follows:

```
from Autodesk.DesignScript.Geometry import *
```



The difference between using 'import X' and 'from X import Y' is discussed on StackOverflow (A very kick-ass site you should get to know) here

* Note the image here is how the node will look out of the Box in Dynamo 1.3.3. This is also the default call for the ProtoGeometry module calling the entire namespace (class) path so as to avoid potential conflicts with other modules.

LIGHTNESS OF TOUCH

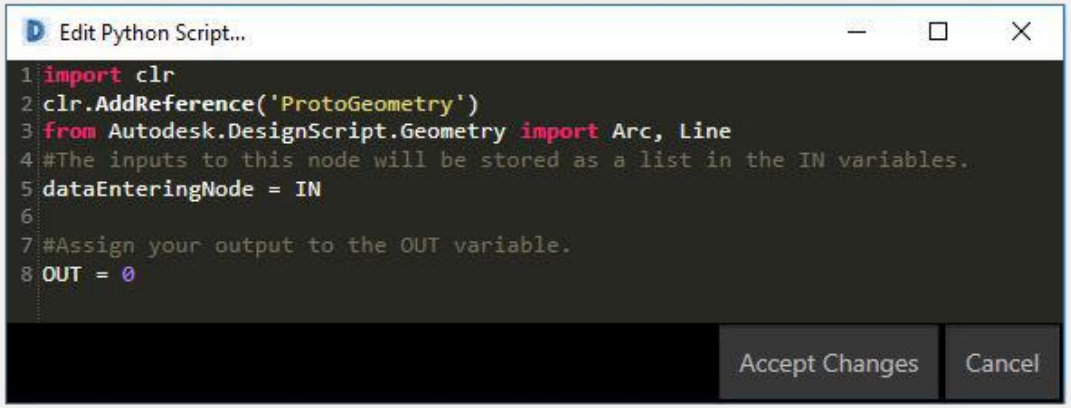
FROM

In the previous import exemplar we use an asterisk (*) to import everything inside that module. In order to retain a lightness of touch and only import relevant items, it's considered best practise to only import elements you are actually using – this stops namespace (Cone, Arc, Line etc) conflicts and the need to write out stuff in full when wanting to use it: e.g. If we only want to use Lines and Arcs from Protogeometry we write the following:

```
from Autodesk.DesignScript.Geometry import Arc, Line
```

If we then use another module that uses a namespace (object) with Geometry.Line we don't have to write DesignScript.Geometry.Line(pt1, pt2) when calling the DesignScript version or X.Geometry.Line(pt1, pt2) when calling the X version

You can always add in more namespaces (objects) when you require them at a later date



Our First Useful Node

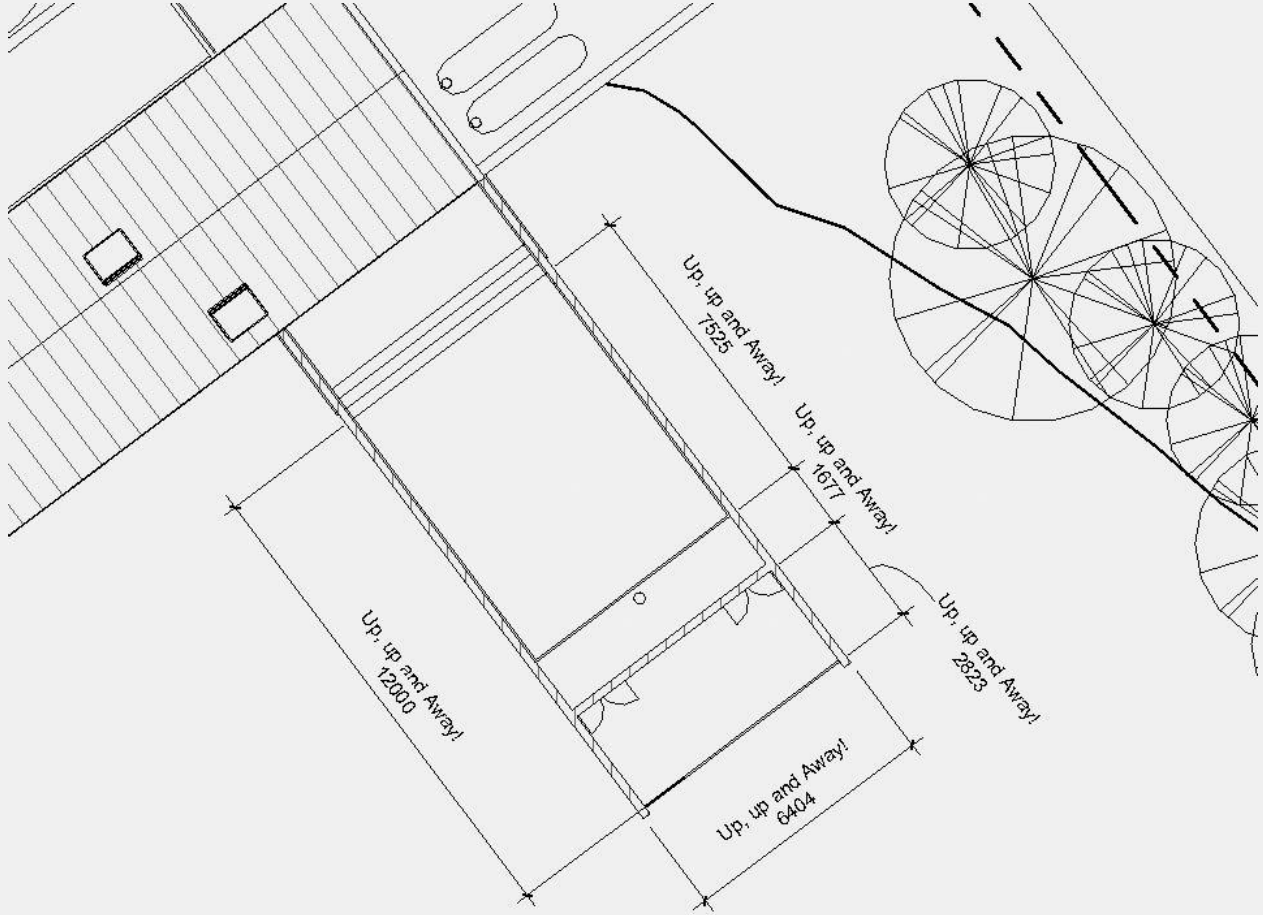
[Creating a Python Node that changes our Revit Project]

MOVING FORWARDS

THE END RESULT

The first Python node we are going to create will allow us to set Dimension text to the **‘ Above’ text field** on a dimension.

Note: This node already exists inside of Dynamo – so we are re-creating content for educational purposes.

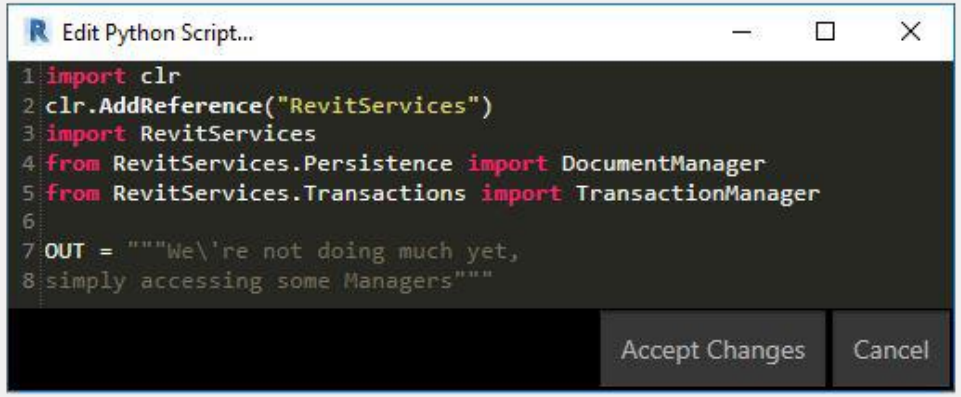


SETTING THE FOUNDATION
IMPORTING MANAGERS

Looking at a tangible example, we are going to build a Python node that allows us to set the '**Above**' text on a Dimension automatically using Python. To do such, we need to import two Managers: The **Document** and **Transaction** Managers.

The **Document Manager** allows us to access the Revit Project (Document)
The **Transaction Manager** allows us to make changes to the Revit Project (Document) – All changes inside of Revit are run via Transactions and this manager is the Dynamo implementation (Wrapper) around that core Revit functionality that stops us from inadvertently breaking things
Note: We only need a Transaction if we are **changing things** in the Revit Project, not if we are just getting stuff (Like querying a property)

Document Manager code available here
Transaction Manager code available here

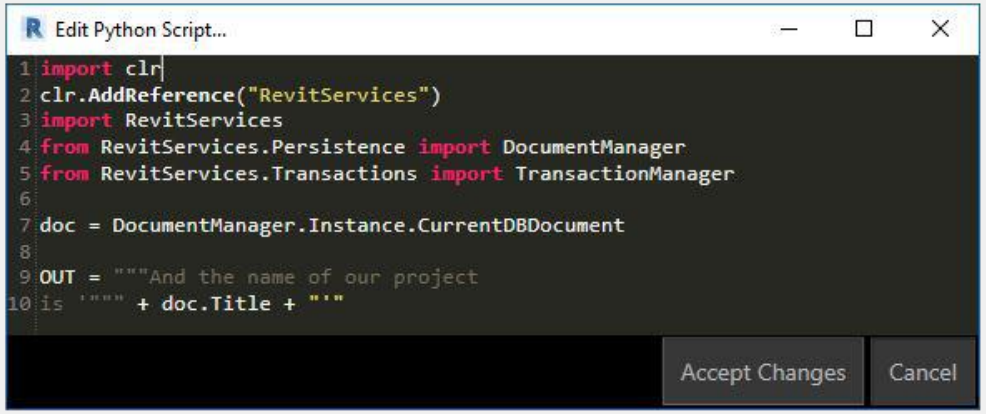


MAKING LIFE EASIER

SETTING VARIABLE NICKNAMES

After we have imported the Document Manager we gain access to the Document. However, this access is through a long namespace object entitled **'DocumentManager.Instance.CurrentDBDocument'** which gets tiresome to write or copy + paste and takes up a lot of screen real estate. One way we can cut down on repetition is by giving that entire object it's own variable 'nickname' – that way we can reference that entire object simply by typing **'doc'**.

