SD5000080

El Poder de la Automatización: Python en Dynamo para Revit y Civil 3D

Raquel Bascones Recio
Autodesk

David Licona
Autodesk

Objetivos de Aprendizaje

- Descubrir la programación orientada a objetos
- Aprender las bases y elementos de lenguaje de Python
- Diferenciar entre Python, IronPython y CPython
- Comprender las bases de las API de Revit, AutoCAD y Civil 3D

Descripción

Post-COVID 19, el mercado global de la industria AECO tiene un crecimiento esperado del 10.7% en los próximos 7 años, de acuerdo con un estudio de Allied Market Research. Para cumplir este objetivo, la automatización es clave para desarrollar mejores y más proyectos, con menos errores y en menos tiempo. Esta clase cubrirá los aspectos fundamentales de la programación en Python para Dynamo. Descubriremos qué es la programación orientada a objectos, así como la sintaxis básica de Python y los elementos de este lenguaje. Introduciremos conceptos de las API tanto de Revit, como de AutoCAD y Civil 3D, y mostraremos como interactuar con ellas vía Python en Dynamo. En esta sesión, presentaremos varios casos de ejemplo y plantillas para crear tus scripts tanto en Dynamo para Revit como Dynamo para Civil 3D.

Ponentes

Raquel Bascones Recio
Senior Implementation Consultant

2010 – 2017: Architect & Landscape Architect
2017 – 2019: Autodesk Global Product Support
Desde 2019: Autodesk Consulting
raquel.bascones.recio@autodesk.com
LinkedIn

David Licona
Implementation Consultant

2014 – 2017: Project Engineer – Construction
2017 – 2020: Project Manager – Rail infrastructure
Desde 2020: Autodesk Consulting
david.licona@autodesk.com
LinkedIn

Python en Dynamo

¿Qué es Python?

Python es un lenguaje de programación que fue desarrollado en primer lugar a finales de los años 80 por Guido van Rossum. Desde entonces se ha convertido en uno de los lenguajes más populares entre los programadores, ocupando siempre puestos dentro de los 5 lenguajes más usados.

Python es desarrollado bajo una licencia de software abierto aprobada por la OSI, haciendo que sea de uso y distribución gratuita, incluso para uso comercial. La licencia de Python esta administrada por la Python Software Foundation.

Hay una comunidad enorme de usuarios que comparten conocimiento e información en internet, vía videos, blogs y foros. Además, las capacidades nativas de Python pueden ser extendidas con los miles de módulos disponibles que han sido creados por la comunidad.

Implementaciones

Python es un lenguaje de programación interpretado, lo que significa que el código fuente de un programa en Python es convertido en *bytecode* cuando se ejecuta por el interpretador de Python.

En Dynamo, Podemos trabajar con dos implementaciones diferentes de Python.

IronPython

Esta implementación fue creada por Microsoft en la mitad de los 2000 y se enfoca en .NET. Años más tarde, Microsoft abandono el proyecto y, en la actualidad, está gestionado por voluntarios.

Dynamo incluye un editor de Python que usas IronPython 2.7. Esta versión de la implementación usa Python 2.7, que alcanzó el final de mantenimiento el 1 de enero de 2020.

CPython

La versión 2.7 de Dynamo introdujo un nuevo motor de Python: CPython. CPython es la implementación tradicional o estándar de Python y se enfoque en la versión 3 del lenguaje. Python 3 es fundamentalmente el mismo lenguaje que Python 2, pero ha habido cambios en la sintaxis y en la librería central para hacer de Python más consistente y fácil de aprender. Permite la importación de módulos.

Ventajas e inconvenientes

Ventajas

- Fácil de leer, cercano al inglés escrito
- Gram comunidad de usuarios
- Flexible, usado en muchas industrias diferentes
- Gestiona la memoria automáticamente
- Integrable con otros lenguajes

- Gratuitos

Inconvenientes

- Es un lenguaje interpretado (acceso al código base)
- La subsanación de errores lleva largo tiempo ya que los problemas son mostrados uno a uno
- Es más lento que otros lenguajes

Principios de programación

Python es un lenguaje multi paradigma: soporta diferentes estrategias de programación.

Una de las estrategias más populares para solucionar un problema en programación es creando objectos. Esto se conoce como Programación Orientada a Objectos. Las APIs de las aplicaciones anfitrionas de Dynamo están basadas en este paradigma.

El concepto de Programación Orientada a Objectos se centra en la creación de código reutilizable. Este concepto también se conoce como DRY (Don't Repeat Yourself).

Class

Una clase es una colección de objectos. Una clase contiene el prototipo o instrucciones para la creación de objectos. Es una entidad lógica que contiene atributos y métodos.

Las clases se basan en el principio de herencia. Existen clases madres que van derivando en otras clases más específicas que heredan los atributos y propiedades de las madres.

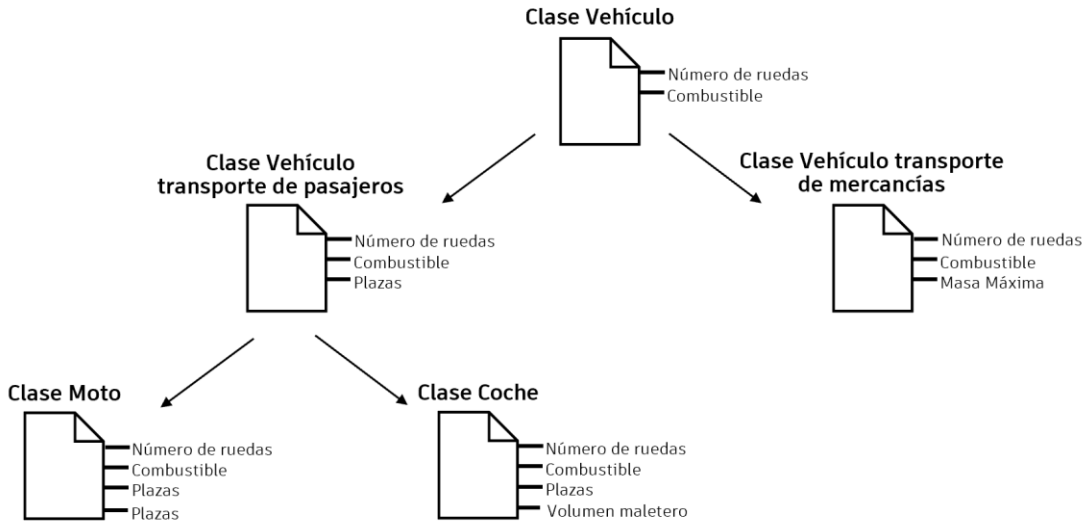


Ilustración 1 - Clases y herencia

Object

El objeto es una entidad que tiene un estado y un comportamiento asociado a él. Es una instancia o copia de la clase.

