

Ignition 7.9 User Manual — Scripting

Generated 2026-10-12 from
<https://www.docs.inductiveautomation.com/docs/7.9/>

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Scripting

What Is Scripting?

Most of the time when we talk about "scripting" in Ignition we are talking about Python scripting, or writing code in the Python language. Python is a general purpose programming language that was developed in the early 90s and has gained significant popularity in the 2000s. We like it because it is extremely readable, elegant, powerful, and easy to learn. As an added bonus, it gracefully interacts with Java, giving programmers an extremely powerful tool when paired with Ignition, which is written in Java.

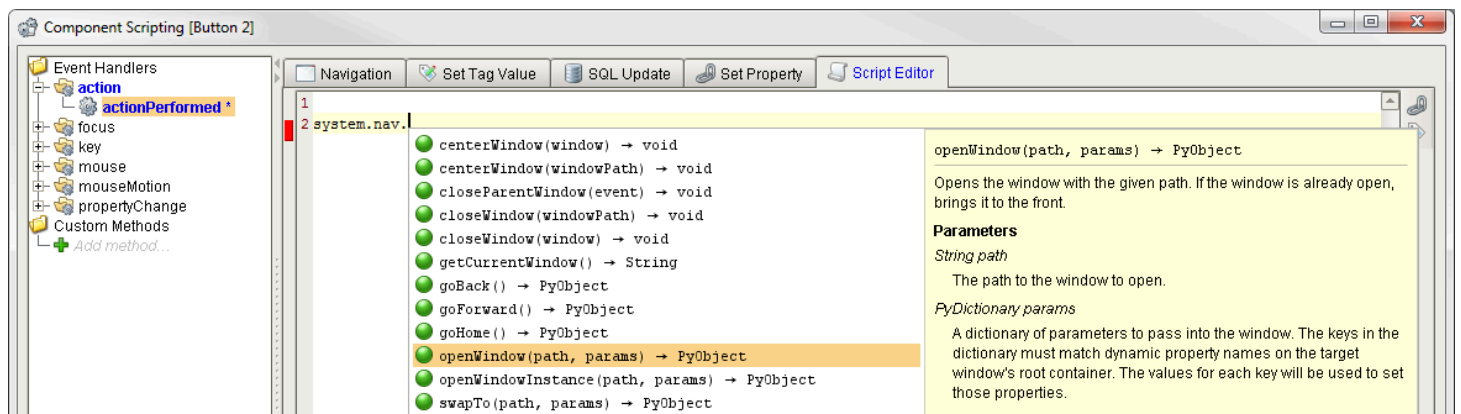
Since Python is such a popular and well-regarded language, there are many high-quality tutorials available on the web. The official [Python tutorial](#), written by the inventor of Python himself, Guido van Rossum, is a great reference. In Ignition, you will be mixing the core Python language with references to other components and a variety of our built-in [system functions](#).

Pseudocode - Interacting With a Component

```
text = event.source.parent.getComponent('My Component').text
if text == "":
    print "You must enter a name"
```

Is It Easy to Use?

Luckily, Python is a simple language to get started with, and using it in an event-driven system takes away a lot of the extra code that normally makes programming time consuming. For those that are already familiar with scripting (and those of you that are learning), we also have a huge list of functions inside Ignition to do some of the common tasks in a single line of code. These [System Functions](#) are available while typing. Just start with **system** and press `ctrl-space` to see a list of available functions. The list will filter itself as you continue typing.



Where Is Scripting Used?

Python is used in many places in Ignition. Each location has its own events that trigger your scripts to run, and add functionality to your projects in different ways.

- **Components** - Add actions to components like buttons, customize the look and feel of charts and tables, and even setup a custom navigation schema.
- **Tags** - Create a script that runs on a Tag change, when an alarm goes active, and more!
- **Reports** - Use scripting to create a customized datasource, or create your own unique action to use with the scheduling system.
- **Alarm Notification** - Create custom rosters using scripting to dynamically change who gets notified with each new alarm event.
- **Client and Gateway** - Add a script that will run when certain events happen, such as when the Client or Gateway starts up, or on certain keystrokes.

Users that are new to Ignition focus mainly on the **Component Binding system**, and for good reason. It's simple, flexible, and generally easy to understand without much of a computer background. However, Ignition has a complete scripting system built into every place you can think of. Using it is not a requirement, but it can add a significant degree of flexibility and customization to your projects. It allows you to create exactly what you need, giving you total control where pre-canned options fall short.

The majority of your scripting will be done in Event Handlers inside of components. This system makes it very easy to get started scripting with little to no experience. With the four **script builders**, basic scripting like Navigation and setting Tag values takes just a few clicks!

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Export Tag Historian to CSV

Obtaining Historian Data

Sometimes, it may be useful to export the values from the Tag Historian. There are many ways that this can be done, but the easiest way is to use the built in `system.tag.queryTagHistory` function. This allows you to specify a list of Tag paths to query history for.

Python - Grabbing History Data

```
# First we start by creating a start and end time that we will be query history for.
endTime = system.date.now()
startTime = system.date.addMinutes(endTime, -30)

# Next we call our queryTagHistory function, using the start and end dates as well as a Tag path.
# The other parameters listed for this function can be altered to fit your need.
data = system.tag.queryTagHistory(paths=["[default]myTag"],
startDate=startTime, endDate=endTime, returnSize=10, aggregationMode="Average",
returnFormat='Wide')
```

This simple script will query Tag History for a Tag called myTag. It will query the last 30 mins of data, return 10 records, each record will be an average of the time slice, and the return format will be wide. For more information on the function parameters and the different options available, see the `queryTagHistory` function page. The data returned is then being stored in a variable `data`, and we can use it later in the script in anyway we want.

Exporting to CSV

Now that we have our data, it is really easy to export it to a CSV file. Ignition has a built-in function to convert a dataset to CSV data called `system.dataset.toCSV`. This CSV data can then be written to an actual file using `system.file.writeFile`.

```
# First we start by creating a start and end time that we will be query history for.
endTime = system.date.now()
startTime = system.date.addMinutes(endTime, -30)

# Next we call our queryTagHistory function, using the start and end dates as well as a Tag path.
# The other parameters listed for this function can be altered to fit your need.
data = system.tag.queryTagHistory(paths=["[default]myTag"],
startDate=startTime, endDate=endTime, returnSize=10, aggregationMode="Average",
returnFormat='Wide')

# Turn that history data into a CSV.
csv = system.dataset.toCSV(data)

# Export that CSV to a specific file path. The r forces it to use the raw path and not require double backslashes.
system.file.writeFile(r"C:\myExports\myExport.csv", csv)
```

History Tag Search and Export

If you have a lot of Tags storing history, it can be difficult to list them all out in the function above. This example makes things easier by specifying a folder of Tags instead of individual Tags. The script first grabs all of the historical Tags from a particular folder and gets the Tag paths using the **system.tag.browseHistoricalTags** function, runs those through the **system.tag.queryTagHistory** function, and then exports the results to a CSV file. This script is great, because it can be run from the **scripting console**, or from a button on the client. However, everything in the script is hardcoded to work with a specific system, though that can be easily changed to fit any system. Below are the list of parts that can be changed with what each of them is used for.

1. `path='histprov:myDB:/drv:myIgnitionGateway:myTagProvider:/tag:myFolderOfTags'`
 - i. The path to the parent folder that we are searching through for history Tags. Each part of the string corresponds to a different part of your system.
 - a. `histprov:myDB:/` - This is the name of your Historical Tag Provider that you are searching in, simply replace **myDB** with the name of your historical Tag Provider.
 - b. `drv:myIgnitionGateway:myTagProvider:/` - The first thing is the name of your Ignition Gateway. Simply replace **myIgnitionGateway** with the name of your Ignition Gateway, which can be found by going to the Gateway Configure Webpage and navigating to **System > Gateway Settings**, and it is the **System Name**. The second thing is the

name of the Realtime Tag Provider where the Tags reside. Simply replace **myTagProvider** with the name of your Tag Provider.

- c. tag:myFolderOfTags - This is the name of the folder you want to search for historical Tags. Simply replace **myFolderOfTags** with the name of a folder in your Tag Browser to search through.

2. `system.tag.queryTagHistory(...)`

- i. This is what will actually query for historical data. While the path parameter should be kept the same, the other parameters can be altered to fit your needs. Currently, the function is querying the last 30 mins of history, has a return size of 10, an aggregation mode of Average, and a return format of wide. Of course, all of this can be changed. See [system.tag.queryTagHistory](#) for more information on the parameters that can be used here.

3. `r"C:\myExports\myExport.csv"`

- i. This is the path that the CSV file will save at. This can be changed to be any location you want. You can even use [system.file.saveFile](#) to first ask the user for a save location and filename, and then passing the path returned from that function into the writeFile function. The r at the beginning of the string is important as it allows us to not have to use double backslashes.

