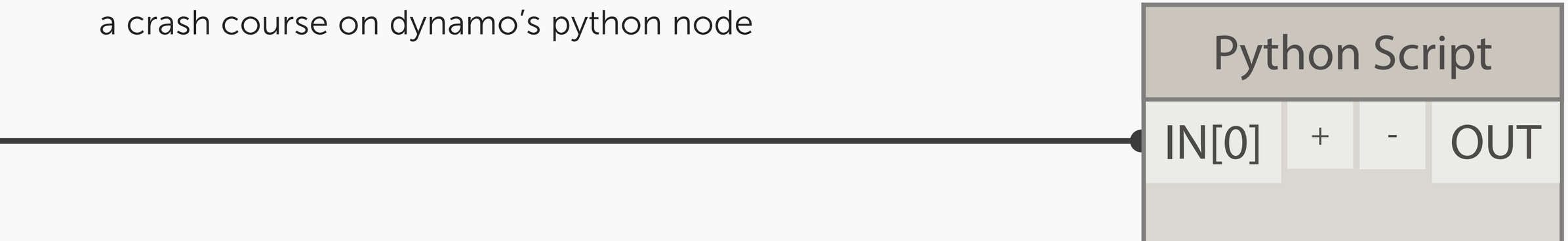
# Untangling Python
# A Crash Course on Dynamo's Python Node

Gui Talarico
Product Development at Wework

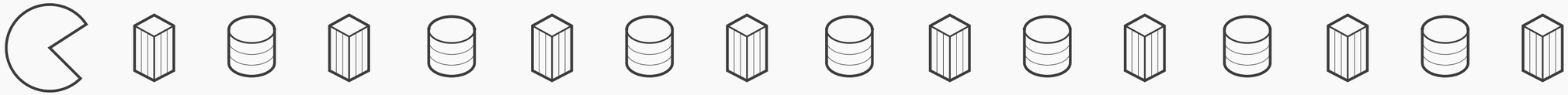
# untangling python

a crash course on dynamo's python node




#AU2017 AS124816-L

me

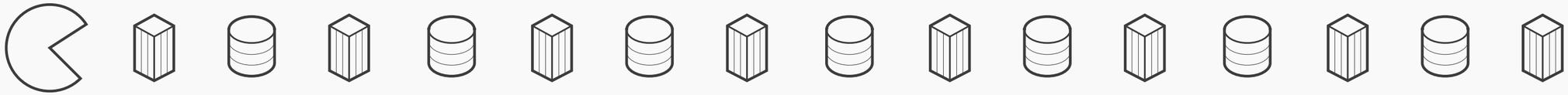


Gui Talarico

Product Development at

wework

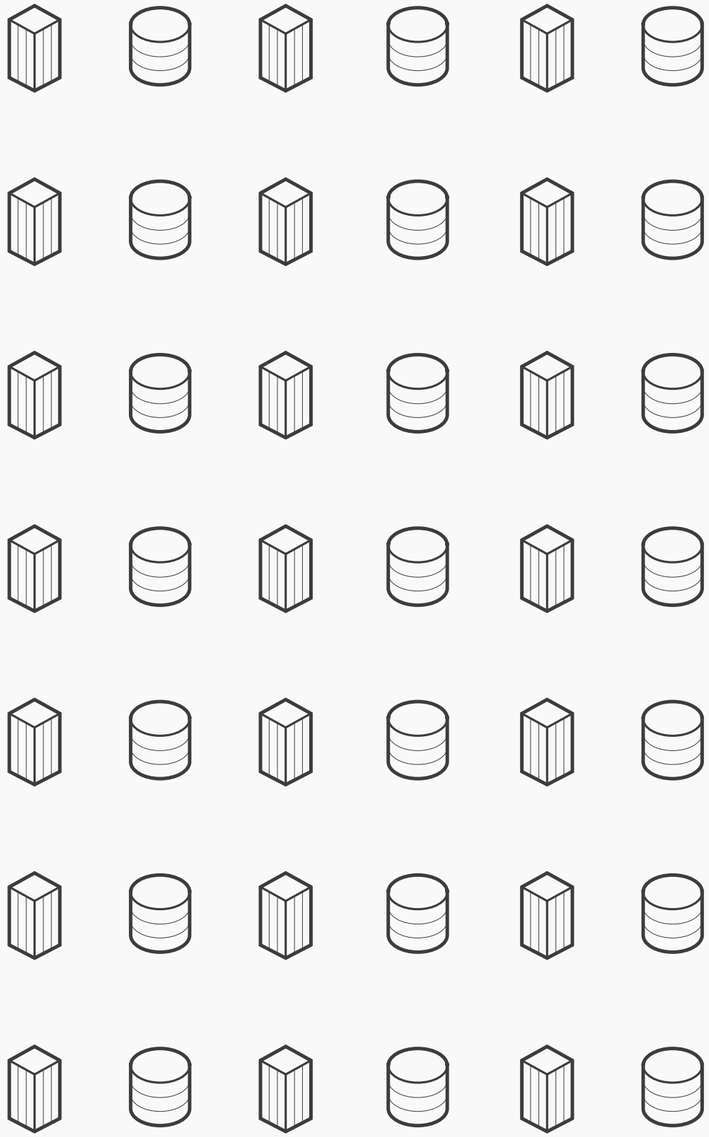
me



Gui Talarico

Product Development at

wework

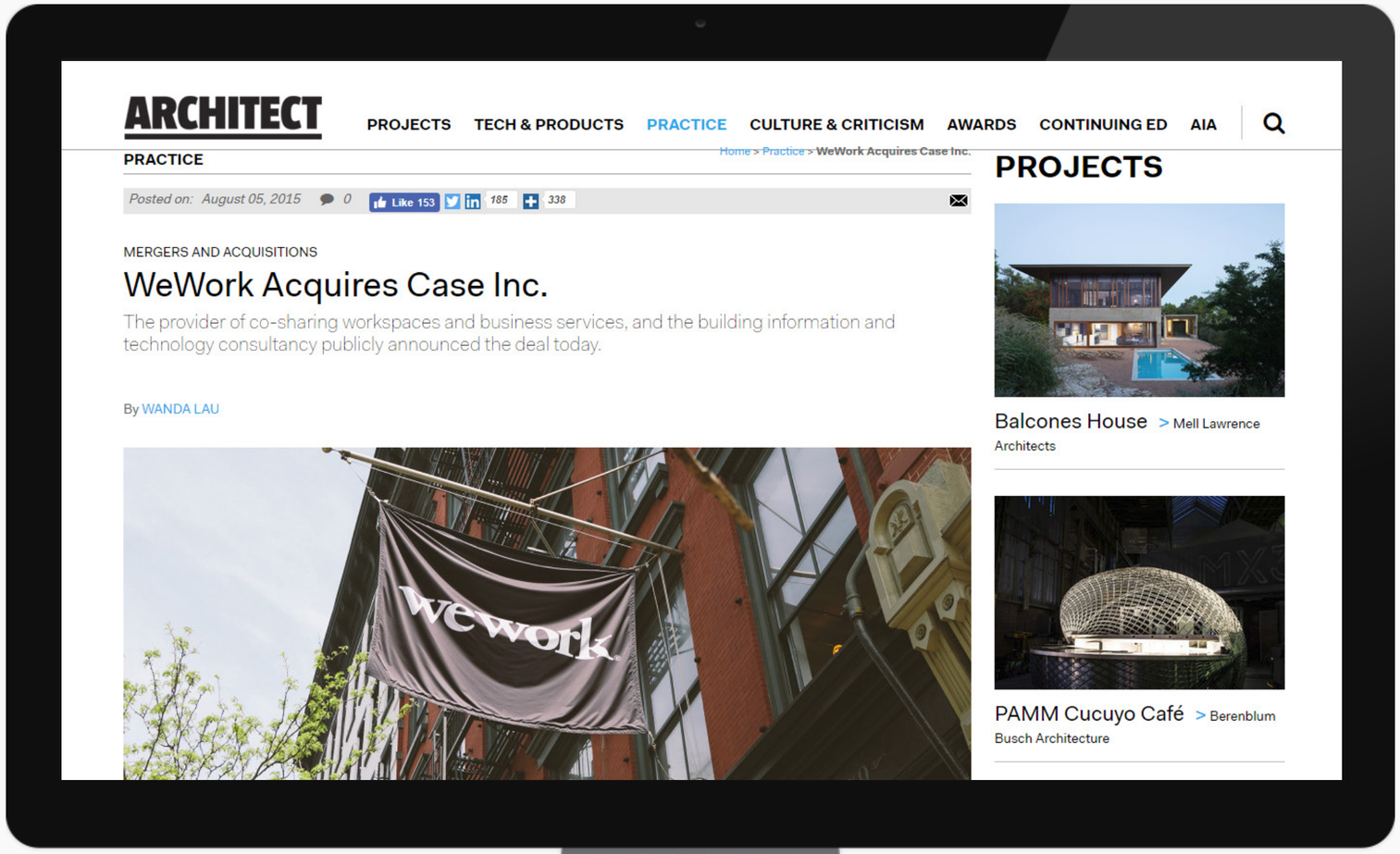


**+170**
locations

**+ 150k**
members

**+10.000.000**
square feet

**A LOT**
of data



ARCHITECT

PROJECTS TECH & PRODUCTS PRACTICE CULTURE & CRITICISM AWARDS CONTINUING ED AIA

PRACTICE

Home > Practice > WeWork Acquires Case Inc.

Posted on: August 05, 2015 0 Like 153 185 338

MERGERS AND ACQUISITIONS

## WeWork Acquires Case Inc.

The provider of co-sharing workspaces and business services, and the building information and technology consultancy publicly announced the deal today.

By WANDA LAU



PROJECTS

Balcones House > Mell Lawrence Architects

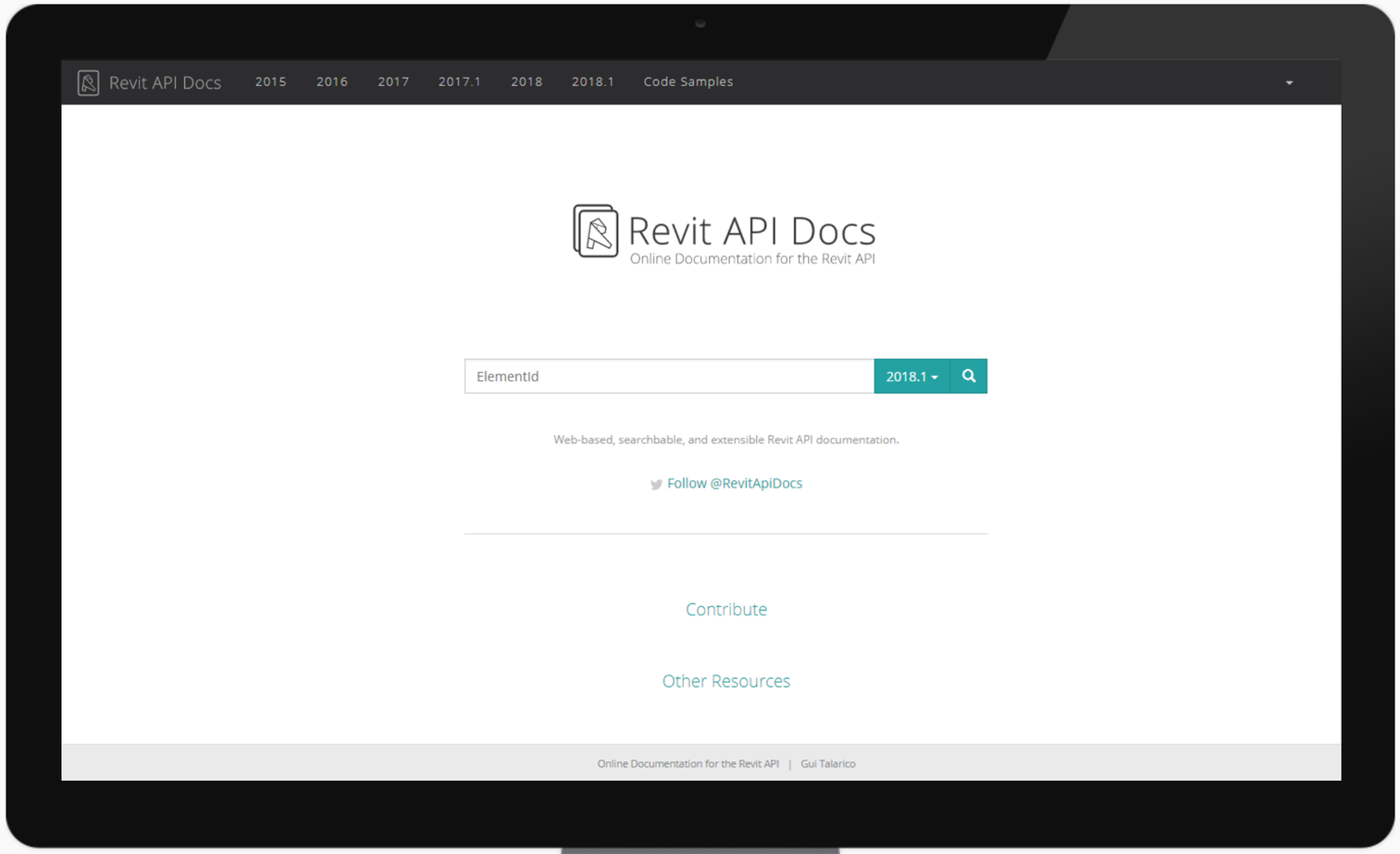
PAMM Cucuyo Café > Berenblum Busch Architecture

