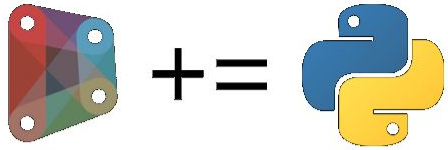
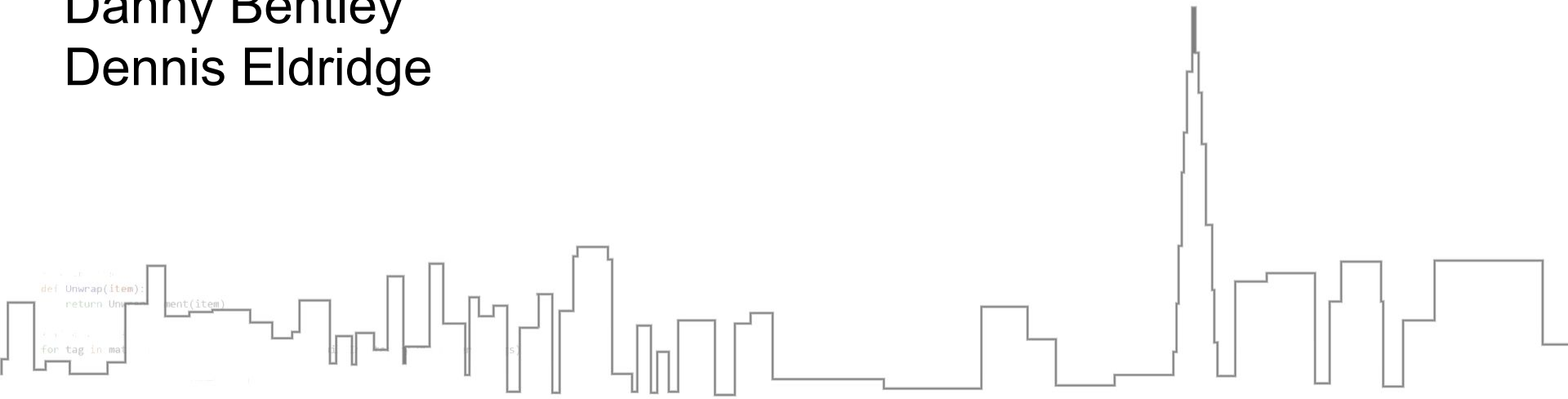
Python Workshop



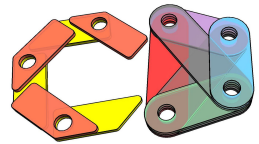
Arman Hadilou
Danny Bentley
Dennis Eldridge



ALL CODE FROM THE WORKSHOP WILL BE AVAILABLE AFTERWARD ON OUR GITHUB:

https://github.com/sfcdug

GITHUB



ARMAN HADILOU, AIA

WRNS Studio

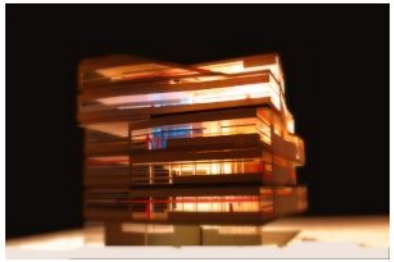
Project Architect/ Project Designer

BENETTON RETAIL STORE

DESIGN MARFA

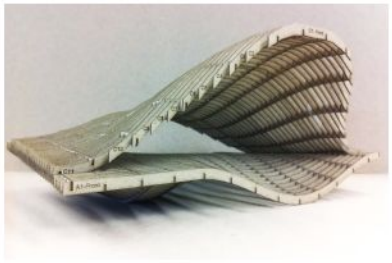
PAT|TRAY

ALEUTIAN ISLAND RESIDENCE



GREEN FOLDS

UT FASHION SHOW 2013



BARKITECTURE

LOUISVILLE CHILDREN MUSEUM

LIGHT SKIN_FUTURE OF SHADE 2017

DANNY BENTLEY

SOM

Structural BIM Designer and Drafter

YOUTUBER

Channel for Revit API and Dynamo using C# and Python.

GITHUB

https://github.com/dannysbentley

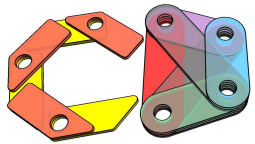
DENNIS ELDRIDGE , AIA

SOM

Digital Designer and Licensed Architect

GITHUB

https://github.com/Dennis-Eldridge



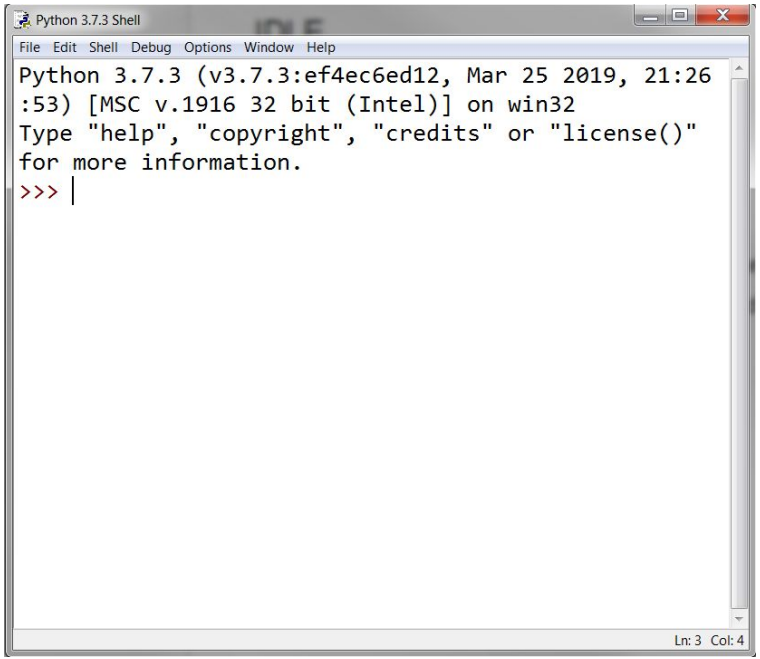
DATA STRUCTURES

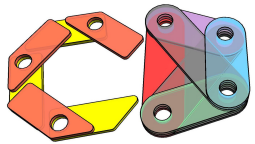
MANAGING INFORMATION

IDLE

INTEGRATED DEVELOPMENT & LEARNING ENVIRONMENT

This is the default development environment for python. It comes with python when you install it, and is suitable for beginners. Go to the start menu and start up IDLE. We'll begin learning the basics of python here.





VARIABLES

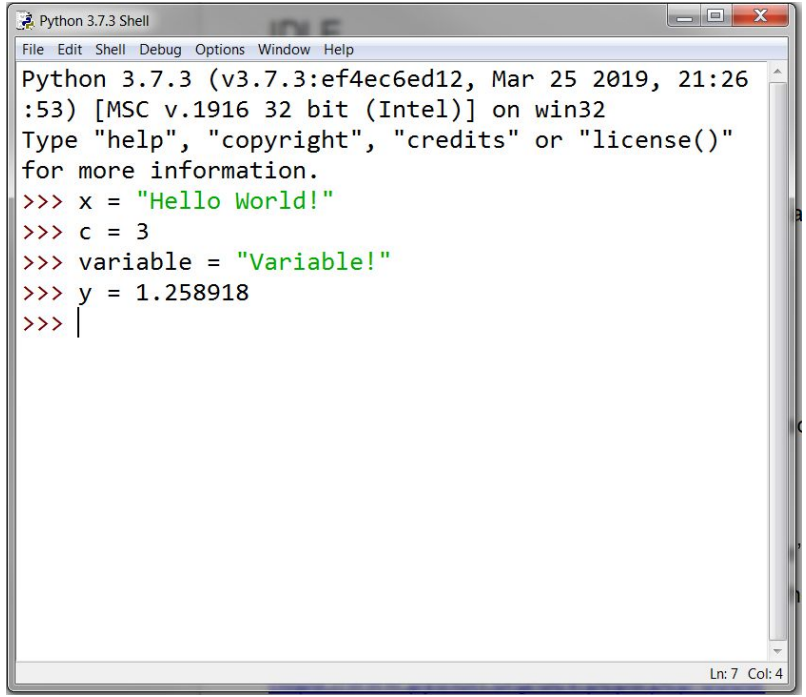
ASSIGNING

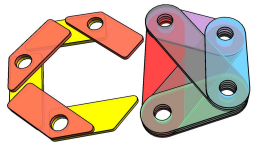
Similar to algebra, a variable is a shorthand name given to an object. Variables can be created from any data-type. Unlike other languages variables don't need to be given a type or declared in advanced.

Variables must start with a letter, but can contain letters, numbers, dashes, and underscores. Variables should be named so it's easy to identify what they're used for.

When you assign a variable python dynamically determines which data type it should be.

Every language has its own style conventions. We're not going to worry about it now, but you can read the official documentation:
https://www.python.org/dev/peps/pep-0008/





BASIC DATA TYPES

NUMBERS

Any variable assigned to a number will be automatically be created as an integer. Technically there are several different kinds of numeric data types, which may be important to know

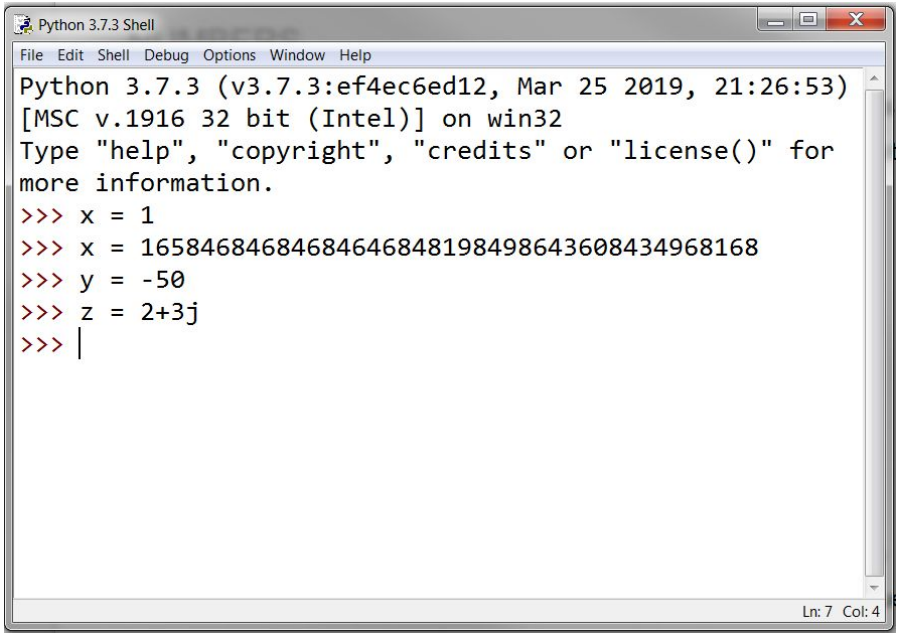
integer (int) : whole number - 5

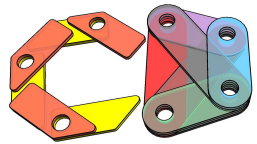
f**loating-point (float) :** decimal point number - 1.258

complex: <real number> + <imaginary number> - 2+3j

long : really long integers (this was phased out of python)

It's important to know about these special types of integers. For example float division is notoriously inaccurate. If you're dividing two decimals (floats) you may notice a tiny discrepancy with the output due to how computers calculate float division. Additionally, when dealing with older versions of python, such as what's used in Dynamo, you may find that an a number is too big to be stored as an integer, and needs to be converted to a long type variable.





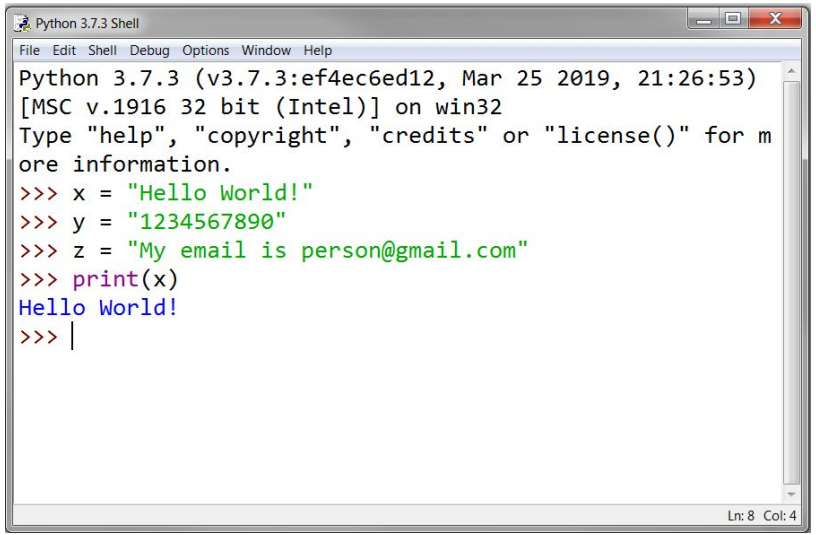
BASIC DATA TYPES

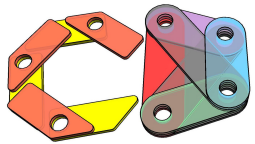
STRINGS

A sequence of characters (text) enclosed by either single (‘) or double quotes (“). Strings can include any character such as letters, numbers, and symbols.

We’ll go more into functions later, but the print() function is an important function to know. It will allow you to print the value of any string, including variables. This can be helpful when debugging, because it allows you to see the values of your variables at certain point in the code.

It’s important to note that the backslash has special meaning in strings. It’s called an escape character, and it’s used to represent special characters. For example, a new line in a string can be represented by “\n”.