DANGER

This script can be dangerous! It recursively looks through a folder to find all historical Tags, and then uses all of those Tags to look up Tag history. Looking through a folder that has too many history Tags, or querying history for too large a period of time can potentially result in locking up your system. Use caution when using this function, and never search through too large a set of Tags or query too much history.

Python - Recursively Browse for History and Export to CSV

```

# Our browse function that will browse for historical tags.
# By setting this up as a function, it allows us to recursively call it to dig
down through the specified folder.
# Pass in an empty list that we can add historical paths to, and the path to the
top level folder.
def browse(t, path):

    # Loop through the results of the historical tag browse, and append the path
to the empty list.
    for result in system.tag.browseHistoricalTags(path).getResults():
        t.append(result.getPath())

        # If the result is a folder, run it through the browse function as well.
        # This will continue until we are as deep as possible.
        if result.hasChildren():
            browse(t, result.getPath())

# Start with an empty list to store our historical paths in.
historyPaths = []

# Call the browse function, passing in an empty list, and the folder that we want
to browse for historical tags.
# This path is a placeholder. It should be replace with your valid path.
browse(historyPaths,
path='histprov:myDB:/drv:myIgnitionGateway:myTagProvider:/tag:myFolderOfTags')

# Create another empty list to store our tag paths that we will pull out of the
historical paths.
tagPaths = []

# Loop through the list of historical tag paths, split out just the tag path part,
# and push it into our tag path list.
for tag in historyPaths:
    tagPaths.append("[myTagProvider]" + str(tag).split("tag:")[1])

# Now that we have a list of tag paths, we need to grab the historical data from
them.
# Start by creating a start and end time.
endTime = system.date.now()
startTime = system.date.addMinutes(endTime, -30)

# Then we can make our query to tag history, specifying the various parameters.
# The parameters listed for this function can be altered to fit your need.
data = system.tag.queryTagHistory(paths=tagPaths, startDate=startTime,
endDate=endTime, returnSize=10, aggregationMode="Average", returnFormat='Wide')

# Turn that history data into a CSV.
csv = system.dataset.toCSV(data)

```

Export that CSV to a specific file path. The r forces it to use the raw path with backslashes.

```
system.file.writeFile(r"C:\myExports\myExport.csv", csv)
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Basic Python Troubleshooting

When learning how to code in Python, most Ignition users tend to place most of their learning efforts on memorizing syntax or other aspects of the language. While being comfortable with the language is useful, there are plenty of references available: countless books and websites that describe syntax and usage already exist.

In truth, the best thing you can do to make yourself a better programmer is to learn some basic troubleshooting behaviors. While syntax examples are all over the Internet, examples detailing exactly what you want your code to do, to the extent you want it to, will be difficult if not impossible to find.

This section details how to troubleshoot a script in Ignition. You won't walk away from this section having exact answers to specific problems, but rather examples and concepts that you can apply to your own scripts.

Coding Best Practices

The following are some general and helpful best practices that can help minimize the amount of time you spend troubleshooting, as they can help better direct you to a problem.

Look for Errors

When code fails mid-execution, it always generates an error message. Where the message appears depends on where the script executed:

- **Client:** The Client Console will contain any errors generated in the client: press **Ctrl + Shift + F7** to open the console, or using the menubar in the client to go to **Help > Diagnostics** then click on the **Console Tab**. Additionally, a red Error Box should appear with details on the error if a Component Event Handler threw the exception: Extension Functions do not generate the Error Box. If you don't see the error box then it might be minimized, or open in the background (behind the Client).
- **Designer:** The Designer reports errors in a similar manner to Clients: errors appear in the **Designer's Console** (**Ctrl + Shift + c**), Component Event Handlers will generate a red Error

Message. Note that events some events, such as [Client Startup Scripts](#), will not trigger in the Designer, so get in the habit of launching a Client when testing non-component scripting events.

- **Gateway:** Scripts on Gateway-scoped resources (Tags, Alarm Pipelines, SFCs, etc.) will appear on the [Logs page of the Status section on the Gateway](#). Additionally, the wrapper.log file in Ignition's installation directory will have these messages. Here are the default wrapper.log file paths for each operating system:
 - **Windows:** Program Files/Inductive Automation/Ignition/logs
 - **Linux:** /var/log/ignition
 - **Mac OS X:** /Users/UserName/Documents/Ignition-osx-x.x.x/logs

What happens if I don't see any errors?

If there truly isn't an error message somewhere, but your code isn't doing what you expected, then ask yourself the following questions:

1. What is your script supposed to do?
2. What is it actually doing?

Question #2 is harder to answer: if you knew what it was doing, you wouldn't be stuck! The best way to answer this question is by adding `print` statements to your code.

When a script doesn't perform to expectations, it can suggest a problem with the script's workflow. Some of the [pages in this section](#) can offer some suggestions on what to do.

If You Find an Error, Read It!

It is common for users that are new to scripting to see an error message, immediately close it without much thought, and then stare at their code as if the problem will politely make it self known. While looking over your code line-by-line will eventually lead you to the problem, the error messages can provide you with a shortcut to the issue. [Check out some of the pages](#) in this section for more information on reading an error message.

Use Print Statements

When testing your scripts, the print command can help you verify that your code is behaving the way it should. This allows you to reconstruct what your code did when it executed. Don't be afraid to add helpful print statements to your code.

Once your code works as expected, remember to either remove or comment out the print statements, so they don't flood the console during normal use as this makes troubleshooting other issue more difficult.

```
myVar = system.tag.read("folder/tag")

# printing out variables you are going to use in an if-statement later allows you
# to confirm that the values are what you expect them to be.
print "myVar is set to: " + str(myVar)

# Sometimes viewing the data type of the variable can prove helpful.
print "myVar is a: " + type(myVar)

# Should your code have multiple if-statements, adding a print statement before
# and after can show you where the flow of the script went.
print "Starting if-statement"

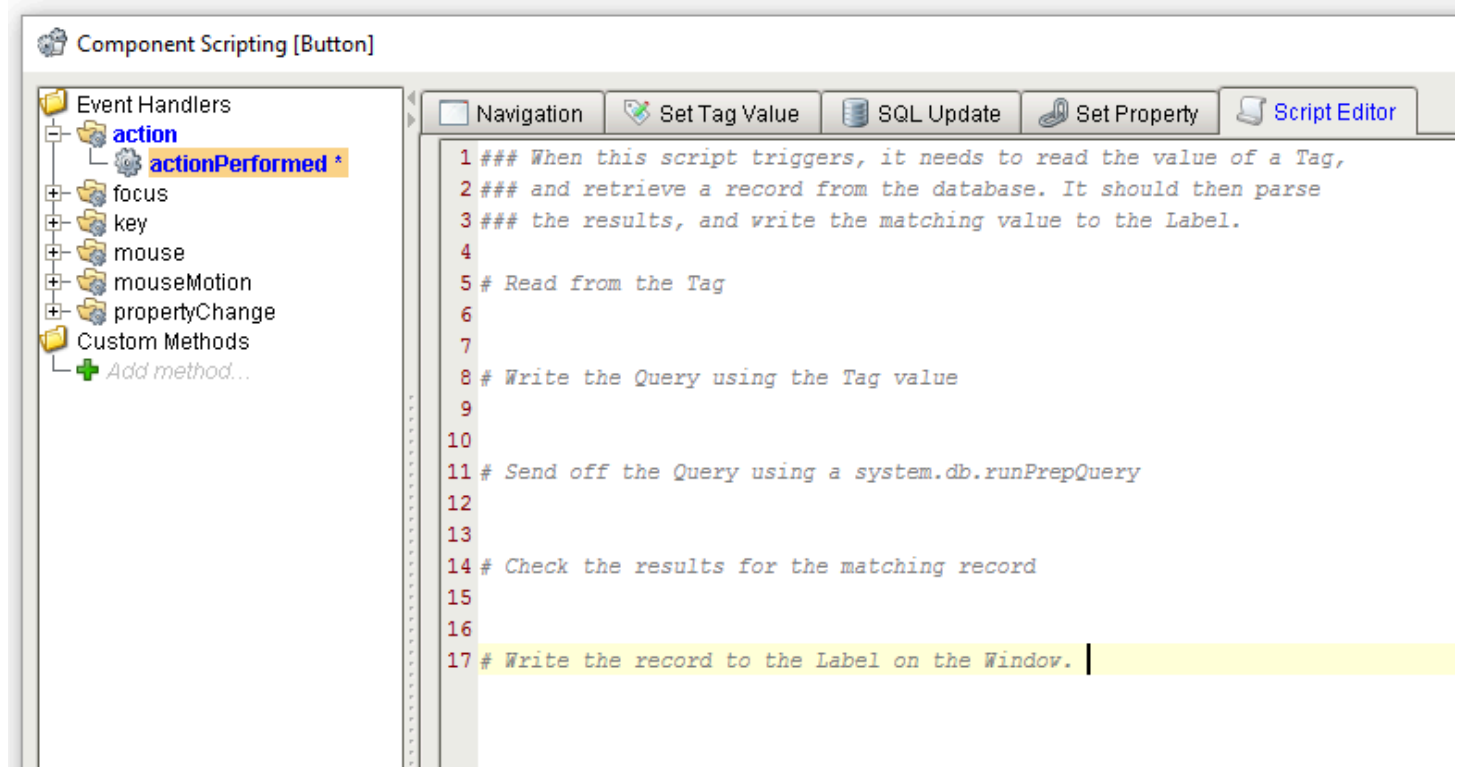
if myVar > 100:

    # If you don't see this print statement, then your if-statement evaluated to
    # False.
    print "Inside if-statement"
    doWork()

# Printing the end of your script doesn't give you any useful troubleshooting
# information, but if you need to trigger the script multiple times, it helps
# delineate each execution.
print "Script Ended"
```

Add Comments

While comments are useful to remind you how your code works, you can also use them to plan your script before you write any code. Break down what you want the code to do into several smaller steps, and then leave comments describing those steps in order. This provides you a chance to review the script's workflow before worrying about syntax. It also provides natural points to stop and test your code, to make sure it is doing what you think it should do.



Test Early, Test Often

Unless the script is very simple, avoid writing the entire script and then testing at the end. You may have missed an important line early on, so now you have to adjust all of the code below that line to make the script run. As mentioned above, add print statements, and run your script to make sure it is doing what you think it should.

Additionally, stopping to test your code provides an excellent opportunity to **save your project**. Get into the habit of saving before you execute any new code.

Avoid Hard-Coding Arguments: Use Variables Instead

Instead of doing this:

Hard Coded Message Box

```
system.gui.messageBox("Hello you. Glad to see you")
```

Try to get into the habit of doing this:

```
message = "Hello you. Glad to see you"  
system.gui.messageBox(message)
```

Simple examples like the above don't make for the best use case, but when you have a large script that references a value multiple times, it is easier to declare the value once in a variable, and then just reference the variable throughout your code. If you need to change the value later, then you can simply change it once where you initialize the variable, and you don't have to search every line of your code looking for the value.

Decide on a Naming Convention

When creating variables, try to adopt a naming convention that comes natural to you, and stick to it. If you consistently write variables using the same conventions, you will be less likely to end up with a typo when referencing that variable later in your code. Remember that code is case-sensitive, so something as simple as forgetting to capitalize a letter will cause an error.

This is especially important if you are working in a group on the same project. It's better to get together with your colleagues and agree upon some naming conventions before you write any code.

Ignition's *system.functions** use the Camel Case (camelCase) naming convention. That is, the first letter of each word is capitalized except for the very first letter. We recommend that you use it for variable names because it is easy to remember to use Camel Case for both functions and variable names instead of for just one.

The Simplest Approach Really Is the Best Approach

When learning how to code, it's not uncommon to run into multiple issues that require you to find a workaround. However, these workarounds can cascade into other issues and make your script more complicated. Consider the following:

"The goal of my script is to access A and then output B... Shoot, B requires C, so I'll add C... Wait, C requires that D exists, so I'll create that... Oh, D needs interfaces E, F, and G, so let's add those in... Hmm. E needs H and I, F needs J and K, while G needs Y, Z and...A **again?!?**"

Take a step back and ask yourself "what is this script doing"? If you can't do that in a sentence or two, you may want to rethink the script. If the scope of the script is too large, then it considerably increases the complexity of the code, which in turn could add a plethora of problems later.

If you keep adding workarounds, but your code is not getting any closer to achieving its end goal, there may be an easier way to accomplish what you're trying to do with a different approach.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

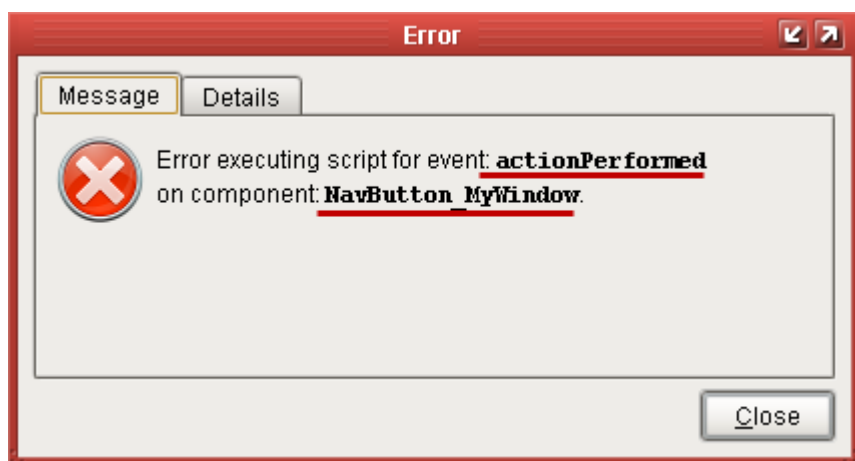
Reading Error Messages

When an error occurs in the execution of a script, as in this example for a button, an Error Message Box will popup. If you have a lot of windows open, the Error Message Box could be hidden behind one of your open windows. To check, hover over the Ignition application icon on your menubar to find the Error Message Box.

Error Message Box Overview

Exceptions usually include a line number. Take note of the number in the Details tab, and start your search for the problem there. Be aware, however, that the line reported may not be the cause of the issue. The actual problem may be higher up in the code due to a faulty initialization, or some other issue. When troubleshooting, always start looking at the line reported, and work your way back towards the top.

When testing a script, you may eventually come across an Error Message Box like the one shown below.

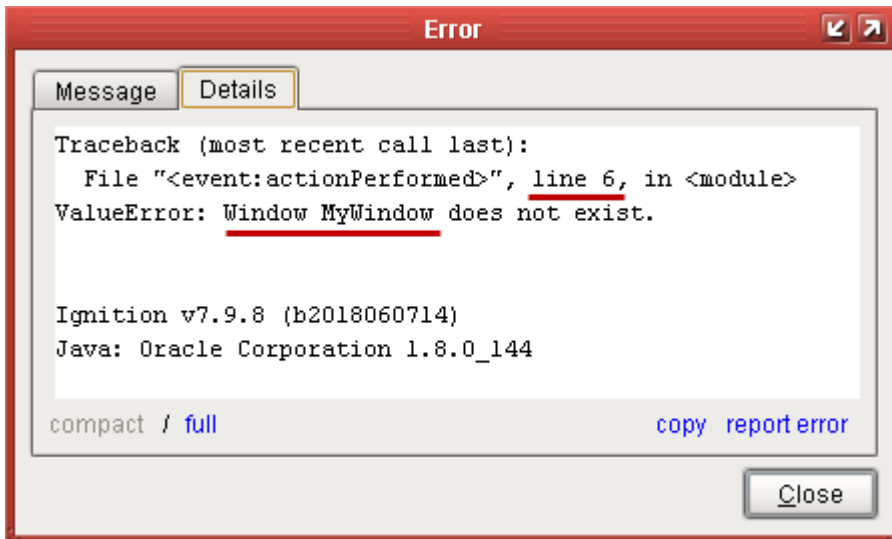


Your first inclination will be to close the error without closer examination: resist it!

The error messages generated from failed script executions are incredibly helpful! The **Message** tab on the error describes the Event Handler that encountered the exception; in this case, the actionPerformed event handler. This is important in cases where multiple Event Handlers on the same component have scripts. Without knowing which Event Handler generated the script, we could waste time trying to troubleshoot the wrong script.

Additionally, it shows the name of the component, which is "**NavButton_MyWindow**". Again, this is helpful if multiple scripts from multiple components are triggering in quick succession. The **Message** tab will clearly point you towards the source of the exception. This is another reason to give meaningful names to your components: doing so makes the process of tracking an error much easier.

The **Details** tab has even more information, specifically the **line number** that the exception occurred on, as well as the **error message**. The message here states that the window at path "MyWindow" does not exist, so we can check the Project Browser to see if we simply mistyped the name of the window.



Troubleshooting Errors Using the Error Message Box

When we don't get our expected results from our script, as in the following examples, always read both the Message and Details tabs in the Error Message Box. They are pretty good about pointing you to the root cause of your error. From there, you can easily find and fix any errors in your script.

The following examples show how to troubleshoot some of these error messages from the Error Message Box. Keep in mind that every script is unique, but at least you'll become familiar with what some of the error messages mean, and gain a little insight of the troubleshooting process.

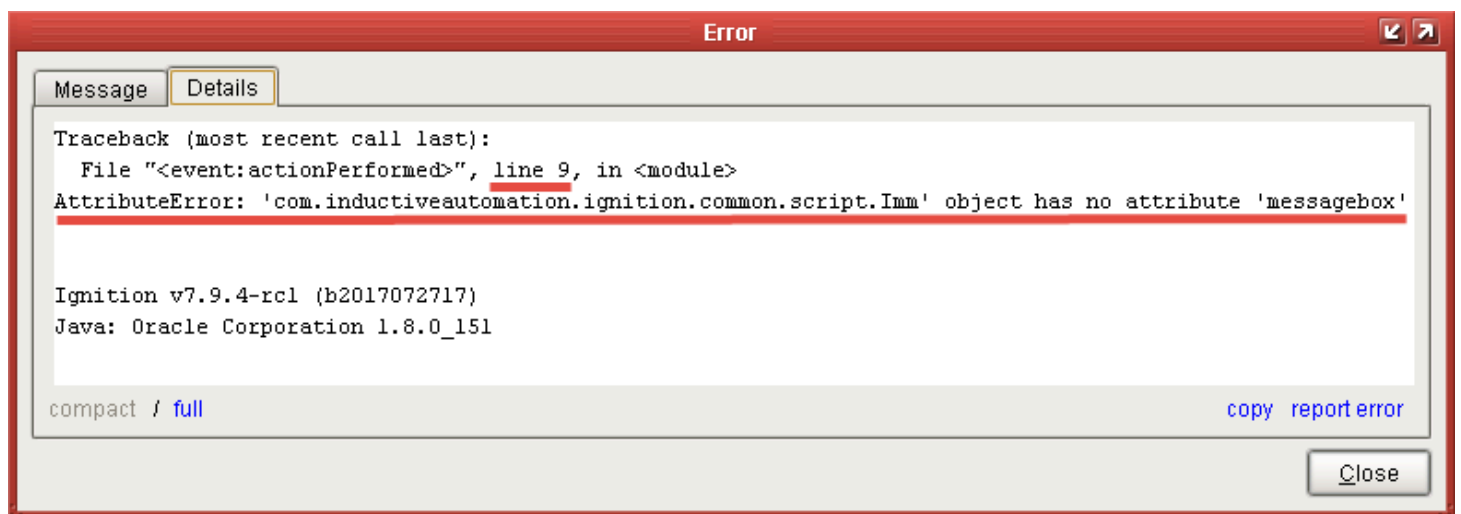
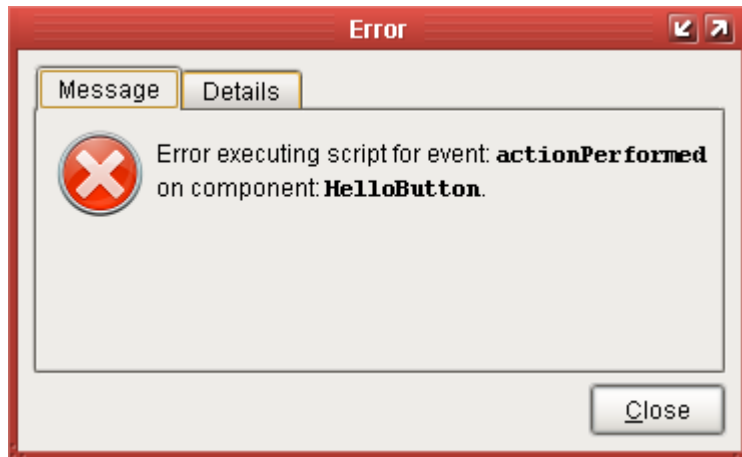
Broken Example - Incorrect Attribute

Here is a script that when a Button is pressed, it's supposed to open a Message Box on a window and display "Hello World".