wework

Revit API Docs
Online Documentation for the Revit API

rpw
Revit Python Wrapper

**IronPythonStubs**
Autocompletion




# me

after dark

@gtalarico

# you want to learn (+) python

why ?

# nodes are not enough

# make your own tools

python is awesome

1 Understand the difference between python and ironpython

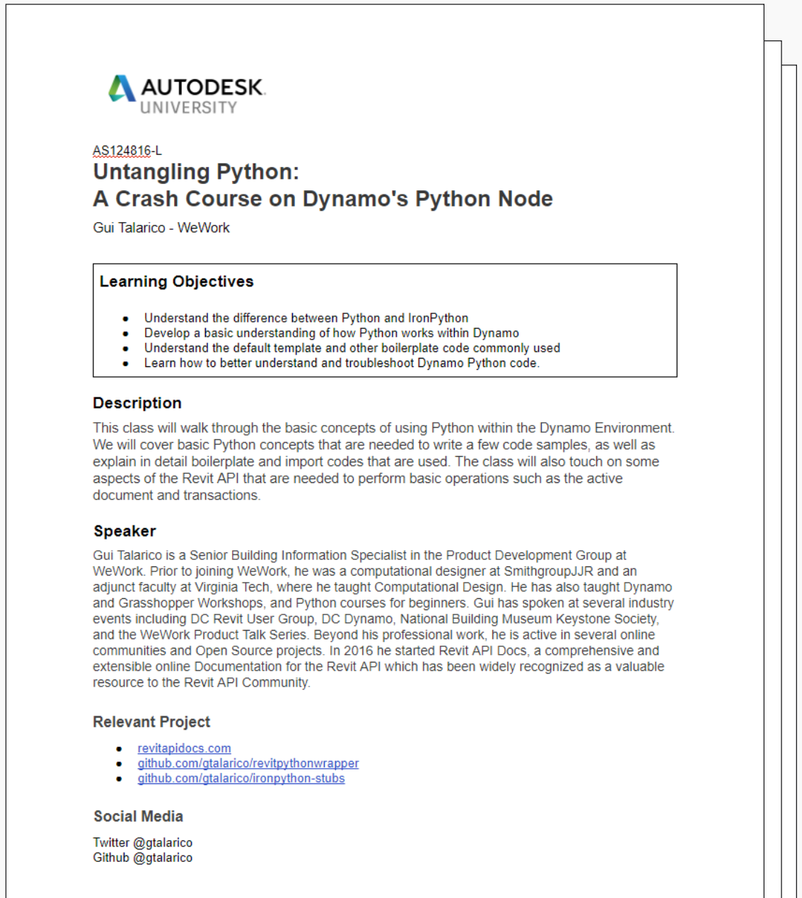
2 Develop a basic understanding of how python works within dynamo

3 Understand the default template and other boilerplate code commonly used

4 Learn how to better understand and troubleshoot dynamo python code

# learning objectives

obligatory slide



AUTODESK UNIVERSITY

AS124816-L

**Untangling Python:**
**A Crash Course on Dynamo's Python Node**

Gui Talarico - WeWork

**Learning Objectives**

- Understand the difference between Python and IronPython
- Develop a basic understanding of how Python works within Dynamo
- Understand the default template and other boilerplate code commonly used
- Learn how to better understand and troubleshoot Dynamo Python code.

**Description**

This class will walk through the basic concepts of using Python within the Dynamo Environment. We will cover basic Python concepts that are needed to write a few code samples, as well as explain in detail boilerplate and import codes that are used. The class will also touch on some aspects of the Revit API that are needed to perform basic operations such as the active document and transactions.

**Speaker**

Gui Talarico is a Senior Building Information Specialist in the Product Development Group at WeWork. Prior to joining WeWork, he was a computational designer at SmithgroupJJR and an adjunct faculty at Virginia Tech, where he taught Computational Design. He has also taught Dynamo and Grasshopper Workshops, and Python courses for beginners. Gui has spoken at several industry events including DC Revit User Group, DC Dynamo, National Building Museum Keystone Society, and the WeWork Product Talk Series. Beyond his professional work, he is active in several online communities and Open Source projects. In 2016 he started Revit API Docs, a comprehensive and extensible online Documentation for the Revit API which has been widely recognized as a valuable resource to the Revit API Community.

**Relevant Project**

- revitapidocs.com
- github.com/gtalarico/revitpythonwrapper
- github.com/gtalarico/ironpython-stubs

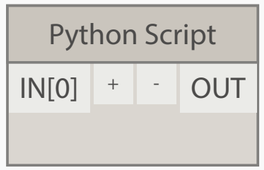
**Social Media**

Twitter @gtalarico
Github @gtalarico

in other words...

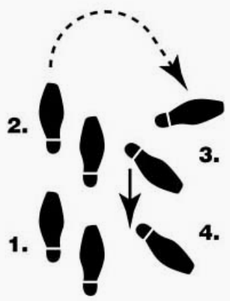
# intro to python

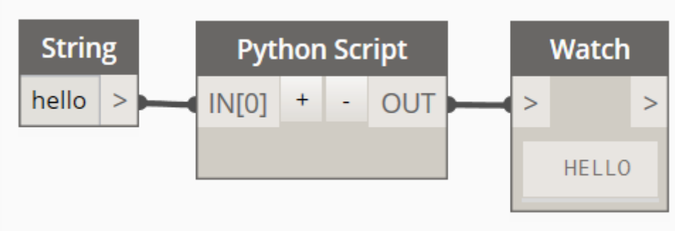
act 1




# intro to python node

act 2




# work through 3 examples

act 3

# intro to python

act 1

open-source, interpreted, general-purpose, object-oriented, high-level, dynamically-typed, programming language

# what is python?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | open-source, | interpreted, | general-purpose, | object-oriented, | high-level, | dynamically-typed, | programming language |
| **not** | proprietary | compiled | domain-specific | functional | low-level | statically-typed | |

# what is python?

# what is ironpython?

and why does that matter

**short answer**

An implementation of the python language specification created by microsoft, written in C#.

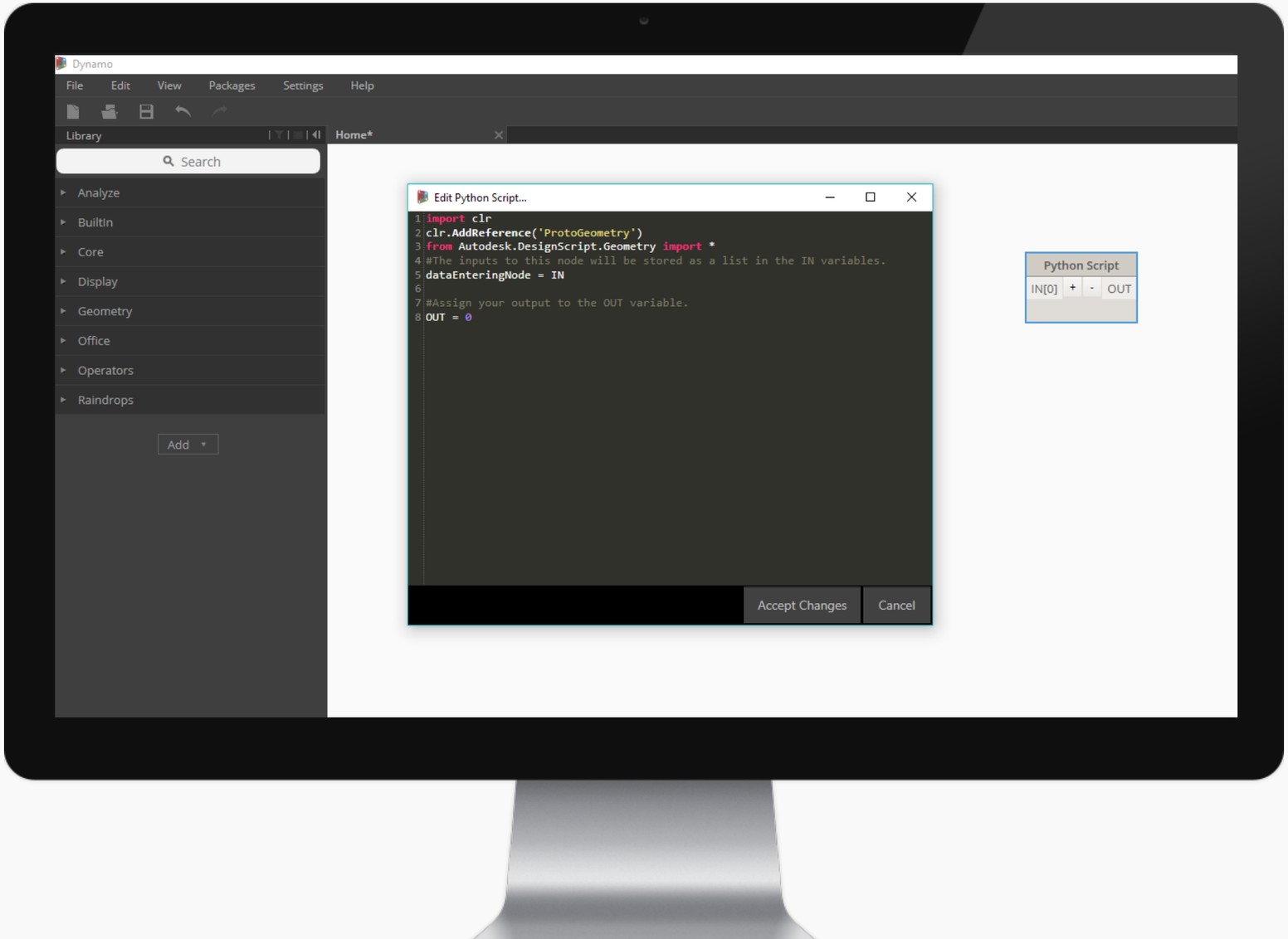
The C# implementation allows it to use the Common Language Runtime (clr) to talk directly to other .NET applications and libraries.