Nodo de Python Script

Dynamo contiene dos nodos en la instalación para ejecutar código de Python: *Python Script* y *Python Script from string*.

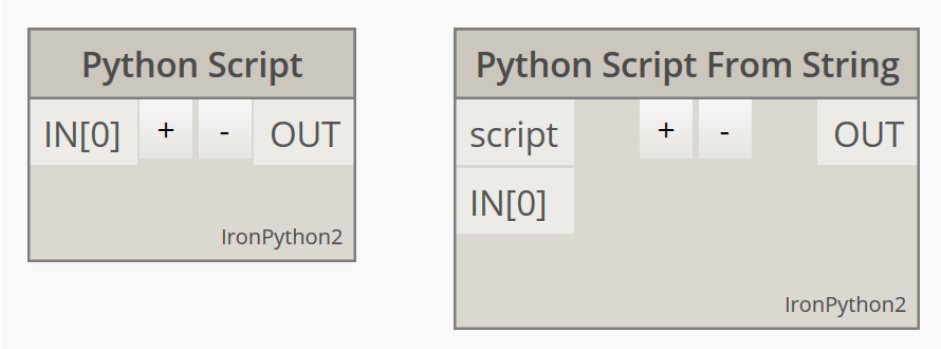


Ilustración 2 - Nodos Python script

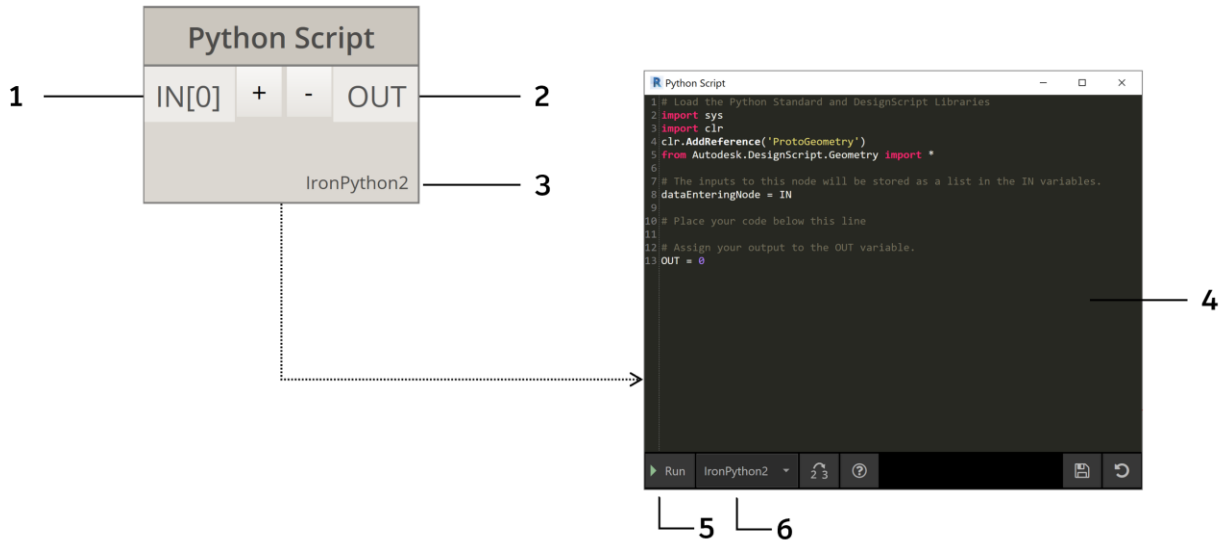


Ilustración 3 - Partes del nodo de Python Script

1. Entradas: Inputs definidos por el usuario controlados por los botones +/-. Dentro del Código las variables IN[0]…IN[n] pueden ser asignadas a otras variables.
2. Puerto de salida: Hay un único Puerto de salida, el resultado mostrado en Dynamo será el asignado a la variable OUT en el código.
3. Implementación: IronPython 2 o CPython 3. Sólo disponibles en Dynamo 2.7+.
4. Editor: Código para ser ejecutado. La ventana se muestra después de hacer doble clic en la parte gris clara del nodo.
5. Run: Ejecución
6. Selector de implementación: Sólo disponible en Dynamo 2.7+.

Editor de Python

Dynamo incluye un editor de código como explica la sección anterior. Sin embargo, este editor tiene unas funcionalidades reducidas en comparación con otros IDE (Integrated Development Environment). Algunas de las ventajas de otros IDE son:

- Autocompletado de funciones y variables
- Revisión de sintaxis y ortografía
- Muestra de errores y advertencias
- Colapso de bloques de código para mejorar la navegación y visualización
- Revisión de estilo

Existen multitud de IDE gratuitos y disponibles. Una vez que el código está escrito, puede ser tanto copiado en el editor de Python en Dynamo o ejecutado con un módulo externo.

Python y Revit API

Assemblies y Namespaces

Documentos

Existen dos documentos diferentes en Revit API:

Document

- Es el documento RVT o RFA como tal
- Contiene propiedades con el nombre del archivo, la ruta, si está almacenado en la nube
- Tiene los métodos para la creación y modificación de elementos

UIDocument

- Es el archivo tal y como es presentado en la interfaz de usuario
- Contiene métodos para interactuar con el usuario como mensajes o selección directa de elementos

Elementos

La mayor parte de los objectos de Revit derivan de una clase llamada *Element*. De esta clase derivan objectos tan diferentes como muros y vistas.

Una diferenciación importante debe hacerse entre los elementos que son familia sistema y las familias cargables. Las familias sistema tienen sus propias clases en la API, tanto para los objectos como para sus tipos. Por ejemplo, en el caso de los muros los objectos son de la clase *Wall* y los tipos de muro son de la clase *WallType*.

En el caso de las familias cargables o componentes, esto sería imposible ya que pueden crearse millones de familias diferentes. Por tanto, existen unas clases determinadas para todas estas familias. Los objetos de las familias cargables son de la clase *FamilyInstance* y los tipos son de la clase *FamilySymbol*. Las familias, definidas como un conjunto de tipos, tienen también su propia clase: Family. Para identificar mejor las familias cargables, puede añadirse la categoría para su diferenciación.

	Familia Sistema	Familia Cargable
Element Type	*WallType* *FloorType*	*FamilySymbol* *&* *Category - Doors, Windows*
Instance	*Wall* *Floor*	*FamilyInstance* *&* *Category - Doors, Windows*

Selección mediante filtros

El objecto *FilteredElementCollector(document)* permite hacer selecciones de objectos en un documento de Revit. Este objecto es un colector que necesita al menos de un filtro para funcionar, de lo contrario dará un error.

