AUTODESK **UNIVERSITY** 2015

CP10574

Customizing Fusion 360 Using its Programming Interface

Brian Ekins
Autodesk

Learning Objectives

- Learn how to create Fusion 360 programs using Python
- Discover the core principles of the Fusion 360 API
- Learn how to use the documentation to understand the Fusion 360 object model
- Learn how to create custom Fusion commands

Description

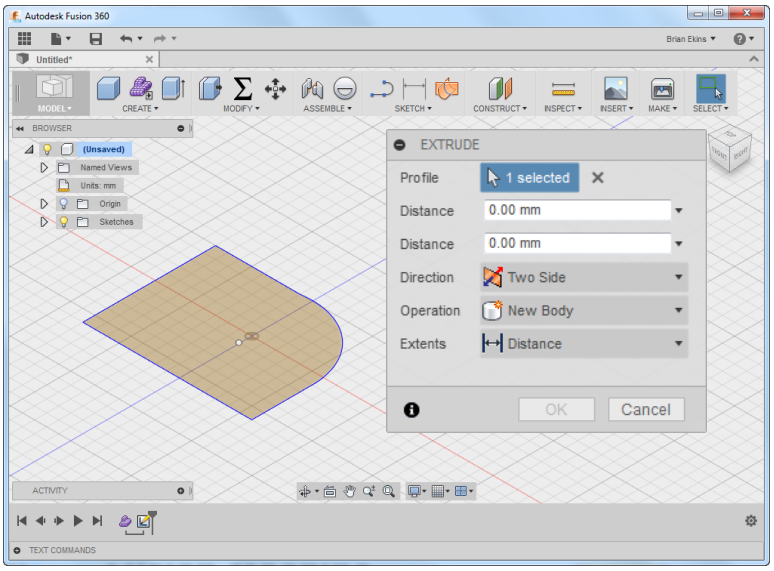
By writing programs using Fusion 360 software's Application Programming Interface (API), you can customize Fusion 360 software to more efficiently accomplish the tasks you need it to do. You can also create new applications that can be provided to Fusion 360 software users through the Autodesk Exchange Store. This class will begin with the basics by looking at how you can write a program in either JavaScript API or Python software, but then will focus on using Python by looking at the basic syntax and statements you'll need in most programs and discovering how to debug your Python programs. The class then will dive into Fusion 360 software's programming interface where its core principles are described and demonstrated. Finally, we'll discuss the more advanced topic of creating custom commands.

Your AU Expert

Brian Ekins *began working in the CAD industry in the 1980s and has worked in various positions, including CAD administrator, applications engineer, instructor, CAD API designer, and consultant. He has worked for Autodesk, Inc., since 1998. He is the designer of the programming interfaces for Inventor and Fusion 360, and he also supports developers and customers in their use of these technologies. Brian also enjoys designing real-world things and writing programs for fun at home.*

Fusion's Programming Interface

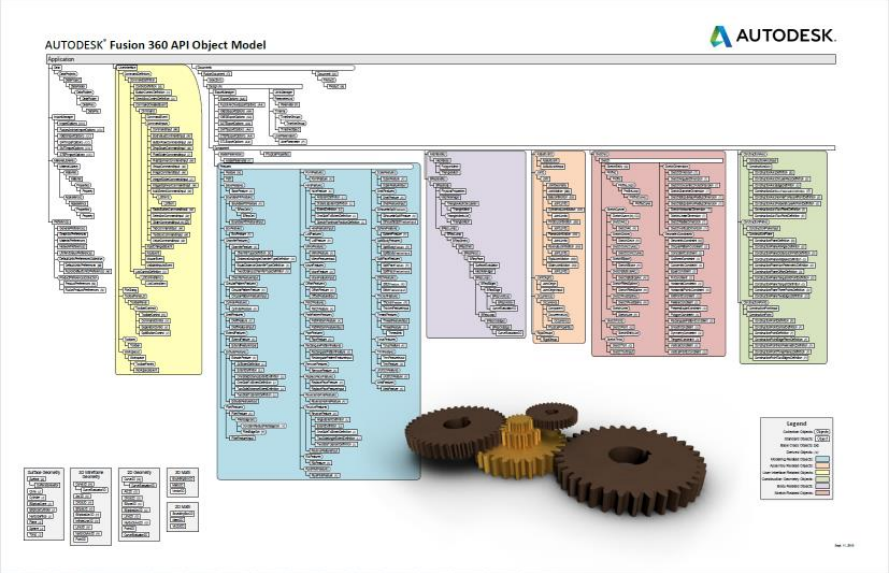
As a user of Fusion you drive Fusion through its "user" interface. Within Fusion is an engine that does the work when you create or modify a design. When you run a command in the user interface, the command collects information from the user and then performs the desired action. For example, the Extrude command guides the user through selecting the profiles, specifying the operation, and the various extent options. Once the required input has been specified and the "OK" button is clicked, the command calls an internal "request", passing it the information it has gathered. The request does the actual work to create the extrusion geometry.



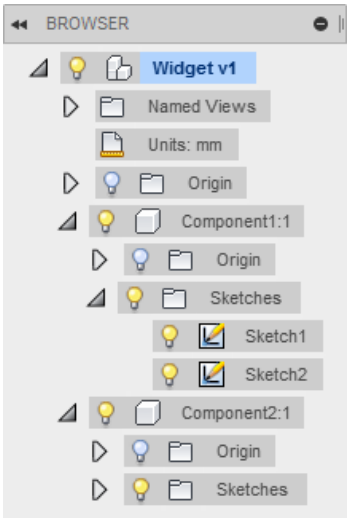
The Fusion "programming" interface is similar to the user interface in that it also collects input and then calls the same request to do the actual work. Understanding that the user and programming interfaces have similarities will help in understanding and using the programming interface.