```
system.gui.messagebox("Hello World")
```

From the Error Message Box, you know two important things right away: the script is using the `actionPerformed` event on the Hello Button, and line 9 displays the description of the error. Check to be sure you used the correct attribute. Then check the spelling, case sensitive letters, spacing, and operators in your script. If you are familiar with Ignition's Built-in Scripting Functions, you can probably spot the error immediately. If not, you might want to use the [autocomplete popup feature](#) to retype your 'messageBox' scripting function. This will automatically fix any syntax errors.

The error in this example is in the attribute name. It uses case sensitive letters (i.e., message**B**ox).



In this case, "messagebox" should be spelled with a capital "B" since we're trying to use Ignition's [system.gui.messageBox](#) function. We corrected the script by changing our code to the following:

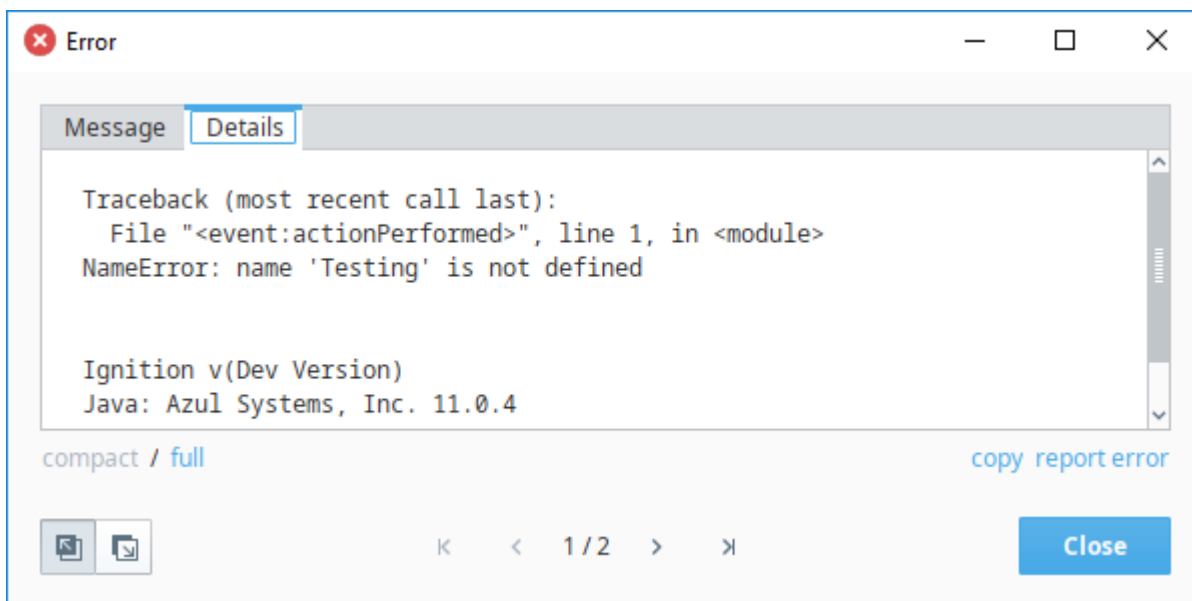
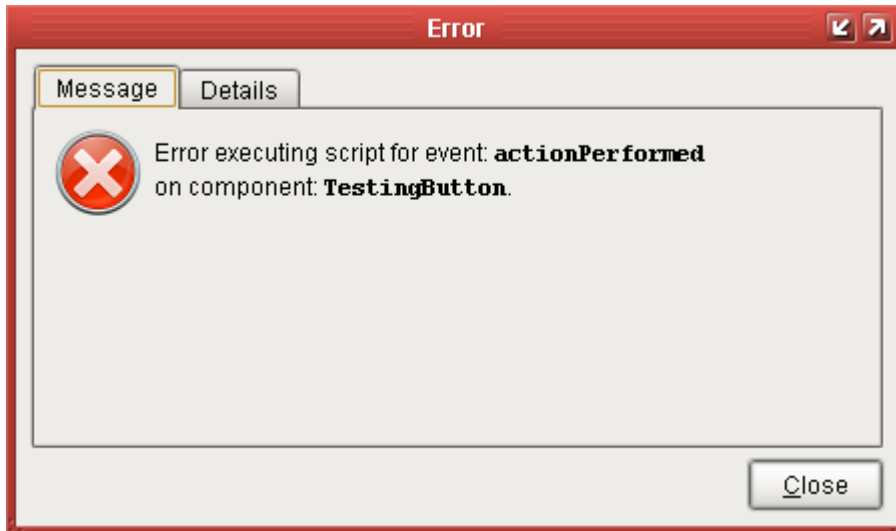
```
system.gui.messageBox("Hello World")
```

Broken Example - Undefined

Here is a similar script that when a Button is pressed, it's supposed to open a Message Box on a window and display "Testing".