Existen tres tipos de filtros:

- Filtros rápidos (Quick Filter): No es necesario expandir el elemento en la memoria para comprobar si es seleccionado.
- Filtros lentos (Slow Filter): Necesita expandir el objeto en la memoria para comprobar su estado o atributos.
- Filtros lógicos: Combinan dos o más filtros con los operadores lógicos AND y OR.

Es recomendable usar primero filtros rápidos y continuar con filtros lentos para mejorar el rendimiento y la velocidad de procesado. Así los filtros lentos tendrán menos elementos que verificar.

Cada tipo de filtro dentro de estas tres categorías tiene su forma de ser creado y debe ser aplicado al colector con el método *WherePasses(filter)*.
Existen una serie de métodos shortcut que permiten la aplicación de filtros rápidos de una manera directa, sin tener que usar el método *WherePasses(filter)*. Los más usados son *OfClass(class)* y *OfCategory(built in category).*

Es importante destacar que esta selección de objectos no se refleja en la interfaz de usuario.

El objeto *FilteredElementCollector(document)* es un iterable por lo que tiene que ser convertido en una colección de Python, bien mediante los métodos *ToElements()* o *ToElementIds()*, o bien a través de bucles de Python (for loop).

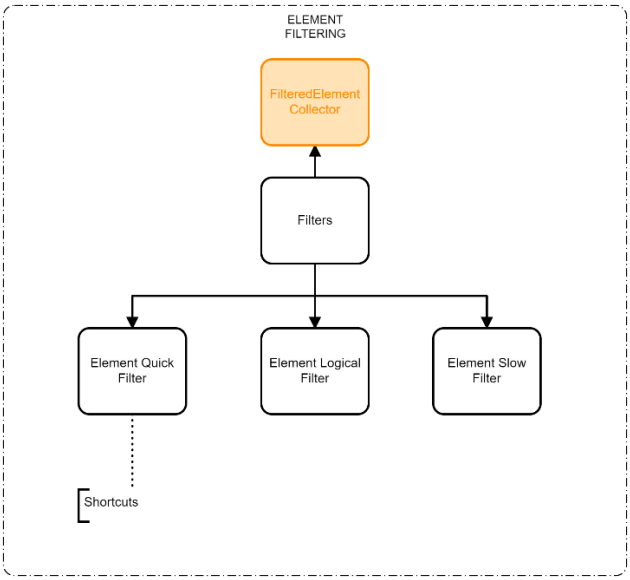


Ilustración 4 - Filtered Element Collector

Parámetros

Todos los objetos de Revit tienen parámetros, que pueden definirse como objectos que almacenan un valor.

Todos los parámetros de Revit, tanto de tipo como de ocurrencia, son de la clase *Parameter*. Su diferencia radica solamente en el elemento al que están asociados.

Los parámetros, como objecto, se obtienen desde el elemento en sí mismo mediante diferentes métodos:

- Element.GetOrderedParameters(): Devuelve una colección de todos los parámetros del elemento en el orden que aparecen en la UI.
- Element.Parameters: Devuelve una colección de todos los parámetros del elemento.
- Element.LookupParameter(name): Devuelve el primer parámetro encontrado con el nombre introducido.
- Element.GetParameters(name): Devuelve todos los parámetros encontrados con el nombre introducido.
- Element.get_Parameter(builtinParameter): Retorna el parámetro nativo de Revit para este elemento. Hay una enumeración en la API con todos los disponibles.

  Con los métodos anteriores se obtiene el objeto parámetros. Para conseguir el valor se debe utilizar alguno de los métodos disponibles en la clase, dependiendo del tipo de almacenamiento del parámetro:
- AsDouble()
- AsString()
- AsElementId()
- AsInteger()
- AsValueString(): Devuelve el valor como se muestra en la interfaz de usuario, incluyendo unidades

  Para la modificación del valor debe usarse el método Set(value) aplicado al parámetro. El valor debe ser del tipo de variable admitido en el parámetro. Esta operación debe llevarse a cabo dentro de una transacción.

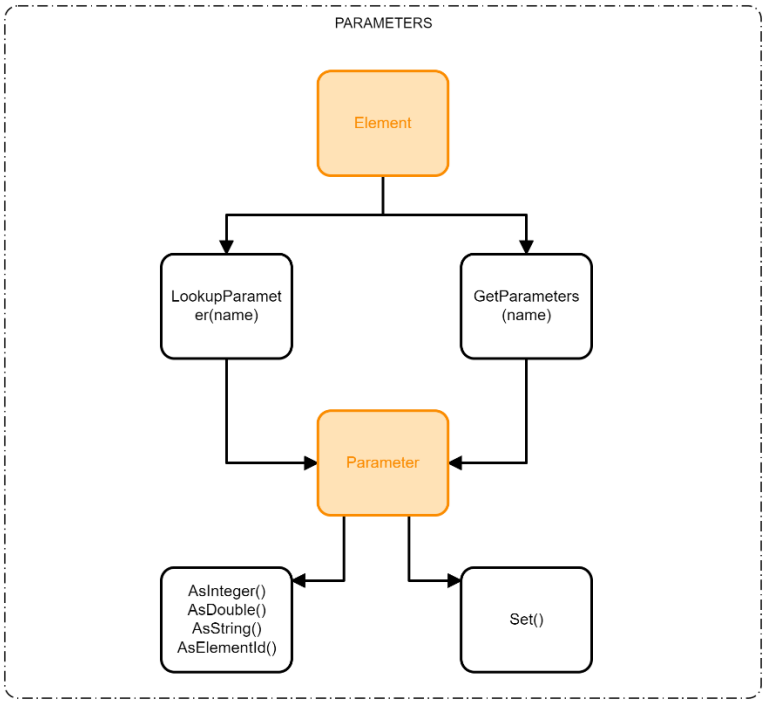


Ilustración 5 - Parámetros

Modificación de elementos

Se pueden realizar modificaciones de elementos en tres niveles:

Documento

A nivel documento se pueden realizar operaciones de rotación, transformación, simetría. Estos métodos se encuentran dentro de la clase *ElementTransformUtils.*

Elemento

A nivel elemento se pueden modificar la localización, el tipo de elemento y los parámetros.

Sketch

Desde Revit 2022, también se pueden modificar elementos basados en sketches (suelos, muros con perfil, techos, tejados, aberturas).

Creación de elementos

Los métodos para creación de elementos pueden encontrarse en tres lugares distintos dentro de la API de Revit:

Proyecto

- *Document.Create*
- Para la creación de
  - Group
  - FamilyInstance
  - DetailLine
  - Floor
  - ….

Familia

- *Document.FamilyCreate*
- Para la creación de
  - Sweep
  - Extrusion
  - Lofts
  - ….