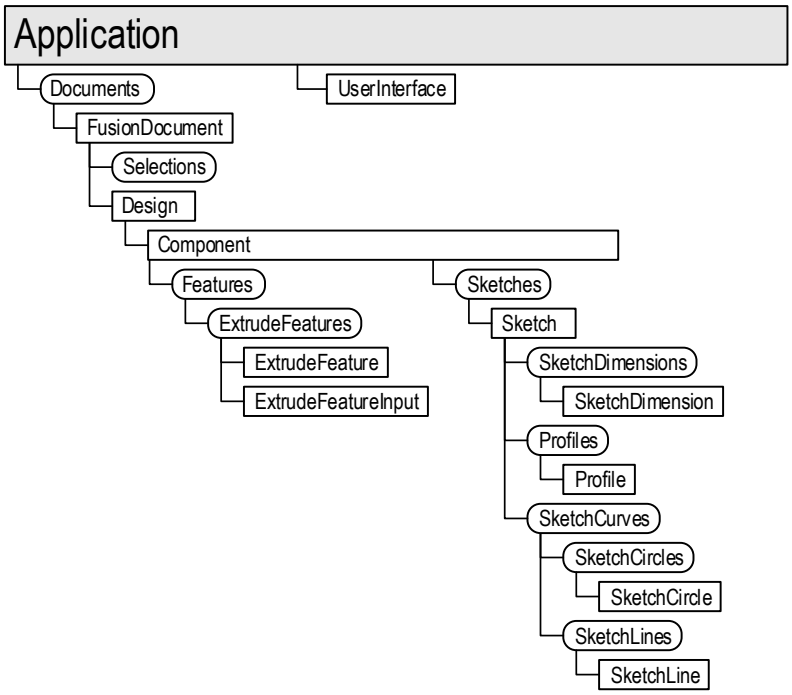
Fusion's Programming Object Model

In the user interface, you access Fusion's functionality through commands and interact with the results through the graphics, browser, and timeline. The API provides access to this same functionality through a set of programming *objects*. These objects are accessed through something known as the *Object Model*. The entire Fusion object model is shown to the right. As of the November 2015 update there are 502 different Fusion programming objects.



The size of the object model chart above can be intimidating at first but you don't need to understand the whole thing in order to start programming Fusion. Instead, you just need to understand the small set of objects related to whatever task you want to do. To better understand how this works a small portion of the object model is shown below. The structure of the object model defines ownership. As a Fusion user, most of these relationships wiltl be fairly obvious if you take the time to think about them. For example, what owns a sketch line? The line is owned by the sketch it is in and the sketch is owned by a component, which is owned by a design, which is owned by a document. Many of these same relationships are also represented in the browser. That same ownership is what's reflected in the object model.





The top object is the Application object. This represents Fusion and is the gateway to everything else. Each object has various properties and methods. Properties let you get and set information associated with that object. For example, a Component object has a "name" property and using it you can get and set the name of the component. A method is an action that a particular object can perform. For example, a common method that many objects support is "deleteMe" which will delete that object. Some objects also support events. These allow the object to notify you when a certain action occurs. We'll look at some examples of using events when we look at creating custom commands.

In the object model above there are two types of objects represented; those in square cornered boxes and those in rounded boxes. All of them are objects, but the objects in rounded corner boxes are a special kind of object referred to as a "collection" object. Collection objects are unique to the API and don't have an equivalent in the user interface. Collections provide structure to the object model by providing access to a group of objects of a particular type. For example, the SketchLines collection shown above provides access to all of the lines in a specific sketch. It's also through collections that you create new objects. The SketchLines collection object supports several methods to create new sketch lines, such as the addByTwoPoints and addTwoPointRectangle methods.

To use the object model you need to first get access to the Application object. It supports properties that let you get to the objects it owns. Those objects in turn have properties to get to their children and so on. You can navigate this hierarchy to get to the specific object you need. We'll look at some specific examples that do this after the discussion about languages and the user interface to the API.

Choosing a Programming Language

Fusion supports writing scripts and add-ins using JavaScript, Python, and C++. Here are a few points to consider when choosing a language.

JavaScript

JavaScript is typically thought of as the language that is used to program web pages, which is true, but for Fusion it is used as a general purpose programming language. In fact when using JavaScript with Fusion you can't display any associated HTML.

Because of what JavaScript was designed for is has some limitations. The primary limitation being that it doesn't provide support to access the file system or system resources. For example, you can't read or write a local text file. This is by design because you don't want to browse to a web page and have it read all of the files on your system. But when writing programs to interact with a CAD system you often need to be able to read and write files. To work around this limitation the Fusion API for JavaScript supports some basic file system functions.

In the JavaScript implementation for Fusion, your JavaScript program runs in an invisible browser outside of Fusion. Because the script is running out-of-process there is an overhead for the browser to communicate with Fusion. Because of this, most JavaScript programs will run much slower than Python or C++.

One other limitation is that not all of the Fusion events are supported in JavaScript.

A reason for using JavaScript is if it's a language you are already familiar and comfortable with. Another reason is if there is an existing JavaScript library that you need to use. In most cases, JavaScript is not the best choice when programming Fusion Python is usually a better choice for writing scripts.

Python

Python is a widely used, general-purpose programming language. Because of its general-purpose nature it doesn't have the same limitations as JavaScript but instead has a rich library for accessing the file system and other system resources. It also has libraries for most other typical programming tasks. In addition there are also other libraries you can download and use. Fusion is currently using Python 3.3.5.

Python runs within a Python interpreter that is running in-process to Fusion. This allows Python programs to run faster than the equivalent JavaScript program. I ran a test that made a trivial API call to Fusion so that the overhead of making the call is being measured rather than the time it takes for Fusion to handle the call. In this case Python was 275 times faster than JavaScript. A slightly more complex sample created one hundred sketch lines where Python was 23 time faster. The time for Fusion to handle a call will be the same regardless of which language is making the call, so in this case more time is being spent by Fusion to react to the call.

For someone writing scripts or add-ins for themselves or their company, Python is a very good choice. This is especially true if you're new to programming. Python is the simpler language of the three and will be the easiest to get started. Although any language has a learning curve and that is especially true if you're new to programming. For this class I'm going to focus primarily on Python.

C++

C++ is a general-purpose, powerful programming language. It is not typically thought of as an easy-to-use language. Both JavaScript and Python programs are interpreted and executed at runtime. C++ programs are compiled into machine code and then that is directly run at runtime. Like Python, it also runs in-process to Fusion but because it is already compiled it will be faster than Python so C++ is the fastest of the three languages.

Another advantage to compiling is that the user of your program doesn't need your source code to run the program. With both JavaScript and Python, you're delivering your source code which is then interpreted at runtime. For professional applications this typically isn't acceptable so C++ is a good solution.

Writing programs in C++ takes more code and is typically more complex. It's primarily attractive for professional developers that need to protect their code, are already familiar with C++, or need the performance benefit of C++.

A Single API

Even though there are three languages they all use the same API. There are some slight differences in how the API is exposed for each of the languages because of differences in the languages themselves but regardless of the language used the API is much more the same than it is different. Below is some JavaScript code that gets the Fusion Application object and then traverses the object model to get the objects needed to create a new sketch and draw a circle with a diameter dimension controlling its diameter.