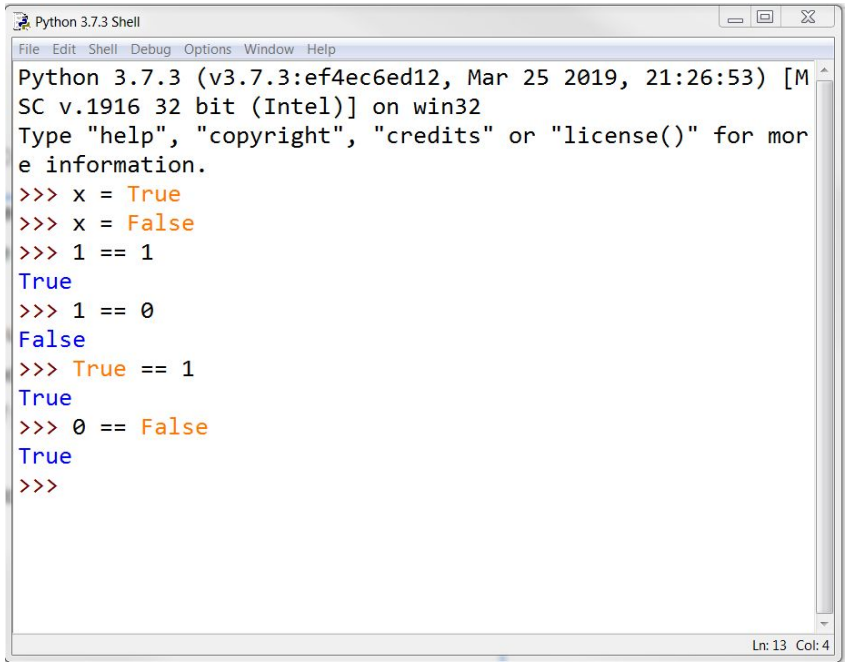
BASIC DATA TYPES

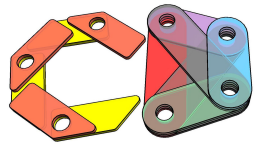
BOOLEAN (BOOL)

This data type is used to represent true or false values. Therefore they can either have a value of “True” or “False”.

Boolean values are used a lot when evaluating expressions. For example, if you typed the expression 1=1 into python it would return True. Conversely, if you typed in 1=0 python would return False. (Technically equality is typed using double equals since a single equal sign sets the value of a variable.)

It’s also interesting to note that 1 and True are equal to each other, as-is 0 and False.



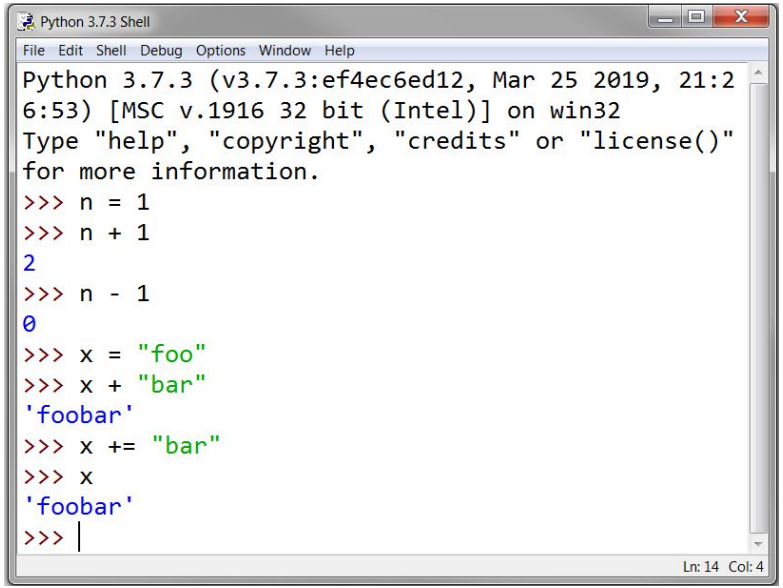


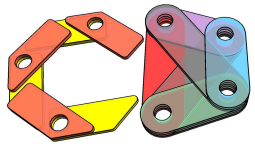
BASIC DATA TYPES

ADDITION / SUBTRACTION & CONCATENATION

Now that we have the basic data types, what can we do with them? For integers, all typical mathematical operations can be performed on them. You can add or subtract, and python will act as a calculator.

Interestingly, you can also add strings together using the “+” character. This is called concatenation. Using the “+=” symbols together will add the additional characters to the value of your variable.



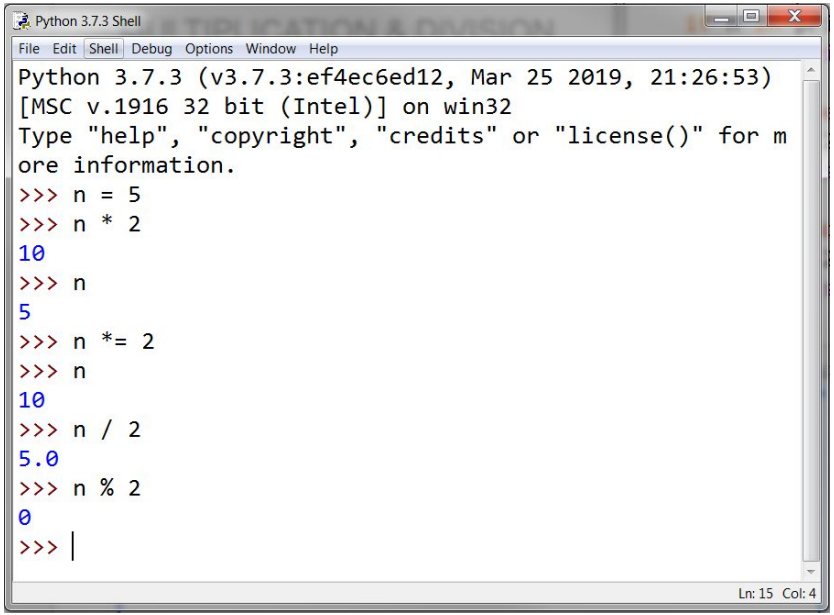


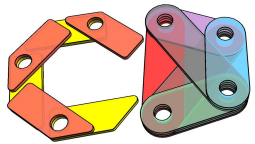
BASIC DATA TYPES

MULTIPLICATION & DIVISION

You can also use the multiply and divide operations. As with other programs “*” is multiply and “/” is divide. Just like with addition you can group those signs with an equal sign to set your variable equal to the multiplication or division of the value.

There is an additional operation that may be helpful to know. The symbol “/” returns the result of a division between two values. There is also an additional using the “%” character that returns the remainder of division between two numbers. You could use this to check if a number is even.

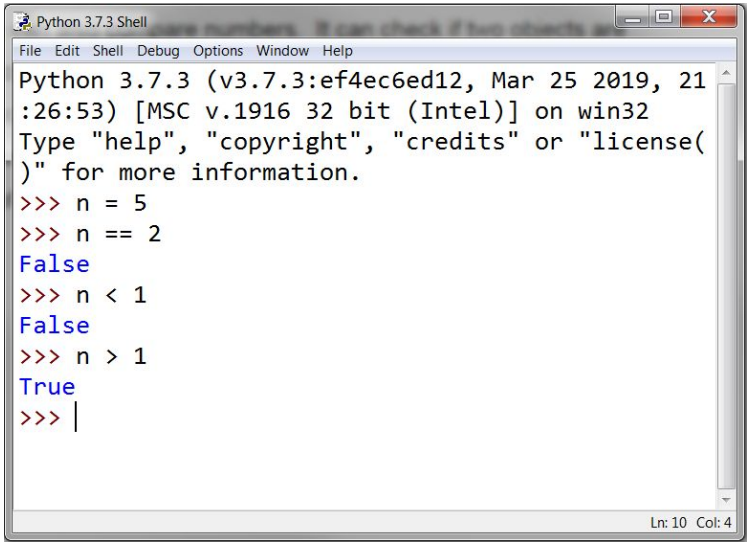


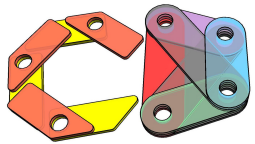


BASIC DATA TYPES

COMPARISONS

Python can also compare numbers. It can check if two objects are equal to each other “==”, if an operand is less than another “<”, or if an operand is greater than another “>”. You can also combine them to get less than or equal to “>=” or greater than or equal to “<=”. The result of these comparisons will be a boolean (True or False) value.



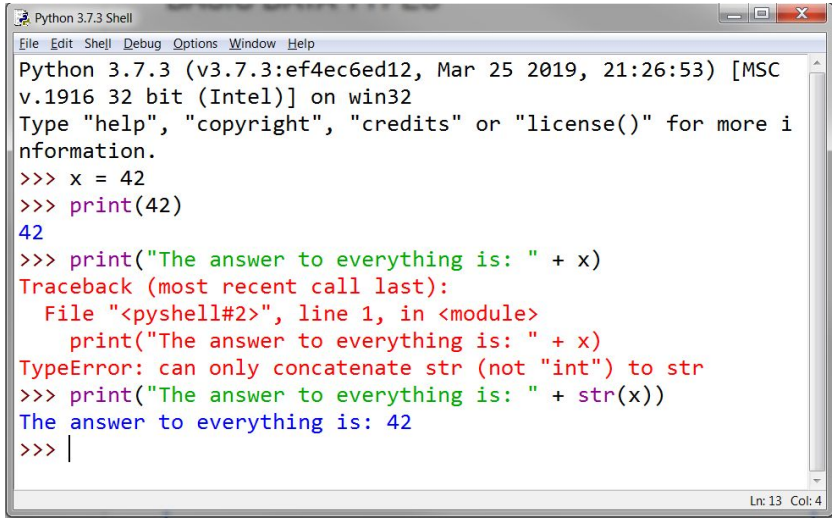


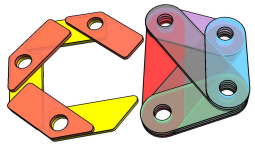
BASIC DATA TYPES

TYPE CONVERSION

Some data types can be converted to other types. Most of the time you will be converting an integer or boolean to a string. If you want to print the value of an integer along with other text you'll need to convert it to a string using this function str(int).

Converting a string to an integer is a little more complicated, because not all strings contain integers. That's beyond the scope of this course.





ADVANCED DATA TYPES

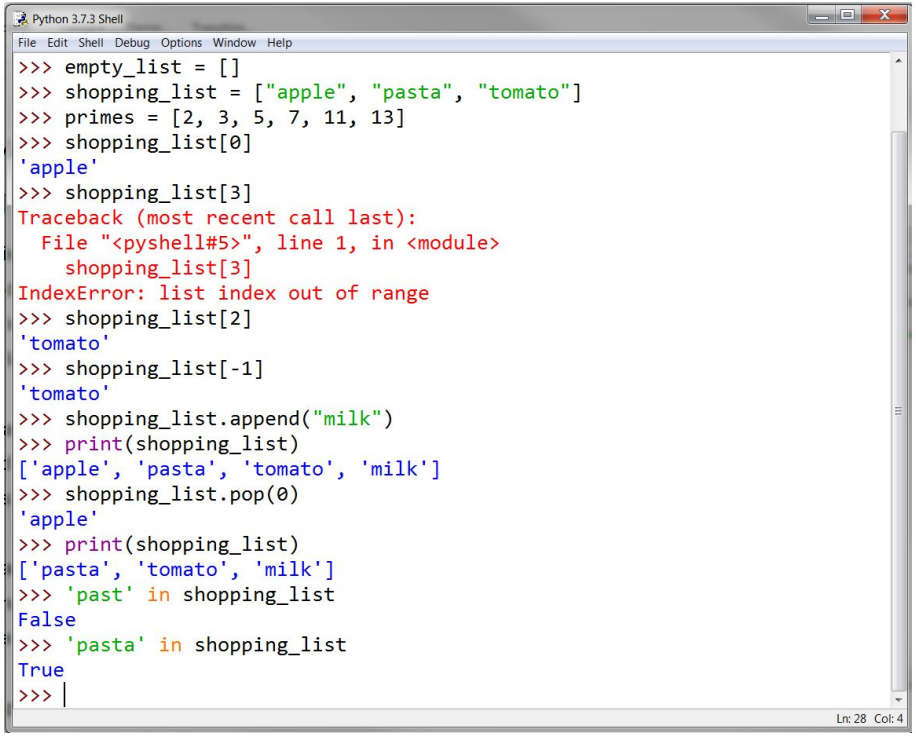
LIST

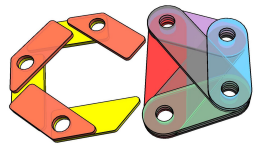
A list is an ordered sequence of elements. A list can contain any other data type, and they can be combined. It's possible to create a list of integers, strings, or boolean values. You could even create a list of lists!

You can create an empty list by assigning a variable to an open and closed bracket "[]". When you create the list you can define its initial values by separating each value with a comma.

Lists are mutable, meaning that they can be changed. You can add or remove items from a list. It's important to note that elements in the list are in a specific order, but the list can be re-ordered.