Métodos específicos

- Métodos de creación en la misma clase del objecto
- Para la creación de
  - Wall
  - AdaptiveComponent
  - …

Python y Civil 3D/AutoCAD API

La utilización de Python al utilizar Dynamo para Civil 3D, permite expandir las capacidades existentes con Dynamo y paquetes como el Civil 3D Toolkit. Esto permite interactuar con objetos para los cuales hoy en día no existe ningún nodo en Dynamo como las líneas de tabulación o los Data Shortcuts de Civil 3D. También se puede utilizar Python para efectuar ciertos comandos AutoCAD como utilizar el método WBLOCK para exportar objetos hacia un nuevo DWG

Assemblies y Namespaces

Civil 3D está formado por 3 tecnologías de base : AutoCAD, AutoCAD Architecture Desktop y Map 3D. Por lo tanto, para poder interactuar con todos los aspectos del software, es necesario no solo agregar los Assemblies correspondientes a las entidades de Civil 3D, pero también los de AutoCAD y los de AutoCAD Architecture.

AutoCAD
Acmgd.dll
Acdbmgd.dll
Accoremgd.dll

AutoCAD Architecture
AecBaseMgd.dll
AecPropDataMgd.dll

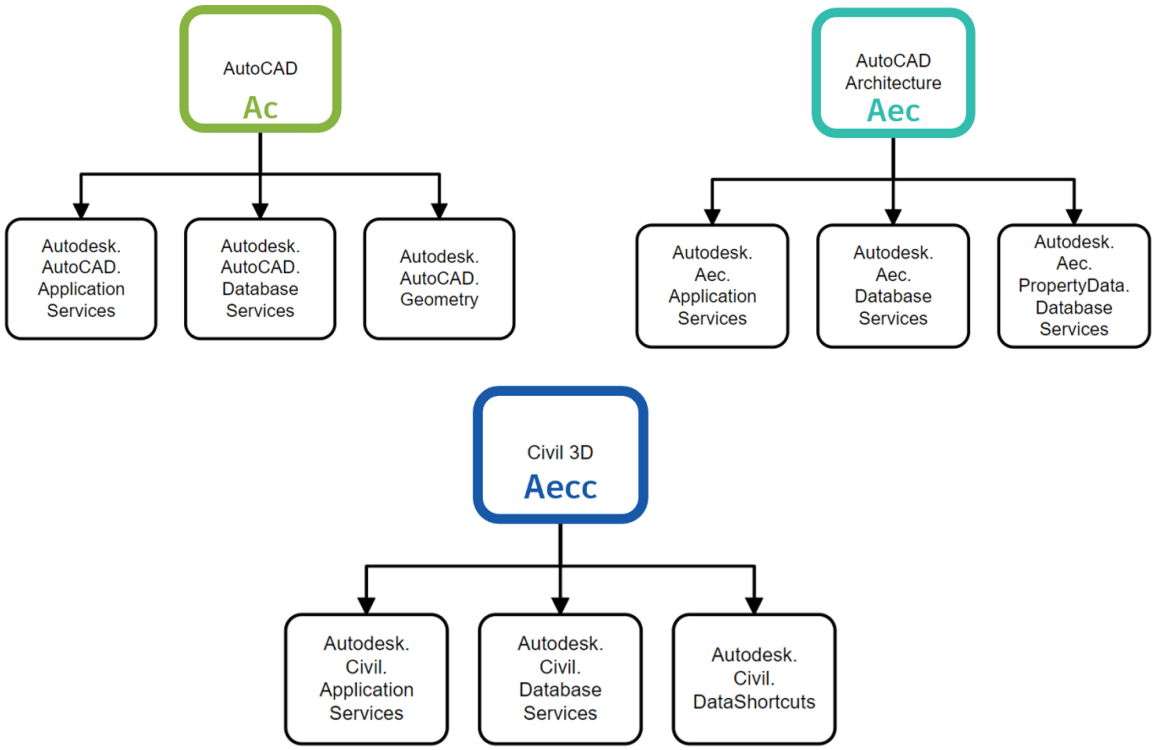
Civil 3D
AeccDbMgd.dll
AeccPressurePipes.dll
AeccDataShortcutMgd.dll

De manera general, los assemblies contienen clases que permiten interactuar con diferentes aspectos :

- AutoCAD : Visualización en el espacio objeto de Civil 3D, entidades básicas de geometría como círculos/polilíneas, barra de comando, etc.
- AutoCAD Architecture : Juegos de propriedades (property sets)
- Civil 3D : Objetos Civil 3D (superficies, drenajes, corredores, ejes, perfil longitudinal, datashortcuts, etc.)

De la misma manera que en Revit, una vez añadidos los assemblies, debemos importar los Namespaces, que no es otra cosa que una colección de clases. Podemos importar todo el namespace o solo las clases con las que queremos interactuar.

La siguiente imagen muestra algunos de los namespaces correspondientes a los assemblies mencionados anteriormente.



Los siguientes ejemplos de código Python, muestran como agregar los assemblies e importar namespaces :

- Añadir el *assembly* :
  - *clr.AddReference("AecBaseMgd")*
  - *clr.AddReference("AeccDbMgd")*

- Importar los objectos en el namespace:
  - *from Autodesk.AutoCAD.Geometry import **
  - *import Autodesk.AutoCAD.ApplicationServices.Application as acapp*

Documentos

Existen dos documentos distintos con los cuales se necesitar interactuar :

AutoCAD Document

- Es el documento activo, el DWG
- Contiene propiedades con el nombre del archivo, la ruta, etc.
- Contiene la Base de Datos con todos los objetos AutoCAD entre otros elementos y los métodos para interactuar con esta base de datos

- Contiene el Editor que permite entre otras cosas la interacción con el usuario a través de la barra de comandos
- Contiene el transaction manager, que permite realizar operaciones en la base de datos de manera segura