JavaScript

```
var app = adsk.core.Application.get();
var design = app.activeProduct;

// Get the root component of the active design.
var rootComp = design.rootComponent;

// Create a new sketch on the xy plane.
var sketches = rootComp.sketches;
var xyPlane = rootComp.xYConstructionPlane;
var sketch = sketches.add(xyPlane);

// Draw a circle.
var circles = sketch.sketchCurves.sketchCircles;
var circle1 = circles.addByCenterRadius(adsk.core.Point3D.create(0, 0, 0), 2);

// Constrain the circle size.
sketch.sketchDimensions.addDiameterDimension(circle1, adsk.core.Point3D.create(3, 3, 0));
```

Below is the equivalent Python code. If you compare the two programs, you'll find very few differences. Primarily just syntax differences because JavaScript requires using the var statement for new variables, uses a different way to define a comment, and requires a semi-colon at the end of each line.

Python

```
app = adsk.core.Application.get()
design = app.activeProduct

# Get the root component of the active design.
rootComp = design.rootComponent

# Create a new sketch on the xy plane.
sketches = rootComp.sketches
xyPlane = rootComp.xYConstructionPlane
sketch = sketches.add(xyPlane)

# Draw a circle.
circles = sketch.sketchCurves.sketchCircles
circle1 = circles.addByCenterRadius(adsk.core.Point3D.create(0, 0, 0), 2)

# Constrain the circle size.
sketch.sketchDimensions.addDiameterDimension(circle1, adsk.core.Point3D.create(3, 3, 0))
```

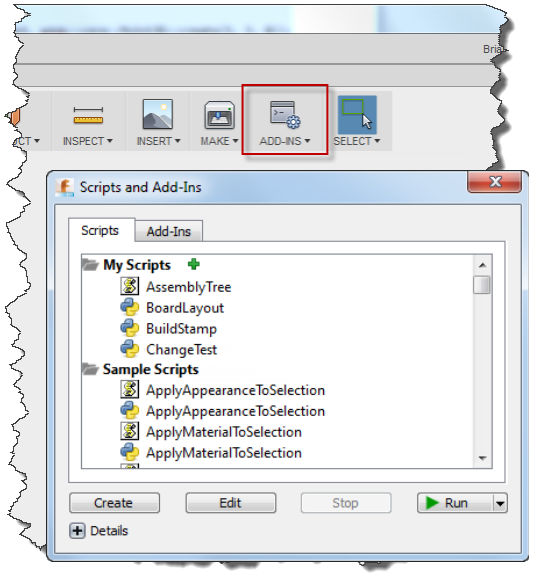
And for comparison, here's the equivalent C++ code. You can see that the code is more verbose but you can also see that it's doing the same thing, using the same objects, and calling the same methods and properties as the other languages.

C++

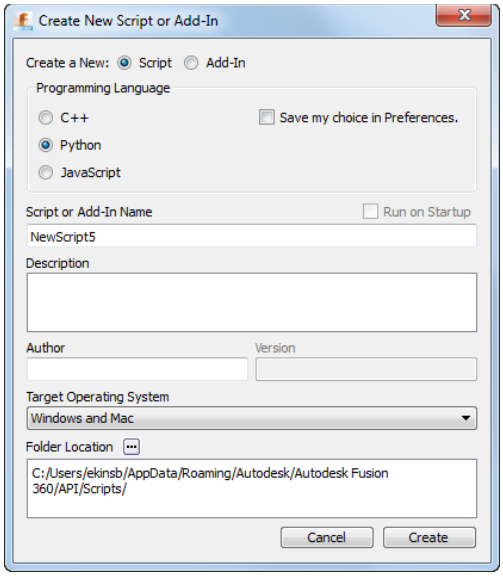
```
app = Application::get();
Ptr<Design> design = app->activeProduct();

// Get the root component of the active design.
Ptr<Component> rootComp = design->rootComponent();

// Create a new sketch on the xy plane.
Ptr<Sketches> sketches = rootComp->sketches();
Ptr<ConstructionPlane> xyPlane = rootComp->xYConstructionPlane();
Ptr<Sketch> sketch = sketches->add(xyPlane);

// Draw a circle.
Ptr<SketchCircles> circles = sketch->sketchCurves()->sketchCircles();
Ptr<SketchCircle> circle1 = circles->addByCenterRadius(adsk::core::Point3D::create(0, 0, 0), 2);

// Constrain the circle size.
sketch->sketchDimensions()->addDiameterDimension(circle1, adsk::core::Point3D::create(3, 3, 0));
```

Creating a Program

No matter which language you choose, the heart of the API in Fusion's user interface is the "Scripts and Add-Ins" command which displays the "Scripts and Add-Ins" dialog, as shown below. This dialog has two tabs, one for Scripts and one for Add-Ins. Each tab has a list of programs that are divided into two categories. The first category contains the scripts or add-ins that you've written or installed and the second category contains sample scripts or add-ins that are delivered with Fusion as programming samples.



New scripts or add-ins are created using the "Create" button in the lower-left corner of the "Scripts and Add-Ins" dialog. When clicked, the dialog shown below is displayed. Using this dialog you can choose whether to create a script or an add-in and in which programming language. For this paper I'll be focusing on Python scripts, which is what is selected in the example below. You should change the name to a name that makes sense and you can optionally enter a description and author.



Using Python

Once you've created a script you can also begin editing it from the "Scripts and Add-Ins" dialog by selecting the script in the list and clicking the "Edit" button. For Python this will open a development environment called "Spyder" which is an open source Integrated Development Environment (IDE) that's delivered with Fusion. There are a series of videos that teach Python using the Spyder IDE on YouTube by Professor George Easton that can be good place to start to learn both Spyder and Python. There is also a good tutorial on the official Python website at https://docs.python.org/3.3/tutorial/. Both of those teach the general use of Python but do not discuss Fusion 360 at all. At the end of this document I've included a few Python basics that I would have found helpful when I first started using Python.