Be careful with your naming conventions as a huge chunk of writing good code is in the way you name: We could use a nickname of 'd' but that makes less sense legibly than 'doc'



LETTING THE OUTSIDE IN
INPUT PORTS AND WRAPPING

We then want to set up our two input ports: One for the Dimension Elements themselves and one for the Text we wish to put above them. To allow data to enter a node, there is a special variable called **'IN'** – this correlates with the toggled input ports outside the Python textual environment. To set data to a variable, we simply call it with the syntax:

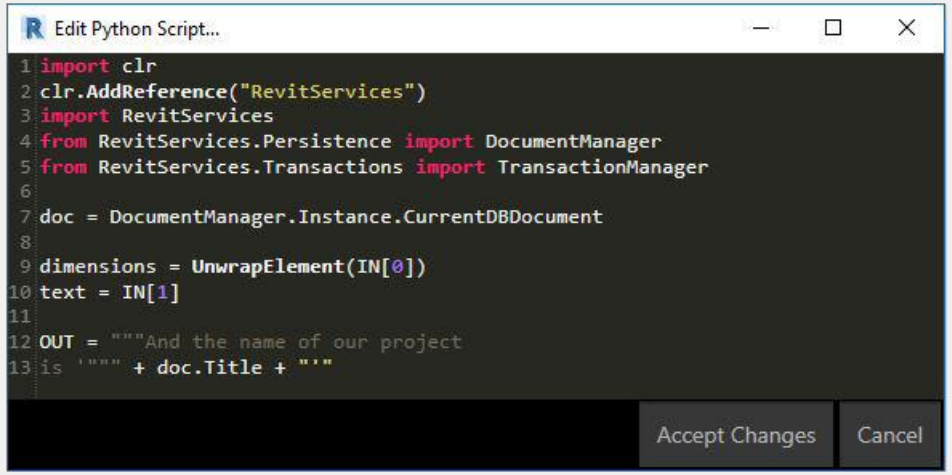
variableName = IN[portNumber* **]**

If any of the information coming through is a Revit Element which we want to change **inside** the Revit Project (Document), then we need to unwrap it**. Dynamo wraps up Revit elements in order to use them so we need to unwrap with the following syntax:

variableName = UnwrapElement(IN(portNumber))

*As with DesignScript, Python and the Python Node are zero-indexed. Read more about zero indexing here.

** Read about Unwrapping on this blog post made by the Dynamo team.

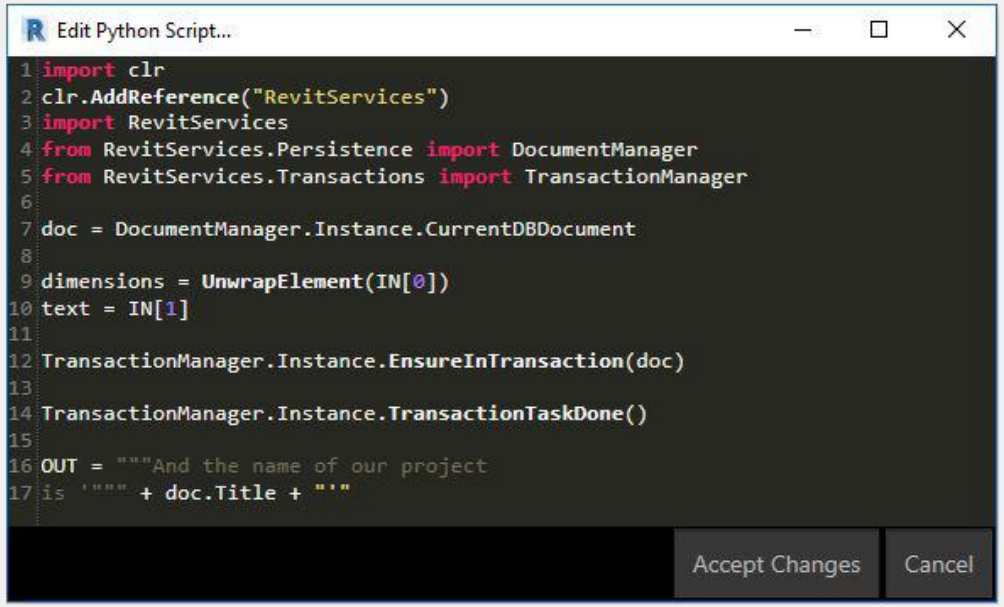


CAN'T TOUCH THIS
TRANSACTIONS

Because we want to manipulate Dimensions inside of Revit, we need to use Transactions to do so. Transactions are utilised inside of Revit for all actions, and will rollback (Undo) if something goes wrong. Dynamo uses a Transaction Wrapper (Special version) as discussed above in **'Importing Managers'**. To use Transactions inside of Dynamo, we use the following syntax:

TransactionManager.Instance.EnsureInTransaction(doc)
<do some stuff to a Revit Project here>
TransactionManager.Instance.TransactionTaskDone( )

Note we have to call the Transaction on our Revit Project (Handily nicknamed to 'doc' previously)



Looping & Conditionals

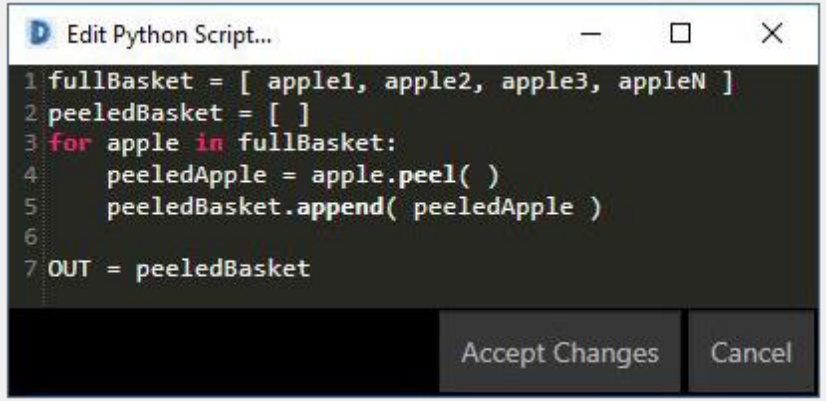
[Iterating a function across a list of items if Conditions are met]

FOR LOOP

WHAT IS A LOOP?

A loop is when you want to do something (Perform an action) to every item inside of a list (Iterate across a list) – Say you have a basket of apples, and you want to peel them all: In looping terms you would take every single apple, one by one, peel it, then add it to another empty basket. The syntax in Python this would be:

```
fullBasket = [ apple1, apple2, apple3, apple4 ]
peeledBasket = [ ]
for apple in fullBasket:
    peeledApple = apple.peel( )
    peeledBasket.append( peeledApple )
```

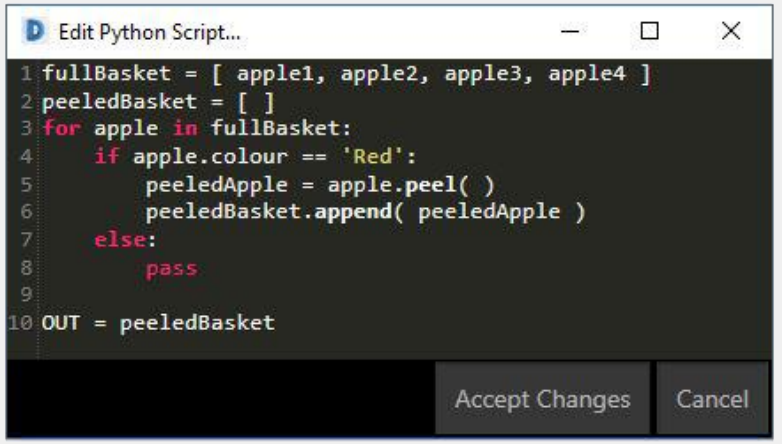


Syntax is: for item in container: do the following
Note the indentation. The loop will result in an error if indentation isn't adhered to correctly
This example will not work in Python – its simply to illustrate process
Note the red words inside the Python node (Reserved keywords)

CONDITIONAL
ONLY PEEL RED APPLES

Say you only want to peel the Red apples in this basket, you would have to use a Conditional Statement (If) to check if they are red first. We also want to have a default action to perform (Even if it's nothing) to the apples that are not Red:

```
fullBasket = [ apple1, apple2, apple3, appleN ]
peeledBasket = [ ]
for apple in fullBasket:
    if apple.colour == 'Red':
        peeledApple = apple.peel( )
        peeledBasket.append( peeledApple )
    else:
        pass
```



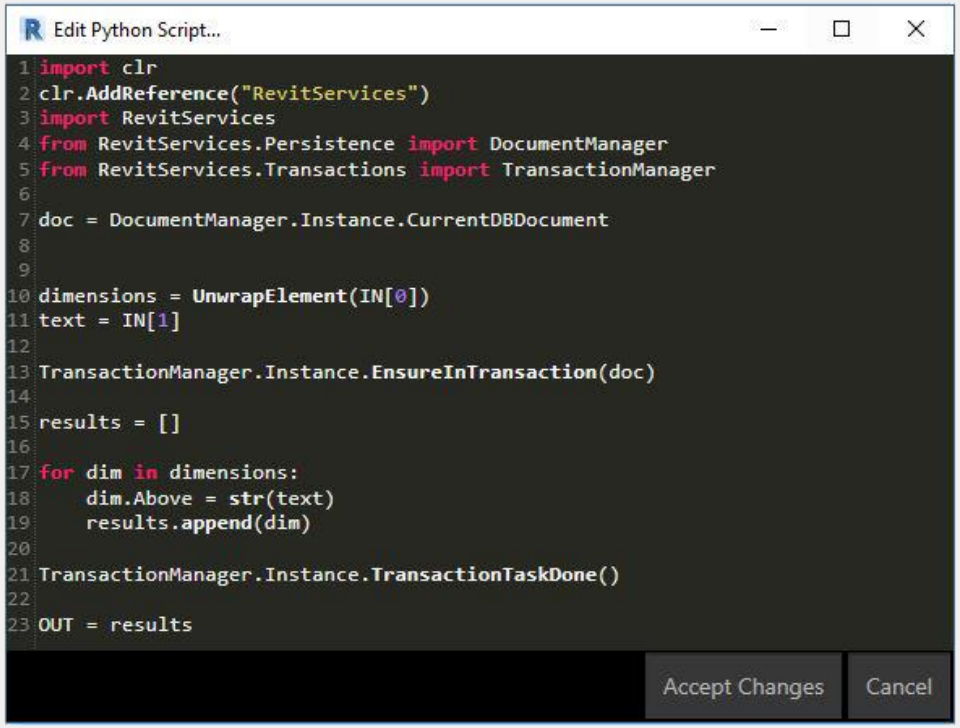
We actually don't need the ' else ' in Python if we wish to simply pass as this is a default action for terminating (closing) an ' If ' statement and is only shown here for explanation
We also assume that each Apple here has a property called ' colour '