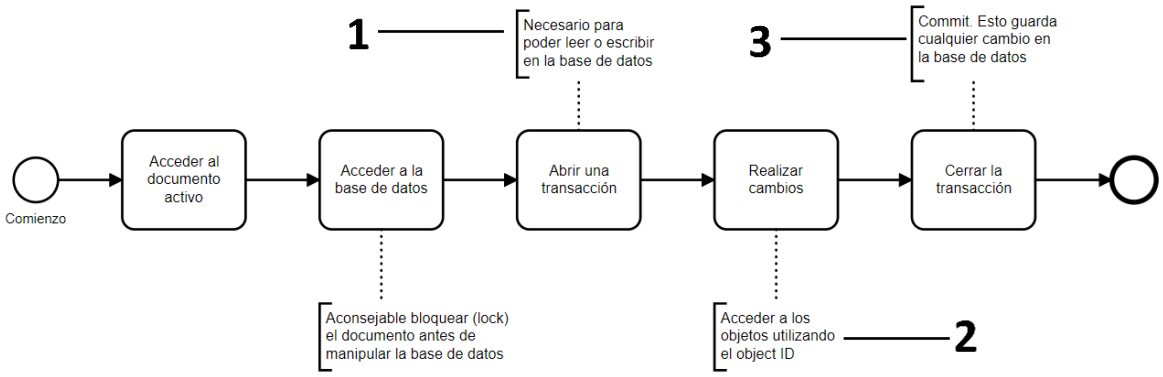
Civil Document

- Contiene métodos para interactuar con las entidades Civil 3D – superficies, corredores, ejes, perfiles longitudinales, etc.
- Contiene métodos para crear, modificar, definir los parámetros (estilos, etiquetas) de las entidades Civil 3D

Transacciones

Las transacciones permiten interactuar con la base de datos de manera segura. Para acceder a la base de datos es necesario :

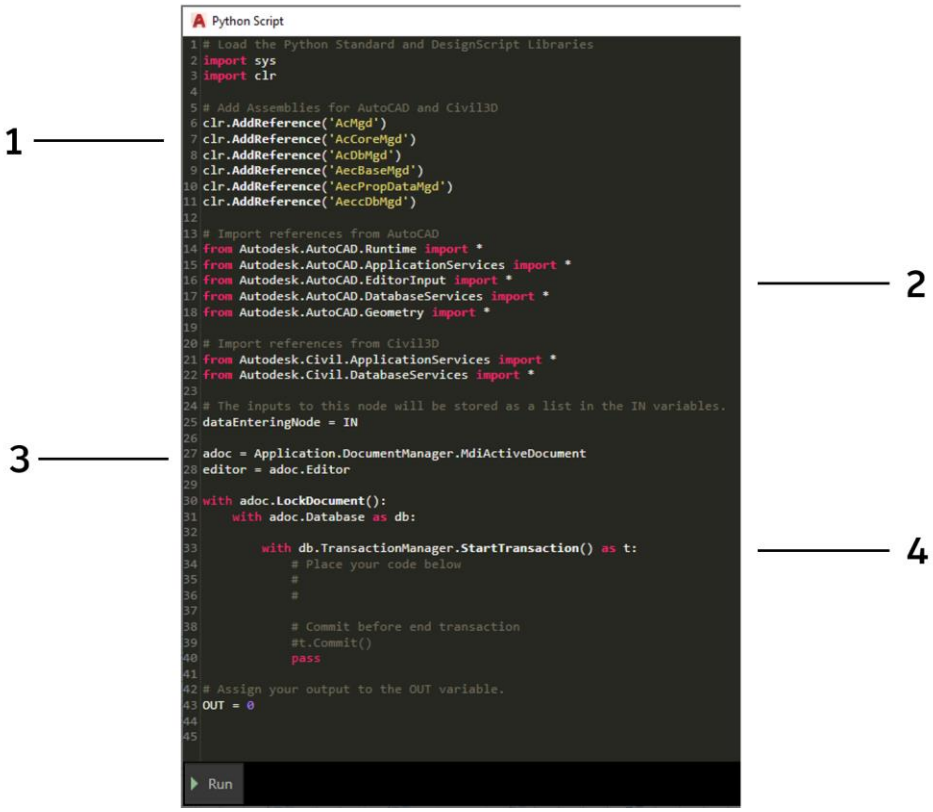
1. Abrir una transacción, lo cual permite
2. Acceder a los objetos en la base de datos para leer sus propiedades, o para modificar/agregar/eliminar objetos de la base de datos
3. Finalizar la transacción con el método “Commit” *.Commit()* para que los cambios realizados surjan efecto



Plantilla Python Dynamo

La plantilla estándar de Python en Dynamo para Civil 3D contiene los elementos descritos anteriormente :

4. Assemblies
5. Namespaces
6. Acceso al documento activo
7. Transacciones




```
# Load the Python Standard and DesignScript Libraries
import sys
import clr

# Add Assemblies for AutoCAD and Civil3D
clr.AddReference('AcMgd')
clr.AddReference('AcCoreMgd')
clr.AddReference('AcDbMgd')
clr.AddReference('AecBaseMgd')
clr.AddReference('AecPropDataMgd')
clr.AddReference('AeccDbMgd')

# Import references from AutoCAD
from Autodesk.AutoCAD.Runtime import *
from Autodesk.AutoCAD.ApplicationServices import *
from Autodesk.AutoCAD.EditorInput import *
from Autodesk.AutoCAD.DatabaseServices import *
from Autodesk.AutoCAD.Geometry import *

# Import references from Civil3D
from Autodesk.Civil.ApplicationServices import *
from Autodesk.Civil.DatabaseServices import *

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

adoc = Application.DocumentManager.MdiActiveDocument
editor = adoc.Editor

with adoc.LockDocument():
    with adoc.Database as db:

        with db.TransactionManager.StartTransaction() as t:
            # Place your code below
            #
            #


            # Commit before end transaction
            #t.Commit()
            pass

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

Y el ejemplo siguiente publicado en el foro de Dynamo por Zachri Jensen, demuestra la realización de una transacción

```
def AddSurfaceBoundaries(surface, boundaries, midOrdinateDistance, boundaryType,

    global adoc

    if not isinstance(boundaries, list):
        boundaries = [boundaries]