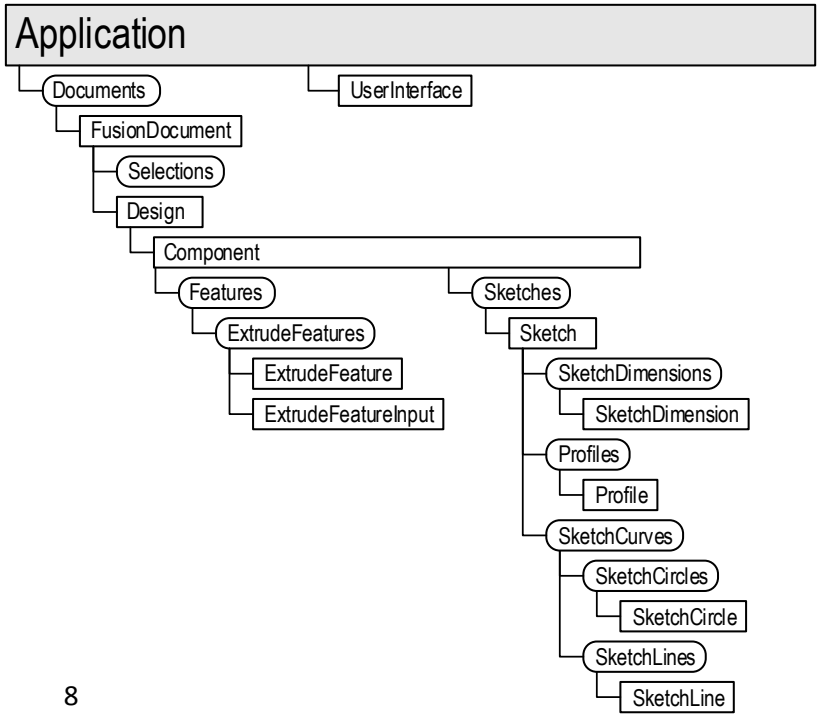
When you create a new Python script you'll end up with code similar to the following.

```
#Author-Brian Ekins
#Description-Sample Script

import adsk.core, adsk.fusion, traceback

def run(context):
    ui = None
    try:
        app = adsk.core.Application.get()
        ui  = app.userInterface

        ui.messageBox('Hello script')
    except:
        if ui:
            ui.messageBox('Failed:\n{}'.format(traceback.format_exc()))
```

The part of the program that does the work is the following three lines.

```
1)        app = adsk.core.Application.get()
2)        ui  = app.userInterface

3)        ui.messageBox('Hello script')
```

Let's go back to the portion of the object model we looked at earlier to see how this code uses the object model to display a message box. The first thing the code above does is use the get function of the Application object. The program now has the Application object and is connected to the object model. In line 2 it calls the userInterface property of the Application object which returns a UserInterface object and assigns it to the variable named "ui". And line 3 calls the messageBox method of the UserInterface object to display a message box with the message "Hello script".



Now let's look at a script that draws a circle in a sketch and extrudes it and dissect the steps.

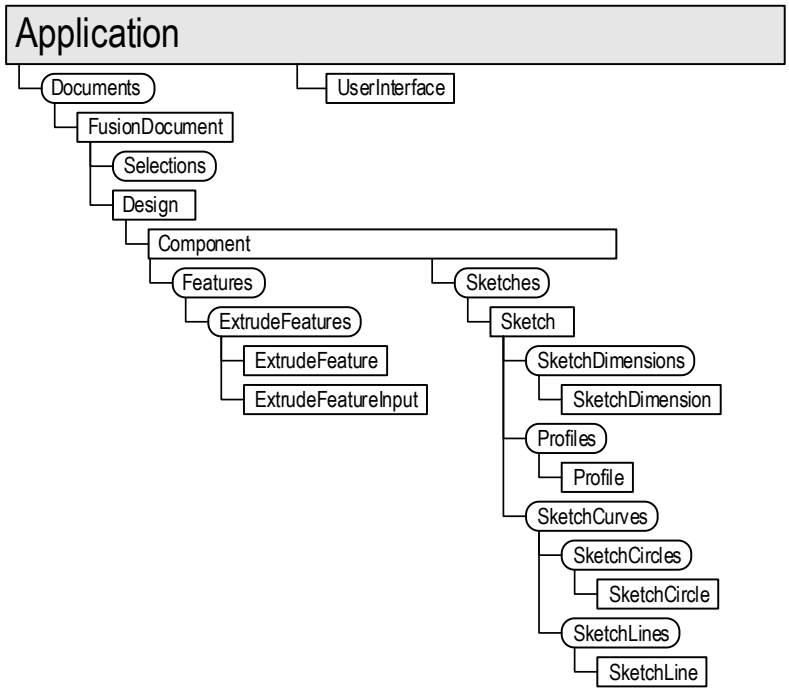
```
1)  app = adsk.core.Application.get()

2)  design = app.activeProduct

    # Get the root component of the active design.
3)  rootComp = design.rootComponent

    # Create a new sketch on the xy plane.
4)  sketches = rootComp.sketches
5)  xyPlane = rootComp.xYConstructionPlane
6)  sketch = sketches.add(xyPlane)

    # Draw a circle.
7)  circles = sketch.sketchCurves.sketchCircles
8)  circle1 = circles.addByCenterRadius(adsk.core.Point3D.create(0, 0, 0), 2)

    # Get the profile defined by the circle.
9)  prof = sketch.profiles.item(0)

    # Create an extrusion input to be able to define the input needed for an extrusion
    # while specifying the profile and that a new component is to be created.
10) extrudes = rootComp.features.extrudeFeatures
11) extInput = extrudes.createInput(prof, adsk.fusion.FeatureOperations.NewComponentFeatureOperation)

    # Define that the extent is a distance extent of 5 cm.
12) extInput.setDistanceExtent(False, adsk.core.ValueInput.createByReal(5))

    # Create the extrusion.
13) ext = extrudes.add(extInput)
```

Line1 gets the Application object and Line 2 takes a shortcut using the ActiveProduct property of the Application object to go directly from the Application object to the Design object currently being edited.

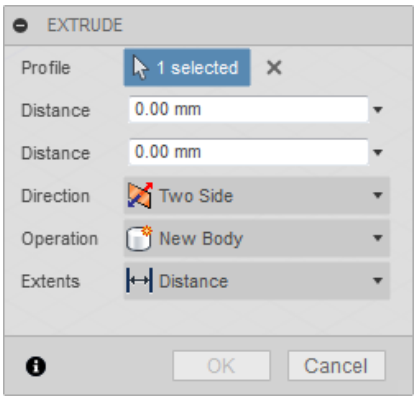
Line 3 calls the rootComponent property of the Design object to get the top-level component object. This is the component represented as the top node in the browser.

Lines 4-6 gets the Sketches collection object from the root component, gets the x-y construction plane from the root component, and finally uses the add method of the Sketches object to create a new sketch.

Lines 7-9 use the Circles collection to create a new circle and then gets the profile defined by the circle. The profile is what you select in the user interface when creating an extrusion.