The append method can be used to append an item to a list.
my_list.append("New Item")
The pop method can be used to remove an item using it's index
my_list.pop(0)





ADVANCED DATA TYPES

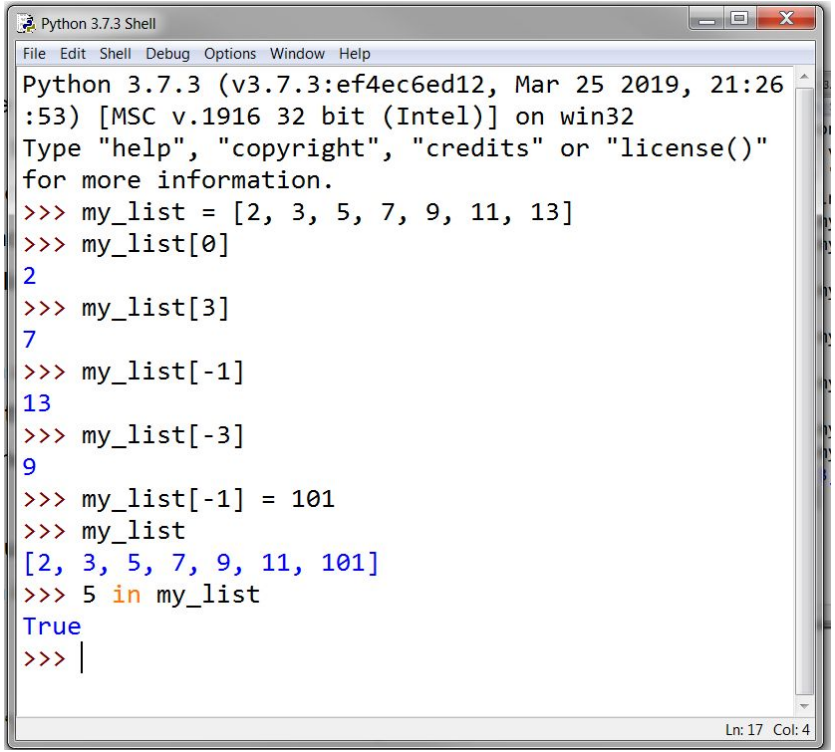
LIST

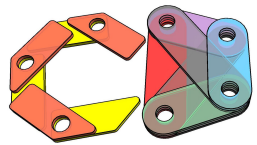
Since lists are sequences of elements you can iterate through all the elements in a list. You can also select an element from a list by it's index. The index is the order in which an item appears in the list. It's important to note that the index starts at zero. To get the value of the first item in a list you would type my_list[0].

Elements can also be index from the end of the list using a negative index. Since the first item in the list is 0 then last item would be -1. You can get the last item in the list by typing my_list[-1].

You can also use the index to change the value of an object in a list, and re-assign it to a different value:
my_list[0] = 5

The keyword "in" can be used to determine if a given object is contained in a list





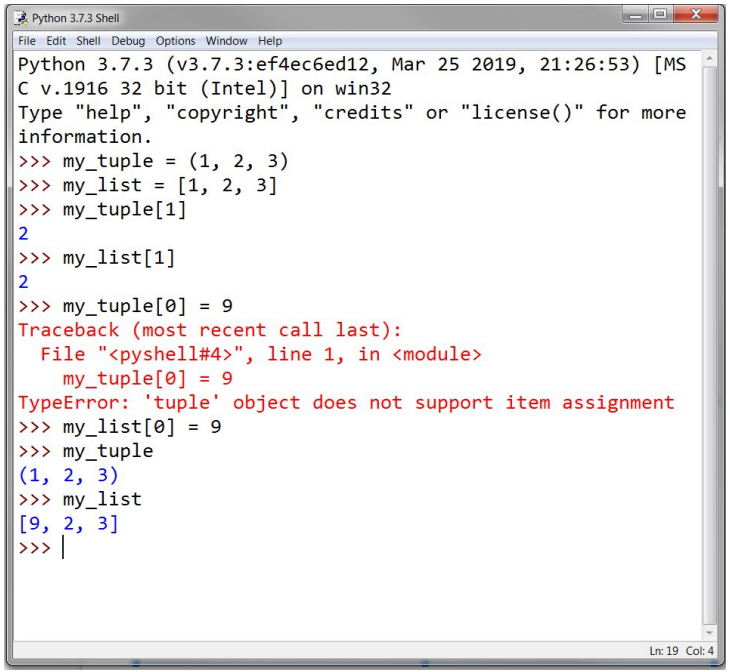
ADVANCED DATA TYPES

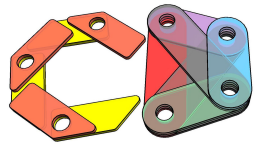
Tuple

Tuples aren't used very often, but it's good to know what they are. A tuple is the same thing as a list except that tuples are immutable. This means that they can't change after they have been created.

Tuples are created by defining them with parenthesis:

my_tuple = (1, 2, 3)





ADVANCED DATA TYPES

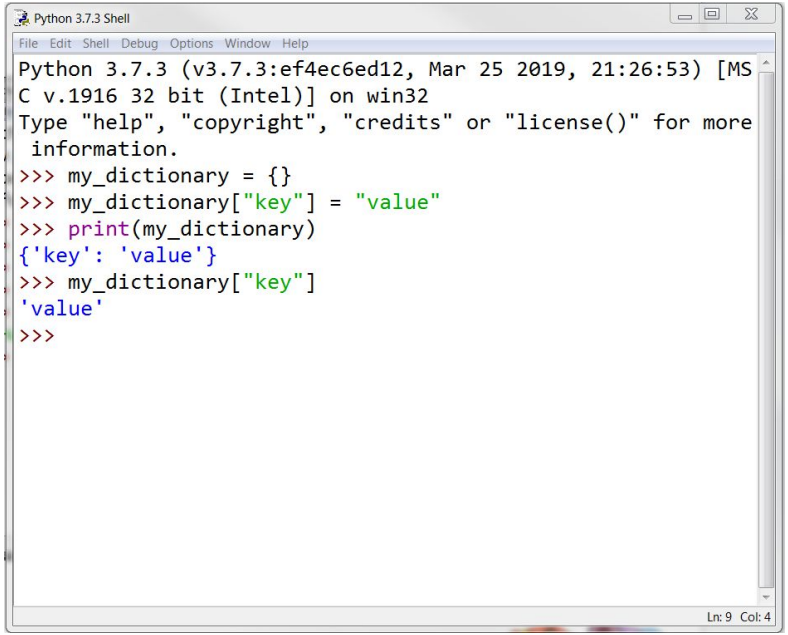
DICTIONARY

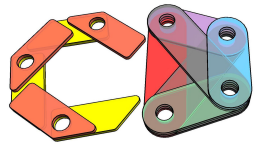
A dictionary is similar to a list in that it is a sequence of elements; however, each element in a dictionary has a key and a value pair. For every key in a dictionary there will be a corresponding value. Just like how every word in a dictionary has definition. Keys and values can be any data type.

Dictionaries are defined with curly braces “{}”. To create a new variable as an empty dictionary you would type my_dictionary = {}. You can also define initial values when you create the dictionary: my_dictionary = { “foo” : “bar”, “green” : “blue”, “Hello” : “World”}

You add elements to a dictionary by defining the key and setting it’s value:
my_dictionary[“key”] = “value”

The above code would add a key called “key”, with a corresponding value of “value”





ADVANCED DATA TYPES

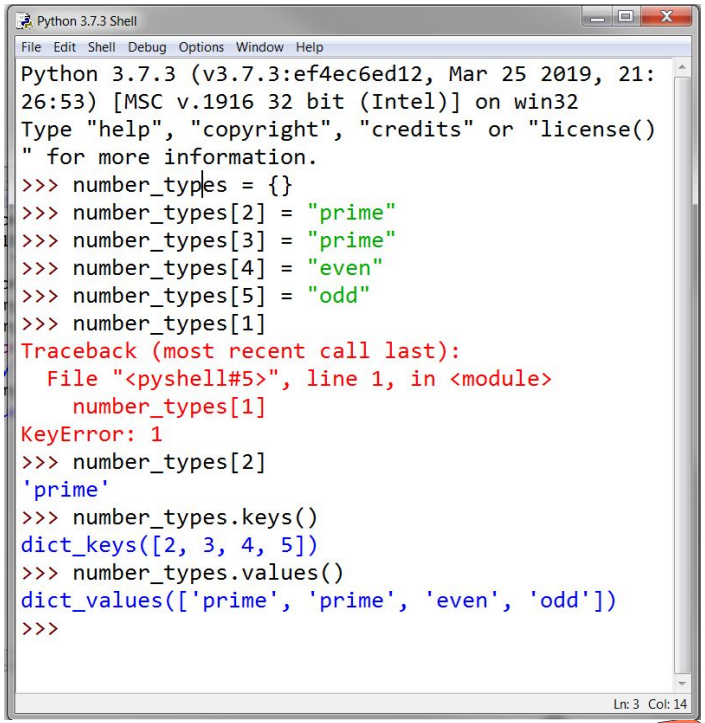
DICTIONARY

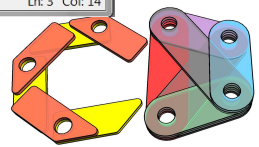
Elements in a dictionary are retrieved by indexing their key. If you have a key of 2 in your dictionary you can use the index 2 to retrieve the corresponding value.

You can use the method keys() to get all the keys in the dictionary. Similarly, you can use the method values() to get all the values.

It was mentioned that dictionaries are sequences. It's important to know that unlike lists dictionaries have no order. If you retrieve all the keys of a dictionary the order may not always be the same.

Similar to lists you can iterate over all the keys in a dictionary.



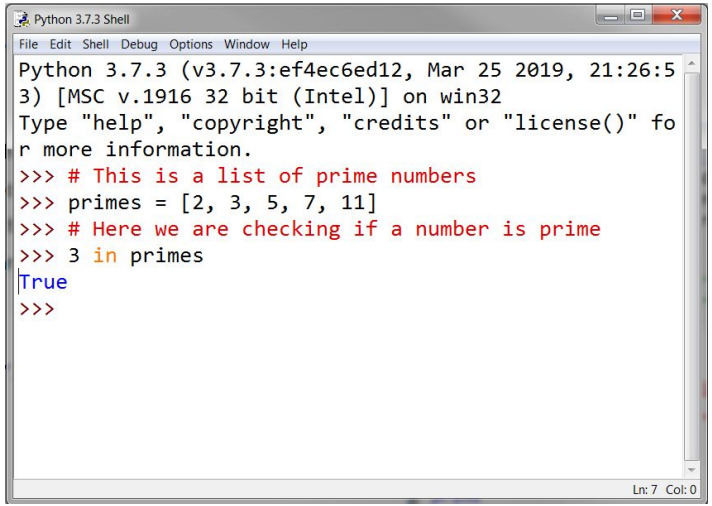


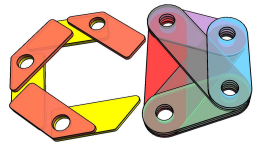
ADVANCED DATA TYPES

COMMENTS

Comments are notes to the person who is reading your code. It's helpful to write a lot of comments in your code so that you or somebody else can understand what the code is doing when you look back at it months or years later. Commenting your code is really important.

In python a comment is defined by "#". Anything following that character is a comment.





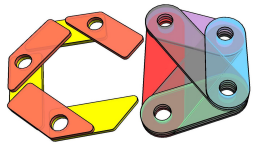
CONTROL STRUCTURES

PROGRAM FLOW

CONTROL STRUCTURES

CONDITIONAL STATEMENTS & LOOPS

Most of the work done by the code is done through control structures. Python scripts, and all code in general, executes sequentially. Control structures allow you to define when certain parts of the code should run, and allow you to loop over the code multiple times to run the same operation of a list of elements. Control structures are what allow you to do that.



CONTROL STRUCTURES

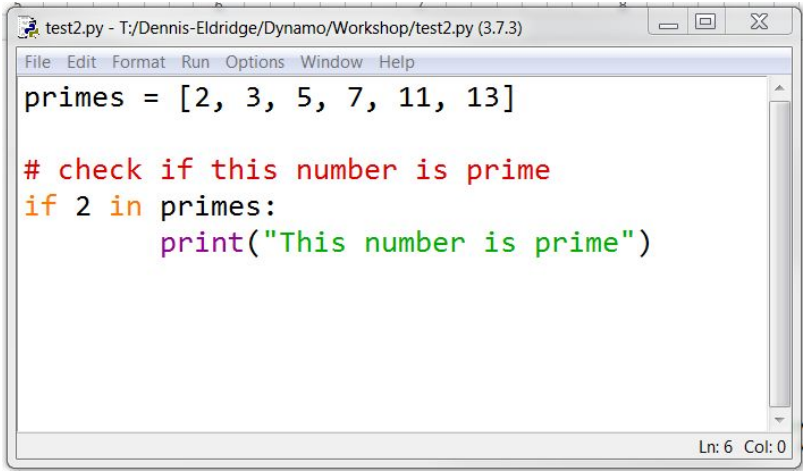
IF THEN

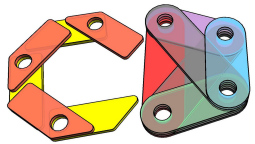
The if then control structures defines some condition in which the subsequent code will run. Here is the format:

if condition:

    conditional code

Python is very particular about the format so it's important to take note of a few things. The if statement always starts with a lowercase if. The condition must evaluate to a True or False statement, and it must end with a colon ":". If the condition is true it will run the conditional code. The conditional code **must** be indented by either 1 tab or 4 spaces or Python will return an error.