    with adoc.LockDocument():
        with adoc.Database as db:
            with db.TransactionManager.StartTransaction() as t:
                surfId = surface.InternalObjectId
                surf = surfId.GetObject(OpenMode.ForWrite)
                surfBoundaries = surf.BoundariesDefinition
                # Create collection of boundary entity IDs
                boundaryEntities = ObjectIdCollection()
                for object in boundaries:
                    boundaryEntities.Add(object.InternalObjectId)
                # Parse enumeration
                boundaryType = System.Enum.Parse(SurfaceBoundaryType, boundaryTyp
                # Add boundary entities to surface boundary collection
                surfBoundaries.AddBoundaries(boundaryEntities, midOrdinateDistanc
                surf.Rebuild()
                t.Commit()
                pass
```

Objetos

Existen diferentes posibilidades para interactuar con los objetos de la base de datos :

Object ID

- **Referencia un objeto en la base de datos**, lo que permite identificarlo. Cada objeto tiene su propio ID
- Es un **entero válido solamente para la sesión activa** de AutoCAD o Civil 3D. El object ID puede cambiar si se cierra el programa.
- Se utiliza para **interactuar con un objeto** (modificar, eliminar)
- Una de las propiedades del Object ID es el Handle

Handle

- **Referencia un objeto en la base de datos**, lo que permite identificarlo. Cada objeto tiene su propio Handle
- Es un **hexadecimal válido para todas las sesiones** de AutoCAD o Civil 3D. El Handle siempre será el mismo para un objeto, aunque se cierre el programa
- Se utiliza principalmente para **serializar información,** es decir si queremos exportar propriedades de un objeto (ej. volumen de un sólido) y necesitamos después identificar el objeto particular al que la propiedad pertenece
- Una de las propiedades del Handle es el Object ID

Object ID Collection

- **Referencia a una colección (grupo) de objetos en la base de datos**
- Es una **lista de Object IDs**
- Cuando interactuamos con las entidades Civil 3D, generalmente obtenemos primero un Object ID Collection, y tenemos que acceder a esta lista para obtener el object ID y poder interactuar con el objeto

CivilDocument Methods

The CivilDocument type exposes the following members.

Methods

	Name	Description
	GetAlignmentIds	Gets the objectId collection of all Alignment objects in the drawing.
	GetAlignmentTableIds	Gets the objectId collection of all alignment tables.
	GetAllPointIds	Gets the objectId collection of all points in the drawing.
S	GetCivilDocument	Gets the CivilDocument object from the AutoCAD Database object.
	GetGeneralSegmentLabelIds	Gets the objectId collection of all GeneralSegmentLabel objects in the drawing.
	GetIntersectionIds	Gets the objectId collection of all intersection objects in the drawing.
	GetLegendTableIds	Gets the objectId collection of legend tables.
	GetNoteLabelIds	Gets the objectId collection of all NoteLabel objects in the drawing.
	GetParcelSegmentTableIds	Gets the objectId collection of parcel segment tables.
	GetParcelTableIds	Gets the objectId collection of parcel tables.
	GetPipeNetworkIds	Gets the pipe network object id collection.
	GetPointTableIds	Gets the objectId collection of point tables.
	GetSiteIds	Gets the objectId collection of all site objects in the drawing.
	GetSitelessAlignmentId	Gets the objectId of a siteless Alignment object by name.

Navegar el API

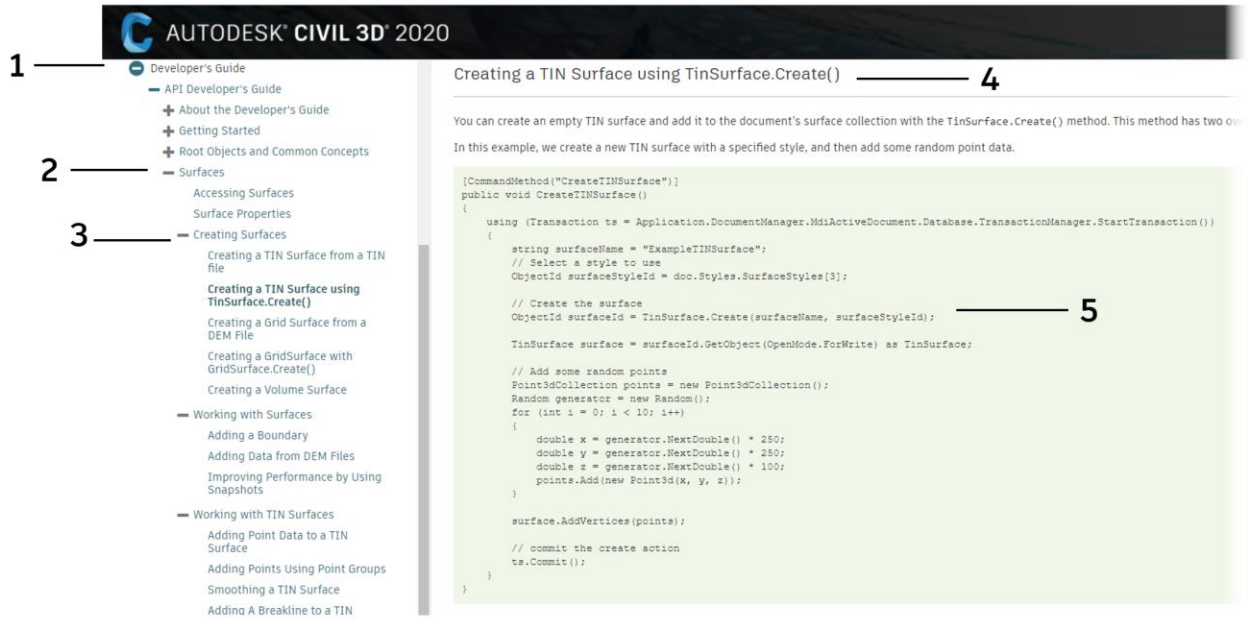
La documentación oficial de los APIs está dividida en dos partes principales. Esto es válido tanto para el API de AutoCAD como para el de Civil 3D. Es importante resaltar que la documentación oficial **no contiene ejemplos** utilizando Python.

Developer's Guide

- Contiene información sobre el API y explica los conceptos principales
- Provee ejemplos (en C# o VBA)

El siguiente ejemplo muestra diferentes aspectos del Developer's Guide :

8. Acceso al Developer's Guide dentro de la ayuda de Civil 3D
9. Información sobre una entidad Civil 3D (Superficies en este caso)
10. Información sobre cómo crear superficies
11. Método específico que permite crear superficies
12. Ejemplo de utilización de este método

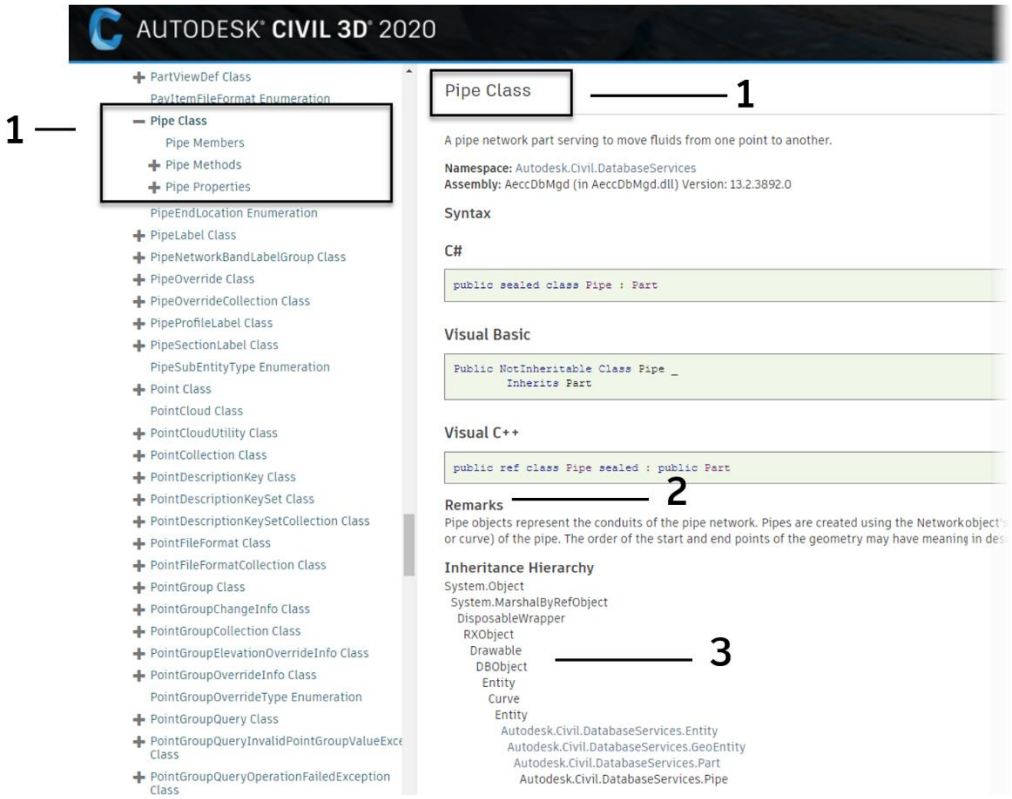


API Reference Guide

- "Diccionario" para navegar el API
- Contiene los namespaces, las clases que pertenecen a cada uno de ellos y los métodos y propriedades que tiene cada clase

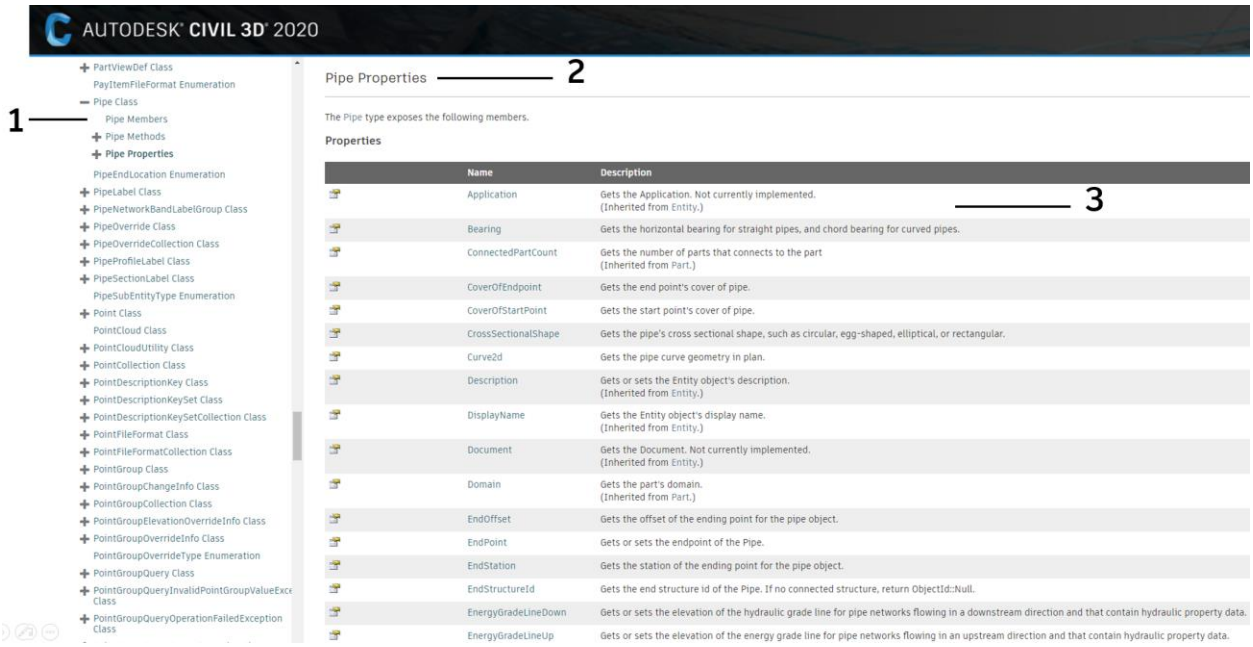
El siguiente ejemplo muestra diferentes aspectos del Reference Guide:

13. Acceso a las clases dentro de un namespace (en este caso la clase Pipe perteneciente al namespace Autodesk.Civil.DatabaseServices
14. Información importante para tener en cuenta para trabajar con esta clase
15. Jerarquía de las clases



Dentro de cada clase se encuentran los métodos y propriedades de cada clase :

16. Acceso a las propriedades de la clase Pipe
17. Propiedades de la clase Pipe
18. Lista de las diferentes propiedades y una descripción de la información que se puede obtener



En el caso de los métodos la documentación permite saber cuáles son parámetros de entrada y de salidos – los argumentos y resultados de la función :

19. Acceso a los métodos de la clase Network