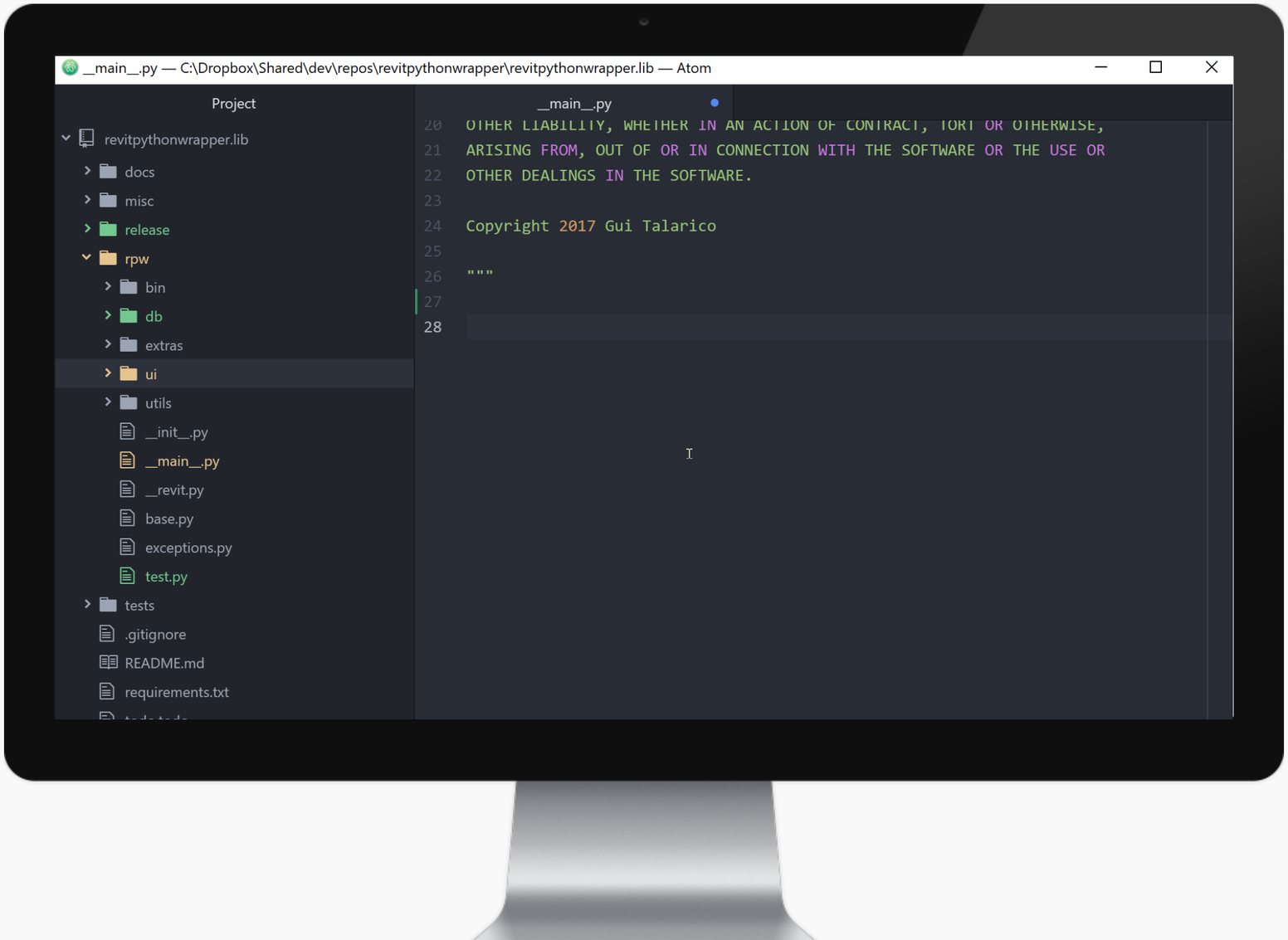
This language interoperability has made Ironpython a popular embedded-scripting-language .

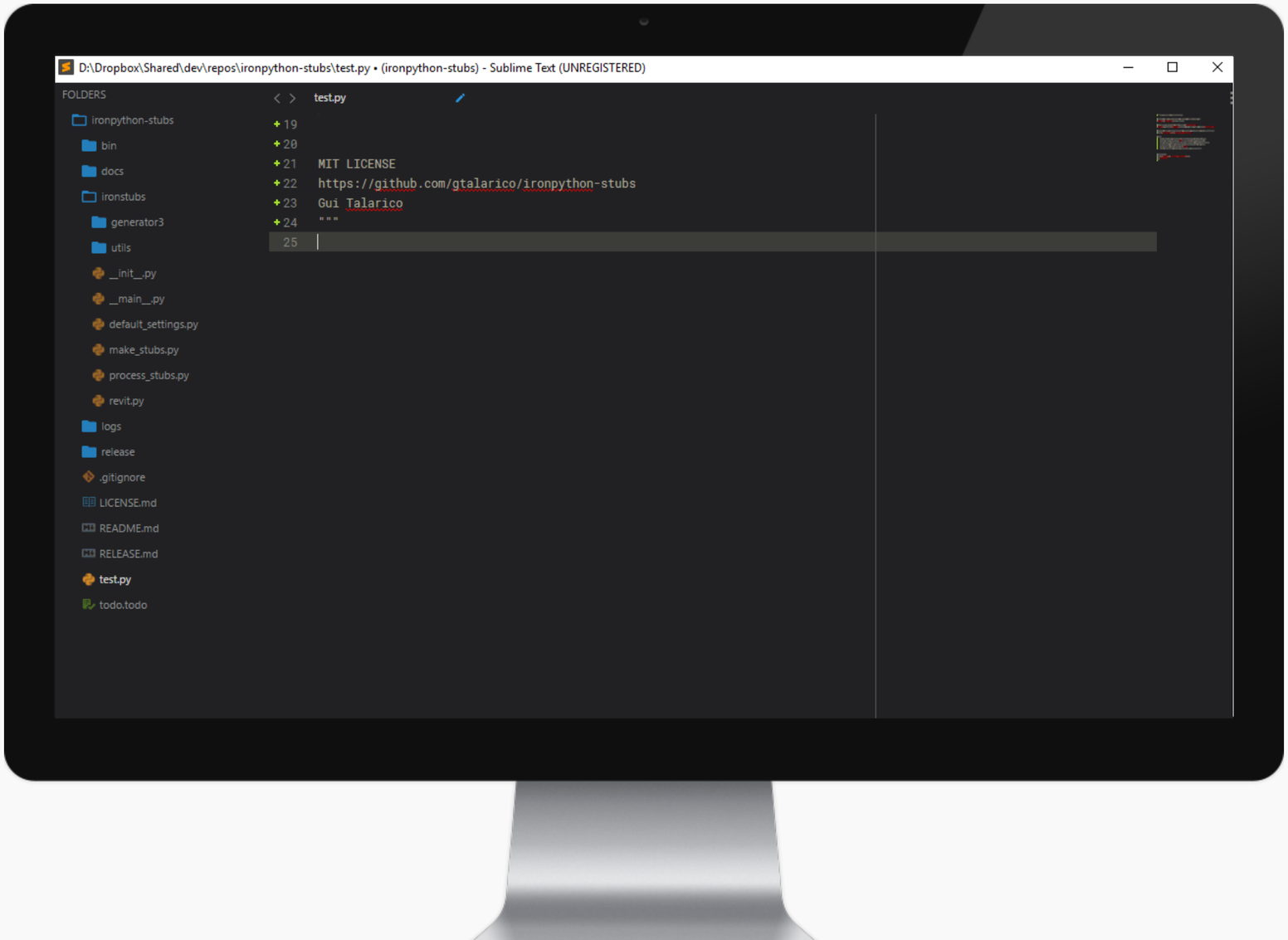
# what is ironpython?

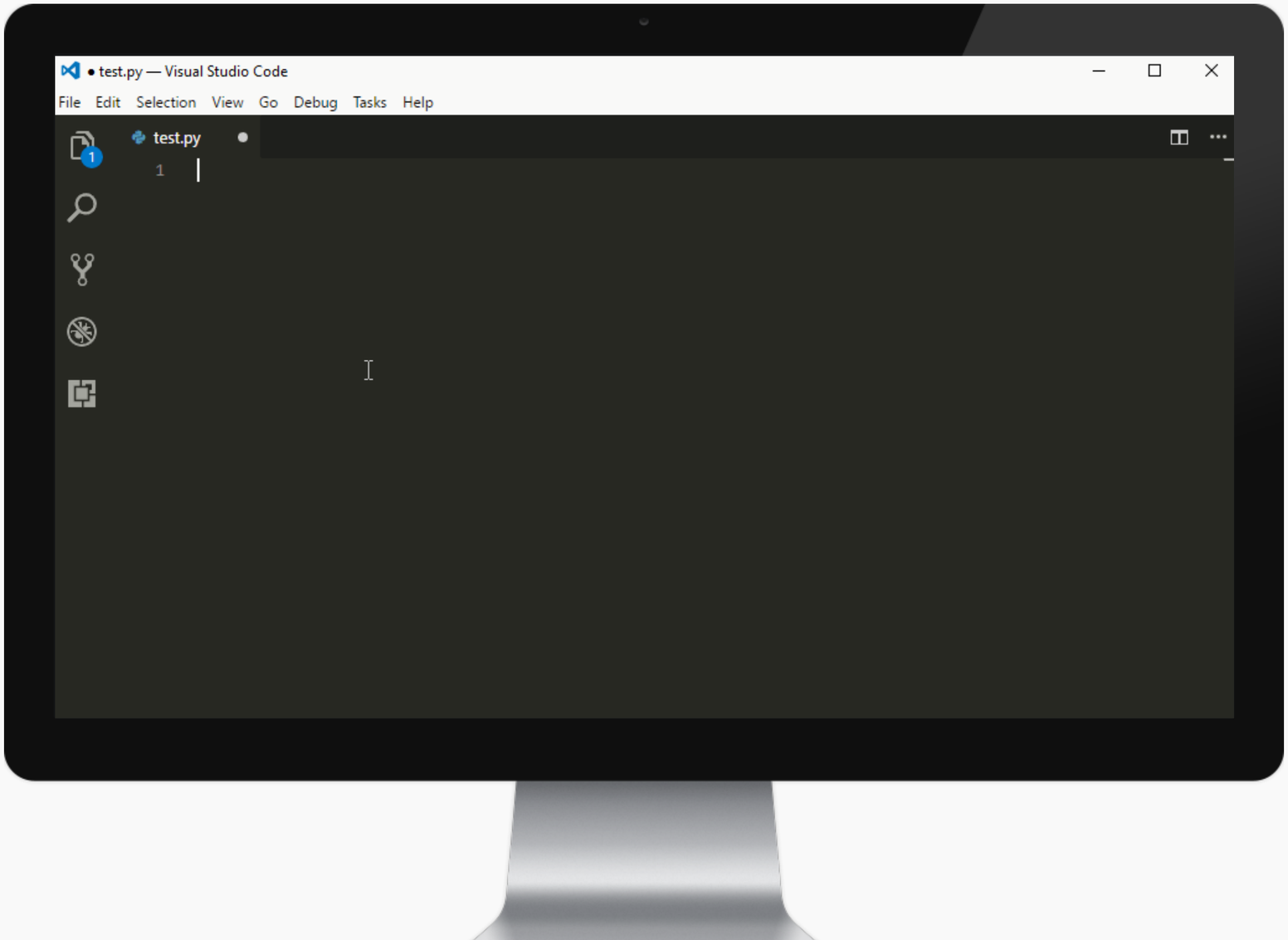
and why does that matter

# writing ironpython code


Dynamo
File
Edit
View
Packages
Settings
Help
Library
Home*
Search
Analyze
BuiltIn
Core
Display
Geometry
Office
Operators
Raindrops
Add
Edit Python Script...
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN
#Assign your output to the OUT variable.
OUT = 0
Accept Changes
Cancel
Python Script
IN[0]
+
-
OUT