CONTROL STRUCTURES

IF THEN ELSE

If the condition is False, then it may be useful to do something different. In this case you can use the else statement following the conditional code. Here is the format:

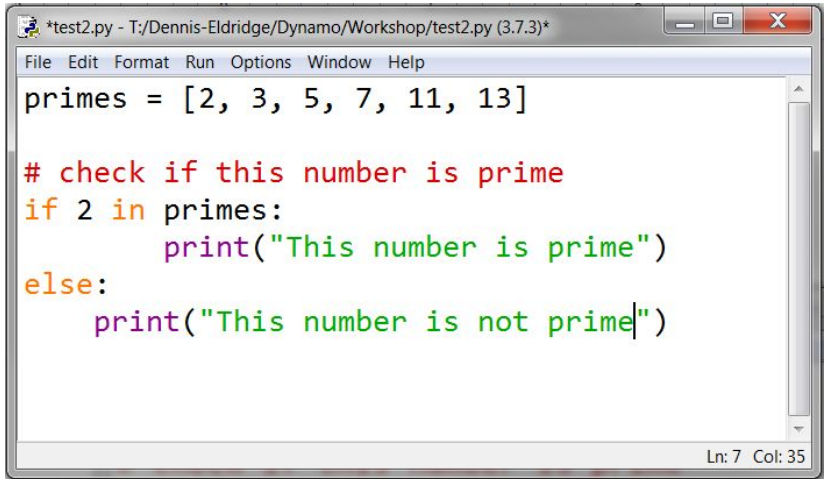
if condition:

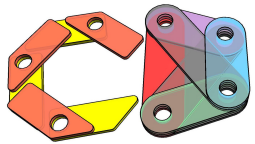
    conditional code

else:

    conditional code

As you can see if the condition is false then the else conditional code code will run. The else statement must be on the same line as the if statement, and must be followed by a colon ":". Any conditional code following the else statement must be indented by 1 tab or 4 spaces.



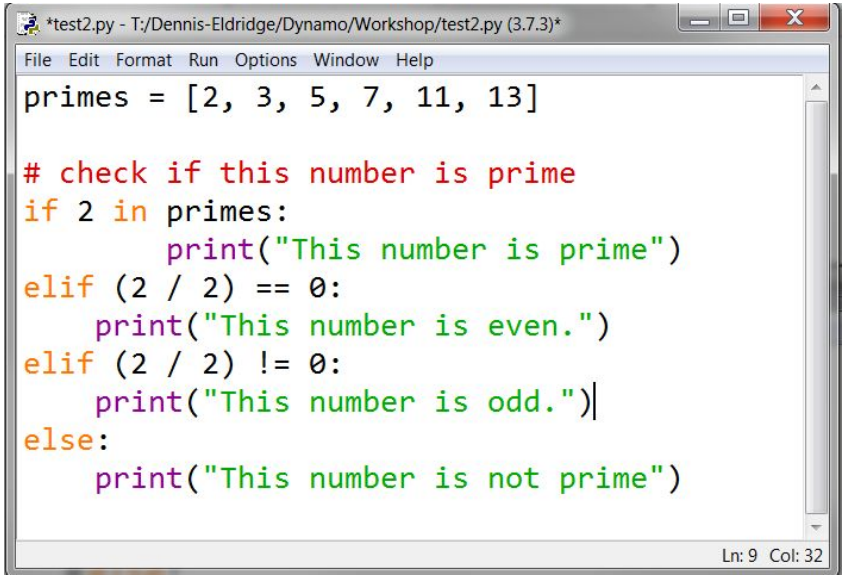


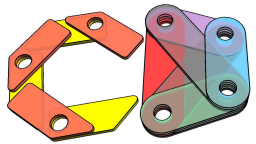
CONTROL STRUCTURES

IF THEN ELIF ELSE

What if you want to test multiple conditions? Python has an elif statement that will run if the first statement is false. You can have as many elif statements as you want. After all your elif statements you can have a final else statement. The format is similar to the if statement:

```
if condition:
        conditional code
elif condition:
        conditional code
elif condition:
        conditional code
else:
        conditional code
```



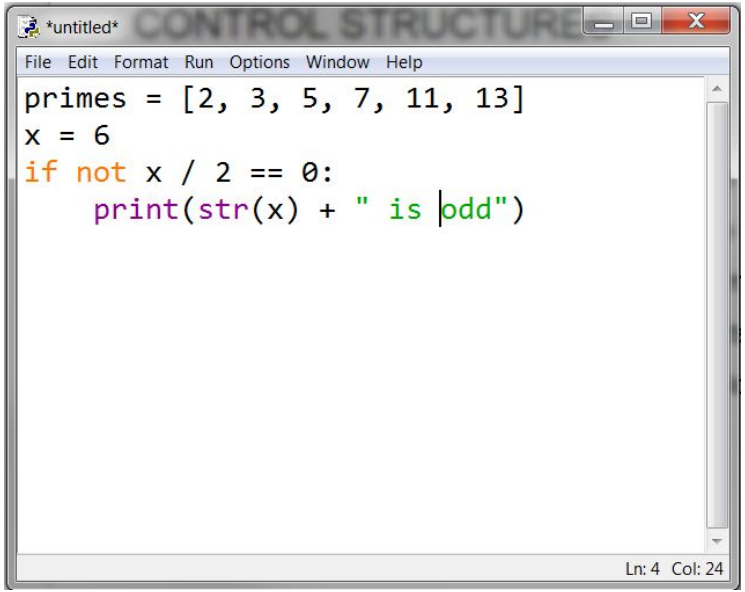


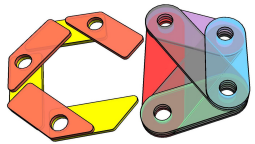
CONTROL STRUCTURES

NOT

Python also has a few addition options for more complex and powerful conditional statements. These are the logical “and”, “or”, and “not”. Those of you use use a lot of filters may be familiar with how they work. The not statement simply inverts the boolean value. not True would equal False, and not False would equal True

Truth table	
X	NOT X
False	True
True	False



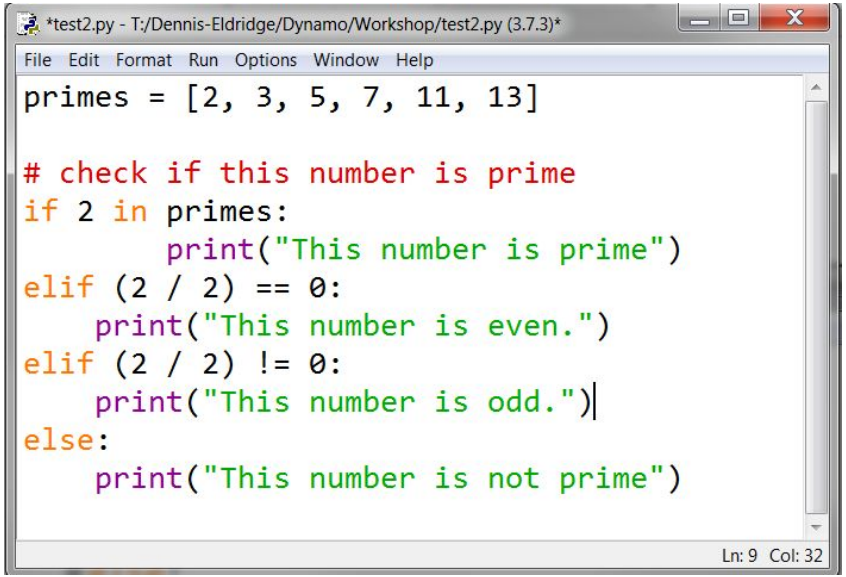


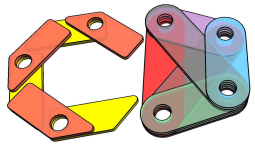
CONTROL STRUCTURES

AND

For an “and” statement to be true both operators need to be true.

Truth table		
X	Y	X AND Y
False	False	False
False	True	False
True	False	False
True	True	True



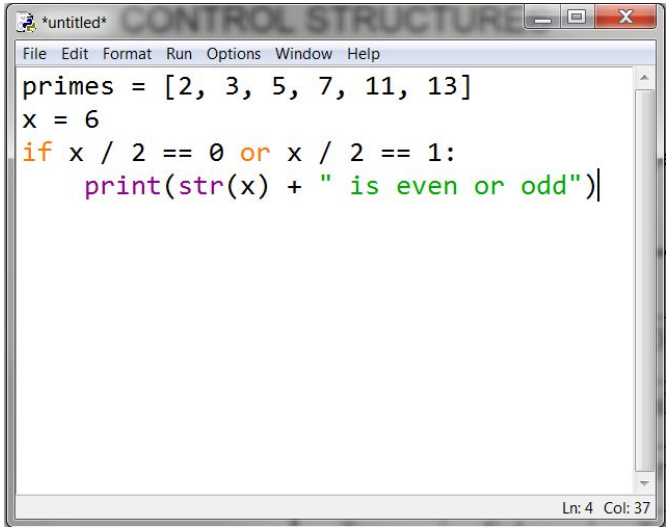


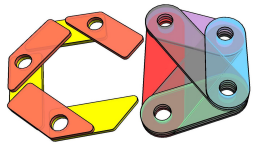
CONTROL STRUCTURES

OR

Or statements are true if either or both of the operators are true

Truth table		
X	Y	X OR Y
False	False	False
False	True	True
True	False	True
True	True	True





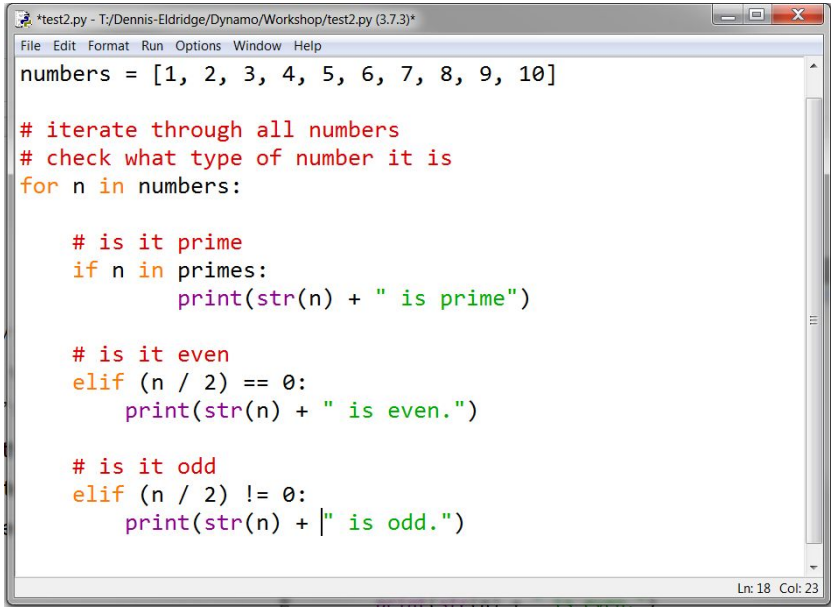
CONTROL STRUCTURES

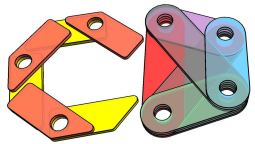
FOR LOOP

The next control structure, the “for” loop, will allow you to iterate over every value in a list. The syntax of “for” loops are very similar to that of “if” statements. You start with the keyword “for” then you define a variable for the instance of the list you will iterate through (n below), and then you write “in” which list you want to iterate over. Similar to an “if” statement any code within the “for” loop needs to be indented with 1 tab or 4 spaces.

for variable in list:

    iterative statements on variable





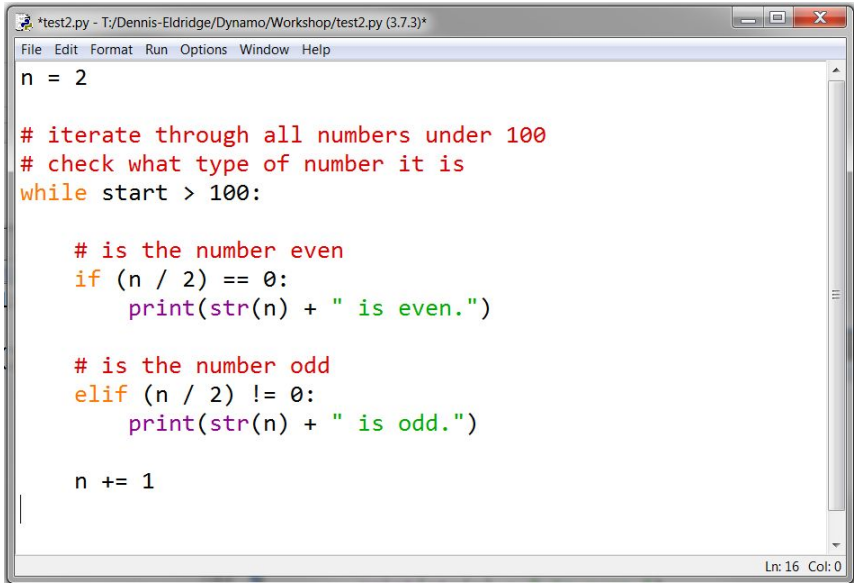
CONTROL STRUCTURES

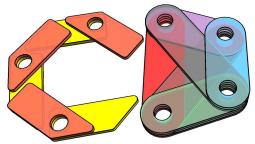
WHILE LOOP

A while loop is a mix between a for loop and a conditional statement. It will keep iterating as long as some condition is true. You need to be careful with your conditions when creating while loops, because it's possible to create infinite loops if the conditional statement never becomes false. As you can see below the syntax for while statements are similar to the other control structures. Here is the format of while statements:

while condition:

    conditional statements





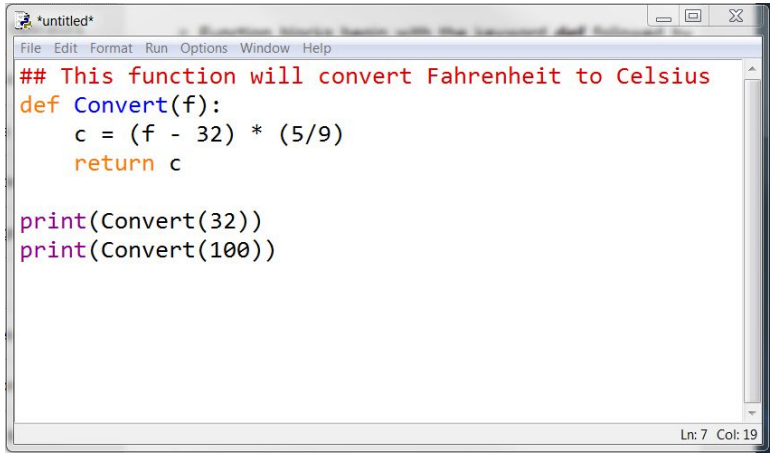
CONTROL STRUCTURES

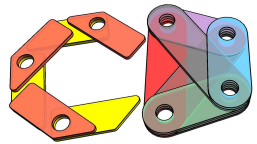
FUNCTIONS

A function is a similar to a formula in math. It's a predefined sequence of operations that output some value, or make some change to the data. Functions are really useful for some action that needs to be repeated a large number of times. Using them can also make your code easier to read. Here is the syntax for a function:

```
def FunctionName(input):
        function code
        return output
```

Functions start with the "def" keyword followed by the name of the function. Then you need two parentheses followed by a colon. If the function takes any input then you list each input variable name separated by a comma. Then you'll indent and write your code for the function. A function can return a value or variable, which is indicated with the "return" keyword.