Lines 10-12 create an ExtrudeInput object and define some of its settings. The ExtrudeInput is the equivalent to the Extrude command dialog, shown to the right. It's used to collect all of the information needed to create an extrusion.



Line 13 calls the add method of the ExtrudeFeatures collection and passes it the ExtrudeInput object. The add method calls the request inside Fusion to create the actual extrude feature.

An important thing to notice is that the program is essentially going through the same steps that you would interactively, it's just doing them through code rather than the user-interface.

Code Hints

Many development environments support the ability to help you write your program by giving you hints about valid options. This can be very beneficial because it helps you to understand what methods, properties, and events, are available on an object without going to the documentation. It also shows you the arguments for the function you're calling. All of these speeds up development by making this information quickly available as you're writing your program and also helps to eliminate mistakes by making sure names are spelled correctly.

Although Python is not a strongly typed language the IDE still attempts to figure out the type that a variable represents so it can show the appropriate code hints for that object. For example, in the picture to the right, the IDE knows that the adsk.core.Application.get function returns an Application object so when I type the period after app, it shows the list of methods and properties supported by the Application object. The same is true if I use the ui variable because it knows it's referencing a UserInterface object.

```
app = adsk.core.Application.get()
ui  = app.userInterface

app.
```

```
activeDocument
activeEditObject
activeProduct
activeViewport
cast
classType
data
documentOpened
documentOpening
documents
favoriteAppearances
```

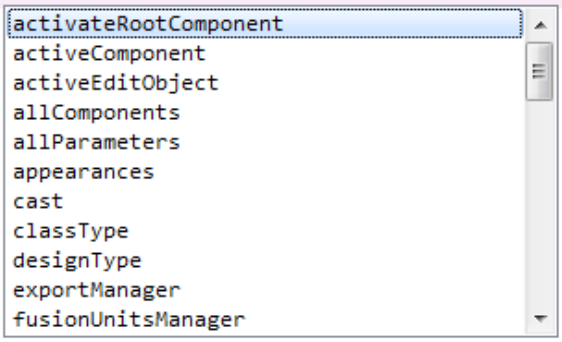
The ability for the IDE to correctly handle code hints falls down when properties are typed to return a base class but actually returns one of the derived classes. A common example of this is the use of the activeProduct property of the Application object. This property is typed to return a Product object, but will always return a more specific object that is derived from Product. For example, when the user is modeling it will return a Design object. The picture to the right shows the code hints given, which are only the methods and properties supported by the Product object but it's most likely that you'll want to access the methods and properties that are supported by the Design object. You can still write the code without using code hints and it will run just the same but having code hints can greatly improve your ability to use the API by quickly showing you what methods and properties an object supports and the arguments that a method requires.

```
des = app.activeProduct
des.
```

```
cast
classType
isValid
objectType
parentDocument
productType
unitsManager
workspaces
__init__
_get_isValid
_get_objectType
```

Although it's not needed for the program to run, you can cast variables to a specific type so the IDE knows what that variable represents and can then show the appropriate code hints. You do this in Python by using the static "cast" function. This is used to the right to cast the variable “des” to be a Design object and you can see that the code hints are now showing all of the methods and properties supported by the Design object. The call of the cast function will return “None” in the case where the active product is not a design.

```
des = adsk.fusion.Design.cast(app.activeProduct)
des.
```



Another common example of where the IDE is unable to show code hints is when using the API to perform a selection. For example, when using the UserInterface.activeSelections property, the actual object being selected is returned by the Selection.entity property which is typed to return a “Base” object, which is what all Fusion objects are derived from and not very useful for code hints. If you know the type of object that will be selected you can use the cast function to let the IDE know what type the variable is, as shown below.

```
sels = app.userInterface.activeSelections

# Cast the selection to an edge.
edge = adsk.fusion.BRepEdge.cast(sels[0].entity)
if not edge:
    ui.messageBox('An edge was not selected.')
    return

# Now there will be good code hints for the variable "edge".
```

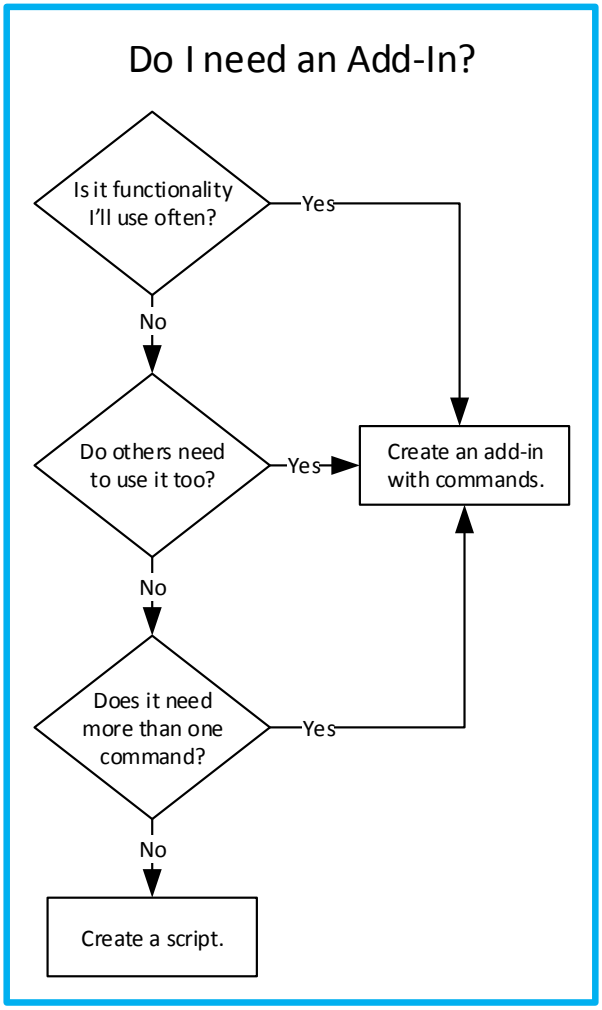
Script or Add-In?

When using the “Scripts and Add-Ins” command to create a new program, you have the choice of creating a script or an add-in. They both have full access to the API and both of them support the creation of commands. The main difference is how they’re run and their lifetime. A script is run by using the “Scripts and Add-Ins” command, choosing a script from the list, and clicking the “Run” button. The script does whatever it’s supposed to do and then terminates.

An add-in is run by Fusion when Fusion starts up. When the add-in is initially loaded it can create buttons and add them to the Fusion UI. Clicking one of these buttons invokes the custom command that’s also been defined within the add-in. The custom command runs and is done, but the add-in continues to run so that you can continue to execute its commands.