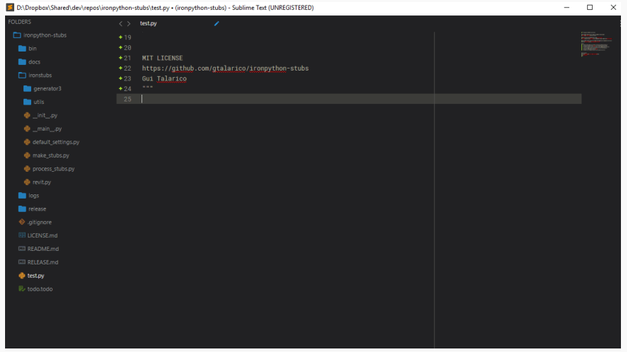

__main__.py — C:\Dropbox\Shared\dev\repos\revitpythonwrapper\revitpythonwrapper.lib — Atom
Project
revitpythonwrapper.lib
docs
misc
release
rpw
bin
db
extras
ui
utils
__init__.py
__main__.py
__revit.py
base.py
exceptions.py
test.py
tests
.gitignore
README.md
requirements.txt
__main__.py
20 OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
21 ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
22 OTHER DEALINGS IN THE SOFTWARE.
23
24 Copyright 2017 Gui Talarico
25
26 """
27
28


D:\Dropbox\Shared\dev\repos\ironpython-stubs\test.py • (ironpython-stubs) - Sublime Text (UNREGISTERED)
FOLDERS
ironpython-stubs
bin
docs
ironstubs
generator3
utils
__init__.py
__main__.py
default_settings.py
make_stubs.py
process_stubs.py
revit.py
logs
release
.gitignore
LICENSE.md
README.md
RELEASE.md
test.py
todo.todo
test.py
19
20
21 MIT LICENSE
22 https://github.com/gtalarico/ironpython-stubs
23 Gui Talarico
24 """
25


• test.py — Visual Studio Code
File Edit Selection View Go Debug Tasks Help
test.py
1

| | dynamo | atom | sublime | vscode |
|---|---|---|---|---|
| **type** | builtin editor | external, general purpose editor | | |
| **language** | ironpython | any language | | |
| **syntax-highlighting** | yes - fixed | yes - many options to chose from | | |
| **autocompletion** | kind of | yes - IronpythonStubs | | |
| **plugins** | - | hell yes | | |

source code




>>>

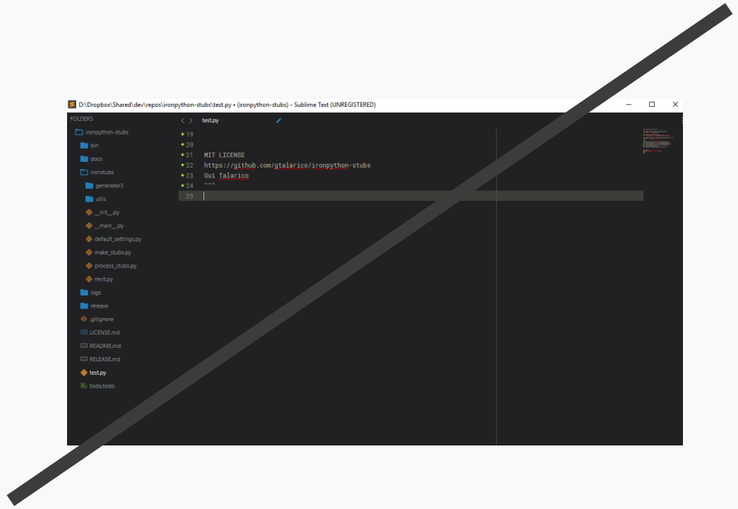
interpreter

>>>

output

# executing code

source code >>> interpreter >>> output




interpreter

>>>

output

# executing code

interpreter >>> output

```
>>>
```

interactive interpreter

```
C:\Program Files (x86)\IronPython 2.7 > ipy.exe
IronPython 2.7.3 (2.7.0.40) on .NET 4.0.30319.42000 (32-bit)
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello AU')
Hello AU
```

# try it out

ipy.exe or ironpython shell

```
>>> print('Hello World')
Hello World
>>> print(2+4)
6
```

# printing

now that you know, you can forget about it.

*thanks dynamo*

```
>>> # Anything after a '#' will be ignored
```

# commenting

use comments to write helpful notes about your code
to your self, and others

# basic data types

`type()`

```
>>> 5 + 5
10
>>> # note: float + int = float
>>> 5.0 + 5
10.0
>>> x = -3
>>> x * 2
-6
```

# integer + floats

aka. numbers

```
# Single Quotes (') or double (") are accepted
>>> '123 Street'
>>> "Another String"
>>> ''' Multiline strings are represented with
        triple quotes '''
>>> """ Multiline can use single or
        double quote """
```

# strings

aka. text

```
>>> True
>>> False
>>> 2 == 2
True
>>> 2 >= 5
False
>>> 'y' in 'python'
True
```

# booleans

true or false

```
>>> None
```

# none

don’t worry about it

```
>>> numbers = [1,2,3]
>>> objects = ['Desk', 'Chair', 'Lamp']
>>> objects[0]  # Retrieves item of index 0 (first)
'Desk'
>>> objects[2]  # Retrieves item of index 2 (third)
'Lamp'
>>> 'Chair' in objects
True
>>> points = [[0,0,0], [2,3,4], [5,5,5]]
>>> points[1][0]
2
```

# list

containers for stuff: numbers, strings, etc, and yes, other lists
lists are defined using square brackets [ ]

```
>>> person = {'Name': 'Mark', 'Age': 20, 'Address': '123 Street'}

>>> coordinates = {'absolute': [0,0,0], 'relative': [20,20,0]}

>>> # Similar to list to retrieve Value but uses Key instead of index
>>> elevations = {'Level 1': 10.0,  'Level 2': 25.0}
>>> elevations['Level 1']
10.0
>>> elevations['Level 1'] < elevations['Level 2']
True
```

# dictionary

key-value pairs, like... well, a dictionary
dictionaries are defined using curly brackets { }

```
>>> 2 * 2                # Evaluates to 4
>>> 'Hello ' + 'You'     # Evaluates to 'Hello You'
>>> [1, 2, 3]            # Evaluates to [1, 2, 3]
>>> 4 > 2                # Evaluates to True
>>> 2 == 3               # Evaluates to False (== checks for equality)
```

# expressions

combination of values and operators that can be evaluated down to a single value

# statements

everything else. often associated with an action

```
>>> x = 5
>>> if x > 3:
...     print('x is larger than 3')
... elif x == 3:
...     print('x is 3')
... else:
...     print('x is less than 3')
```

# statements

conditional statements

ps: white space matters

```
>>> for letter in 'Hello':
...     print(letter)
'H'
'e'
'l'
'l'
'o'
```

```
>>> for number in [1,2,3]:
...     print(number)
1
2
3
```

# statements

for loops

```
>>> def print_text():          # Function Takes no arguments
...     print('My Text')        # Performs a function, but doesn't return any value
>>> print_text()
'My Text'
```

# functions

named blocks of code that performs an action.
can receive arguments
can return values

```
>>> def add_numbers(x, y):      # Function receives 2 argument
...     return x + y            # Returns the sum of the 2 arguments received
>>> total = add_numbers(2, 5)   # total variable will hold the value 7 returned
>>> print(total)
7
```

# functions

named blocks of code that performs an action.
can receive arguments
can return values

```
>>> numbers = [1, 2, 3]
>>> numbers.append(4)   # appends an item to a list
[1, 2, 3, 4]

>>> 'LEVEL 01'.lower()  # converts a string to lower-case
'level 01'
```

# methods

functions that are defined and stored within an object or data type

```
>>> class Human():
... planet = 'earth'
...
... def __init__(self, name):
...     self.name = name
...
... def speak(self):
...     print('Hello. My name is ' + self.name)
```

```
>>> brian = Human('Brian')
>>> brian.speak()
'Hello. My name is Brian'
>>> print(brian.planet)
'earth'
```

# classes

recipes for objects or custom data types

```
>>> import os  # imports the os module
>>> os.listdir('C:/')  # call listdir method to get files in directory
['Program Files', 'ProgramData', 'Users', 'Windows', 'Logs' ]
```

```
>>> import sys # imports the sys module
>>> sys.version
'2.7.3 (IronPython 2.7.3 (2.7.0.40) on .NET 4.0.30319.42000 (32-bit))'
>>> sys.exit() # Execution of program is terminated
```