CLASSES

BRINGING IT ALL TOGETHER

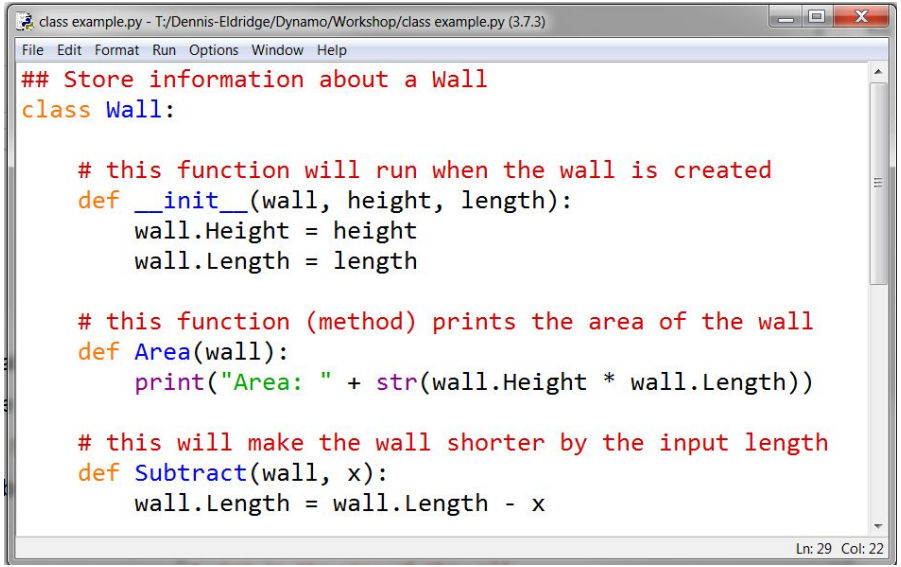
CLASSES

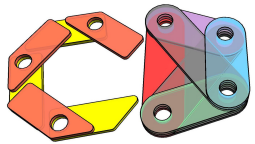
DEFINITION

are a collection of all the things that we've talked about thus far. You can think of it as a template that is predefined to contain specific variables and functions that helps to group information, perform an action, or both. Classes are defined with the this syntax:

```
class ClassName:
        def __init__(self):
                Initialization code
```

Properties of the class are defined as the wall class.Property. Functions within a class are called methods, and are accessed as class.Method() with parenthesis.





CLASSES

CREATION

In the __init__ function we noted the self variable. That variable is used to define properties of the class. If you use the variable name, and then a period that will create a property of the class with that name, which you can then define a value. This value could be changed later.

```
def __init__(self):
        self.Parameter = value
```

Within your class you can also define functions. Functions within a class are defined the same way you would define them outside a class. A function within a class is referred to as a method. The first parameter of a function has to be the self variable followed by a comma separated list of optional variables.

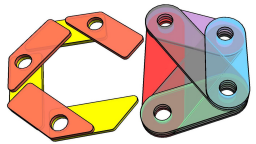
```
def Action(self):
        return code
```

class example.py - T:/Dennis-Eldridge/Dynamo/Workshop/class example.py (3.7.3)

File Edit Format Run Options Window Help

```
## Store information about a Wall
class Wall:

    # this function will run when the wall is created
    def __init__(wall, height, length):
        wall.Height = height
        wall.Length = length

    # this function (method) prints the area of the wall
    def Area(wall):
        print("Area: " + str(wall.Height * wall.Length))

    # this will make the wall shorter by the input length
    def Subtract(wall, x):
        wall.Length = wall.Length - x
```

Ln: 29 Col: 22



CLASSES

INITIALIZATION

Once the class is defined you can create instances from the class template by assigning the class to a variable like so:

exterior = Wall(2, 5)

Once the instance of the class is created you can access the properties of the class with a period, and the name of the property

exterior.Height

You can access the method of the class by using a period, and then the name of the function. For a method you'll also need parentheses and any required parameters

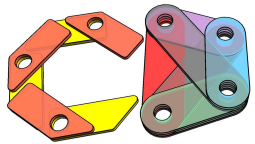
exterior.Area()

class example.py - T:/Dennis-Eldridge/Dynamo/Workshop/class example.py (3.7.3)

File Edit Format Run Options Window Help

```
# create an instance of the class
exterior = Wall(2, 5)

# find the height
print(exterior.Height)
print(exterior.Length)

# what is the size of the wall
exterior.Area()

# change the wall size
exterior.Subtract(5)
print(exterior.Length)
exterior.Area()

# retsore the size
exterior.Length = 5
print(exterior.Length)
```

Ln: 34 Col: 22

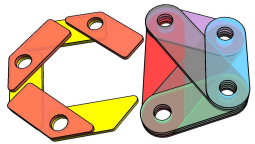


CLASSES

REVIEW

Don’t worry if you get confused about how to create a class. Class creation is not necessary in the vast majority of cases for Dynamo. It’s important to know how classes work though, because everything in the Revit API is a class.

It’s important to remember that a class is a template used to represent a type of object. Classes contain properties which contain useful values, and methods that return some value, or take some action on the class.



CLASSES

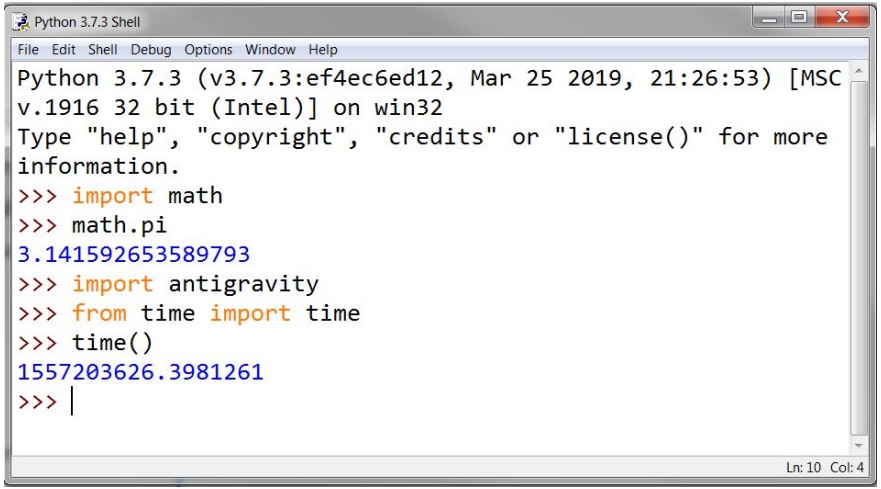
IMPORTING

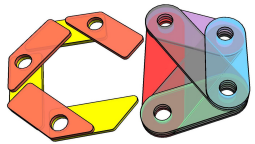
Importing is the process of loading classes from another file into your current project. There are a lot of modules (collections of classes) that come with Python, which you can import. We will also be importing Revit classes from the Revit API soon enough. The syntax is :

import module

from module import class

Start with the keyword “import” and then the module name. If you only want to insert a specific class from a module, then you can use the keyword “from” the module name, and then the keyword “import”, followed by the class name.



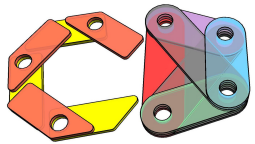


CLASSES

THAT'S IT!

That's all you need to know about the basic of python. You can use what you just learned to create really powerful programs, even if it doesn't seem like it as a beginner. Of course, there are a lot of extra details to learn, but this covers the majority of the language.

As you get more experience you'll learn to build bigger and bigger programs to do more complicated things with your code.



CLASSES

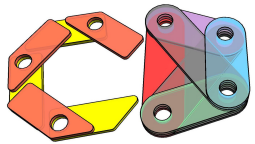
REVIT API

WHY REVIT API?

REVIT NODE LIBRARY

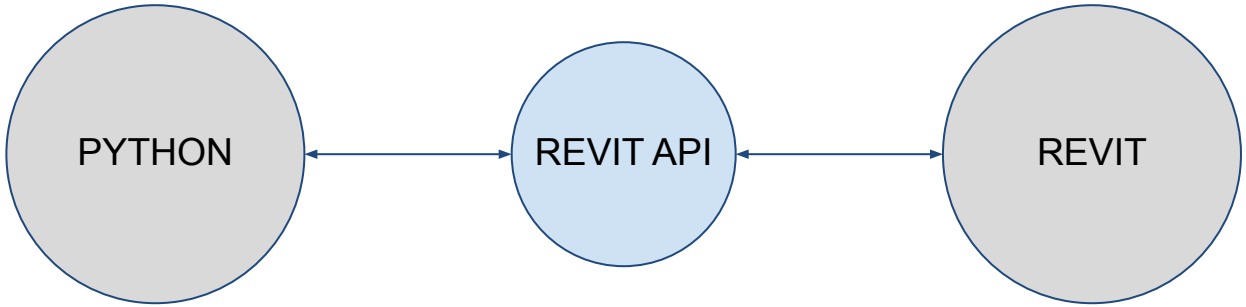
Revit Node Library provides access to many classes and methods within the revit for creating geometries and making changes to revit elements but there are still some limitations; we use Revit API to go beyond what R node and access all classes and methods within the Revit Project

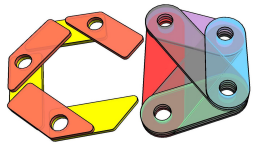
```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

clr.AddReference('RevitNodes')
import Revit

# The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

# Place your code below this line

# Assign your output to the OUT variable.
OUT = dir(Revit.Elements)
```



REVIT API

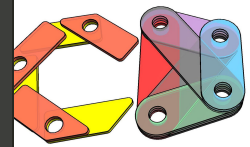
APPLICATION PROGRAMMING INTERFACE





IMPORTING API

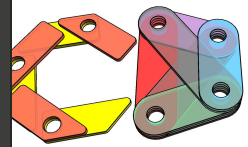
```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#import RevitAPI

clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *

clr.AddReference('RevitServices')
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

# The inputs to this node will be stored as a list in the IN variables.
familytype = UnwrapElement(IN[0])
output = []

# Place your code below this line
#Assign Document
doc = DocumentManager.Instance.CurrentDBDocument

#Start Transaction
TransactionManager.Instance.EnsureInTransaction(doc)


for x in range (0,100,10):
    fam = doc.Create.NewFamilyInstance(XYZ(x,0,0),
    familytype,Structure.StructuralType.NonStructural)
    output.append(fam)

#End Transaction
TransactionManager.Instance.TransactionTaskDone()

# Assign your output to the OUT variable.
OUT = output
```



Document Manager Class

Transaction Manager Class

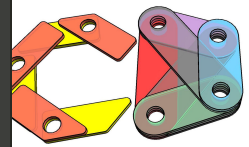
```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#import RevitAPI

clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *

clr.AddReference('RevitServices')
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

# The inputs to this node will be stored as a list in the IN variables.
familytype = UnwrapElement(IN[0])
output = []

# Place your code below this line
#Assign Document
doc = DocumentManager.Instance.CurrentDBDocument

#Start Transaction
TransactionManager.Instance.EnsureInTransaction(doc)


for x in range (0,100,10):
    fam = doc.Create.NewFamilyInstance(XYZ(x,0,0),
    familytype,Structure.StructuralType.NonStructural)
    output.append(fam)

#End Transaction
TransactionManager.Instance.TransactionTaskDone()

# Assign your output to the OUT variable.
OUT = output
```



Assigning document

Starting a transaction

```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#import RevitAPI

clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *

clr.AddReference('RevitServices')
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

# The inputs to this node will be stored as a list in the IN variables.
familytype = UnwrapElement(IN[0])
output = []

# Place your code below this line
#Assign Document
doc = DocumentManager.Instance.CurrentDBDocument

#Start Transaction
TransactionManager.Instance.EnsureInTransaction(doc)


for x in range (0,100,10):
    fam = doc.Create.NewFamilyInstance(XYZ(x,0,0),
    familytype,Structure.StructuralType.NonStructural)
    output.append(fam)

#End Transaction
TransactionManager.Instance.TransactionTaskDone()

# Assign your output to the OUT variable.
OUT = output
```