```
system.gui.messageBox("Testing")
```

The Error Message Box displays an error on the `actionPerformed` script on the button component in line 9 with an **undefined** name. The Details tab is simply telling us that something is not defined in the script. You can either define a variable or create a string by putting "Testing" in quotes to correct the error.



For this example, we corrected the code by turning the argument passed to `system.gui.messageBox` into a string literal with quotation marks:

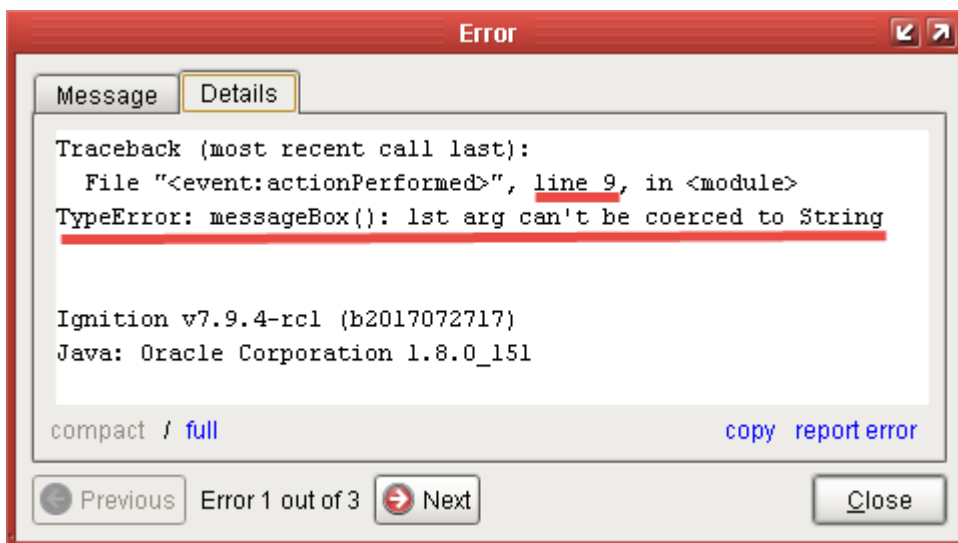
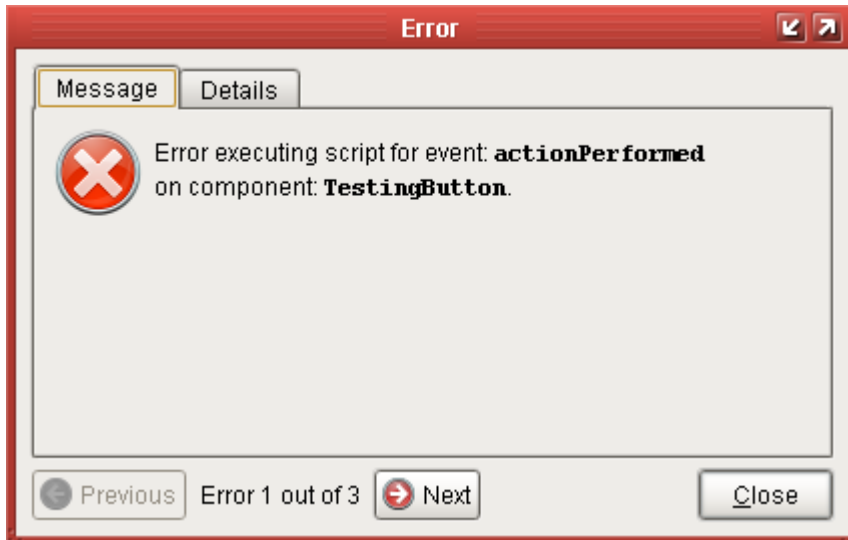
```
system.gui.messageBox("Testing")
```

Broken Example - Type Error

Similar to the other scripts, this script opens a Message Box and is supposed to display "100" when a Button is pressed.

```
system.gui.messageBox(100)
```

Since this script failed to execute successfully, the Error Message Box popped up and displayed an error on the `actionPerformed` script on the button component in line 9 with a **TypeError**. This error tells us that the 1st parameter is expecting a string, not an integer.



For this example, we corrected the code by changing the argument passed to `system.gui.messageBox` into a string literal with quotation marks.

```
system.gui.messageBox("100")
```

You can see from the examples above, that the Error Message Box provides quite a bit of information that points to the root cause of an error on an Event Handler script. As you learn from the Error Message Boxes about what these messages mean, you'll be able to quickly spot them in your script and quickly correct any errors you encounter.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Troubleshooting - Nothing Happened

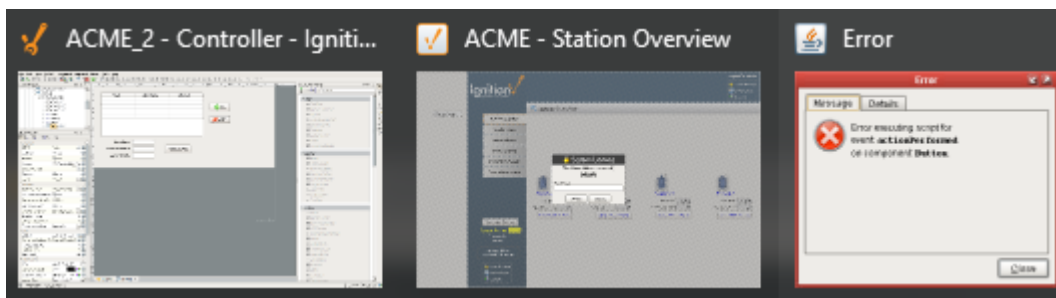
Did It Work?

When testing a script, you may find yourself in a situation where the script appears to be running, but doesn't seem to work. Furthermore, you may not see any error messages stating there is a problem.

When a script doesn't perform to expectations and doesn't throw an exception, it can suggest a problem with the script's workflow. This page describes a couple of common scenarios.

Before You Begin...

Make **absolutely** certain that there isn't an Error Message Box hiding somewhere, otherwise you'll waste time applying the troubleshooting tips outlined below, when the real the error message was hiding in the background behind another window the whole time.



If a button component on a window is running a script, the output will be displayed on the Designer/Client Output Console. You can check for any error messages by navigating to the Console. From the **Client** menu bar, go to **Help > Diagnostics** and select the **Console tab**. From the **Designer** menu bar, select **Tools > Console**.

Common Scenarios When Nothing Happens

An Important Line Is Missing

Some critical part of your code is missing or commented out. It could be something simple, like your code is supposed to increment a value, but you forgot to write the line that increments the value.

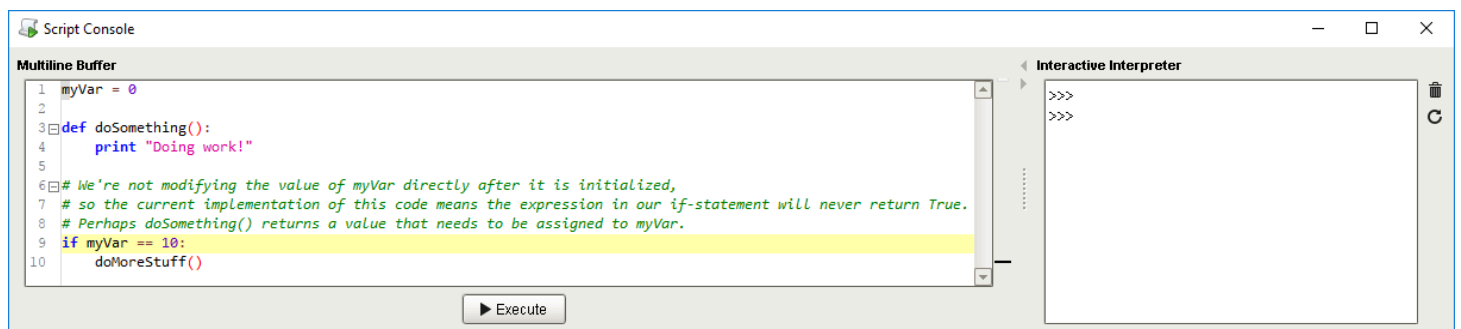
Pseudocode - Missing Code

```
myVar = 0

def doSomething():
    print "Doing work!"

# We're not modifying the value of myVar directly after it is initialized,
# so the current implementation of this code means the expression in our if-
statement will never return True.
# Perhaps doSomething() returns a value that needs to be assigned to myVar.
if myVar == 10:
    doMoreStuff()
```

A quick way to test your code is by running your script in the [Script Console](#) before attaching it to a scripting event or specific component. You can see that running the code above in the Script Console that nothing happened, thus, suggesting an error.



The Script Is Not Being Called

The script doesn't appear to be working because the mechanism that is supposed to be triggered hasn't been called. This can be caused by using the wrong event: (i.e., perhaps you placed the code on a Button's **propertyChange** event, when you meant to place it on **actionPerformed**).

Alternatively, perhaps you defined a function in your script, but you forgot to call the function.

Additionally, if you're testing the script in the Designer, make sure it is in **Preview Mode**. Event based scripts will not run in the Designer unless it is in Preview Mode.

Important Lines Are Being Skipped

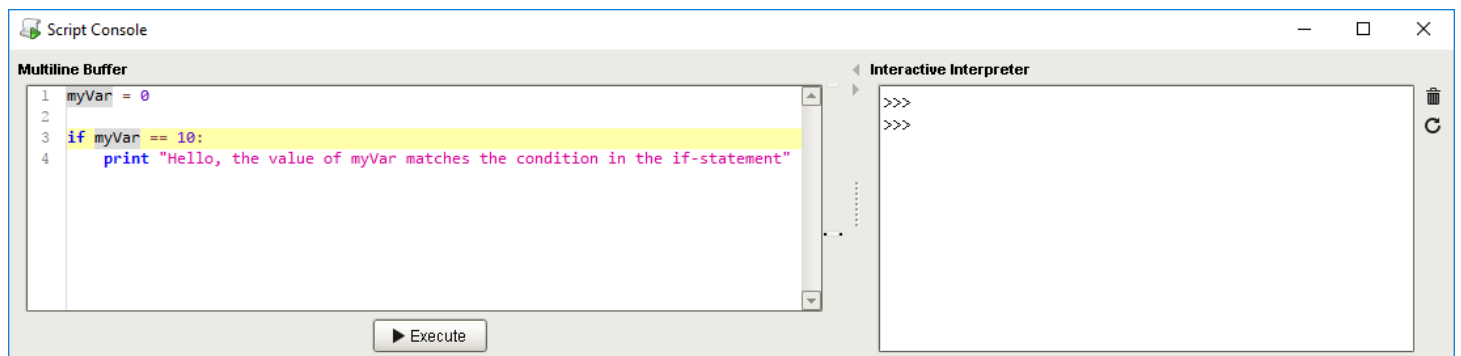
This can be caused by incorrect indentation, or a misconfigured condition in an if-statement's expression. For example, you used "==" when you meant to use "!=". Verify that you are using the correct operators in if-statements.

Pseudocode - Skipping Code