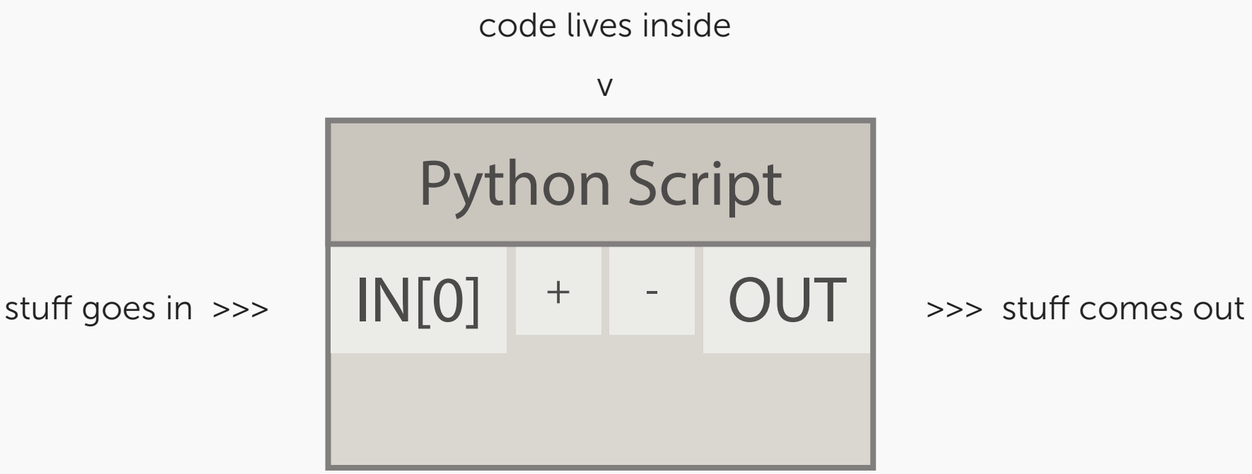
# imports

load additional functionality into your code

* importing modules within Dynamo requires additional code - see handout

# intro to python node

act 2

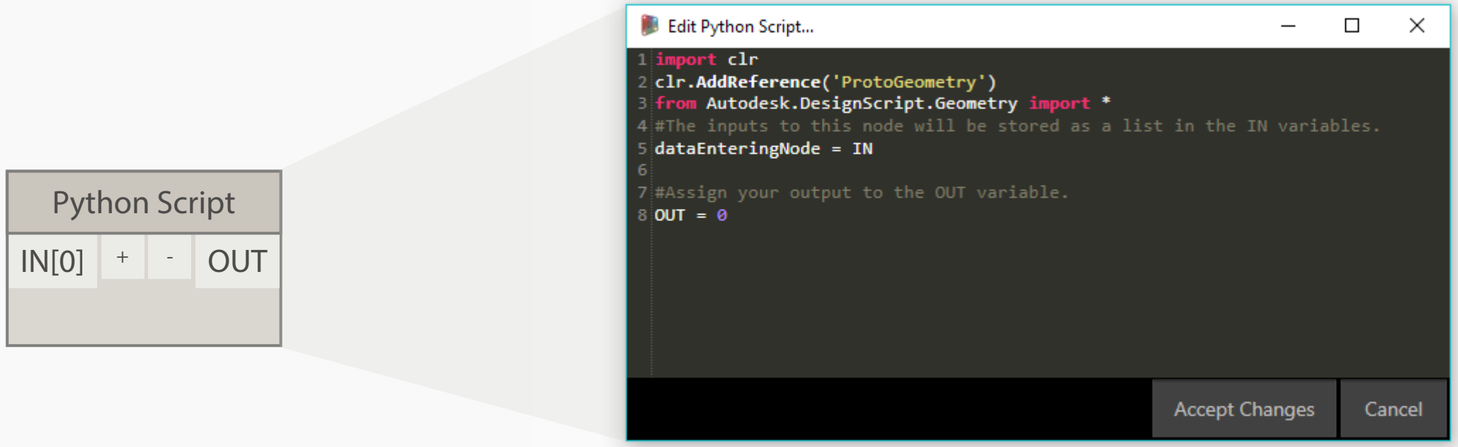



# dynamo python node

python script node

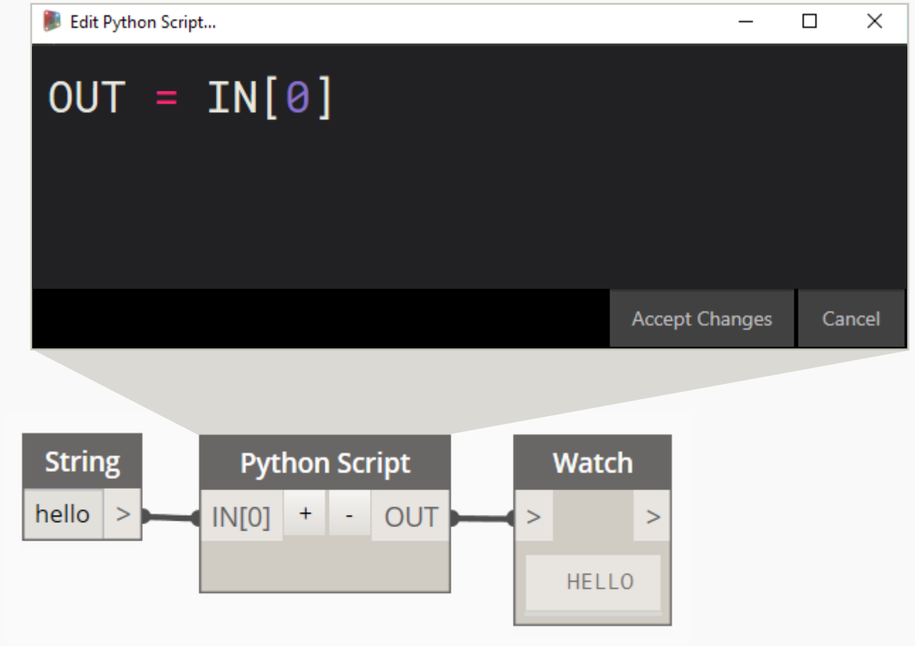
# >>>

dynamo-0-inputs-outputs.dyn



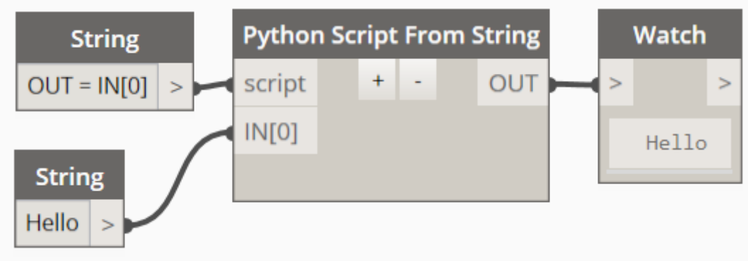

# dynamo python node

python script node



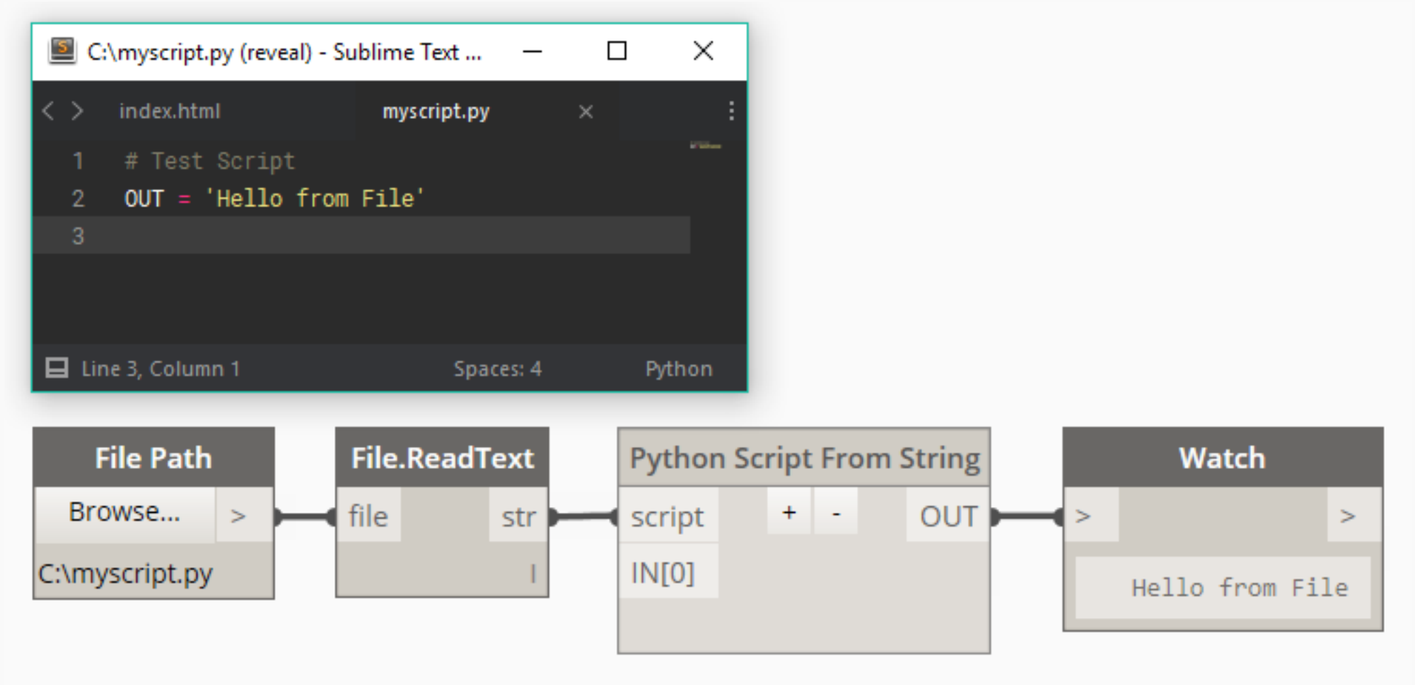

# dynamo python node

python script node




# dynamo python node

python script from string




# dynamo python node

python script from string (external file)

# dynamo import code

understanding the boiler plate code

```python
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

#Assign your output to the OUT variable.
OUT = 0
```

```python
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

#Assign your output to the OUT variable.
OUT = 0
```

imports the Common Language Runtime module.

clr has a `AddReference()` that can be used to load .NET dll references.

```python
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

#Assign your output to the OUT variable.
OUT = 0
```

**`clr.AddReference()`**
must be used to enable libraries that are not native python libraries.

using this clr method, we can add a reference to 'ProtoGeometry'

The actual library is stored here:
`"C:\Program Files\Dynamo\Dynamo Core\1.3\ProtoCore.dll"`

```python
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

#Assign your output to the
OUT = 0
```

Other commonly used references are

```python
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import Vector

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

clr.AddReference('RevitAPIUI')
from Autodesk.Revit.UI import TaskDialog

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

clr.AddReference('RevitNodes')
from Revit import Elements

clr.AddReference('DSCoreNodes')
from DSCore import Color
```

```python
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

#Assign your output to the OUT variable.
OUT = 0
```

Once 'ProtoGeometry' has been added, we load things from the DesignScript library

```python
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

#Assign your output to the OUT variable.
OUT = 0
```

Once 'ProtoGeometry' has been added, we load things from the DesignScript library

PS:

```python
from x import *
```

ಠ_ಠ

```python
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

#Assign your output to the OUT variable.
OUT = 0
```

```python
import clr
# Import RevitAPI Classes
clr.AddReference("RevitAPI")
from Autodesk.Revit.DB import Wall, FilteredElementCollector
# As explained in the previous section, replace * with the class you need separated by comma.

clr.AddReference("RevitNodes")
import Revit
# Adds ToDSType (bool) extension method to Wrapped elements
clr.ImportExtensions(Revit.Elements)
# Adds ToProtoType, ToRevitType geometry conversion extension methods to objects
clr.ImportExtensions(Revit.GeometryConversion)

# Import DocumentManager and TransactionManager
clr.AddReference("RevitServices")
from RevitServices.Transactions import TransactionManager
from RevitServices.Persistence import DocumentManager
# Create variable for Revit Document
doc = DocumentManager.Instance.CurrentDBDocument

TransactionManager.Instance.EnsureInTransaction(doc)  # Start Transaction

# !!!! Code that modifies Revit Database goes Here !!!!

TransactionManager.Instance.TransactionTaskDone()     # End Transaction

OUT = None
```