- Namespaces
  - Autodesk.Revit.ApplicationServices Namespace
  - Autodesk.Revit.Attributes Namespace
  - Autodesk.Revit.Creation Namespace
    - Application Class
    - AreaCreationData Class
    - Document Class
    - eRefFace Enumeration
    - FamilyInstanceCreationData Class
      - FamilyInstanceCreationData Members
      - FamilyInstanceCreationData Constructor
        - FamilyInstanceCreationData Constructor (Object)
        - FamilyInstanceCreationData Constructor (FamilySymbol, IList(XYZ))
        - FamilyInstanceCreationData Constructor (Face, Line, FamilySymbol)
        - FamilyInstanceCreationData Constructor (XYZ, FamilySymbol, StructuralType)
        - FamilyInstanceCreationData Constructor (Curve, FamilySymbol, Level, StructuralType)
        - FamilyInstanceCreationData Constructor (Face, XYZ, XYZ, FamilySymbol)
        - FamilyInstanceCreationData Constructor (XYZ, FamilySymbol, Element, StructuralType)
        - FamilyInstanceCreationData Constructor (XYZ, FamilySymbol, Level, StructuralType)
        - FamilyInstanceCreationData Constructor (XYZ, FamilySymbol, Element, Level, StructuralTy
        - FamilyInstanceCreationData Constructor (XYZ, FamilySymbol, XYZ, Element, StructuralTyp
      - FamilyInstanceCreationData Methods
      - FamilyInstanceCreationData Properties
    - FamilyItemFactory Class
    - ItemFactoryBase Class
  - Autodesk.Revit.DB Namespace
  - Autodesk.Revit.DB.Analysis Namespace

FamilyInstanceCreationData Constructor (XYZ, FamilySymbol, StructuralType)

Initializes a new instance of the FamilyInstanceCreationData class

Namespace: Autodesk.Revit.Creation
Assembly: RevitAPI (in RevitAPI.dll) Version: 19.0.0.0 (19.2.1.1)

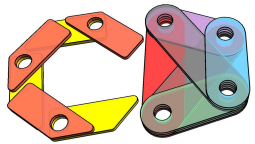
Syntax

C#

```
public FamilyInstanceCreationData(
        XYZ location,
        FamilySymbol symbol,
        StructuralType structuralType
)
```

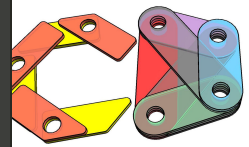
Visual Basic

```
Public Sub New ( _
        location As XYZ, _
        symbol As FamilySymbol, _
        structuralType As StructuralType _
)
```



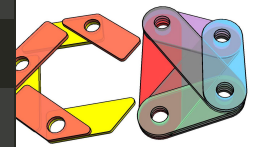
Creating family

```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#import RevitAPI

clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *

clr.AddReference('RevitServices')
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

# The inputs to this node will be stored as a list in the IN variables.
familytype = UnwrapElement(IN[0])
output = []

# Place your code below this line
#Assign Document
doc = DocumentManager.Instance.CurrentDBDocument

#Start Transaction
TransactionManager.Instance.EnsureInTransaction(doc)


for x in range (0,100,10):
    fam = doc.Create.NewFamilyInstance(XYZ(x,0,0),
    familytype,Structure.StructuralType.NonStructural)
    output.append(fam)

#End Transaction
TransactionManager.Instance.TransactionTaskDone()

# Assign your output to the OUT variable.
OUT = output
```



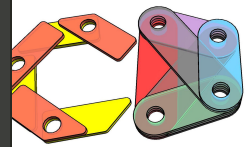
Ending transaction

```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#import RevitAPI

clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *

clr.AddReference('RevitServices')
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

# The inputs to this node will be stored as a list in the IN variables.
familytype = UnwrapElement(IN[0])
output = []

# Place your code below this line
#Assign Document
doc = DocumentManager.Instance.CurrentDBDocument

#Start Transaction
TransactionManager.Instance.EnsureInTransaction(doc)


for x in range (0,100,10):
    fam = doc.Create.NewFamilyInstance(XYZ(x,0,0),
    familytype,Structure.StructuralType.NonStructural)
    output.append(fam)

#End Transaction
TransactionManager.Instance.TransactionTaskDone()

# Assign your output to the OUT variable.
OUT = output
```



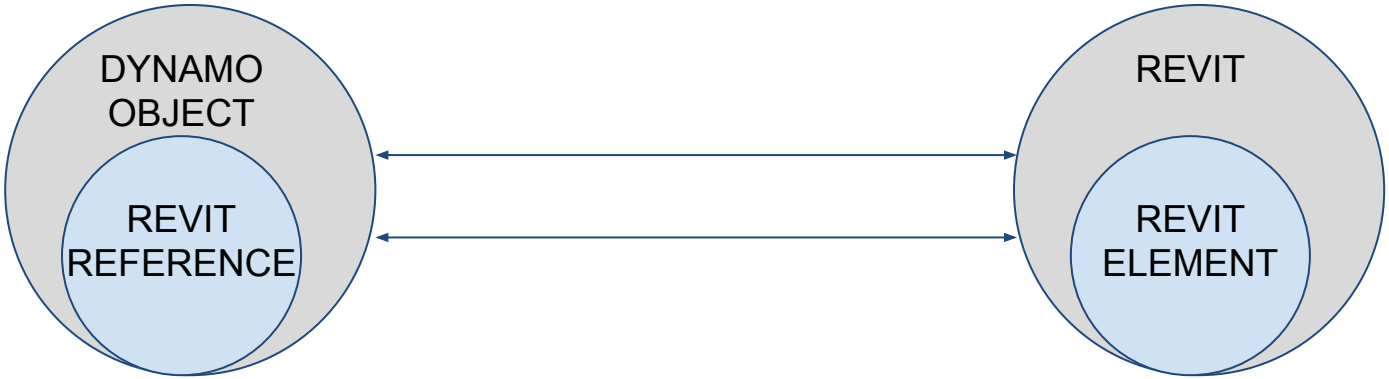
Wrapping and Unwrapping

```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#import RevitAPI

clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *

clr.AddReference('RevitServices')
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

# The inputs to this node will be stored as a list in the IN variables.
familytype = UnwrapElement(IN[0])
output = []

# Place your code below this line
#Assign Document
doc = DocumentManager.Instance.CurrentDBDocument

#Start Transaction
TransactionManager.Instance.EnsureInTransaction(doc)


for x in range (0,100,10):
    fam = doc.Create.NewFamilyInstance(XYZ(x,0,0),
    familytype,Structure.StructuralType.NonStructural)
    output.append(fam)

#End Transaction
TransactionManager.Instance.TransactionTaskDone()

# Assign your output to the OUT variable.
OUT = output
```



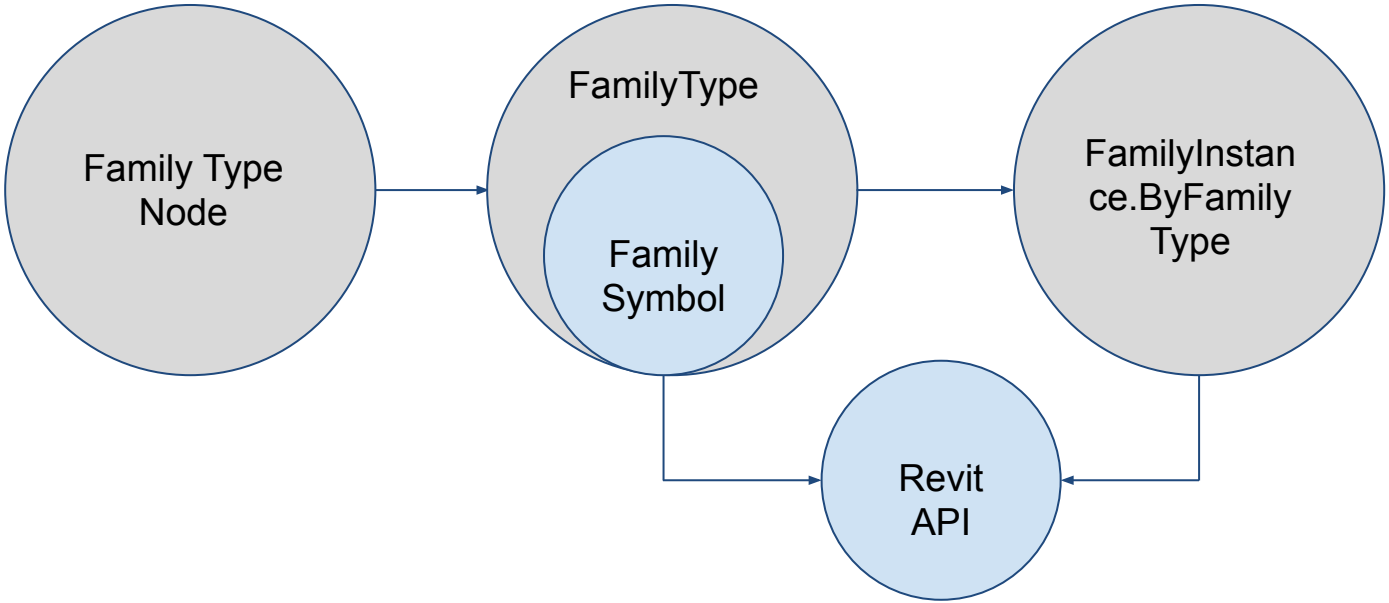
CONVERSIONS

REVIT ELEMENT CONVERSION



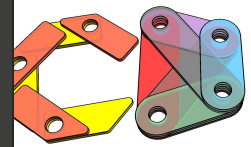
CONVERSIONS

FAMILY TYPE



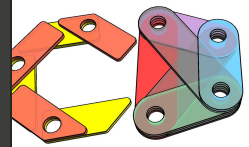
Wrapping and Unwrapping

```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#import RevitAPI

clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *

clr.AddReference('RevitServices')
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

# The inputs to this node will be stored as a list in the IN variables.
familytype = UnwrapElement(IN[0])
output = []

# Place your code below this line
#Assign Document
doc = DocumentManager.Instance.CurrentDBDocument

#Start Transaction
TransactionManager.Instance.EnsureInTransaction(doc)


for x in range (0,100,10):
    fam = doc.Create.NewFamilyInstance(XYZ(x,0,0),
    familytype,Structure.StructuralType.NonStructural)
    output.append(fam)

#End Transaction
TransactionManager.Instance.TransactionTaskDone()

# Assign your output to the OUT variable.
OUT = output
```



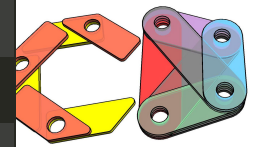
Import Extensions

```
# Enable Python Support and load DesignScript Library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#Import Revit API
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *

#Import Manger classes
clr.AddReference('RevitServices')
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

#Import DSType method
clr.AddReference('RevitNodes')
import Revit
clr.ImportExtensions(Revit.Elements)

#Inputs to this Node
fam = UnwrapElement(IN[0])
output = []

#Assign Document
doc = DocumentManager.Instance.CurrentDBDocument

#Start Transaction
TransactionManager.Instance.EnsureInTransaction(doc)

for x in range (0,50,10):
    famarray = doc.Create.NewFamilyInstance(XYZ(x,x,0),fam,
    Structure.StructuralType.NonStructural)
    wrappedfamarray = famarray.ToDSType(False)
    output.append(wrappedfamarray)
```



```python
# Enable Python Support and load DesignScript Library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#Import Revit API
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *

#Import Manger classes
clr.AddReference('RevitServices')
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

#Import DSType method
clr.AddReference('RevitNodes')
import Revit
clr.ImportExtensions(Revit.Elements)

#Inputs to this Node
fam = UnwrapElement(IN[0])
output = []

#Assign Document
doc = DocumentManager.Instance.CurrentDBDocument

#Start Transaction
TransactionManager.Instance.EnsureInTransaction(doc)

for x in range (0,50,10):
    famarray = doc.Create.NewFamilyInstance(XYZ(x,x,0),fam,
    Structure.StructuralType.NonStructural)
    wrappedfamarray = famarray.ToDSType(False)
    output.append(wrappedfamarray)
```

wrapping elements

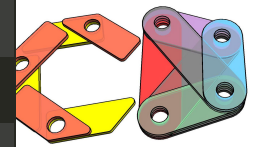


```python
# Enable Python Support and load DesignScript Library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#Import Revit API
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *

#Import Manger classes
clr.AddReference('RevitServices')
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

#Import DSType method
clr.AddReference('RevitNodes')
import Revit
clr.ImportExtensions(Revit.Elements)

#Inputs to this Node
fam = UnwrapElement(IN[0])
output = []

#Assign Document
doc = DocumentManager.Instance.CurrentDBDocument

#Start Transaction
TransactionManager.Instance.EnsureInTransaction(doc)

for x in range (0,50,10):
    famarray = doc.Create.NewFamilyInstance(XYZ(x,x,0),fam,
    Structure.StructuralType.NonStructural)
    wrappedfamarray = famarray.ToDSType(False)
    output.append(wrappedfamarray)
```

wrapping elements

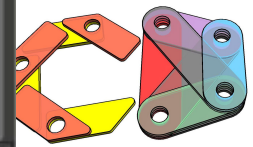
(Fals > Dynamo Bound)

(True > Revit Bound)



Converting Location Coordinates to Points or Vectors in Revit

```
from Autodesk.Revit.DB import *

#Import Manger classes
clr.AddReference('RevitServices')
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

#Import DSType method
clr.AddReference('RevitNodes')
import Revit
clr.ImportExtensions(Revit.Elements)
clr.ImportExtensions(Revit.GeometryConversion)

#Inputs to this Node
fam = UnwrapElement(IN[0])
output = []

#Assign Document
doc = DocumentManager.Instance.CurrentDBDocument

#Start Transaction
TransactionManager.Instance.EnsureInTransaction(doc)

for x in range (0,50,10):
    famarray = doc.Create.NewFamilyInstance(XYZ(x,x,0),fam,
    Structure.StructuralType.NonStructural)
    wrappedfamarray = famarray.ToDSType(False)

    xyz = XYZ(x,x,0)
    output.append(xyz.ToVector())



#Assign your output to the OUT variable
OUT = output
```



CLASSES

CONTAINERS FOR FUNCTIONS AND OTHER DATA TYPES

When accessing the Revit API Python most objects are made up of classes. Classes are a collection of properties and methods.