```
myVar = 0

if myVar == 10:
    print "Hello, the value of myVar matches the condition in the if-statement"
```

Test your code for any errors by running your script in the Script Console. You can see nothing happened by running the code above in the Script Editor, once again suggesting an error.



Determining the Cause

The easier step to take when troubleshooting your script is to start adding print statements to your code. From here, you can start piecing together what the code is doing:

- **Print** the value of variables to make sure they are coming in the way you expect.
- Place **print** statements at the start and end of your code. This allows you to determine when your script is being called, and when it finishes.
- Adding a **print** statement before and after an if-statement can show you the flow of your script went as expected, or was filtered out by the if-statement.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Troubleshooting Workflow

title: Troubleshooting Workflow sidebar_position: 3

Troubleshooting a script is an iterative process. Since the script stops executing at the first error, we won't see if there are any other errors on later lines unless, of course, we spot them ourselves while writing the code. As a result, we will have to keep trying our script until it executes successfully.

It is important to understand that scripts always execute from top to bottom, and each line must complete before the next line may move on. In the event, our code returns an exception, we can assume that our troubleshooting process should always start at the line reported in the error, and then work up until we find the problem.

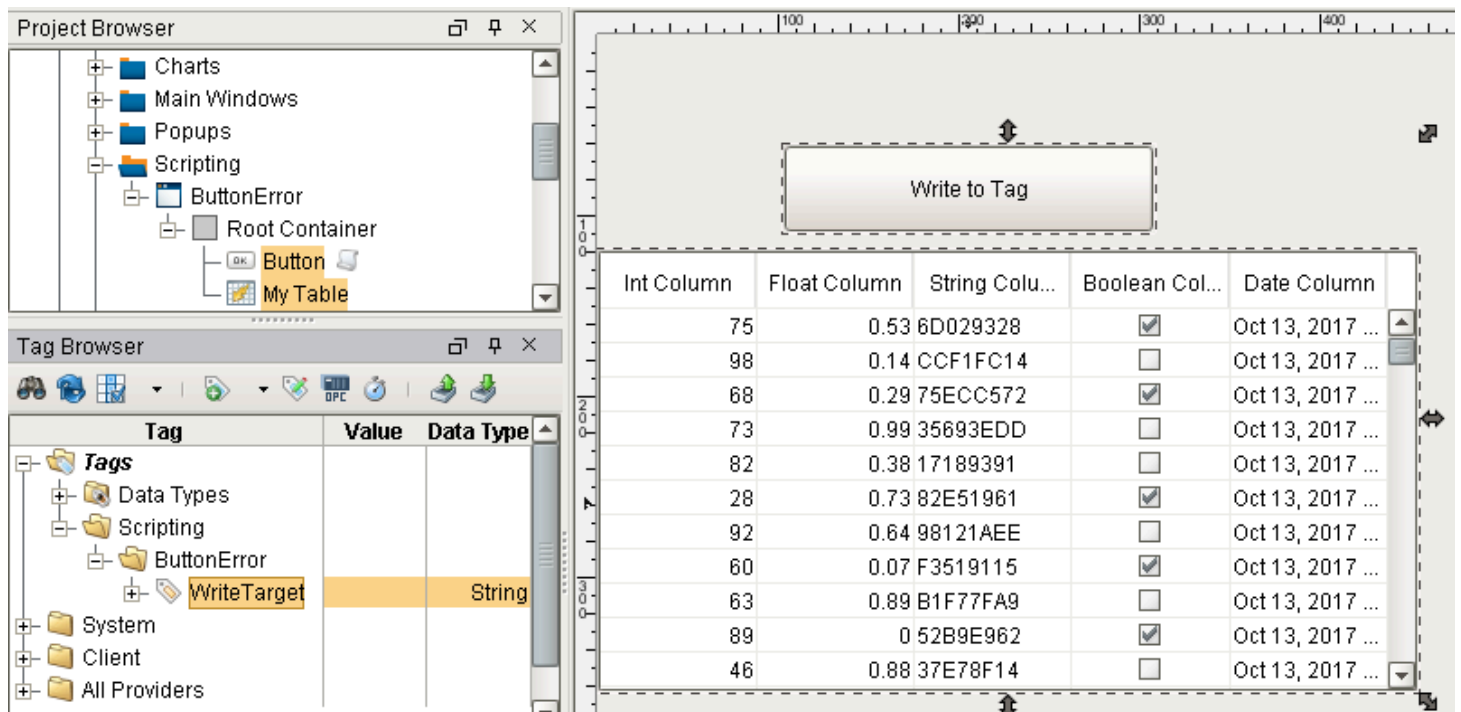
On this page, we will take an in depth look at a script with multiple problems, and work through each one. Note, that this is not a comprehensive list of all possible types of exceptions that could occur when writing a script, but instead, this page attempts to demonstrate the troubleshooting process.

To learn more about troubleshooting specific script errors, refer to the following sections:

+ [Reading Error Messages](#) + [Troubleshooting - Nothing Happened](#)

Scenario Overview

Here we have a window with two components: a **Button** and **Power Table**. The purpose of the button is to find which cell in the Power Table the user selected, and write the value to a Tag.



The code on the button is listed below.

Python - Sample Code Block

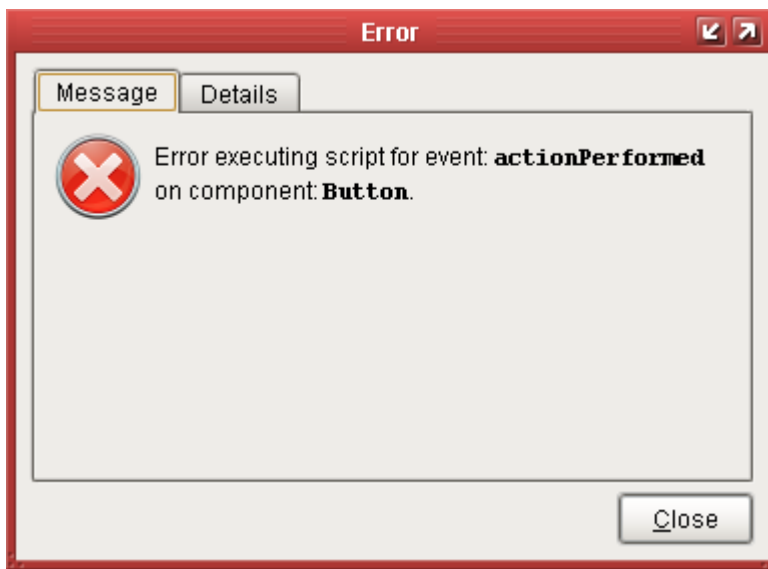
```
# Create a variable that references the Power Table
table = event.source.parent.getComponent('Power Table')

# Find the cell the user currently has selected, and store the value in a variable
userSelectedValue = table.data.getValueAt(table.selectedrow, table.selectedColumn)