FOR LOOP
OUR FIRST SUCCESSFUL NODE

As we want to put our ' **Text** ' on the Above Text Field for all selected dimensions, we need to use a For Loop to do so – As discussed in ' **What is a Loop** ', we need to iterate across our input list (Called ' dimensions ') and perform the same action on every member of that list. To change the text field ' Above ', we use the following syntax:

```
results = [ ]
for dim in dimensions:
    dim.Above = str( text )
    results.append( dim )
```

We create an empty results list, to which we want to append our results
Then for every dimension, we set its property of ' Above ' to our chosen text and make sure it's a string (So that numbers don't come through)
We finally append the resultant dimension to our results list

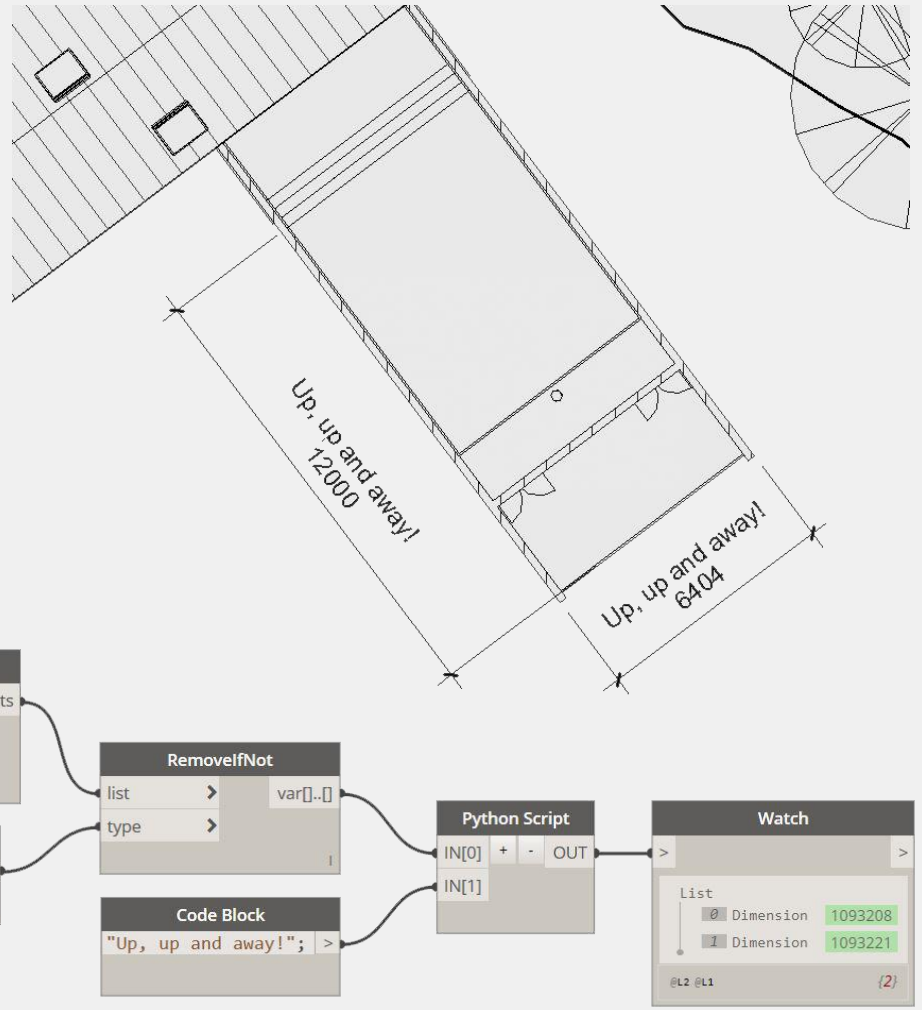


THE RESULT
RUNNING OUR PYTHON NODE

After building a graph (To filter out dimensions only) we can run it to see the selected dimensions back in Revit now have an ' **Above** ' text field added to all selected dimensions* – all through Python alone! Pretty funky.

In order to change the ' Below ' text field instead of ' Above ' only one word needs to change in our code – see if you can figure out what that is now; Same thing applies to ' Prefix ' and ' Suffix ' !

*If you selected a Segmented dimension, you'll notice your graph fail. We address this in the next phase.



Errors & Debugging

[Solving problems when they arise and mitigating where possible]

PAYING ATTENTION
BUT IT HAS ERRORS

If we select a node that has segmented dimensions, this node will fail and result in a yellow warning message – in order to fix this, we need to both understand the problem and find a solution:

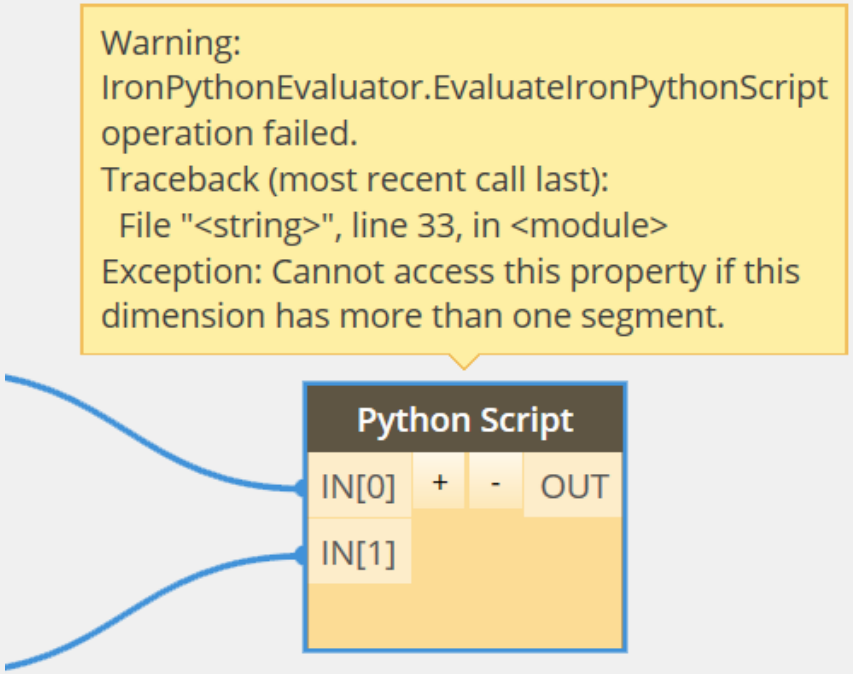
Traceback = Tells us which line of code has the most recent error

Exception = Tells us why our code is failing

Problem: We cannot access the same property on a dimension that has more than one segment

Solution: We have to treat Dimensions that are singular or ones with Segments differently as they have different properties

Note: Sometimes the Exception messages can be cryptic and require a little deciphering
If you want to read more about Errors, look here.



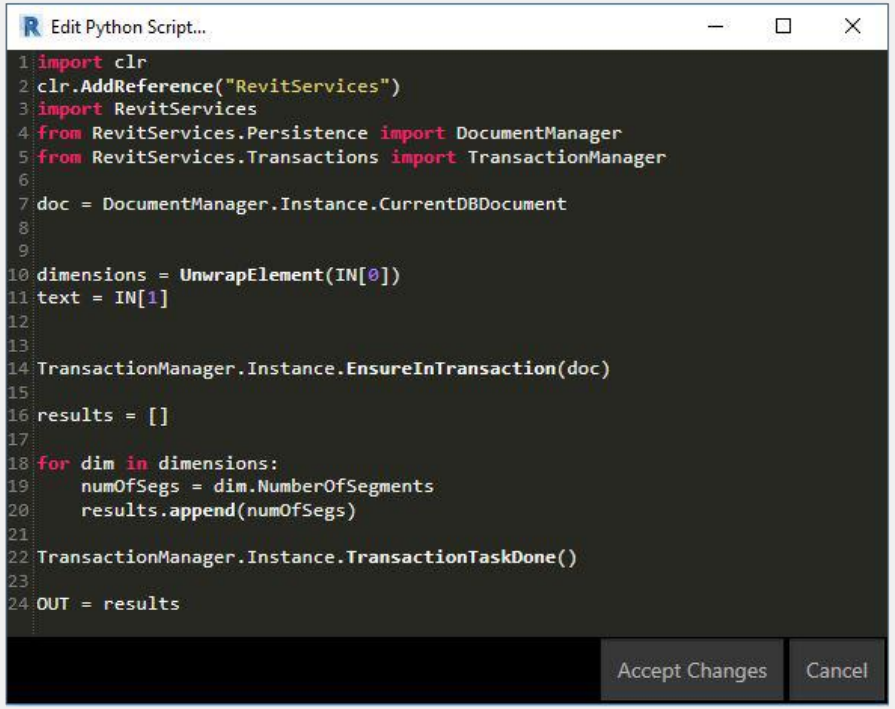
EXTRAPOLATING SOLUTIONS

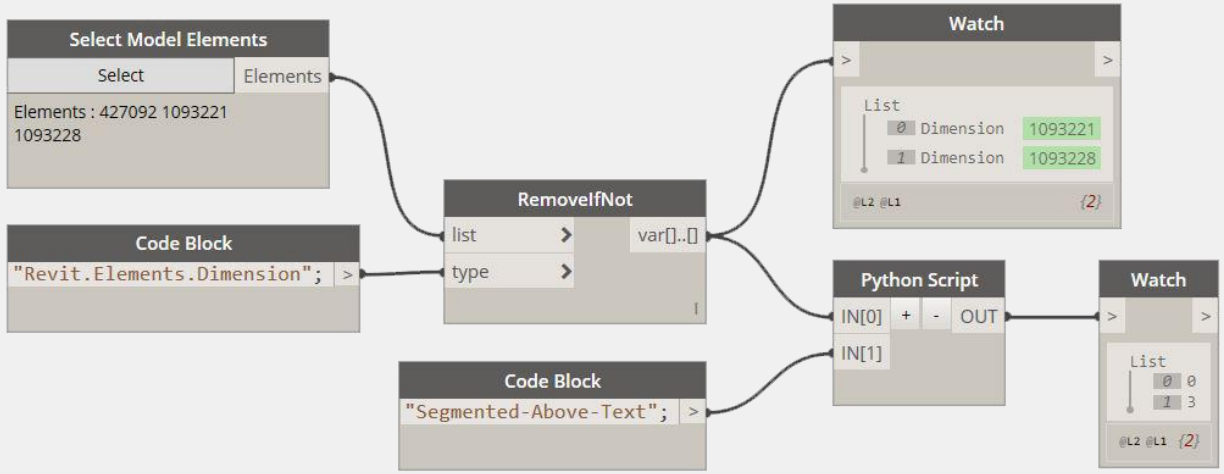
RESOLVING THE ERRORS

To run a working graph that functions if the dimension is singular or segmented we need to use a Conditional Statement and two solutions. Our first solution will cover non-segmented dimensions, and our second must take into account Segments. To do this, we need to check if each dimension is segmented:

```
for dim in dimensions:
    numOfSegs = dim.NumberOfSegments
```

This method will allow us to count the number of Segments by querying that property – it will return zero (0) if there are no segments, and the segment number if they exist.