Full template

* see handout

# hands on exercises

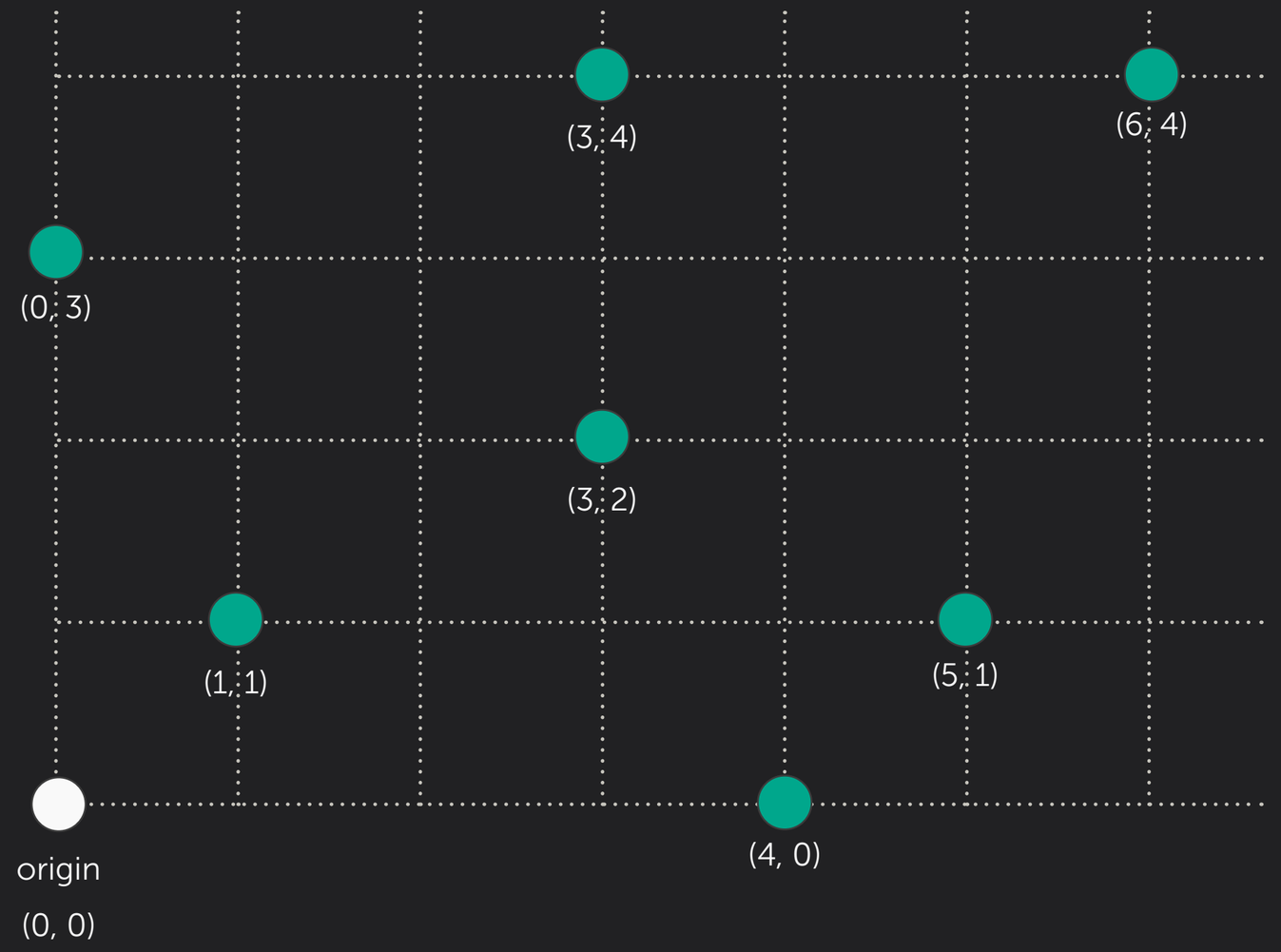
act 3

# exercise #1

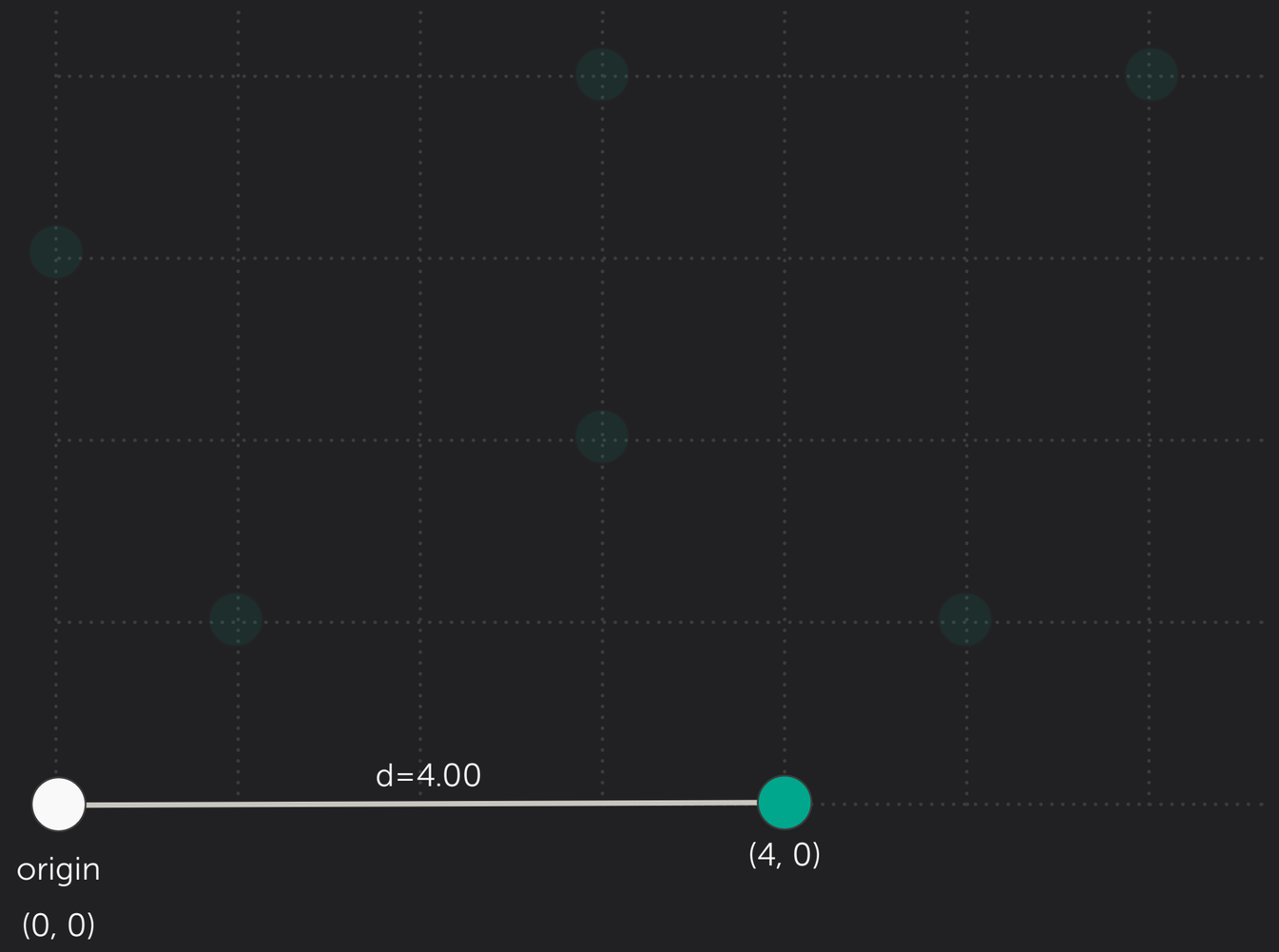
order points sample by distance to origin

```
points = [ [5, 1], [6, 4], [3, 4], [1, 1], [4, 0], [0, 3], [3, 2]]

origin = [0,0]
```

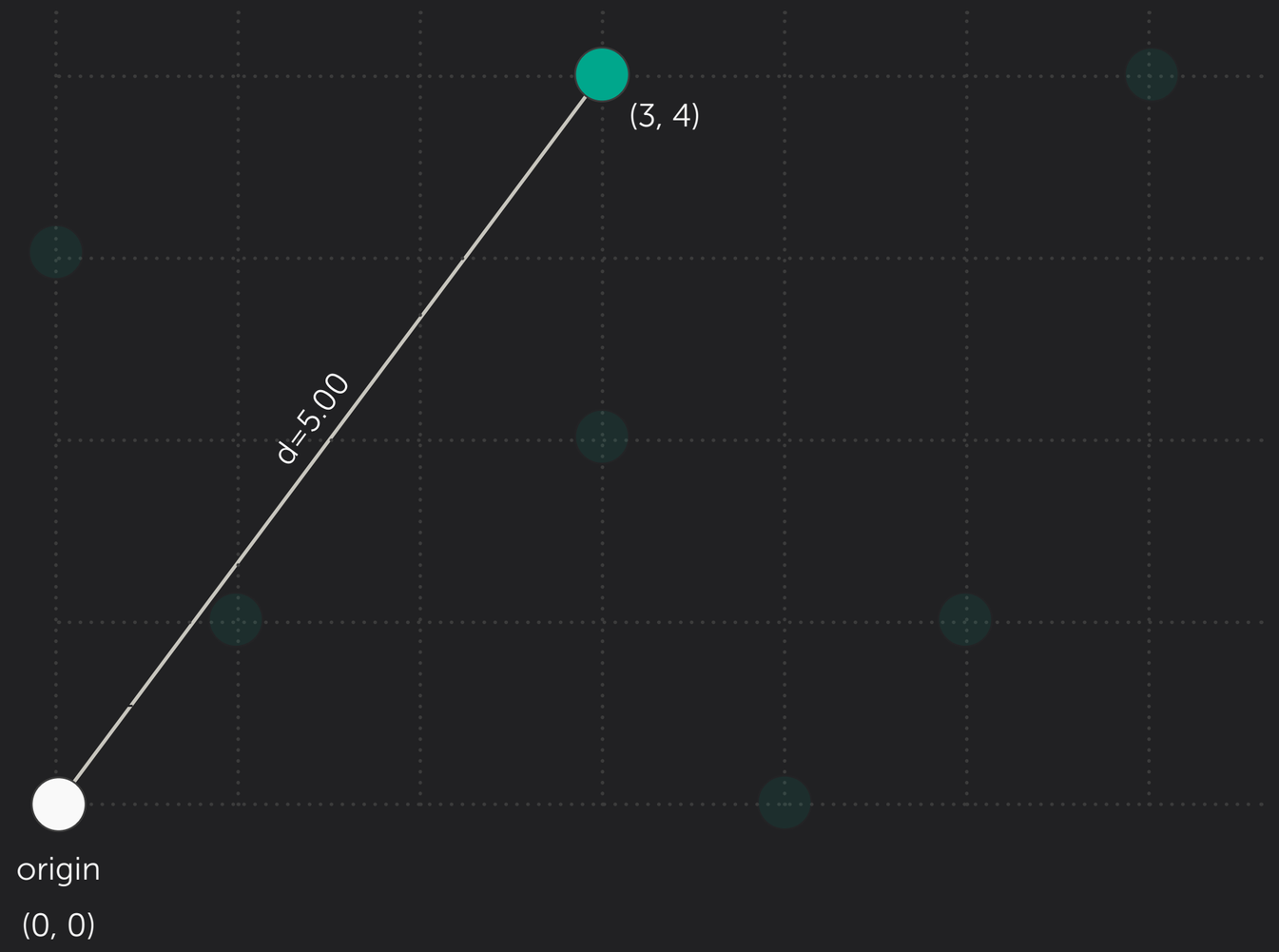


```
points = [ [5, 1], [6, 4], [3, 4], [1, 1], [4, 0], [0, 3], [3, 2]]
origin = [0,0]
```

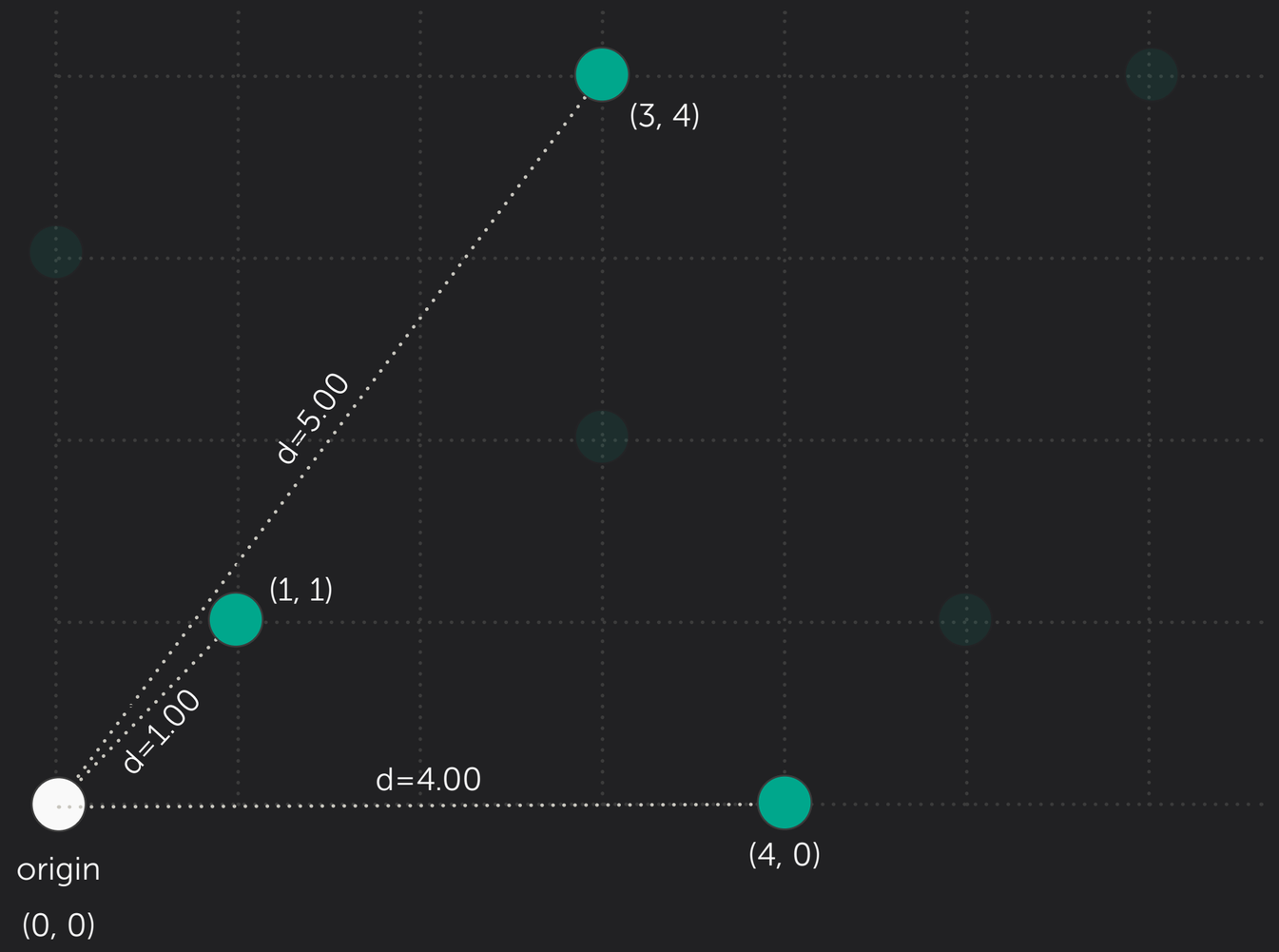


```
points = [ [5, 1], [6, 4], [3, 4], [1, 1], [4, 0], [0, 3], [3, 2]]
origin = [0,0]
```

```
points = [ [5, 1], [6, 4], [3, 4], [1, 1], [4, 0], [0, 3], [3, 2]]
origin = [0,0]
```