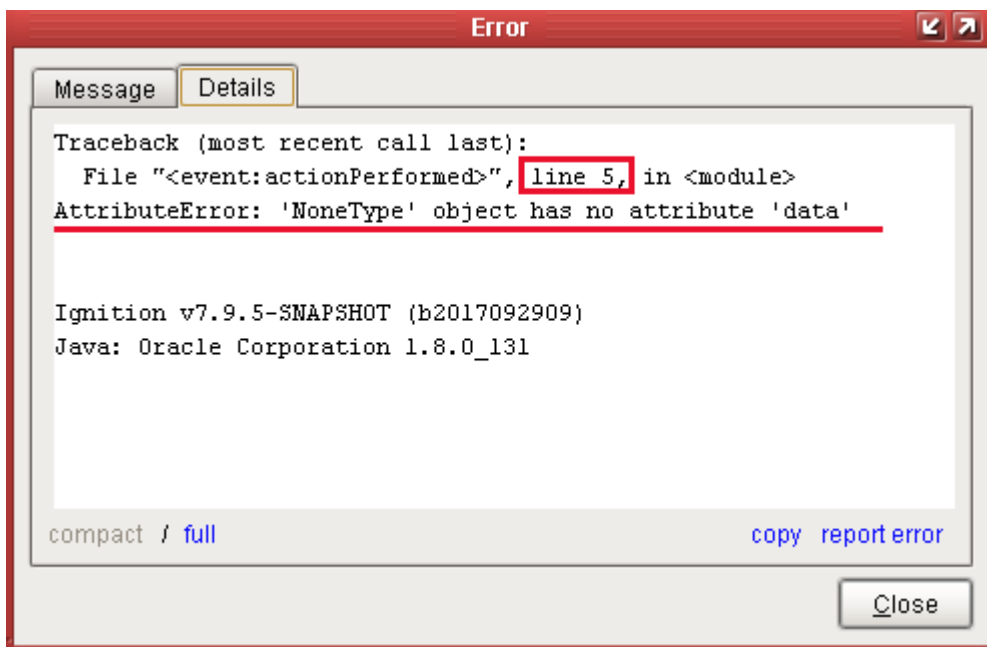
# Write the User Selected Value to a tag
system.tag.write("Scripting/ButtonError/WriteTarget", userSelectedValue)
```

First Error - NoneType Object

When the button is pressed, we are presented with an error. The **Message** tab in the Error Box describes the **Event Handler** that encountered the exception.



The **Details** tab tells us where to start looking, specifically the line number where the exception occurred as well as the error message. Line 5 is referenced, so the troubleshooting process should start there.



Exception Error Explained

When an error message refers to a **NoneType** object, that simply means a null, or None in Python. The word '**attribute**' is used to reference a property on an object. In this case, this means the script was trying to access the '**data**' property on nothing. The message is telling us that it couldn't find a property named '**data**' on a nothing, which is correct since NoneType objects don't have any properties named 'data'.

If we were to reword this message to something a bit more straightforward, it would say the following:
"I tried to access the 'data' property on 'null', but it doesn't have a property by that name."

What Should We Look For?

If your exception is referring to a `NoneType` object, then some line of code probably tried to reference something else (i.e., Tag value, property, another variable, etc.), but couldn't find anything at the location you specified. It is also helpful to note what attribute was mentioned in the error, which was **data**. If we look at our code, we see the following on line 5.

Python - Line 5

```
userSelectedValue = table.data.getValueAt(table.selectedrow,  
table.selectedColumn)
```

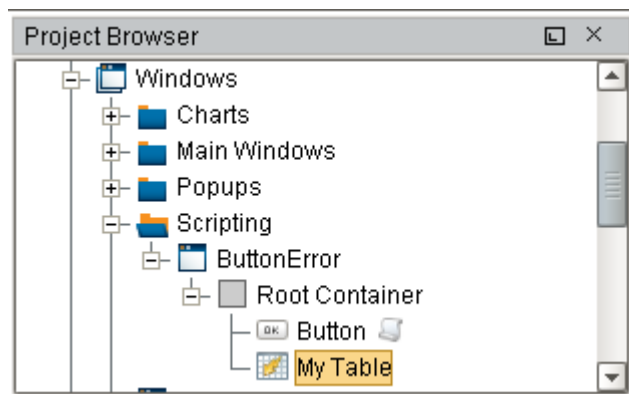
Based on this information, we should take a closer look at the `table` variable since our code is specifying '**data**' as an attribute of the table. The error states that it could not find the `data` attribute on a `NoneType` object, but our code only references `data` in regards to the table object, so maybe there is something wrong with how that table object was referenced or initialized. If we look further up in our code, we see that `table` was initialized on line 2:

Python - Line 2

```
table = event.source.parent.getComponent('Power Table')
```

Since this script was placed on the Button component's `actionPerformed` event, we can trace back to the source of the issue:

+ `event` = `actionPerformed` event + `source` = Button this script is placed on + `parent` = Container that the Button was placed in. Based on the Button's position in the window, this appears to be the Root Container. + `getComponent('Power Table')` = Should return a reference to a component named '**Power Table**' directly inside of the Root Container, assuming one exists. If we take a look at the Project Browser, we see the following:



There is no component named '**Power Table**' in the Root Container. However, there is a Power Table component named '**My Table**', so it appears someone renamed the component, which caused our script to initialize the variable table as a '**NoneType**' instead of a reference to a component.

Solution

To fix this issue, we can simply update our code to use the new name of the Power Table, which is '**My Table**'. We can type this in manually, but since it is case sensitive, it must match exactly. It may be easier to find the **Name** property on the component, copy the name to the system clipboard (**Ctrl-C**), and paste it (**Ctrl-V**) into the script. Our code now looks like the following:

```
# Create a variable that references the Power Table
table = event.source.parent.getComponent('My Table')

# Find the cell the user currently has selected, and store the value in a variable
userSelectedValue = table.data.getValueAt(table.selectedrow, table.selectedColumn)

# Write the User Selected Value to a tag
system.tag.write("Scripting/ButtonError/WriteTarget", userSelectedValue)
```

Second Error - Checking Attributes

If we try our button again, we get the following exception error.



Exception Error Explained

Like the last exception, the problem has to do with our code trying to access an attribute (property) on an object, but the given attribute doesn't exist on the object.

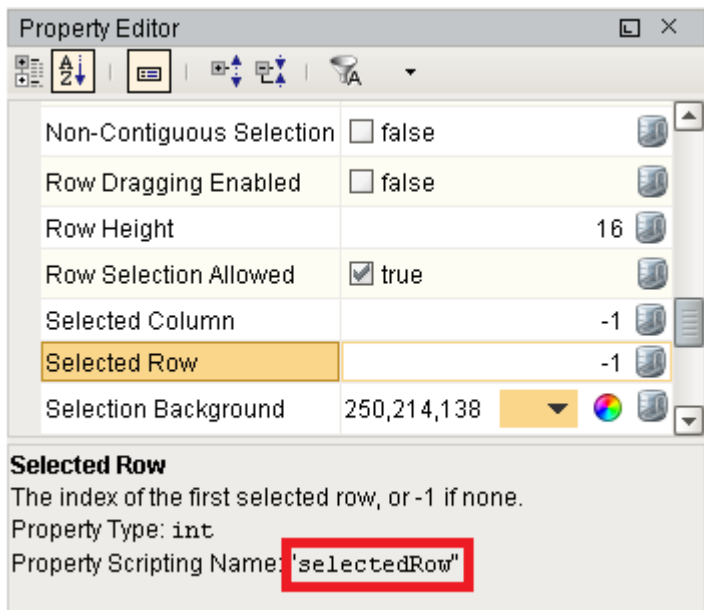
What Should We Look For?

Again, the exception refers to line 5, but note that the error mentions a different object and attribute this time. Without knowing what a `com.inductiveautomation.factorypmi.application.com` object is, we can tell that our script is trying to reference an attribute named `selectedrow`. This gives us a starting point for troubleshooting the error. If we check line 5, we see the following:

Python - Line 5

```
userSelectedValue = table.data.getValueAt(table.selectedrow, table.selectedColumn)
```

We see `table.selectedrow` as the first parameter being passed to the `getValueAt()` function. We checked into the `table` variable earlier, so this doesn't necessarily mean that the variable is the problem, although there is no harm in double checking, especially if you have not personally verified the variable. Assuming that the table variable is correct, let's check the attribute. We can check the Property Editor in the Designer to find a matching property, or head over to the Power Table page in the manual and check the property reference. We can see that there is a **Selected Row** property on the **Power Table**, but it should be spelled "`selectedRow`" with a capital "R".



We also know that the `getValueAt()` function takes a row index as the first parameter, so it would make sense that our code has a typo.

Solution

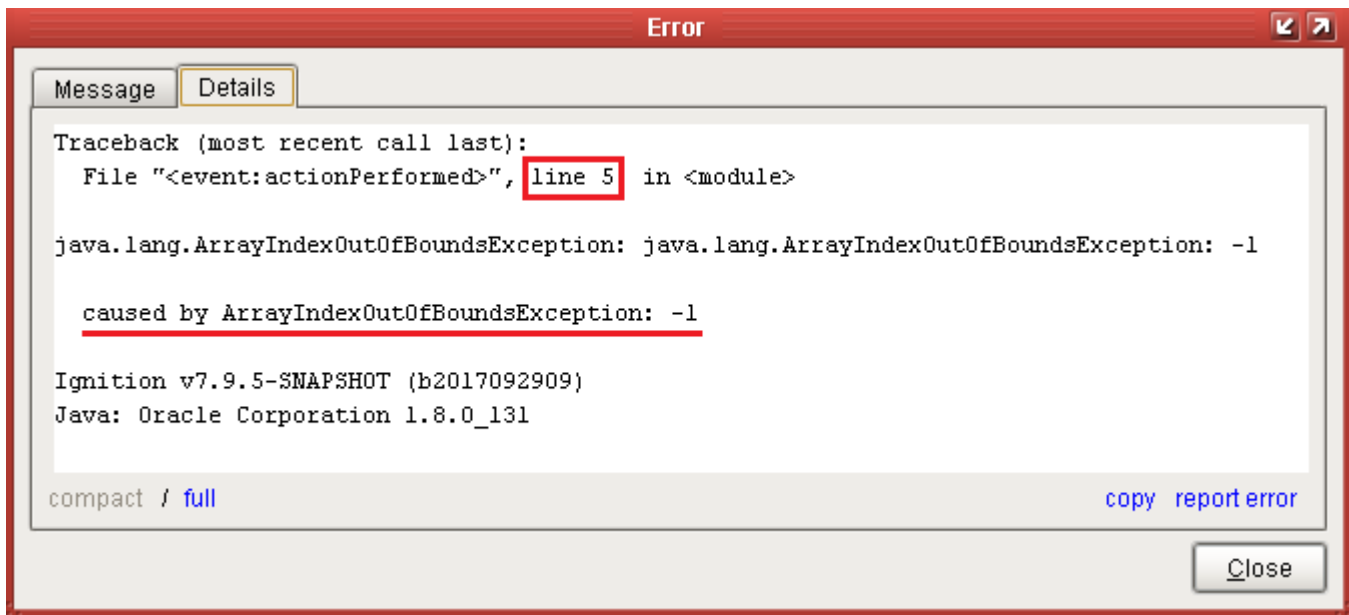
We should fix the typo in the attribute name.

Python - Line 5 Updated