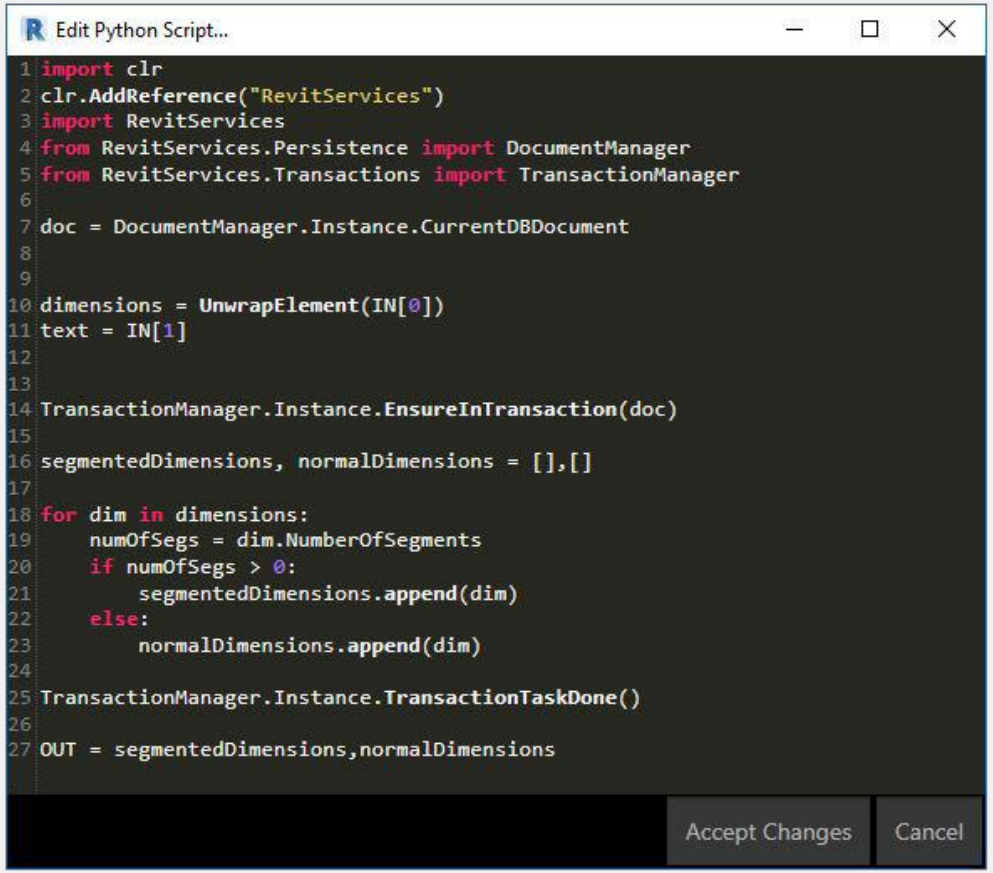




STEP BY STEP
SPLITTING DIMENSION TYPES

After we have our 'Number of Segments' check, we can run an 'If' / 'else' split operation: ' If ' the segments of the dimension are greater than zero (As in, the dimension **has** segments), then append it to a segmented list, ' else ' (otherwise) append it to a non-segmented list:

```
numOfSegs = dim.NumberOfSegments
if numOfSegs > 0:
    segmentedDimensions.append(dim)
else:
    normalDimensions.append(dim)
```



SUBELEMENT MANIPULATION

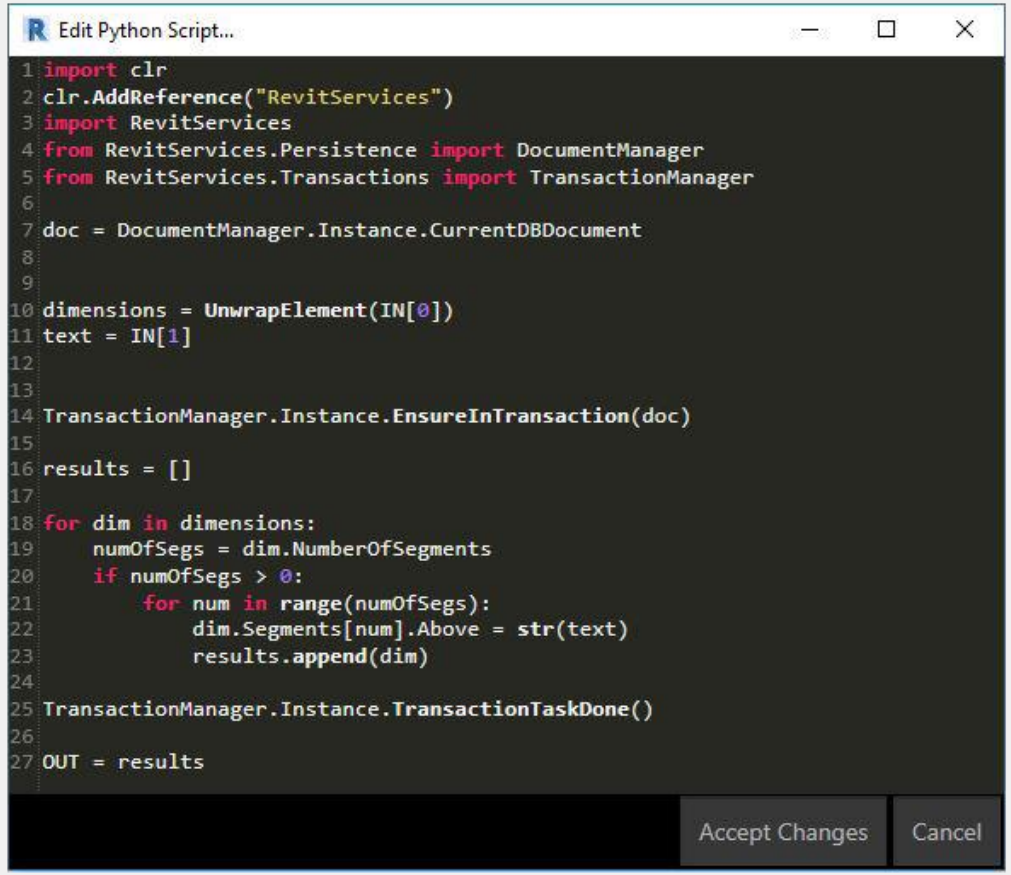
SETTING SEGMENT ABOVE TEXT

As we have our segment count, we can now run another ' for ' loop across only those segmented dimensions. We need to access each individual segment and will do so using a range count and indexing – so our ' for ' loop looks for each number inside of a list of numbers (range) that starts at zero and finishes at the segment count minus one*:

```
for num in range( numOfSegs ):
    dim.Segments[ num ].Above = str( text )
    results.append( dim )
```

The syntax for dim.Segments[num] allows us to set each individual segment in turn.

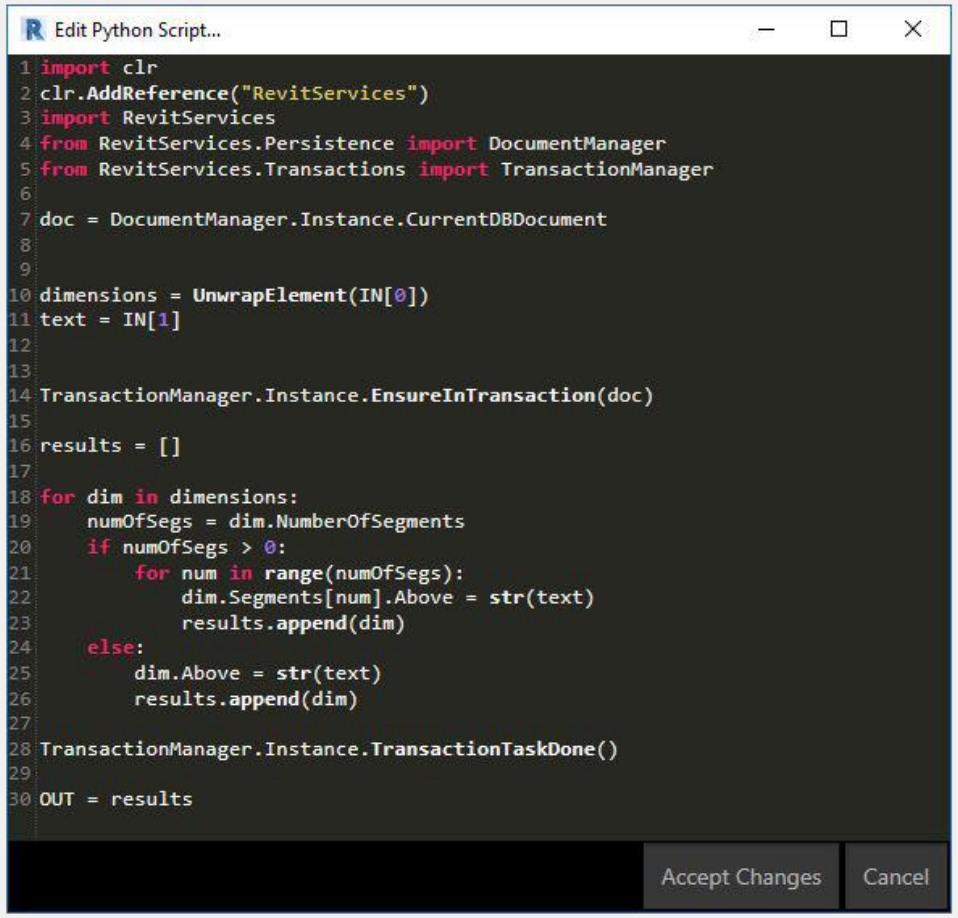
* This is due to how slicing works in Python. Refer to the lab dynamo files for explanation.



THE FINALE
SETTING ABOVE TEXT

After we have successfully actioned our segmented dimensions, we simply copy our previous code into an ' else ' statement. This means that if our Dimension is segmented, it will be actioned in the first ' if ' statement, and iterated across (done for every single segment) by populating the ' **Above** ' text, and if it's not segmented it will populate the ' **Above** ' text by calling the overall dimension.Above property and setting it.

An annotated version of this code is set inside the downloadable Lab zip file.