The flow chart to the right goes through some decisions that can influence whether you should create a script or an add-in.

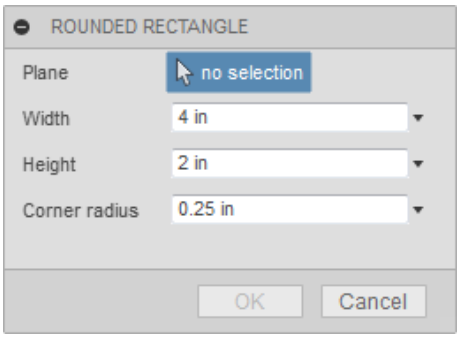
Because scripts are slightly easier to debug, I’ll often start by creating a script to get the basic program working and then convert it to an add-in later so it’s not a critical decision since you can always change your mind.



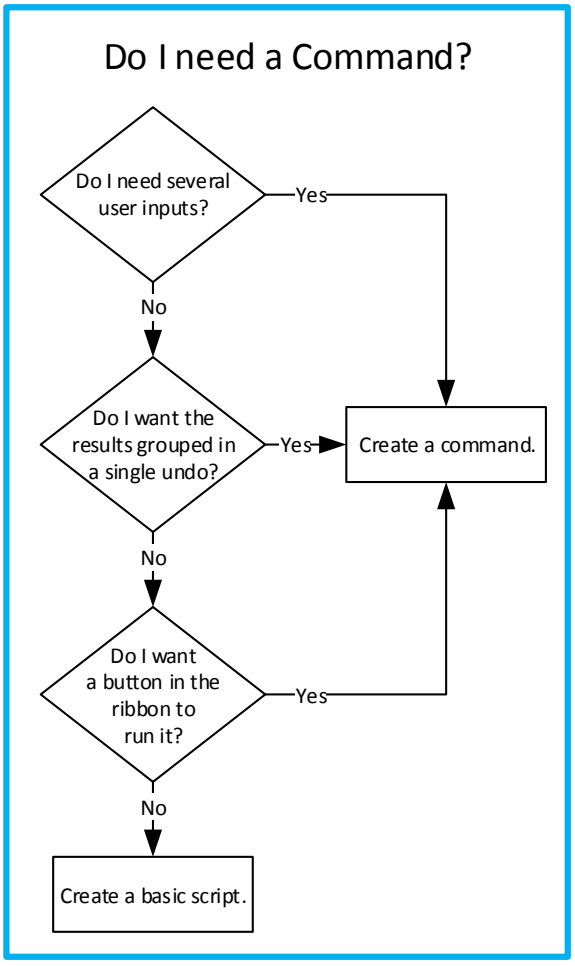
Creating a Command

Fusion has a well-defined concept of what a command is and the API supports the ability to create a true Fusion command. Let's look at a practical example and the differences that a command provides over a simple script. In this example we want to be able to select a face or construction plane and create a rounded corner rectangle where in addition to the plane, the user also specifies the rectangle width, height, and the radius of the corners.

If you write this without using a command you can use some methods supported by the UserInterface object to get the required input from the user. For example, you can use the selectEntity method to have them pick the plane and then use the inputBox method three times to get the three different size values. Once you have the inputs you can then use the API functionality like we've already seen to create a sketch and draw geometry. Each API call that makes a change to the Fusion data is transacted, which means you can undo that operation. In this case, running the command will make 9 changes as it creates 9 new entities (the sketch, four lines, and four arcs). To undo this work the user will need to run the "Undo" command 9 times.



If you write this as a command you can create a dialog like the one shown to the left to get the user input. When the user clicks the button to run a command, Fusion creates a new Command object and calls the commandCreated event associated with the button. In reaction to the creation of a command, you define the different inputs you need in the dialog. In this case there is one selection input and three value inputs. As the user interacts with the dialog, Fusion fires additional events so you can do things like displaying a preview of the results, changing which inputs are displayed in the dialog, validating the inputs, and doing custom filtering on the selections. In addition to having a command dialog to gather the inputs, the result of the command is a single transaction which can be undone with a single undo.



The flowchart to the right can help with the decision of whether to use a command or not. Something not shown in the flowchart but that you should consider is that creating a command is slightly more complicated.

Commands can be used by both scripts and add-ins but they were designed for and are typically used by add-ins.

On page 15 is a flowchart that goes through the various decisions and work needed to create a command.

Events

In order to create a command you'll need to use events. Everything that happens within a command is in response to an event; from the initial execution, the interaction while the command is running, to creating the final results. Conceptually, an event is when Fusion tells your program that something is about to happen or has happened inside Fusion. Technically, it is a function in your program known as an "event handler" that Fusion calls when something happens inside Fusion. Because of differences in the programming languages, events are implemented differently for each language.

No matter which language you're using, there are two basic steps in using events. First, you create the handler function for the event. This is the function that Fusion will call when a specific action occurs in Fusion. This is where your code is that reacts to the event. The second thing is that you need to connect the event handler to the event. None of this is especially hard but there are little details that need to be right for everything to work.

The Fusion API help provides code for all of the events that you can copy and paste that significantly simplifies programming an event. For example, an event that's required when implementing a command is the commandCreated event of the CommandDefinition object. The help topic for this event is shown below.

CommandDefinition.commandCreated Event

LIKE (0) SHARE

Parent Object: CommandDefinition

Description

This event is fired when the associated control is manipulated by the user. A new Command object is created and passed back through this event which you can then use to interact with the user to get any input the command requires.

Syntax

Python | C++ | JavaScript

-------- Global variables ---------

```
# Global variable used to maintain a reference to all event handlers.
handlers = []
```

-------- Event handler class definition ---------

```
# Event handler for commandCreated event.
class MyCommandCreatedHandler(adsk.core.CommandCreatedEventHandler):
    def __init__(self):
        super().__init__()
    def notify(self, args):
        # Code to react to the event.
        ui.messageBox('In MyCommandCreatedHandler event handler.')
```