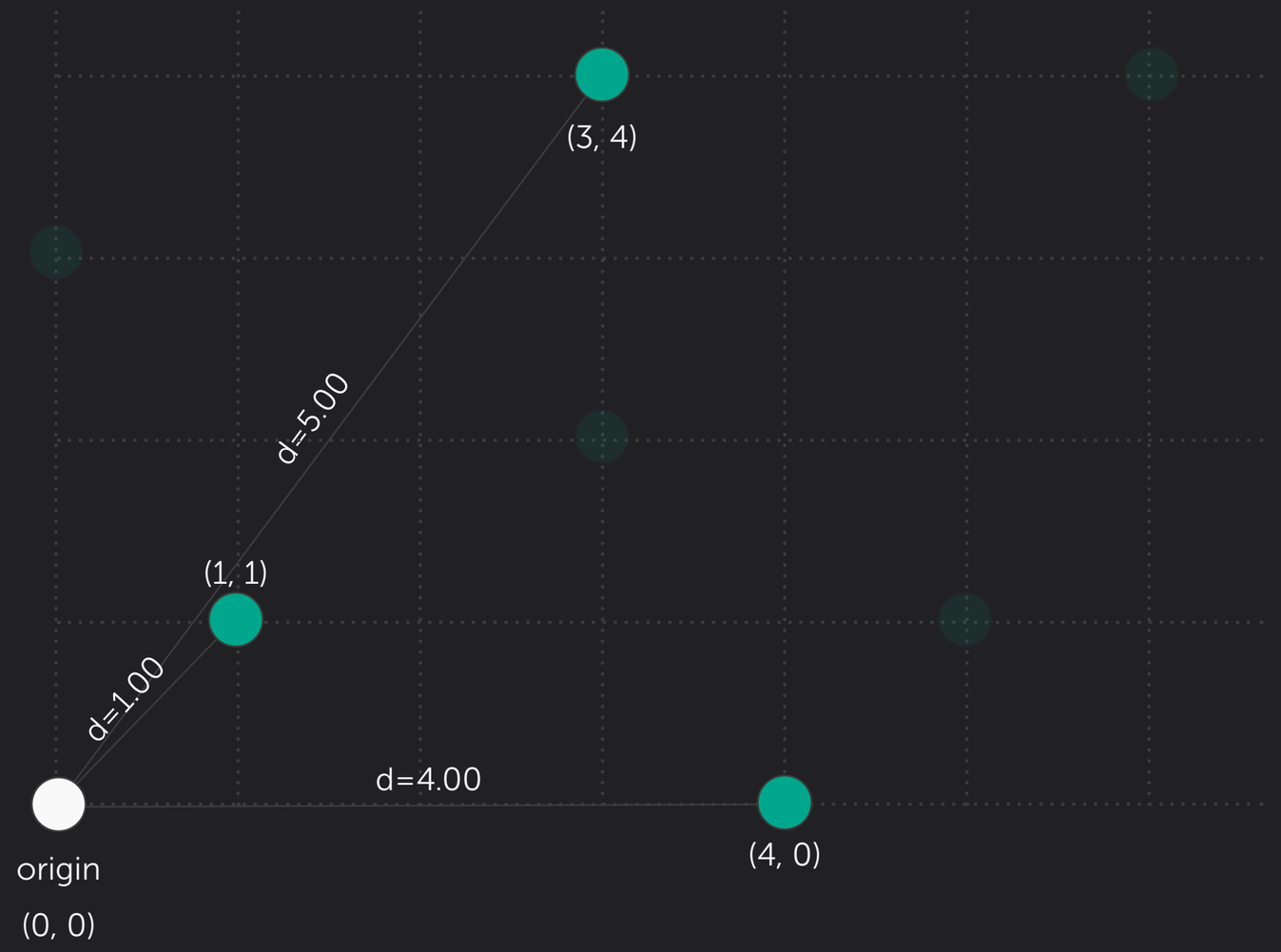


```
d=5.00  d=1.00  d=4.00
points = [ [5, 1], [6, 4], [3, 4], [1, 1], [4, 0], [0, 3], [3, 2]]
```



```
d=5.00  d=1.00  d=4.00
points = [ [5, 1], [6, 4], [3, 4], [1, 1], [4, 0], [0, 3], [3, 2]]
```

```
append dist:   [5.00, [3,4]]  [1.00, [1,1]] [4.00, [4,0]]
```

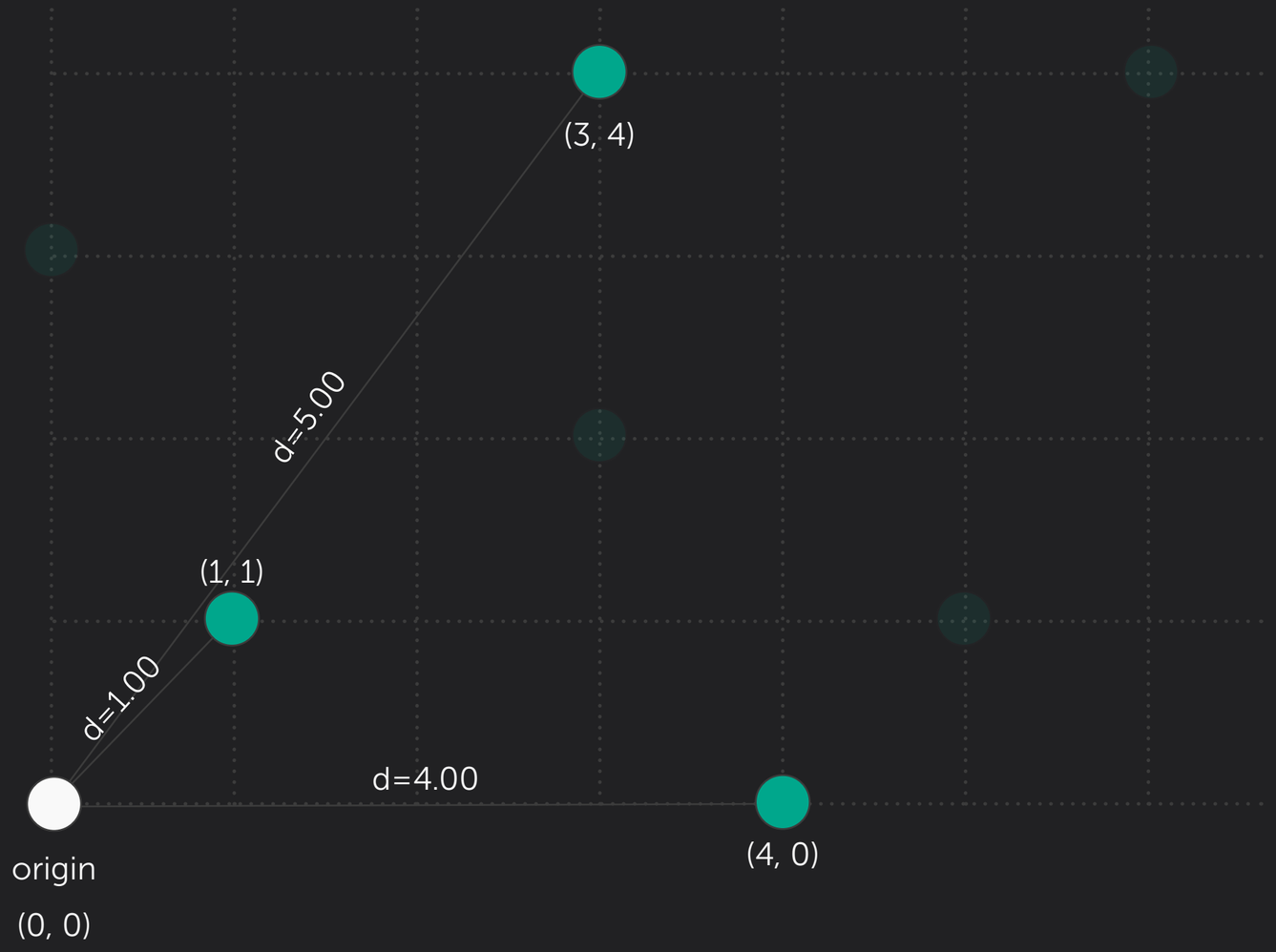


d=5.00 d=1.00 d=4.00

```
points = [ [5, 1], [6, 4], [3, 4], [1, 1], [4, 0], [0, 3], [3, 2]]
```

```
append dist:  [5.00, [3,4]]  [1.00, [1,1]]  [4.00, [4,0]]
```

```
sorted (  [1.00, [1,1]], [4.00, [4,1]], [5.00, [3,4]]  )
```



# >>>

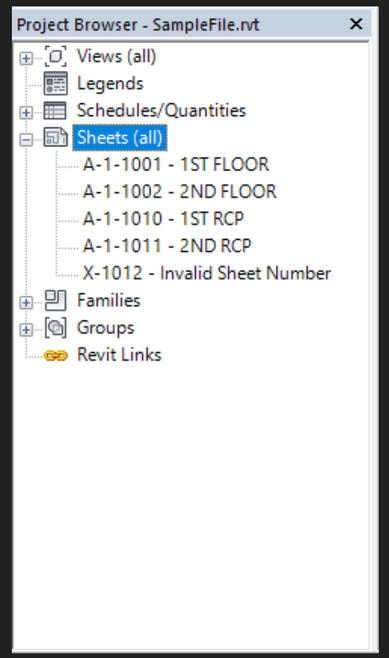
example-dynamo-1-sort_points.dyn

# exercise #2

validate sheet naming pattern

# A-1-1000

letter digit 4 digits



# A-1-1000

letter digit 4 digits

```
>>> name = 'A-1-1000'
>>> chunks = name.split('-')
>>> print(chunks)
['A', '1', '1000']
>>> len(chunks) == 3
True
>>> chunks[0].isalpha()
True
>>> chunks[1].isnumeric()
True
>>> chunks[2].isnumeric() and len(chunks[2]) == 4
True
```

| A | - | 1 | - | 1000 |
|---|---|---|---|---|
| letter | dash | digit | dash | 4 digits |

| \D | - | \d | - | \d{4} |
|---|---|---|---|---|
| letter | dash | digit | dash | 4 digits |

```
pat = '\d-\d-\d{4}
```