Revit and the API

[Talking the same language]

TAKING THE PLUNGE
REVITAPIDOCS

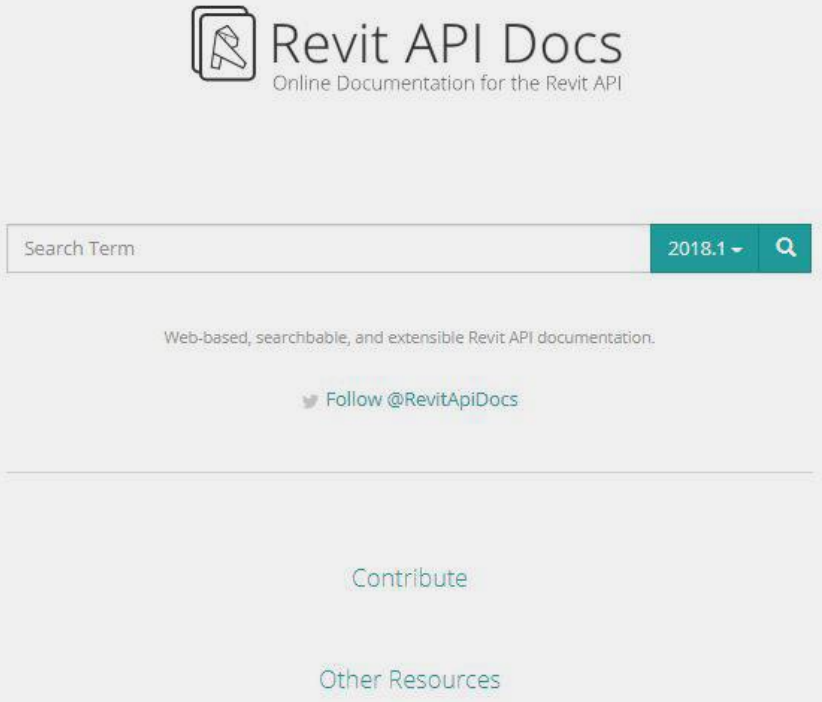
The RevitAPIDocs are a collection of online documents that match the Revit SDK (Software Development Kit*). In simple terms, this is where you can go to explore the Revit API. In not-so-simple terms it covers examples in C++, C# and VB – none of which help the Python Beginner. However, these next few slides will showcase how you can still use them to your advantage!

Accessible at: http://www.revitapidocs.com/

In this example we are looking at a **Filtered Element Collector** for illustrative purposes in order to pick a series of elements directly from the Revit Document.

Shoutout to Gui Talarico (@gtalarico) for creating and maintaining this awesome resource

We will not be touching upon the SDK in this Lab. Please go here to read more about it

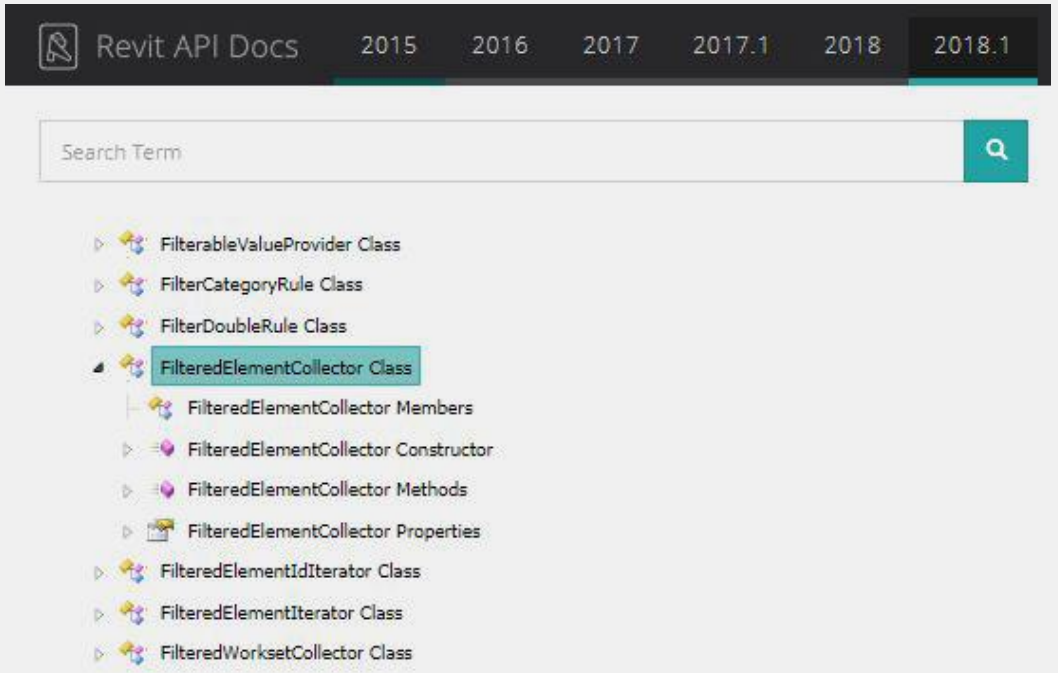


BROWSING THE LIBRARY
A COLLECTOR

We are going to explore the FilteredElementCollector Class (A way in which to collect a list of Elements inside of the Revit project based on set criteria). Inside this class we have four sub-elements:

Members = All items pertaining to the Class*
Constructors = How to Create this thing (Object)
Methods = Actions (Functions) we call on the Class
Properties = Things we can query

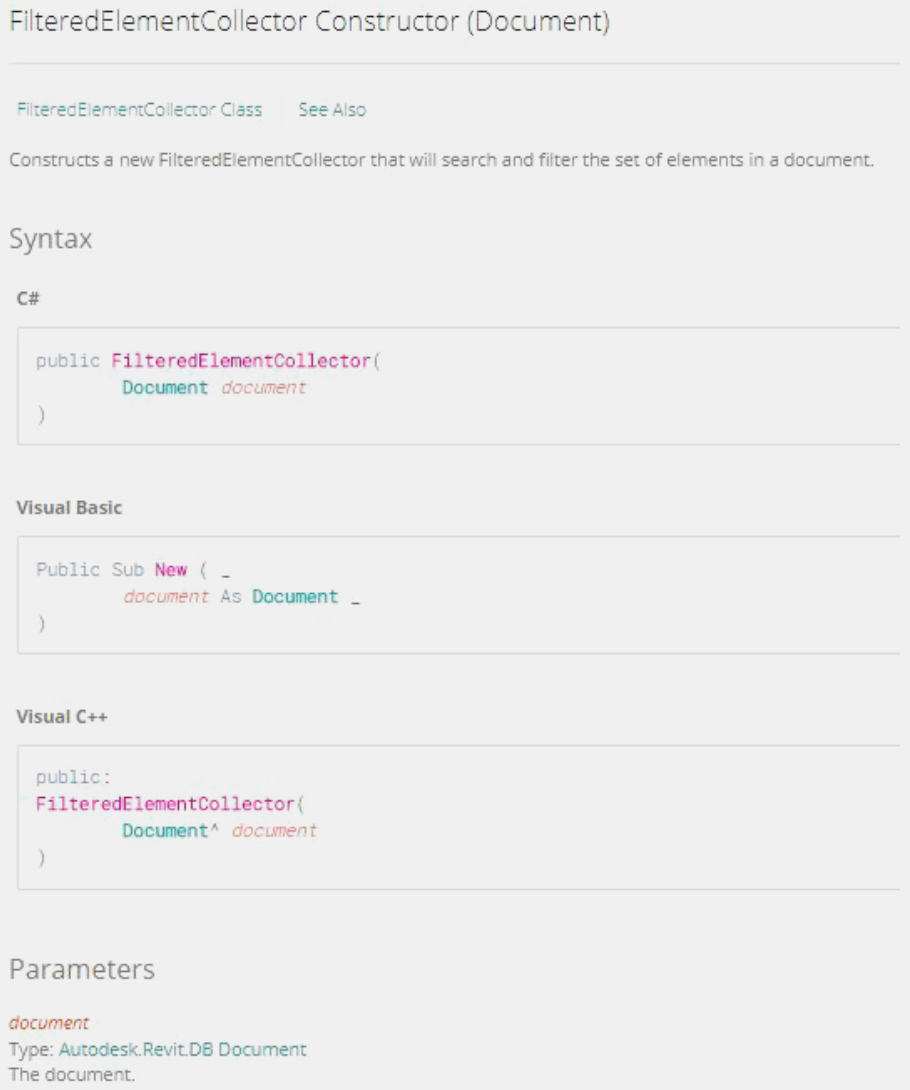
Read more about classes here.



COLLECTING
CREATING A COLLECTOR

We choose our preferred Construction Method of '**FilteredElementCollector Constructor (Document)**' and are a little bit stumped... What is ' public ' ? What is ' New ' ? Good news is that you can ignore the hard coded-typologies of these three languages, and simply look at the C# exemplar for a hint. Using the pink and orange sections as guides (Not always true but a good rule of thumb to begin) we can construct our Collector in Python.

Note: If you look down at the base under 'Parameters' you'll see we need to refer to a 'document' – Good thing we already know how to do this!



FilteredElementCollector Constructor (Document)

FilteredElementCollector Class See Also

Constructs a new FilteredElementCollector that will search and filter the set of elements in a document.

Syntax

C#

```
public FilteredElementCollector(
        Document document
)
```

Visual Basic

```
Public Sub New ( _
        document As Document _
)
```

Visual C++

```
public:
FilteredElementCollector(
        Document^ document
)
```

Parameters

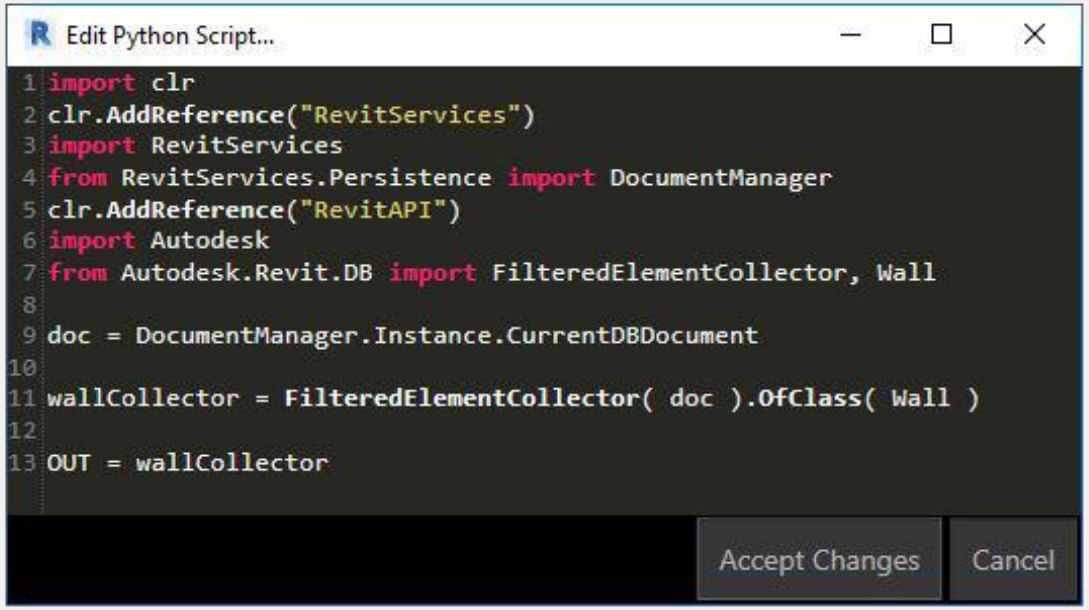
document
Type: Autodesk.Revit.DB Document
The document.