-------- Connect the handler to the event. ---------

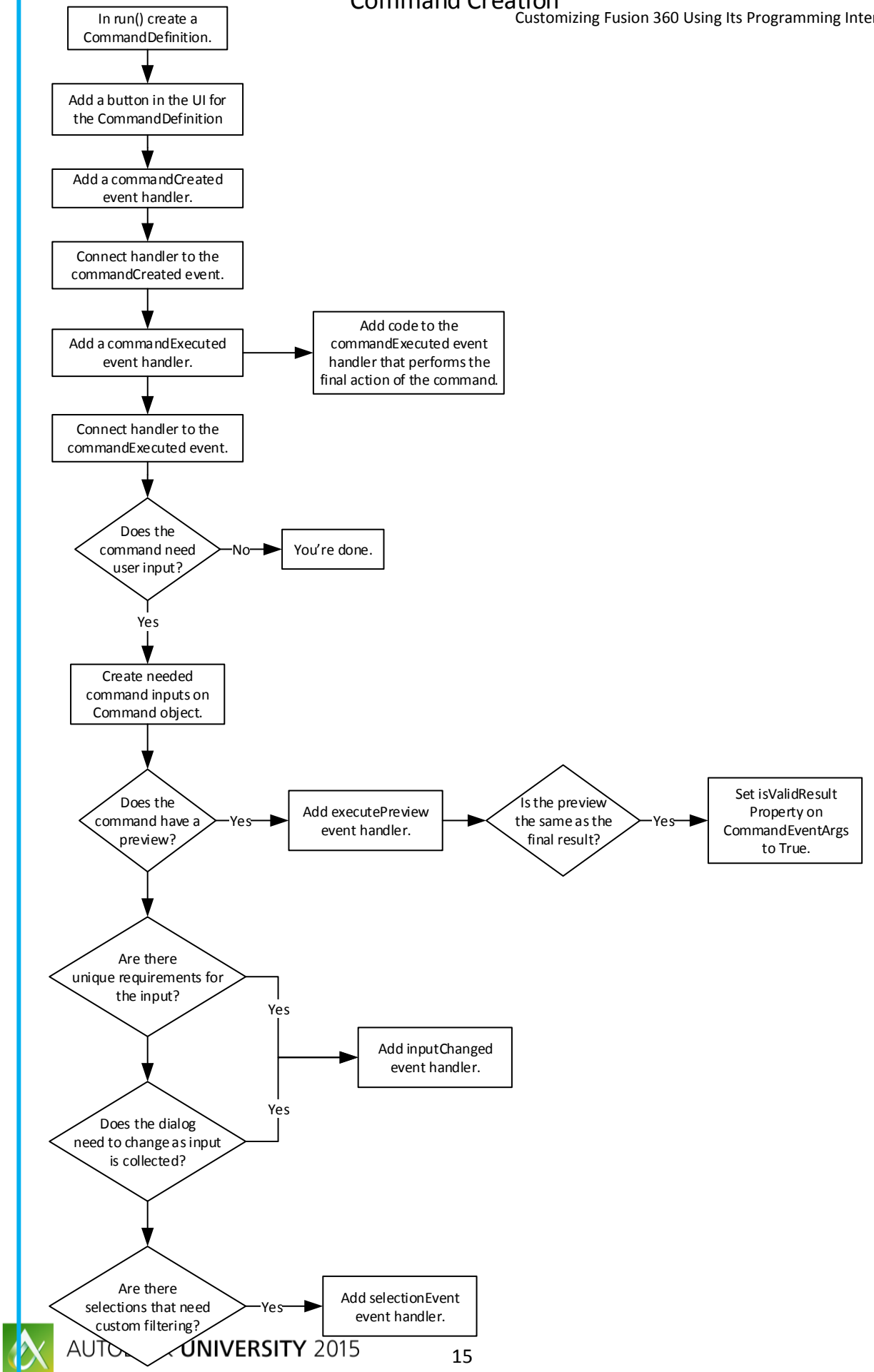
```
# "commandDefinition_var" is a variable referencing a CommandDefinition object.
onCommandCreated = MyCommandCreatedHandler()
commandDefinition_var.commandCreated.add(onCommandCreated)
handlers.append(onCommandCreated)
```

The “Syntax” portion of the help contains the code for the handler, which in the case of Python is implemented as a class. It also contains the code that you’ll use to connect the event up to the handler. The declaration of the “handlers” variable is unique to Python and is shared by all event handlers and is used to keep them in scope so they’ll continue to work. The code below demonstrates the use of copy-paste from the help documentation above to implement the command created event.

```
import adsk.core, adsk.fusion, traceback

# Global variable used to maintain a reference to all event handlers.
handlers = []

# Event handler for commandCreated event.
class MyCommandCreatedHandler(adsk.core.CommandCreatedEventHandler):
    def __init__(self):
        super().__init__()
    def notify(self, args):
        # Code to react to the event.
        app = adsk.core.Application.get()
        ui  = app.userInterface
        ui.messageBox('In MyCommandCreatedHandler event handler.')

def run(context):
    try:
        app = adsk.core.Application.get()
        ui  = app.userInterface

        # Create a command definition.
        cmdDef = ui.commandDefinitions.addButtonDefinition('myCommand', 'Sample Command', 'Sample Command')

        # Connect to the event handler.
        onCommandCreated = MyCommandCreatedHandler()
        cmdDef.commandCreated.add(onCommandCreated)
        handlers.append(onCommandCreated)

        # Add a button for the command in the ADD-INS dropdown in the MODEL workspace.
        addInPanel = ui.allToolbarPanels.itemById('SolidScriptsAddinsPanel')
        addInPanel.controls.addCommand(cmdDef)
    except:
        if ui:
            ui.messageBox('Failed:\n{}'.format(traceback.format_exc()))

def stop(context):
    ui = None
    try:
        app = adsk.core.Application.get()
        ui  = app.userInterface

        # Remove the CommandDefinition and button.
        addInPanel = ui.allToolbarPanels.itemById('SolidScriptsAddinsPanel')
        cmdButton = addInPanel.controls.itemById('myCommand')
        if cmdButton:
            cmdButton.deleteMe()

        cmdDef = ui.commandDefinitions.itemById('myCommand')
        if cmdDef:
            cmdDef.deleteMe()
    except:
        if ui:
            ui.messageBox('Failed:\n{}'.format(traceback.format_exc()))
```

The chart below goes through the decisions and actions that need to be made when implementing a command, which are also demonstrated in the code above.

Command Creation



Python Basics

Python, like any programming language you're new to, can take a little while to get used to. Here are some of the things I found useful when getting started.