- **Wall**

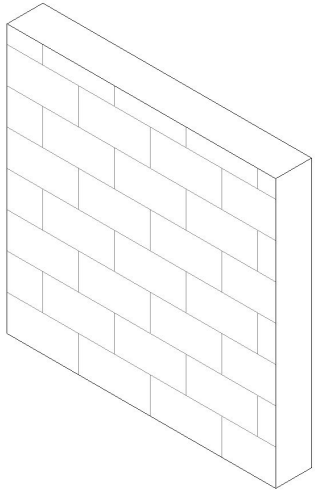
*Properties are similar to Parameters in Revit.

- **Wall.Name**

Methods are functions (actions) you can perform with that class. Methods are always designated with parentheses, and may require input between the parenthesis

- **Wall.Flip()**

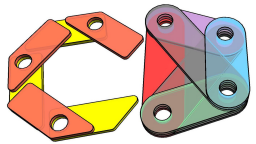
CLASS
WALL



*PROPERTY
Wall.**Name**

METHOD
Wall.**Flip()**

*Technically in Python this is called an Attribute, but in .Net terminology, which the Revit API uses, it's called a Property.



SYNTAX STRUCTURE

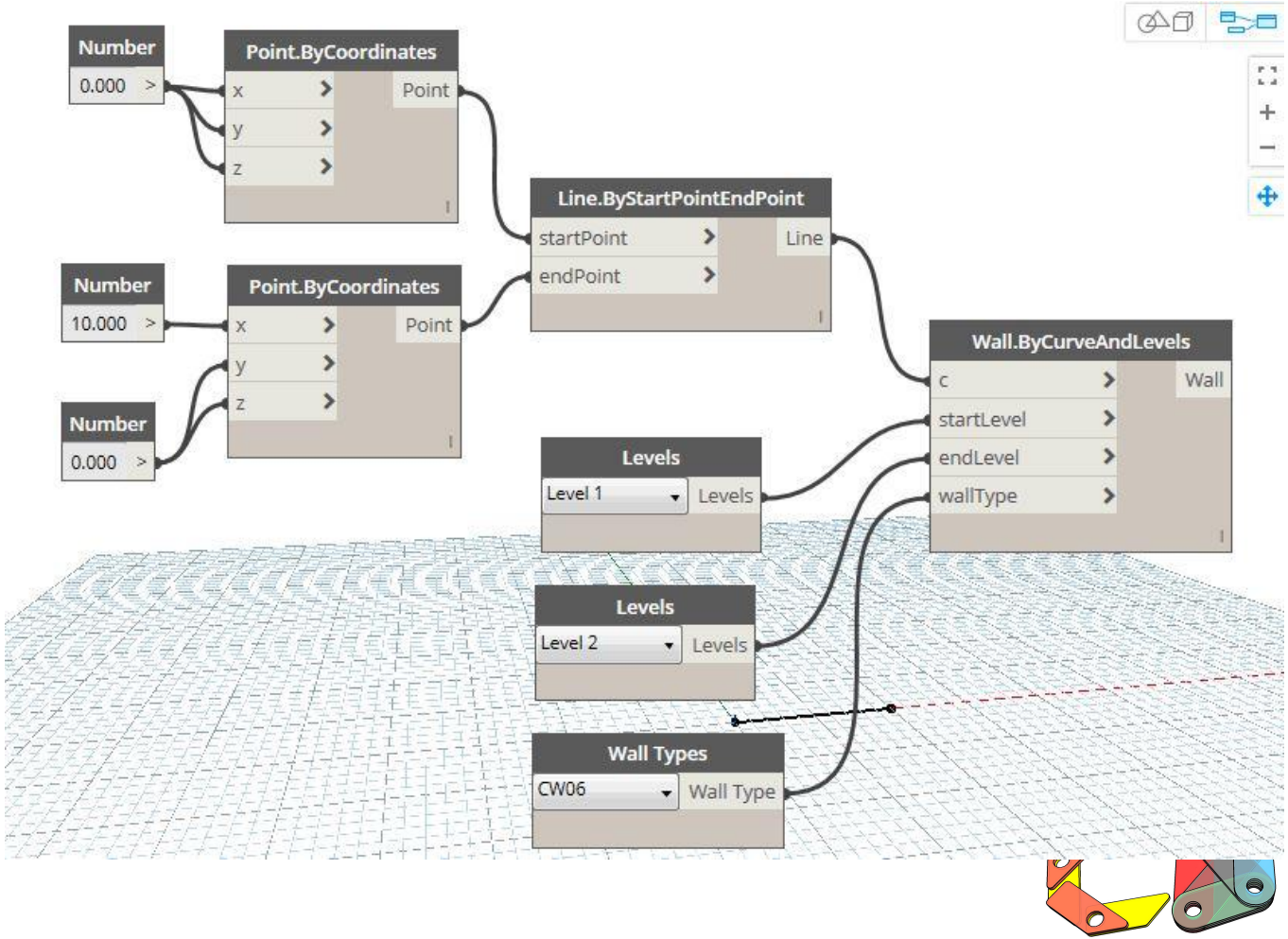
CREATE A WALL SAMPLE

SYNTAX

DYNAMO NODES

Create a simple wall using Dynamo.

Recreate this in Python.



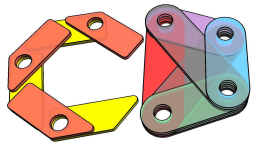
SYNTAX

ADD REFERENCE

Import all the modules we will need for this script.

Keep references in a .txt file and copy and past.

```
import clr
#Import module for Revit
clr.AddReference("RevitNodes")
import Revit
clr.ImportExtensions(Revit.Elements)
#import module for the Document and transactions
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
#import Revit API
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
#get the document
doc = DocumentManager.Instance.CurrentDBDocument
#Dynamo input
baseLevel = UnwrapElement(IN[0])
topLevel = UnwrapElement(IN[1])
wallType = UnwrapElement(IN[2])
#create point for line
pt1 = XYZ(0, 0, 0)
pt2 = XYZ(10, 0, 0)
#use safe transaction with Revit
TransactionManager.Instance.EnsureInTransaction(doc)
#create line
line = Line.CreateBound(pt1, pt2)
#create wall using Revit API
wall = Wall.Create(doc, line, baseLevel.Id, False)
#Set the wall type to Dynamo input
wall.WallType = wallType
#Get the top constarint parameter using built in parameter
#Revit Document shows it as WALL_HEIGHT_TYPE
topConstraint = wall.get_Parameter(BuiltInParameter.WALL_HEIGHT_TYPE)
#Set the top constraint
topConstraint.Set(topLevel.Id)
#Finish Transaction with task done
TransactionManager.Instance.TransactionTaskDone()

OUT = wall
```

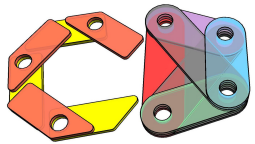


SYNTAX

CURRENT DOCUMENT

From the imported document manager get the instance of the current document.

```
import clr
#Import module for Revit
clr.AddReference("RevitNodes")
import Revit
clr.ImportExtensions(Revit.Elements)
#import module for the Document and transactions
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
#import Revit API
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
#get the document
doc = DocumentManager.Instance.CurrentDBDocument
#Dynamo input
baseLevel = UnwrapElement(IN[0])
topLevel = UnwrapElement(IN[1])
wallType = UnwrapElement(IN[2])
#create point for line
pt1 = XYZ(0, 0, 0)
pt2 = XYZ(10, 0, 0)
#use safe transaction with Revit
TransactionManager.Instance.EnsureInTransaction(doc)
#create line
line = Line.CreateBound(pt1, pt2)
#create wall using Revit API
wall = Wall.Create(doc, line, baseLevel.Id, False)
#Set the wall type to Dynamo input
wall.WallType = wallType
#Get the top constarint parameter using built in parameter
#Revit Document shows it as WALL_HEIGHT_TYPE
topConstraint = wall.get_Parameter(BuiltInParameter.WALL_HEIGHT_TYPE)
#Set the top constraint
topConstraint.Set(topLevel.Id)
#Finish Transaction with task done
TransactionManager.Instance.TransactionTaskDone()

OUT = wall
```

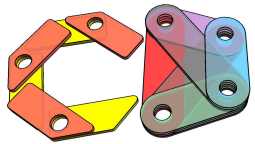


SYNTAX

DYNAMO INPUT

User will input base level, top level, and wall type. We've unwrapped Revit elements with RevitNodes.

```
import clr
#Import module for Revit
clr.AddReference("RevitNodes")
import Revit
clr.ImportExtensions(Revit.Elements)
#import module for the Document and transactions
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
#import Revit API
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
#get the document
doc = DocumentManager.Instance.CurrentDBDocument
#Dynamo input
baseLevel = UnwrapElement(IN[0])
topLevel = UnwrapElement(IN[1])
wallType = UnwrapElement(IN[2])
#create point for line
pt1 = XYZ(0, 0, 0)
pt2 = XYZ(10, 0, 0)
#use safe transaction with Revit
TransactionManager.Instance.EnsureInTransaction(doc)
#create line
line = Line.CreateBound(pt1, pt2)
#create wall using Revit API
wall = Wall.Create(doc, line, baseLevel.Id, False)
#Set the wall type to Dynamo input
wall.WallType = wallType
#Get the top constarint parameter using built in parameter
#Revit Document shows it as WALL_HEIGHT_TYPE
topConstraint = wall.get_Parameter(BuiltInParameter.WALL_HEIGHT_TYPE)
#Set the top constraint
topConstraint.Set(topLevel.Id)
#Finish Transaction with task done
TransactionManager.Instance.TransactionTaskDone()

OUT = wall
```

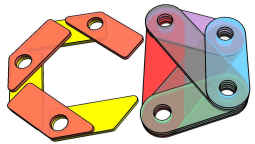


SYNTAX

POINTS

Create two points with arguments X Y and Z.

```
import clr
#Import module for Revit
clr.AddReference("RevitNodes")
import Revit
clr.ImportExtensions(Revit.Elements)
#import module for the Document and transactions
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
#import Revit API
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
#get the document
doc = DocumentManager.Instance.CurrentDBDocument
#Dynamo input
baseLevel = UnwrapElement(IN[0])
topLevel = UnwrapElement(IN[1])
wallType = UnwrapElement(IN[2])
#create point for line
pt1 = XYZ(0, 0, 0)
pt2 = XYZ(10, 0, 0)
#use safe transaction with Revit
TransactionManager.Instance.EnsureInTransaction(doc)
#create line
line = Line.CreateBound(pt1, pt2)
#create wall using Revit API
wall = Wall.Create(doc, line, baseLevel.Id, False)
#Set the wall type to Dynamo input
wall.WallType = wallType
#Get the top constarint parameter using built in parameter
#Revit Document shows it as WALL_HEIGHT_TYPE
topConstraint = wall.get_Parameter(BuiltInParameter.WALL_HEIGHT_TYPE)
#Set the top constraint
topConstraint.Set(topLevel.Id)
#Finish Transaction with task done
TransactionManager.Instance.TransactionTaskDone()

OUT = wall
```

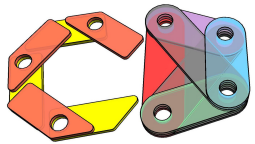


SYNTAX

START TRANSACTION

Using our Revit Services we use our Transaction Manager Instance to Ensure In Transaction our current document.

```
import clr
#Import module for Revit
clr.AddReference("RevitNodes")
import Revit
clr.ImportExtensions(Revit.Elements)
#import module for the Document and transactions
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
#import Revit API
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
#get the document
doc = DocumentManager.Instance.CurrentDBDocument
#Dynamo input
baseLevel = UnwrapElement(IN[0])
topLevel = UnwrapElement(IN[1])
wallType = UnwrapElement(IN[2])
#create point for line
pt1 = XYZ(0, 0, 0)
pt2 = XYZ(10, 0, 0)
#use safe transaction with Revit
TransactionManager.Instance.EnsureInTransaction(doc)
#create line
line = Line.CreateBound(pt1, pt2)
#create wall using Revit API
wall = Wall.Create(doc, line, baseLevel.Id, False)
#Set the wall type to Dynamo input
wall.WallType = wallType
#Get the top constarint parameter using built in parameter
#Revit Document shows it as WALL_HEIGHT_TYPE
topConstraint = wall.get_Parameter(BuiltInParameter.WALL_HEIGHT_TYPE)
#Set the top constraint
topConstraint.Set(topLevel.Id)
#Finish Transaction with task done
TransactionManager.Instance.TransactionTaskDone()

OUT = wall
```

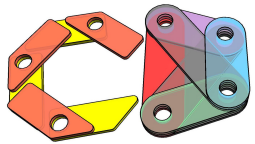


SYNTAX

CREATE LINE

Create a line which has two arguments. A start point and end point.

```
import clr
#Import module for Revit
clr.AddReference("RevitNodes")
import Revit
clr.ImportExtensions(Revit.Elements)
#import module for the Document and transactions
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
#import Revit API
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
#get the document
doc = DocumentManager.Instance.CurrentDBDocument
#Dynamo input
baseLevel = UnwrapElement(IN[0])
topLevel = UnwrapElement(IN[1])
wallType = UnwrapElement(IN[2])
#create point for line
pt1 = XYZ(0, 0, 0)
pt2 = XYZ(10, 0, 0)
#use safe transaction with Revit
TransactionManager.Instance.EnsureInTransaction(doc)
#create line
line = Line.CreateBound(pt1, pt2)
#create wall using Revit API
wall = Wall.Create(doc, line, baseLevel.Id, False)
#Set the wall type to Dynamo input
wall.WallType = wallType
#Get the top constarint parameter using built in parameter
#Revit Document shows it as WALL_HEIGHT_TYPE
topConstraint = wall.get_Parameter(BuiltInParameter.WALL_HEIGHT_TYPE)
#Set the top constraint
topConstraint.Set(topLevel.Id)
#Finish Transaction with task done
TransactionManager.Instance.TransactionTaskDone()

OUT = wall
```



SYNTAX

CREATE WALL

To create a wall the arguments are the current document, a curve, a level ID and structural bool.