COLLECTING
COLLECTOR CODE

Creating a **Wall Collector** (To capture all Wall elements inside our Revit Project) requires us to create a FilteredElementCollector of the Wall Class. The syntax is as follows:

wallCollector = FilteredElementCollector(doc).OfClass(Wall)

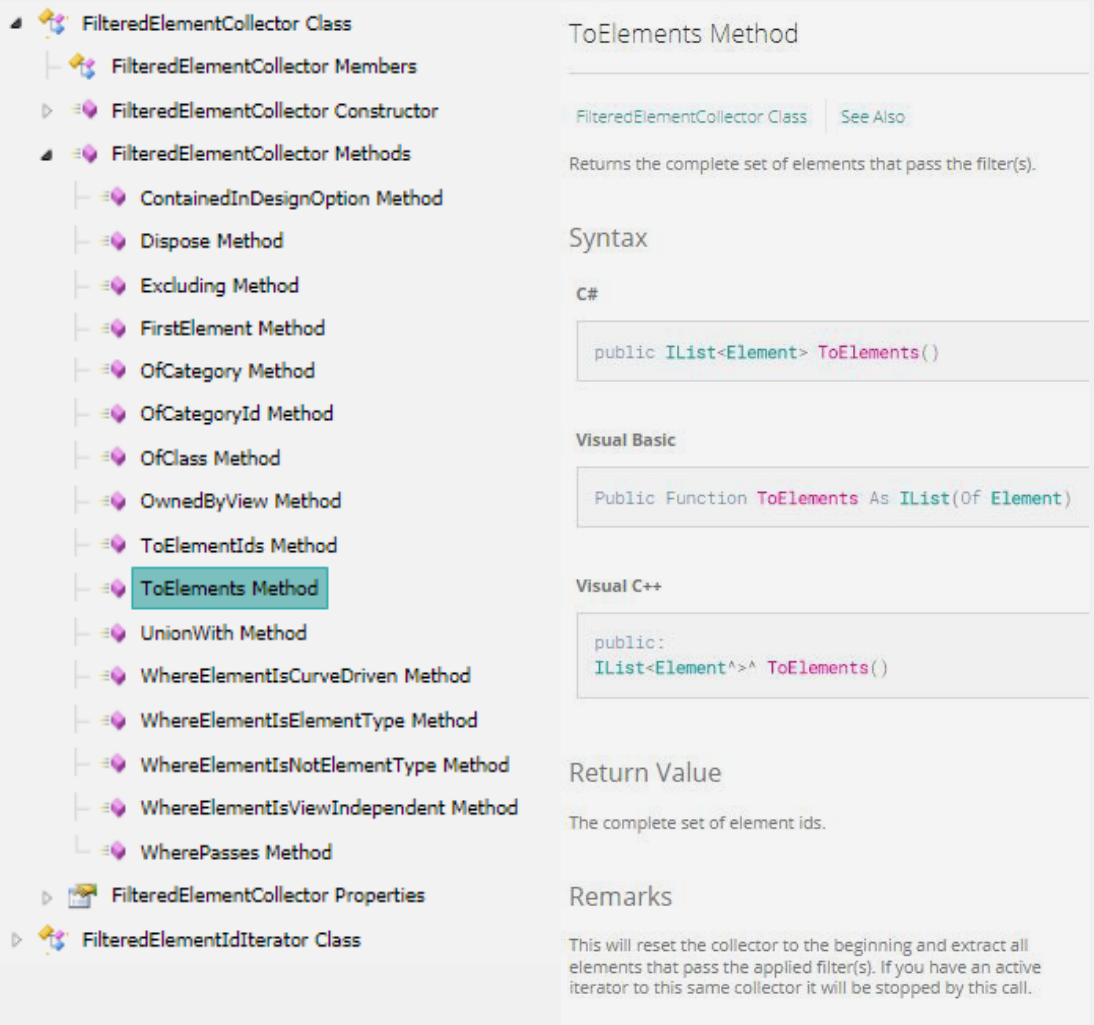
Note: This will return Revit namespaces (Autodesk.Revit.DB.Wall) rather than Elements.



GETTING
RETURNING ELEMENTS

After choosing what we want to Collect (Walls) we need to finish our Collector by getting real Elements which we can then use inside of Dynamo. To do such we need to use the **'ToElements( )'** call:

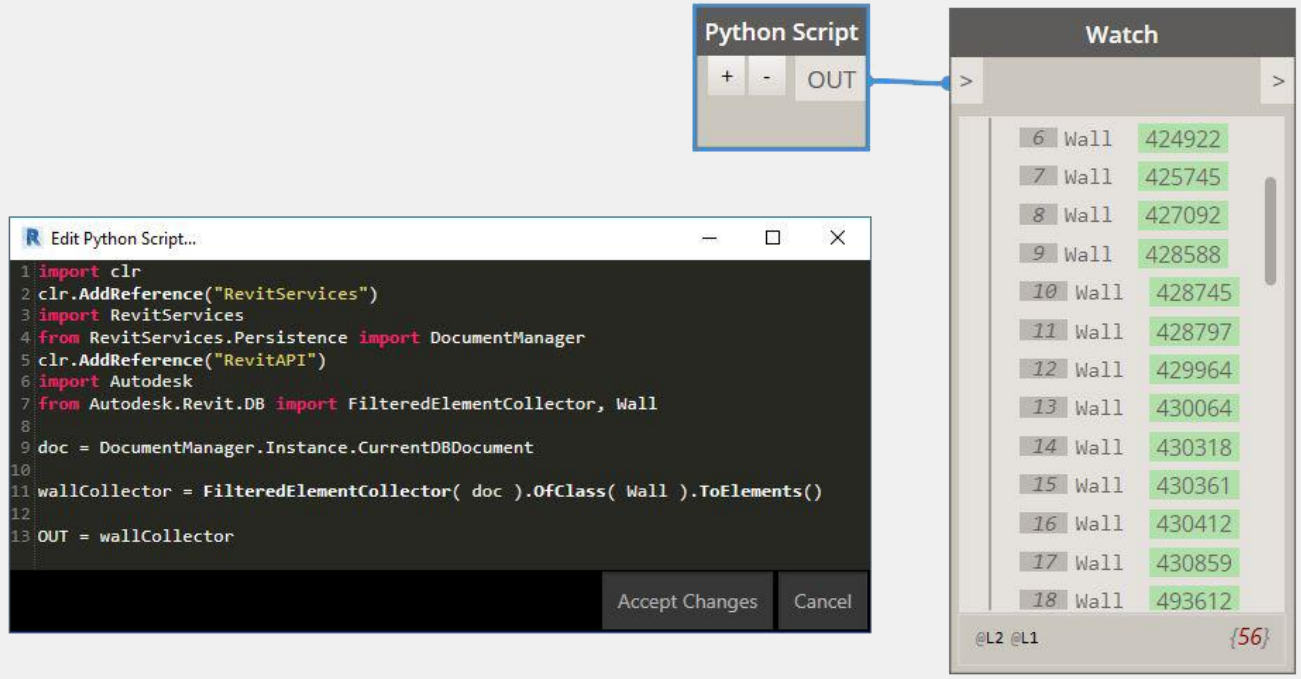
```
wallCollector = FilteredElementCollector( doc )
.OfClass( Wall ).ToElements( )
```



ONWARDS
BACK INTO DYNAMO

This will create a FilteredElementCollector on the entire Revit Project, from which we use a Filter called ' OfClass ' and stipulate ' Walls ' as the Elements we wish to collect and after collection we cast them to elements for subsequent use inside of Dynamo.

An annotated version of this code is set inside the downloadable Lab zip file.



QUESTIONS

[Is everyone's brain still alive... ?]

REFERENCES
STANDING ON THE SHOULDERS OF GIANTS

- https://python.org
- http://python-reference.readthedocs.io/en/latest/
- http://www.revitapidocs.com/
- https://github.com/gtalarico/python-revit-resources
- https://www.datacamp.com/community/tutorials/18-most-common-python-list-questions-learn-python

Addendum

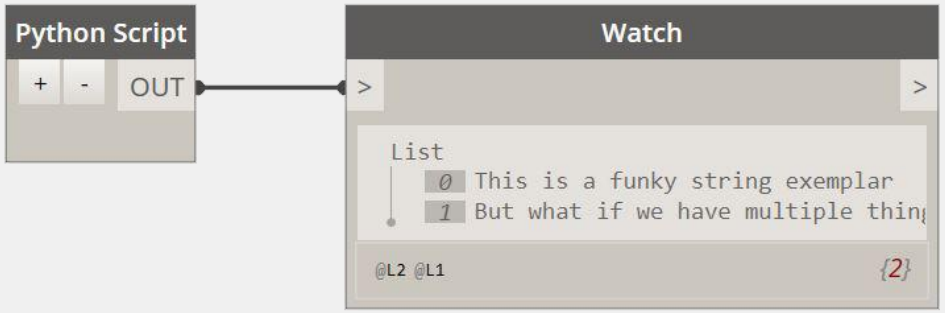
[A few additional toys to play with later]

STRING ACTIONS

STRING FORMATTING

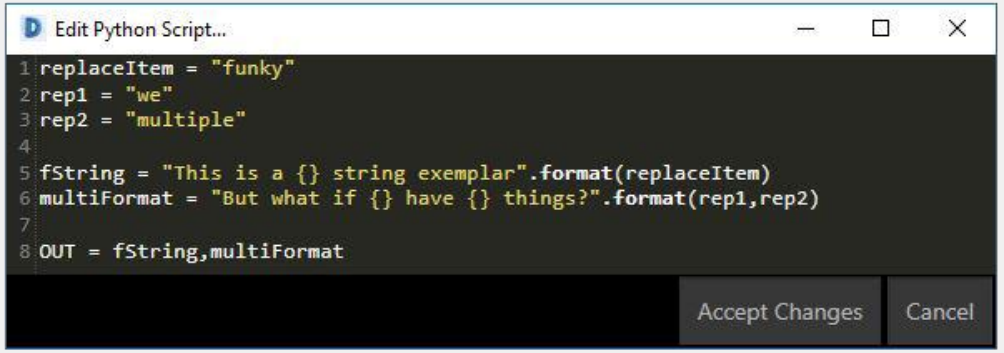
String format (**String.Replace***) | Allows us to use a variable (changeable thing) inside of otherwise static text

```
fString = "This is a { } string exemplar".format( replaceItem )
multiFormat = "But what if { } have { } things?".format( rep1, rep2 )
```



Note: An older method of "string with % in it"%("percentage") still exists, but is being phased out in lieu of the above.