20. Paramétros necesarios para llamar al método, así como el tipo de parámetro (booleano, object ID, etc.)
21. Ejemplo de aplicación del método (en C#). No todos los métodos contienen ejemplos de aplicación

Ejemplos de utilización

Ciertos ejemplos de utilización se podrán encontrar en anexo de este documento.

Conclusiones

Para aventurarse a utilizar Python dentro de Dynamo, recomendamos empezar a partir de un problema práctico que necesite el uso de Python. Es mucho más fácil lanzarse cuando se tiene un objetivo en mente. La mejor manera de empezar es describiendo el problema y cuáles son

los pasos que debemos realizar para resolver este problema. Esta etapa es muy importante porque nos permite visualizar el código que queremos escribir. Por experiencia, recomendamos comenzar lo más detallado posible y con la práctica, se puede ya solo realizar a un nivel macro. Para esto, generalmente en Autodesk Consulting, utilizamos diagramas de flujo en el formato BPMN. Nuestro colega Paolo Serra, realizó este ejemplo descrito en esta entrada del foro de Dynamo.

Una vez descrito el problema, la siguiente etapa es comenzar a crear la solución en Python. Esto es un proceso iterativo, donde se prueba la solución dentro del espacio de Dynamo y luego regresamos a hacer cambios ya que hasta los programadores más experimentados pueden tener errores en el código que crean.

Al principio, el proceso será lento y tedioso, sobre todo por la falta de conocimiento de la API y los diferentes métodos y propriedades que podemos utilizar. Para esto, lo más fácil es ir al foro de Dynamo y hacer preguntas. La comunidad es muy activa y probablemente alguien ya hizo la misma pregunta.

Fuentes y recursos

Las siguientes fuentes pueden ser útiles para el aprendizaje de Python y su utilización dentro de Dynamo :

- Documentación de Iron Python
- Cursos en línea para aprender Python de plataformas como Coursera, Udemy, EDX, etc.
- Blogs :
  - The Building Coder – Jeremy Tammick (Revit API en C#)
  - Through the interface – Kean Wamsley (API AutoCAD en C#)
- Python Primer de Dynamo
- Clases AU :
  - Diving Deeper: A Beginner's Look at Python in Dynamo – Sol Amour
  - Untangling Python: A Crash Course on Dynamo‘s Python Node – Gui Talarico
- Documentación oficial :
  - Civil 3D – Developer’s Guide
  - Civil 3D – API Reference Guide
  - AutoCAD – Managed .NET Developer’s Guide
  - AutoCAD – Managed .NET API Reference Guide
  - Revit Developer Center
- Herramientas :
  - Revit Lookup Tool

Anexo: Python Cheat Sheet

Base Data Types

DYNAMO	PYTHON	
Integer	int	*5*
Double	float	*5.5555*
String	str	*"hello" or 'hello'*
Boolean	bool	*True, False*
Null	None	*None*

Collection Types

COLLECTION	EXAMPLE	MUTABILITY
List	*[1,2,3] or* ***list****(1,2,3)*	Mutable
Tuple	*(1,2,3)*	Inmutable
Dictionary	*{"key": value} or* ***dict****("key":value)*	Mutable
Set	***set****(1,2,3) or {1,2,3}*	Mutable (unless declared *frozenset*)

Assignment of variables

- Binding of a name with a value

Unique assignment	*x = 1*
Multiple assignments	*x, y, z = 1, 2, 3*
Assignment to the same value	*x = y = z = 0*
Increment	*x+=1 equals x = x + 1*
Decrement	*x-=1 equals x = x - 1*
Remove variable	***del** x*
Empty variables	*x = None* *x = ""* *x = []* *x = {}* *x = **set**()*

Variable conversions

- Change variable type

To integer	***int**(x)*
To float	***float**(x)*
Round float	***round**(number,#digits)*
To str	***str**(x)*
List from string	***list**("abc") > ["a", "b", "c"]* *x.**split**(separator)*
String from list	*separator.**join**(list)*
Check type	***isinstance**(x, type)*

Indexing lists

Working with values from lists

Value at index	*x = ["a", "b", "c"]* *x[0] = "a"*
Index counting from end	*x = ["a", "b", "c"]* *x[-1] = "c"*
Remove value	***del*** *x[2]*
Modify value	*x[1] = 25*
Length of a list	*x = ["a", "b", "c"]* ***len****(x) = 3*
Slicing lists	*list[start slice : end slice : step]* *x = [1, 2, 3, 4, 5]* *x[2::] = [3, 4, 5]* *x[1:3] = [2, 3, 4]* *x[::-1] = [5, 4, 3, 2, 1]* *x[::2] = [1, 3, 5]*

Boolean Logic

negates A	**not** A
True only if all the arguments are True	A **and** B
True if at least one of the arguments is True	A **or** B
Equality	A **==** B
Inequality	A **!=** B
Comparisons	A **<** B A **<=** B A **>** B A **>=** B

Math

- Import math module

Operators	**+** **-** ***** **/** **%** *(remainder of division)* **//** *(integer value of division)* ******(pow, a^b)
Absolute value	***abs***(x)
Round up to integer	***math.ceil***(x)
Round down to integer	***math.floor***(x)

Main operations on lists

Length of the list	***len***(x)
Minimum / maximum value	***min***(x) ***max***(x)
Iterator returning (index,value)	***enumerate***(x)
Iterator returning values at same index of all collections	***zip***(x, y, z)
Evaluate existence/not existence in list, returns bool	a ***in*** x a ***not in*** x
Index of an item in a list	x.***index***(a)
Count occurrences of an item in a list	x.***count***(a)

Append a value at the end	x.***append***(a)
Add a sequence of values at the end	x.***extend***([a,b, c])
Insert item at index	x.***insert***(index, a)
Remove first item of value a	x.***remove***(a)
Remove item at index and return item	x.***pop***(index)
Sort the list in place	x.***sort***()
Make a sorted copy of the list	***sorted***(x)

Conditional Statements