Declaring Variables

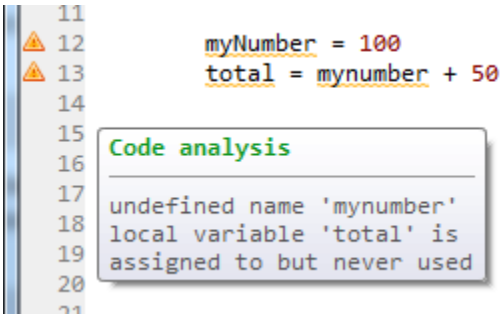
Python has variables but they aren't explicitly declared or typed. They are created on-the-fly as-needed and can reference any type. For example, the statement below creates the variable named "myNumber" and assigns it the number 100 and the next statement creates the variable named "total" and assigns it the result of the equation which is 150.

```
myNumber = 100
total = myNumber + 50
```

Variable names are case sensitive. If you have the code below, (notice that the "n" for number in the second line is lower case), you'll get an error when you run the program complaining that the "global name 'mynumber' is not defined".

```
myNumber = 100
total = mynumber + 50
```

Spyder also helps with these kinds of problems as you're writing your code. Watch for the little warning or error symbols to left of your code. If you move the cursor over the symbol it will display a message indicating what the problem is, as shown below. In this case there are two issues that are being reported for line 13. The first is that "mynumber" is undefined and the second is that the variable "total" is being declared but is not used anywhere in the program. The second one is just a warning but can also be a clue that you have a misspelled name somewhere.



Variable Types

As I said earlier, when a variable is created you don't specify the type of data it can reference. Python variables can reference any type of date. Even though variables themselves don't have a type, the data it references does have a specific type. Python has a simplified set of types compared to many other languages. Here are the common types you're most likely to use for a Fusion program.

Numeric Types – There are two types of numbers that you'll be using; integer and float. Integers are whole numbers and have unlimited precision. In practice you usually don't need to worry about the different number types. Python will automatically convert from one type to another when you perform an arithmetic operation with two different types or when comparing two different types.

Boolean – A variable can be set to True or False. Technically, Booleans are implemented as integers, but in practice I believe it's useful to think of it as a distinct type since it's used so differently than other numbers.

String – Any string.

List – A list is very similar to an array in other languages. It has the same flexibility as other variables in that it can hold data of any type and can even hold mixed types. For example a list can contain numbers, strings, and even other lists, all at the same time.

Tuple – A tuple is similar to a list except its contents can't be changed. It's used by Fusion when passing back the results of a method call.

None – When a variable is not referencing any data it will return "None" and you can also set variables to "None" to clear their current reference.

I mentioned above that Python does automatic conversions between numeric types. This isn't true for other types and if you need to combine them for some reason you'll need to convert them. For example, a common case is when you have a number that you want to display, which means converting it to a string. The following combines a string with a number and assigns it to another string by using the "str" function to convert the number to a string.

```
result = 'The result is: ' + str(number1):
```

You can also convert a string to a number using the float or int function depending on the type of number the string represents as shown below.

```
myNumber = 100
strNum = '50'
newNum = myNumber + float(strNum)
newNum = myNumber + int(strNum)
```

Program Structure

Python is very unique in how you define the structure of your program. For example, specifying the block of code that is to be executed within an "if" statement. This is done purely by code indentation. In the example below, if "number1" is equal to "number2", "match" is set to True and then regardless of the previous statements, "number3" is set to 5. The fact that the line "number3 = 5" is a different indent level than the previous line signals the end of if statements code block.

```
if number1 == number2:
    match = True
number3 = 5
```

In this example, "match" is set to True and "number3" is set to 5 only when "number1" equals "number2". In this case the if statement's code block includes both lines because they have the same indent level.

```
if number1 == number2:
    match = True
    number3 = 5
```

The only difference between the two examples is the indentation of the last line. Python is picky about the indentation so you need to be consistent. This means you need to use either spaces (the most common) or tabs but you can't use both. When using spaces, you also need to use the same number of spaces within a code block. In the example above, 4 spaces are being used for each indent level.

Making Comparisons

An “if” statement is the most common method of using a comparison to make a decision.

Below is a simple “if” statement.

```
if number1 == number2:
    match = True
```

And here’s a more complex “if” statement with several comparisons.

```
if number1 == number2:
    match2 = True
elif: number1 == number3:
    match3 = True
elif: number1 == number4:
    match4 = True
else:
    noMatch = True
```

The operators for making comparisons are shown below.

Operation	Meaning
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Not equal
is	Objects are equal
is not	Objects are not equal

Looping

Looping through a list of things is very common and Python supports it in a few ways. The “for” statement is probably the most common method of looping through a list or set of values. For example, if you have a list you can iterate over its contents using a “for” statement like below. A “for” statement can also be used when iterating over collections from the Fusion API.

```
for listItem in myList:
    print('Item: ' + listItem)
```

Another common use of a “for” statement is when iterating a specific number of times. To do that in Python you combine the “for” statement with the “range” function. The example below will run the block of code associated with the “for” statement ten times. The variable “i" will vary from 0 to 9. There are other options with the range function to control the starting value and the step value. This uses the default which starts at 0 and steps by 1.

```
for i in range(10):
    print('i: ' + int(i))
```

Another commonly used looping statement is “while” when you need to loop until a condition is met rather than loop a certain number of times.

```
myVal = 100
stopVal = 79
while myVal != stopVal:
    myVal = myVal – 1
```

Error Handling

When an error occurs in Python, python fires an “assert” and continues to execute the program. Most often when an error occurs you don’t want it to continue running. To do that you can use the “try” statement. When you create a new script, it will already have a “try” statement around the main block of code, as can be seen below. The lines of code between the “try:” and “except:” statements will be executed and if any of them fail, the execution of the program will immediately jump to the lines in the “except:” code block where the code below the prints out details of the exception in a message box.

```
import adsk.core, adsk.fusion, traceback

def run(context):
    ui = None
    try:
        app = adsk.core.Application.get()
        ui  = app.userInterface

        ui.messageBox('Hello script')
    except:
        if ui:
            ui.messageBox('Failed:\n{}'.format(traceback.format_exc()))
```

You should always use “try except” statements around your code. Otherwise it can be frustrating when your code doesn’t work as expected because an error is occurring that you’re not aware of.

Tips When Using Spyder

Below is the Spyder window, which is configurable. The picture below shows how I have it configured on my computer. The code window is to the left and on the right are the “Variable explorer” and “Console” windows. You can see in the “Variable explorer” that Python lists all of the variables that currently exist, along with their type and value. The console window is useful for debugging where you can type in statements. This is particularly useful for more closely examining Fusion objects.

The buttons in the toolbar that I’ve highlighted let you run and step through your code in order to debug it. You can also run and begin a debugging session from the “Scripts and Add-Ins” dialog. For add-ins, you should also being debugging from the “Scripts and Add-Ins” dialog.