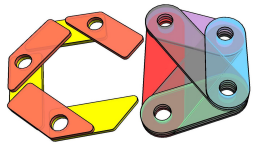
Revit API Docs website is a great place to find the documentation on the Revit API.

revitapidocs.com

Revit API Docs
Online Documentation for the Revit API

```
public static Wall Create(
        Document document,
        Curve curve,
        ElementId levelId,
        bool structural
)
```

```
import clr
#Import module for Revit
clr.AddReference("RevitNodes")
import Revit
clr.ImportExtensions(Revit.Elements)
#import module for the Document and transactions
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
#import Revit API
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
#get the document
doc = DocumentManager.Instance.CurrentDBDocument
#Dynamo input
baseLevel = UnwrapElement(IN[0])
topLevel = UnwrapElement(IN[1])
wallType = UnwrapElement(IN[2])
#create point for line
pt1 = XYZ(0, 0, 0)
pt2 = XYZ(10, 0, 0)
#use safe transaction with Revit
TransactionManager.Instance.EnsureInTransaction(doc)
#create line
line = Line.CreateBound(pt1, pt2)
#create wall using Revit API
wall = Wall.Create(doc, line, baseLevel.Id, False)
#Set the wall type to Dynamo input
wall.WallType = wallType
#Get the top constarint parameter using built in parameter
#Revit Document shows it as WALL_HEIGHT_TYPE
topConstraint = wall.get_Parameter(BuiltInParameter.WALL_HEIGHT_TYPE)
#Set the top constraint
topConstraint.Set(topLevel.Id)
#Finish Transaction with task done
TransactionManager.Instance.TransactionTaskDone()

OUT = wall
```



SYNTAX

SET PARAMETER

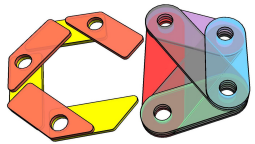
The get_Parameter method which needs the built in parameter name. We can find the name in the Revit API Doc website.

Revit API Docs

"Top Constraint"

WALL_HEIGHT_TYPE

```python
import clr
#Import module for Revit
clr.AddReference("RevitNodes")
import Revit
clr.ImportExtensions(Revit.Elements)
#import module for the Document and transactions
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
#import Revit API
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
#get the document
doc = DocumentManager.Instance.CurrentDBDocument
#Dynamo input
baseLevel = UnwrapElement(IN[0])
topLevel = UnwrapElement(IN[1])
wallType = UnwrapElement(IN[2])
#create point for line
pt1 = XYZ(0, 0, 0)
pt2 = XYZ(10, 0, 0)
#use safe transaction with Revit
TransactionManager.Instance.EnsureInTransaction(doc)
#create line
line = Line.CreateBound(pt1, pt2)
#create wall using Revit API
wall = Wall.Create(doc, line, baseLevel.Id, False)
#Set the wall type to Dynamo input
wall.WallType = wallType
#Get the top constarint parameter using built in parameter
#Revit Document shows it as WALL_HEIGHT_TYPE
topConstraint = wall.get_Parameter(BuiltInParameter.WALL_HEIGHT_TYPE)
#Set the top constraint
topConstraint.Set(topLevel.Id)
#Finish Transaction with task done
TransactionManager.Instance.TransactionTaskDone()

OUT = wall
```



SYNTAX

TRANSACTION DONE

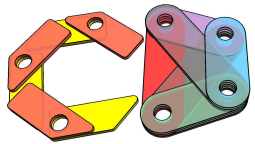
Complete our transaction using

Transaction Manager Instance

Transaction Task Done.

Then we send out our new wall.

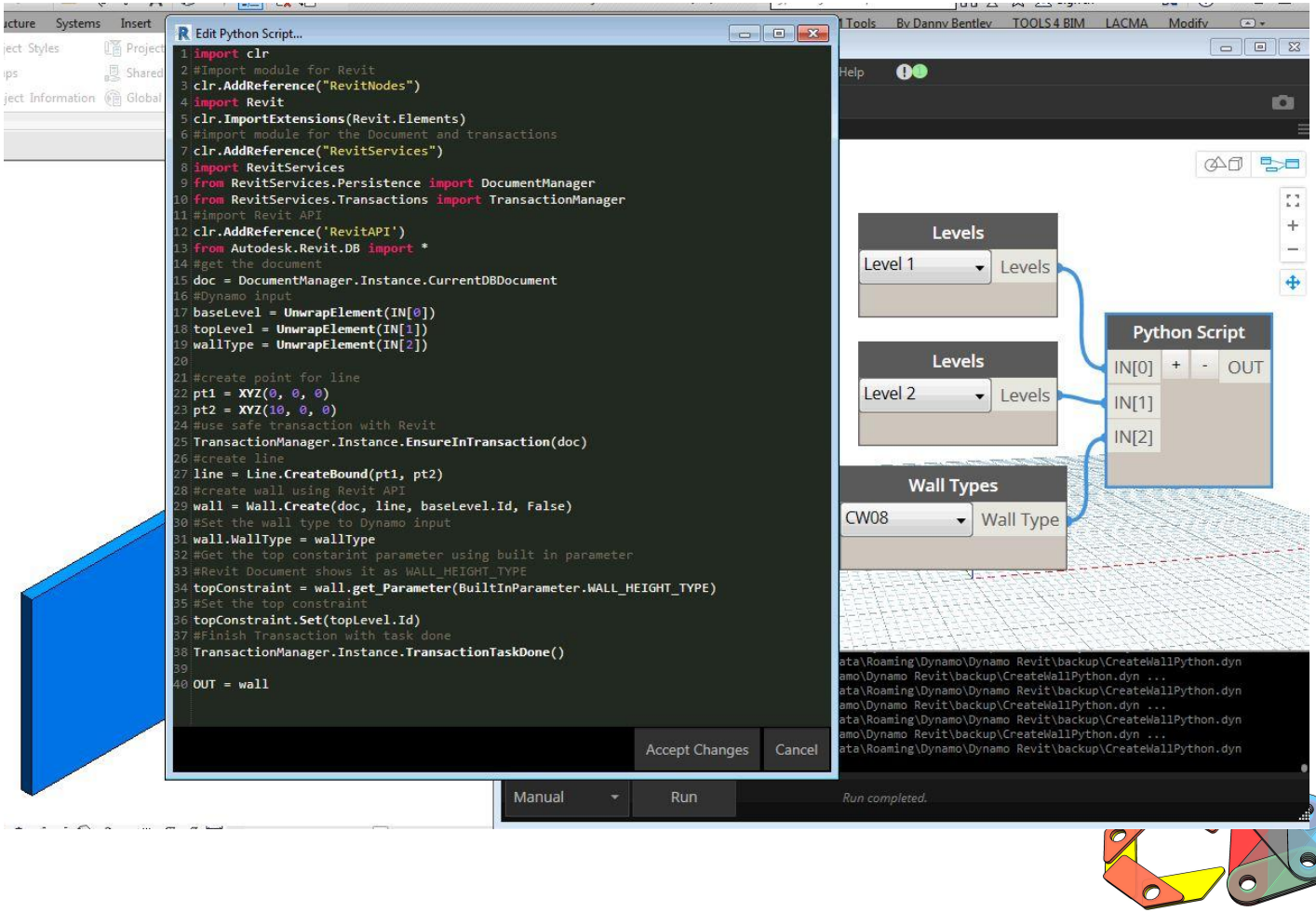
```
import clr
#Import module for Revit
clr.AddReference("RevitNodes")
import Revit
clr.ImportExtensions(Revit.Elements)
#import module for the Document and transactions
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
#import Revit API
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
#get the document
doc = DocumentManager.Instance.CurrentDBDocument
#Dynamo input
baseLevel = UnwrapElement(IN[0])
topLevel = UnwrapElement(IN[1])
wallType = UnwrapElement(IN[2])
#create point for line
pt1 = XYZ(0, 0, 0)
pt2 = XYZ(10, 0, 0)
#use safe transaction with Revit
TransactionManager.Instance.EnsureInTransaction(doc)
#create line
line = Line.CreateBound(pt1, pt2)
#create wall using Revit API
wall = Wall.Create(doc, line, baseLevel.Id, False)
#Set the wall type to Dynamo input
wall.WallType = wallType
#Get the top constarint parameter using built in parameter
#Revit Document shows it as WALL_HEIGHT_TYPE
topConstraint = wall.get_Parameter(BuiltInParameter.WALL_HEIGHT_TYPE)
#Set the top constraint
topConstraint.Set(topLevel.Id)
#Finish Transaction with task done
TransactionManager.Instance.TransactionTaskDone()

OUT = wall
```



SYNTAX

COMPLETE CODE

It’s not so intimidating.



PROJECT WORKFLOW WITH PYTHON & DYNAMO

BY DANNY BENTLEY

READ CSV FILE

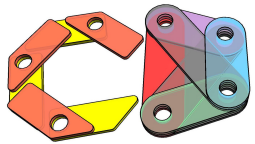
USING CSV FILE TO

CREATE AN OBJECT CLASS.

Read file using the system, input/output, stream reader
System.IO.StreamReader(filePath)

return byte from the stream. If -1 nothing left to read
Peek()

Close file
Close()



CREATE STRUCTURAL FRAMING

USING OBJECT DATA TO CREATE REVIT STRUCTURAL FRAMING ELEMENTS.

Collect all levels in project to host new framing

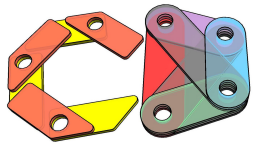
```
FilteredElementCollecto(doc)
.OfCategory(BuiltInCategory.OST_Levels)
.WhereElementIsNotElementType().ToElements()
```

Collect structural framing types to set new framing type

```
FilteredElementCollector(doc)
.OfCategory(BuiltInCategory.OST_StructuralFraming)
.OfClass(FamilySymbol).ToElements()
```

Create a new family instance in Revit.

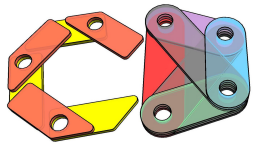
```
NewFamilyInstance(XYZ,FamilySymbol,Level,StructuralType)
```



UPDATE STRUCTURAL FRAMING

USING OBJECT DATA TO UPDATE REVIT STRUCTURAL FRAMING ELEMENTS.

```
Dictionary of key CSV Id and value Revit Element
dict = {}
dict.Add(key, value)

Set new type to element
Element.Symbol = type
```



GRAPHIC OVERRIDE

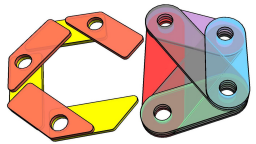
SET GRAPHIC OVERRIDE IN A VIEW.

Setting to override display of elements in a view
OverrideGraphicSettings()

Set the color of the surface foreground pattern
SetProjectionFillColor(Color)

Sets the ElementId of the surface foreground pattern
SetProjectionFillPatternId(ElementId)

Set new type to element
Element.Symbol = type

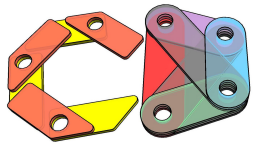


ROTATE ELEMENTS

ROTATE A GROUP OF ELEMENTS IN A PROJECT

Rotates a set of elements about a given axis and angle

```
ElementTransformUtils
.RotateElements(Document, ICollection<ElementId>,
                Line axis, double angle)
```

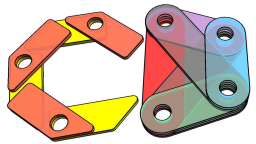


MOVE ELEMENTS

MOVE A GROUP OF ELEMENTS IN A PROJECT

Rotates a set of elements about a given axis and angle

```
ElementTransformUtils
.MoveElements(Document, ICollection<ElementId>,
                XYZ)
```



CREATE SHEETS

CREATE NEW SHEETS AND SET THE NAME AND NUMBER PARAMETERS

ViewSheet class and Create method

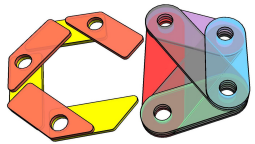
```
ViewSheet
.Create(Document, ElementId titleblock)
```

Set sheet number

```
newSheet.SheetNumber = sheetNumber
```

Set sheet name

```
newSheet.Name = sheetName
```



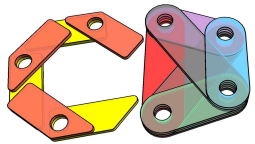
APPLY VIEW TEMPLATE

APPLYING VIEW TEMPLATES TO MULTIPLE VIEWS.

Collect all the views from project
FilteredElementCollector(Document).OfClass(View)

Test view to ensure it’s a view template
view.IsTemplate

Set view template onto a view
view.ViewTemplateId = viewTemp.Id



COPY LINKED MODEL ELEMENTS

COPY ELEMENTS FROM A LINKED MODEL TO THE CURRENT MODEL

Collect all linked models in project
FilteredElementCollector(Document)
.OfClass(RevitLinkInstance)

Return the linked model document
linkDoc.GetLinkedDocument

Get’s the total transform, which includes the true north transform for instances like import instances
linkDoc.GetTotalTransform()

A collection of utilities allowing transformation of elements. Copies a set of elements from source document to destination document
ElementTranformUtils
.CopyElements(Document linkedDoc, ICollection<ElementId>,
Document currentDoc, Transform,
CopyPasteOptions)