* Format doesn't really have a proper equivalent out-of-the-box.



LIST ACTIONS

APPEND & EXTEND

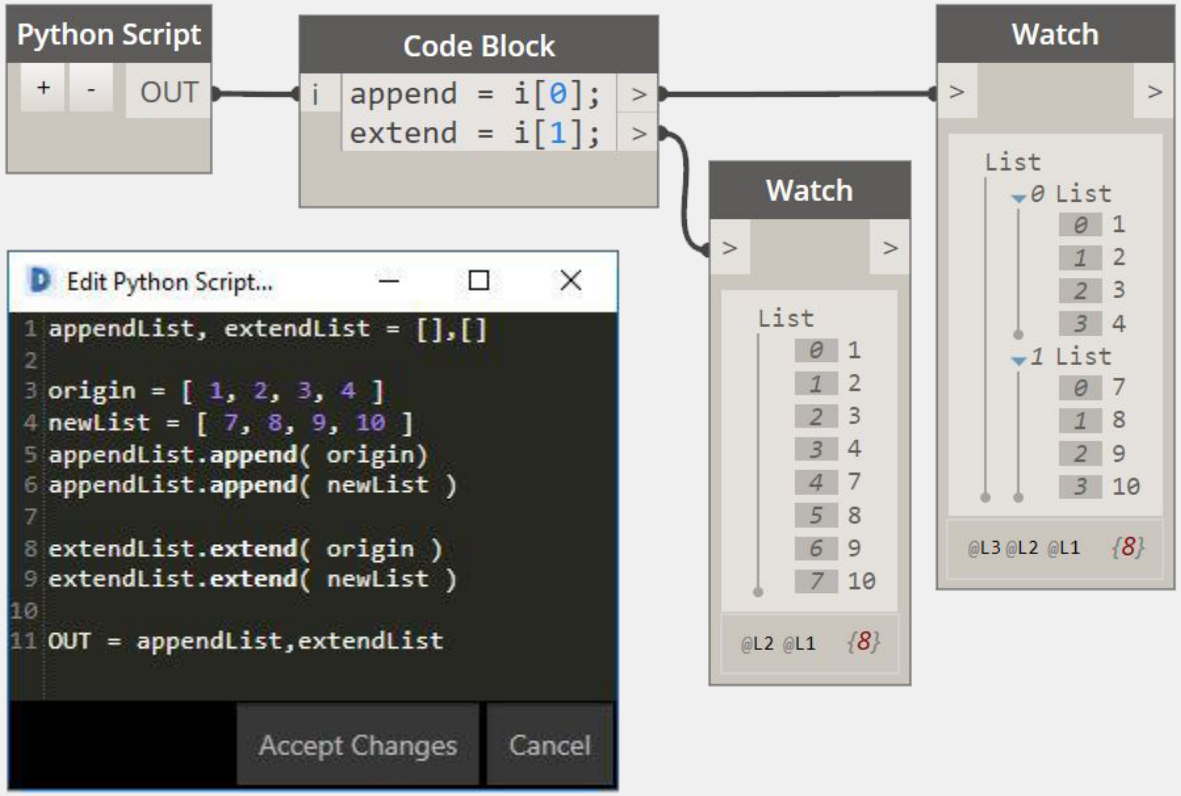
To add an item to a list in Python, you can either Append or Extend

```
appendList, extendList = [ ],[ ]
origin = [ 1, 2, 3, 4 ]
newList = [ 7, 8, 9, 10 ]
appendList.append( origin )
appendList.append( newList )

extendList.extend( origin )
extendList.extend( newList )
```

Append adds elements to the list including the list the elements are contained in

Extend adds elements to the list excluding the list the elements are contained in



LIST ACTIONS

OTHER LIST ACTIONS

List.insert(i, x): Inserts an element at a particular index
List.remove(x): Removes first element in the List with a value of 'x'
List.pop(*[i]*): Remove item at index (optional). If no index supplied, return last element
List.reverse( ): Reverses the list
List.count(x): Will return the number of times 'x' appears in the list
List.sort( ): Sorts a list by its numeric values
List.index(x): Returns the first index of the value 'x' in a list
List.clear( ): Empties a list of it's elements
List.copy( ): Copies a list so that manipulation doesn't affect the origin list

List.insert() List.remove() List.pop()

List.reverse() List.count() List.sort()

List.index() List.clear() List.copy()

Read more about data structures here.

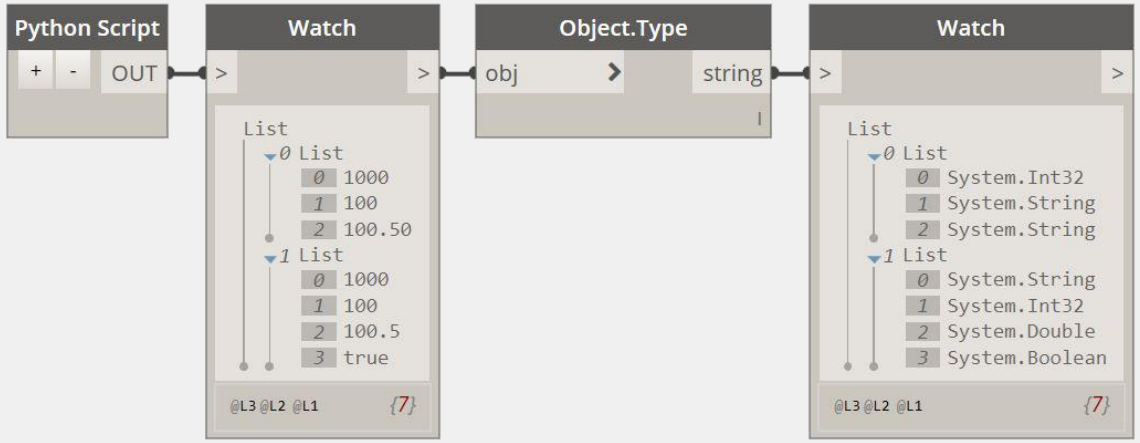
CASTING

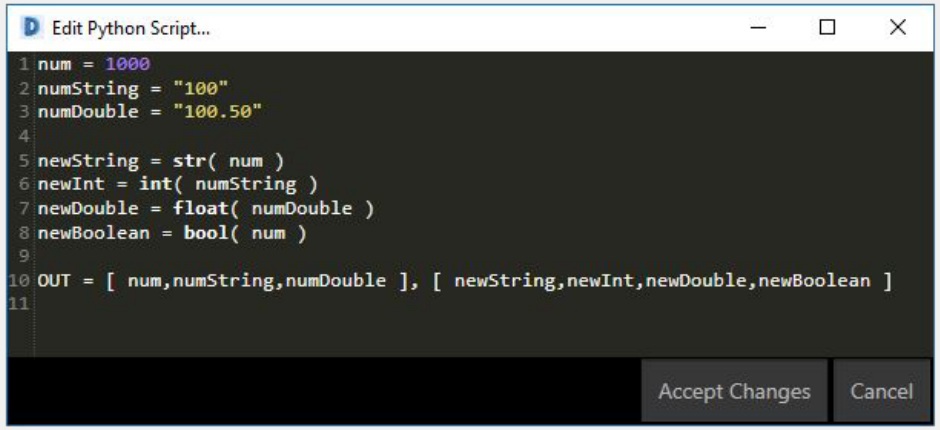
TYPE CONVERSION*

Casting is when you want to change the Object Type of a variable e.g. you have a number of 100 but you want it to be a string (text) instead.

```
newString = str( x )
newInt = int( x )
newDouble = float( x )
newBoolean** = bool( x )
newList = list( x )
```

* We are only covering basic casting here. For additional type conversion read more here.
** Casting anything to a Boolean will only ever result in False if an empty string, or True for all other elements.



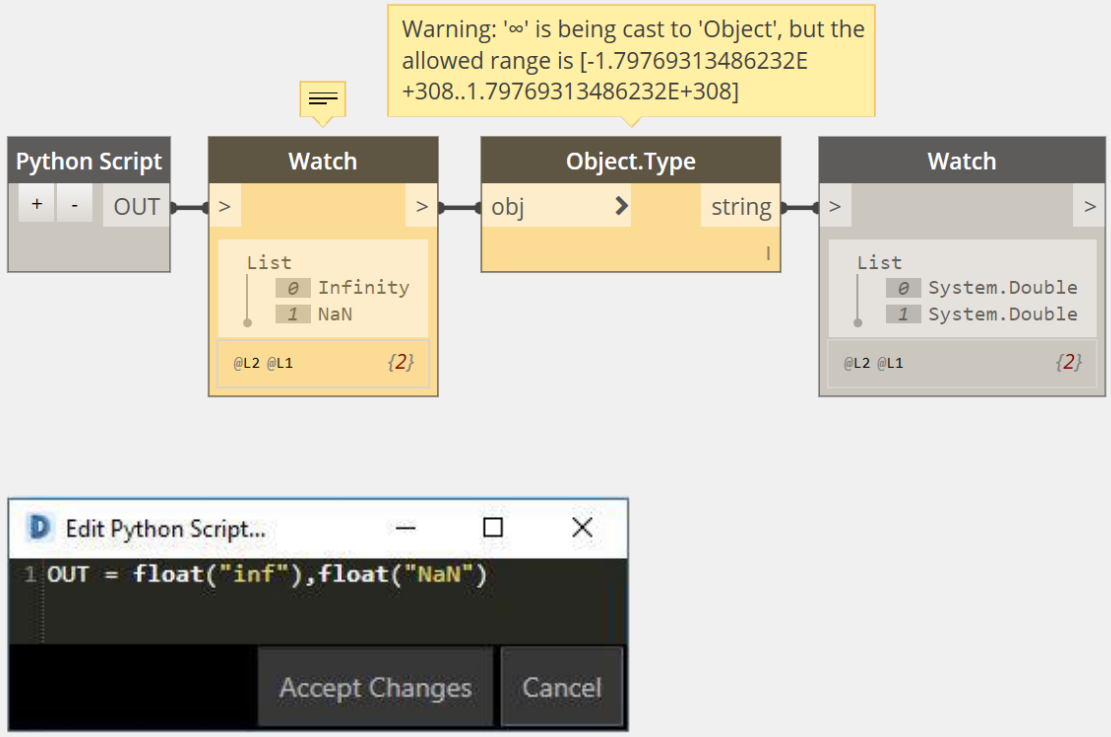


CASTING
OUTLIERS

You can even cast some fun things such as Infinity and Not a Number (NaN):

```
infinity = float( "inf" )
notANumber = float( "NaN" )
```

These result in a funky error in Dynamo – but do correctly cast.