# regular expressions

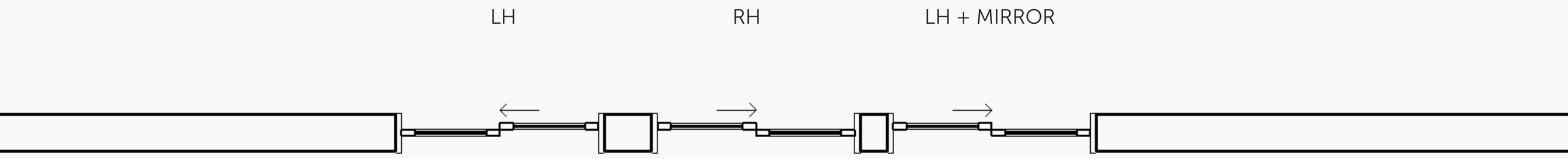
```
import re
```

>>>

Sample.rvt + example-dynamo-2-sheet-number.dyn

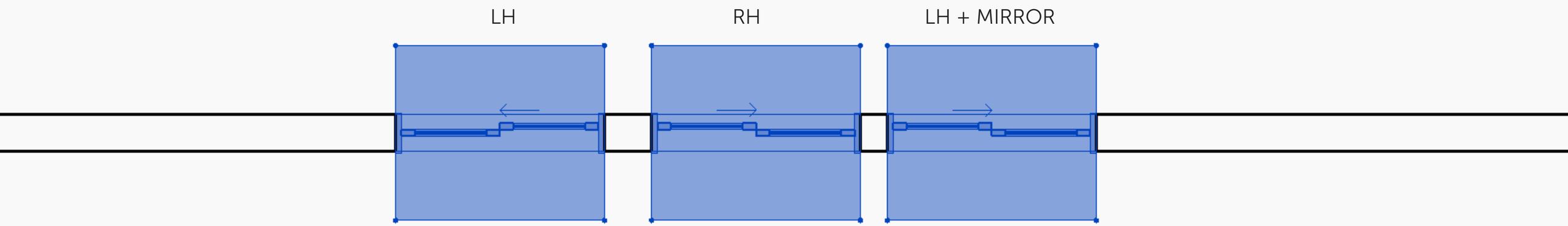
# exercise #3

check for mirrored doors and set parameter



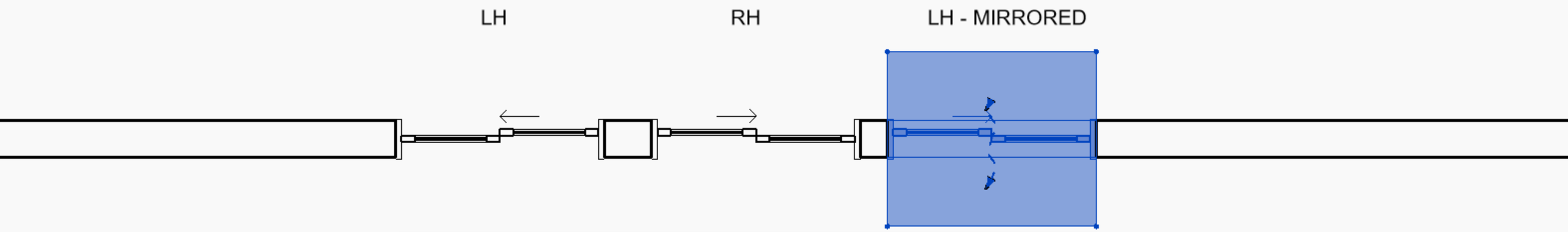

# find mirrored door elements

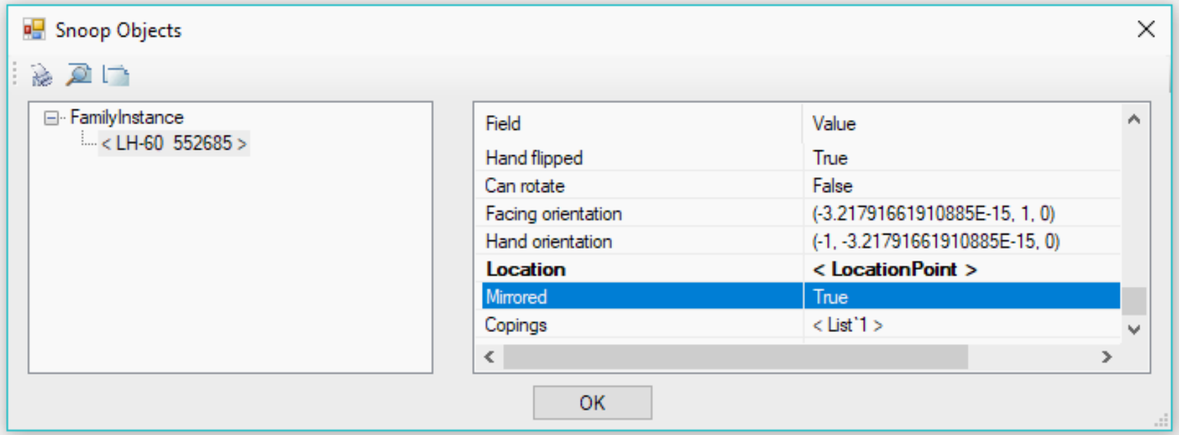
the issue



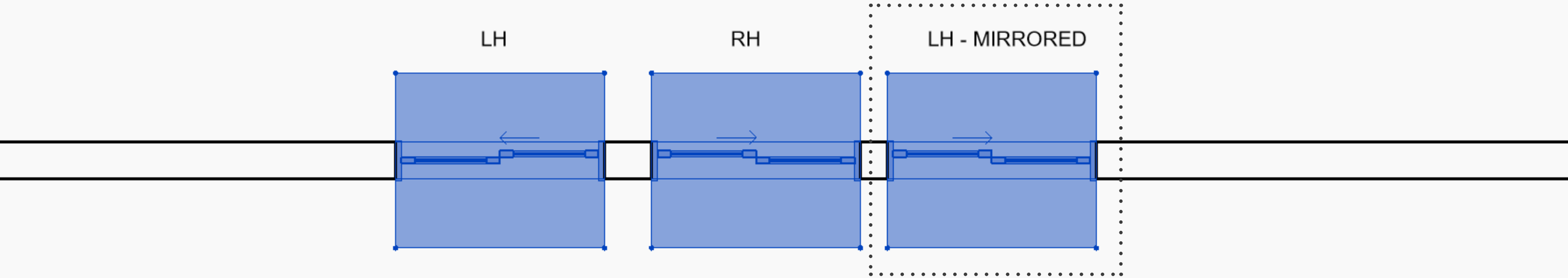

# selecting all doors

step 1


LH
RH
LH - MIRRORED


Snoop Objects
FamilyInstance
< LH-60 552685 >
Field
Value
Hand flipped
True
Can rotate
False
Facing orientation
(-3.21791661910885E-15, 1, 0)
Hand orientation
(-1, -3.21791661910885E-15, 0)
Location
< LocationPoint >
Mirrored
True
Copings
< List`1 >
OK

step 2




# add parameter to mirrored doors

step 3

>>>

Sample.rvt + example-dynamo-3-door-mirrored.dyn

# misc + tips

```
>>> import clr
>>> clr.AddReference('ProtoGeometry')
>>> from Autodesk.DesignScript.Geometry import Vector
>>> dir(Vector)
['Add', 'AngleAboutAxis', 'AngleBetween', 'AngleWithVector', 'AsPoint',
'ByCoordinates', 'ByTwoPoints', 'CheckArgsForAsmExtents', 'ComputeHashCode',
'Cross', 'Dispose', 'DisposeDisplayable', 'Dot', 'Equals', 'GetHashCode', 'GetType',
'IsAlmostEqualTo', 'IsParallel', 'Length', 'MemberwiseClone', 'Normalized', 'ReferenceEquals',
'Reverse', 'Rotate', 'Scale', 'Subtract', 'Tags', 'Tessellate', 'ToString', 'Transform',
'X', 'XAxis', 'Y', 'YAxis', 'Z', 'ZAxis', '__add__', '__class__', '__delattr__',
'__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__getattribute__',
'__hash__', '__init__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__',
'mConstructor', 'scaleFactor']
```

# dir()

prints methods and attributes

```
>>> def add_two(x, y):
...     """ This function takes 2 numbers and returns the sum """
...     return x + y
...
>>> add_two.__doc__
' This function takes 2 numbers and returns the sum '
```

```
>>> FilteredElementCollector.__doc__
'''
This class is used to search, filter and iterate through a set of elements.
FilteredElementCollector(document: Document, viewId: ElementId)
FilteredElementCollector(document: Document, elementIds: ICollection[ElementId])
FilteredElementCollector(document: Document)
'''
```

# __doc__

shows docstring of class or function

# Transaction

misc

```python
clr.AddReference("RevitServices")
import RevitServices

from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
doc = DocumentManager.Instance.CurrentDBDocument

# "Start" the transaction
TransactionManager.Instance.EnsureInTransaction(doc)


# "End" the transaction
TransactionManager.Instance.TransactionTaskDone()
```

GitHub, Inc. [US] | https://github.com/DynamoDS/Dynamo/wiki/Python-0.6.3-to-0.7.x-Migration#transactions

### Transactions

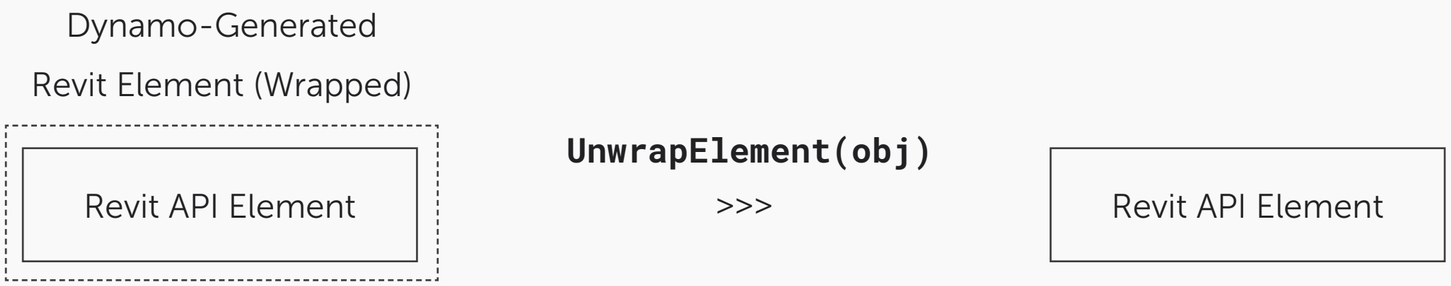
Dynamo provides its own Transaction framework for working with the RevitAPI. This means that your Python script will be executing in the context of an overall Dynamo Transaction.

If you are writing RevitAPI code that requires a Transaction, then you may use the Dynamo `TransactionManager`.

```python
clr.AddReference("RevitNodes")
import Revit
# Adds ToDSType (bool) extension method to Wrapped elements
clr.ImportExtensions(Revit.Elements)
# Adds ToProtoType, ToRevitType geometry conversion extension methods to objects
clr.ImportExtensions(Revit.GeometryConversion)
```

# UnwrapElement( )

misc




UnwrapElement() is function and is always available. It can be called on single elements or lists

# UnwrapElement( )

misc

Dynamo Geometry

**`obj.ToRevitType() *`**

>>>

Revit API Geometry

# ToRevitType( )

misc

**`* ToRevitType()`** method has to be manually imported

```
import clr
clr.AddReference("RevitNodes")
import Revit
# Import ToProtoType, ToRevitType geometry conversion
extension methods
clr.ImportExtensions(Revit.GeometryConversion)
```

Revit API Geometry

**`obj.ToDSType(bool) *`**

>>>

Dynamo Geometry

# ToProtoType( )

misc

**`* ToDSType()`** must also method has to be manually imported

Furthermore, this method receives a boolean (True or False)
to indicate whether the element is “Revit-owned” or not.

# If element **WAS CREATED** in script, use False (non-Revit-owned)

# If element was **NOT CREATED** in script, use True (Revit-owned)

# RevitLookUp

misc

# RevitPythonWrapper

misc

# closing

# remember

as you learn, there are only 4 things you need to remember

- ☑ docs.python.org
- ☑ ironpython.net
- ☑ dynamo wiki + forum
- ☑ revit api docs

# read the manual

skim through; revisit later

# use an interactive interpreter

the interpreter allows you to explorer the language interactively

# use revitpythonshell + lookup

let’s you explore revit objects, understand the internal data structures and relationships

# leverage communities

stackoverflow, github, dynamobim.org, etc

thank you