```
if Test :
        # do something
elif newTest :
        # do something different
else :
        # final case
```

```
# Import modules and namespaces to add references to the code
import clr
clr.AddReference('System.Windows.Forms')
from System.Windows.Forms import MessageBox


a = 35

output = 'Conditional Statements\na = {0}\n\n'.format(a)

if a > 50:
    output += 'a is greater than 50\n'
elif a < 25:
    output += 'a is less than 25\n'
else:
    output += 'a is between 25 and 50'
MessageBox.Show(output)
```

Loops | Conditional

Repeat instructions in the body until a condition is met

```
while conditional statement :
        # do something
```

- The loop will break only when the Test returns False
- It is very easy to make a mistake and create infinite loops

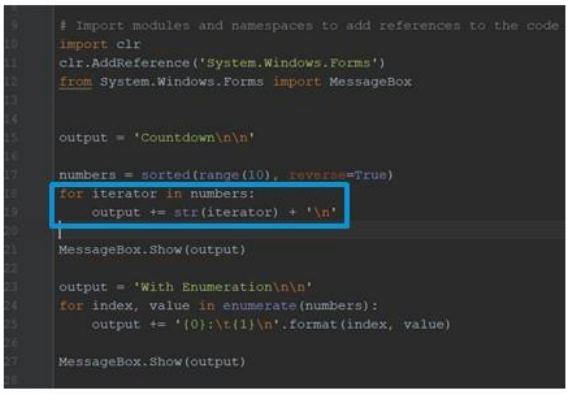
```
# Import modules and namespaces to add references to the code
import clr
clr.AddReference('System.Windows.Forms')
from System.Windows.Forms import MessageBox


a = 3
MessageBox.Show('Countdown')

while a > 0:
    MessageBox.Show(str(a))
    a -= 1
```

Loops | for … in …

Traverse a list and repeat the instructions a given amount of times

```
for iterator in collection :
    # loop instructions
```

- Iterator is a local variable only valid in the scope of the loop
- The iterator takes all the values in order from the collection



```
# Import modules and namespaces to add references to the code
import clr
clr.AddReference('System.Windows.Forms')
from System.Windows.Forms import MessageBox


output = 'Countdown\n\n'

numbers = sorted(range(10), reverse=True)
for iterator in numbers:
    output += str(iterator) + '\n'

MessageBox.Show(output)

output = 'With Enumeration\n\n'
for index, value in enumerate(numbers):
    output += '{0}:\t{1}\n'.format(index, value)

MessageBox.Show(output)
```