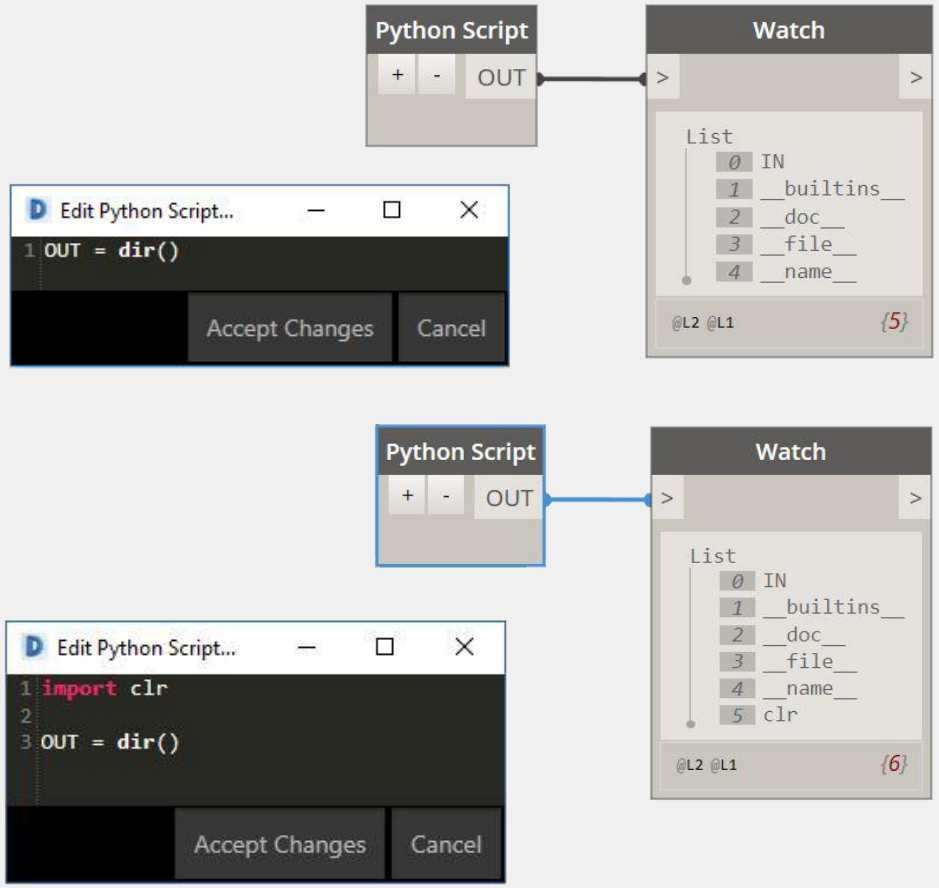


FOR THE CURIOUS

USING DIR( ) TO EXPLORE

In Python we can use a command (function) called 'dir()' to explore the 'directory' – i.e the Python node itself. We do this by calling dir on nothing:

```
OUT = dir( )
```



This will give us the results of what exists inside of the Python node by default – the 'under-the-hood' stuff.

The top exemplar showcases the directory (dir) of the out-of-the-box node in a non failing state (OUT being called). Here we simply have the 'IN' variable and some built-in functions

The bottom exemplar showcases the directory (dir) of the node with an imported 'clr' module

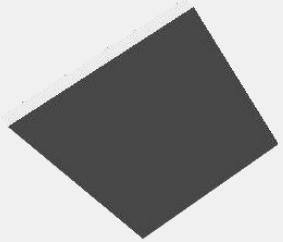
Read more about dir( ) here

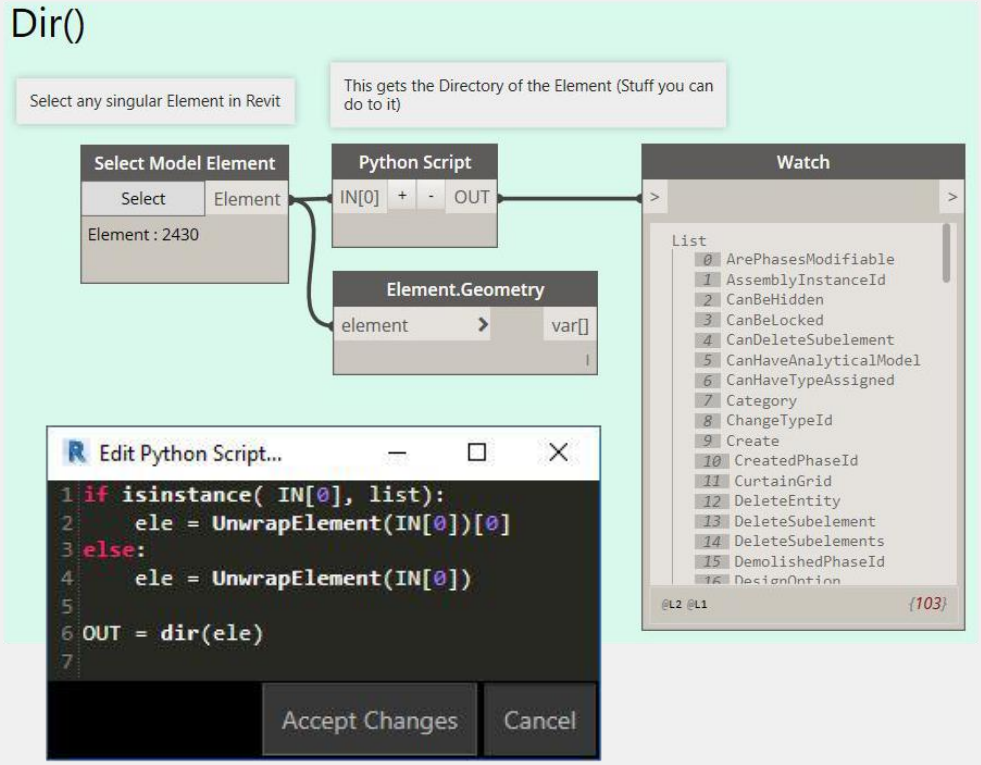
DYNAMO SNOOP

REVIT ELEMENT DIR( )

The graph to the right will allow you to select any element inside of Revit (Such as a Wall, Dimension, Sheet etc) and it will show you all of the actions / methods you can run on that Element.

You are ' **snooping** ' (Similar to the Revit Lookup add-on)



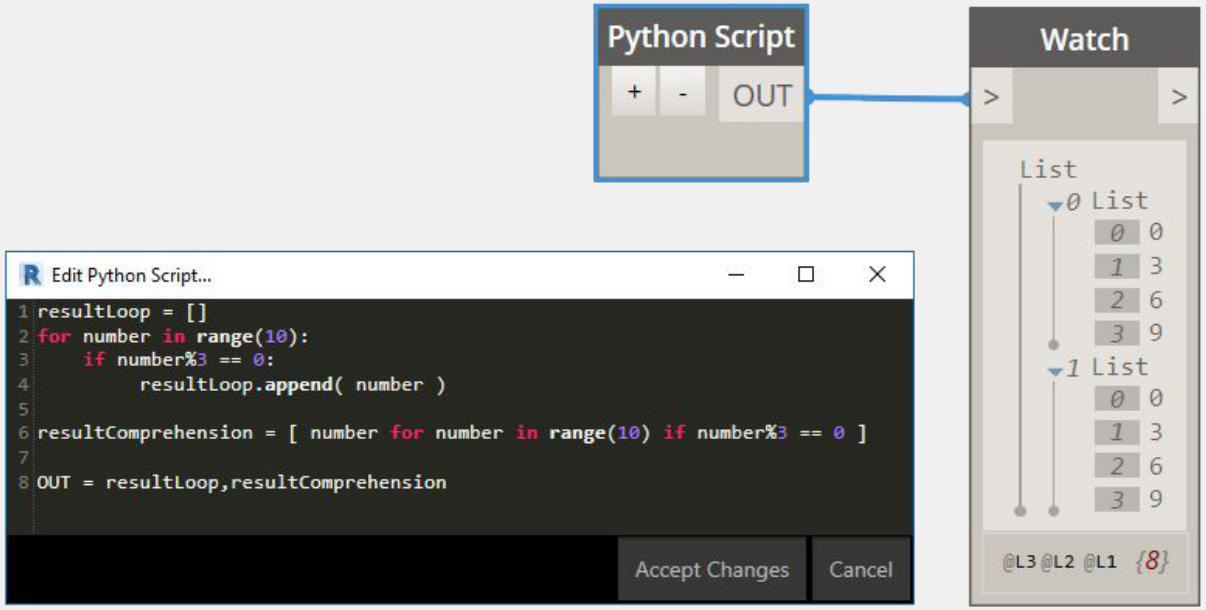


COMPREHENSIONS
LIST COMPREHENSION

List comprehensions are a concise way to create lists. They will always return a list. The comprehension syntax is [*expression* for *item* in *list* if *conditional*] and they cut down on the required number of lines of code by neatly wrapping up conditional checks:

```
results = [ ]
for number in range( 10 ):
    if number % 3 != 0:
        results.append( number )

results = [ number for number in range( 10 ) if number % 3 != 0 ]
```



Note: You can also do Dictionary and Tuple comprehensions.
Read more about comprehensions here.

FREEBIES

DYNAMO TO PYTHON FUNCTIONALITY

Attached to this Lab class are a series of Dynamo Graphs containing functionality inside of Python. This functionality covers what has been showcased today in this Lab and also mimics commonly used workflows from Dynamo inside of Python, such as:

- Fundamentals
- Ranges and Sequences
- Slicing and Items at Indexes
- Loops
- Sorting Grouping
- List Actions
- Directory operations
- Comprehensions


designtech.io

AUTODESK®

Make anything.