```
userSelectedValue = table.data.getValueAt(table.selectedRow, table.selectedColumn)
```

Third Error - ArrayIndexOutOfBoundsException

After making our most recent change, the button works great! Users are able to select a cell, and write to the Tag. The one exception, is if the user does not have a cell selected before clicking the button. In this case, the following error occurs:



Exception Error Explained

Our code attempted to look up something in a collection of some sort (i.e., sequence, dataset, etc.,) by index, but was told to check index -1, which is out of bounds. Indexes in Ignition are typically zero-based, meaning they start at 0, and increment from there, so an index of -1 doesn't exist. Commonly, when a property in Ignition refers to index -1, that means nothing is selected, so our code failed because nothing was selected.

What Should We Look For?

Fortunately, the exception told us what the issue is: the user did not have anything selected before pressing the button, so we don't have to search further.

Solution

The solution to this issue is more open ended, as there are many ways to suggest to our user that they need to select a cell in the table first. The **Selected Row** and **Selected Column** properties on

the Power Table will have -1 values if none of the cells are selected, so we could use an 'if' statement checking the values of one of those properties. Our new code could look like the following:

```
# Create a variable that references the Power Table
table = event.source.parent.getComponent('My Table')

# Make sure a cell in the Power Table is selected first
if table.selectedRow != -1:
    # Find the cell the user currently has selected, and store the value in a variable
    userSelectedValue = table.data.getValueAt(table.selectedRow,
table.selectedColumn)

    # Write the User Selected Value to a tag
    system.tag.write("Scripting/ButtonError/WriteTarget", userSelectedValue)

# If a cell isn't selected, then let the user know
else:
    system.gui.messageBox("Please select a cell in the table first!")
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Scripting Vs. SQL Vs. Expressions

title: Scripting Vs. SQL Vs. Expressions
sidebar_position: 4

Expression Language & SQL Queries vs Scripting

There are three major languages in Ignition, the **Expression language**, the **SQL Queries**, and **Python Scripting**. It is important to understand the differences between the three and to know where each is used. Scripting is used in the event handlers that are available all over Ignition, but Expressions and SQL are in the **Property Binding** locations shown here.

Spot the Difference - Comments

When starting out with Ignition, it can be difficult to know which syntax you should be using for a particular area of text. One little trick that may help is to be familiar with how each language handles comments, and then utilize the **Ctrl + /** command which automatically adds the characters that comments out a line of code in the language are you are typing in. Once you remember to use **Ctrl + /**, you simply need to be familiar with the characters that each language uses.

Expression Language - Comment

```
// The Expression Language uses double forward slashes.
```

SQL - Comment

```
-- Areas in Ignition that accept SQL syntax use double dashes.
```

Python - Comment

```
# Python uses the pound/number/hash character.
```

In addition to comments, interfaces for each of the languages usually contain other signs or reminders about the language. These will be covered in their respective sections on this page.

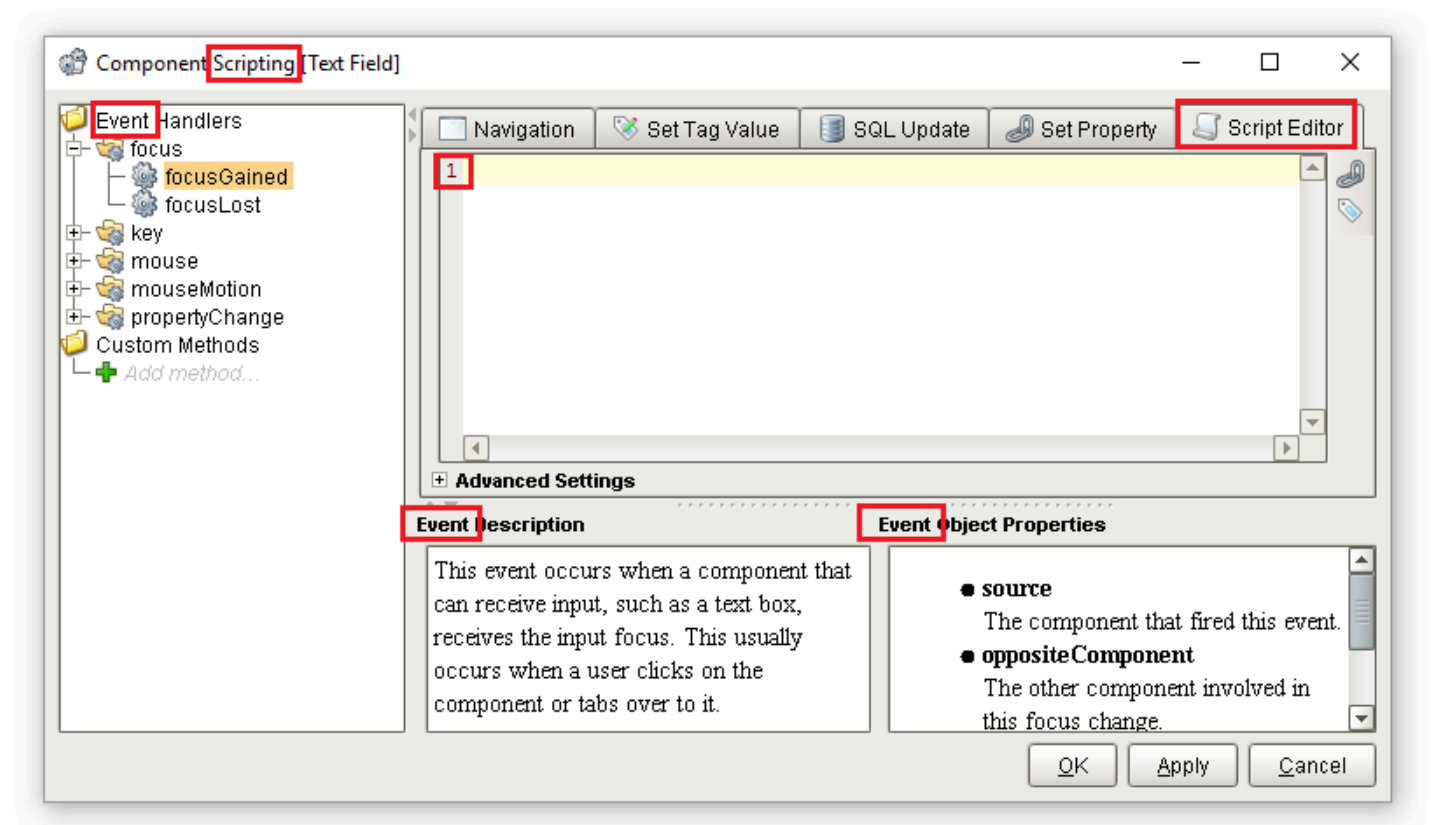
Python Scripts

Overview

Python is featured prominently throughout Ignition and many different resources can contain a Python Script.

How Can I Tell If I'm Writing a Python Script?

Python Scripts typically use the words **Script** or **Event** in the interface. Additionally, Python requires a particular Event to be selected so if you see event Handlers on the left, you know you are looking at a Python script.



Where Are Python Scripts Used?

Python Scripts are used all throughout Ignition. Some resources, such as components, even have multiple places to write scripts! Below are some common locations for scripts:

* **Component Scripts** * **Extension Functions** * **Tag Event Scripts** * **Client and Gateway Event Scripts**

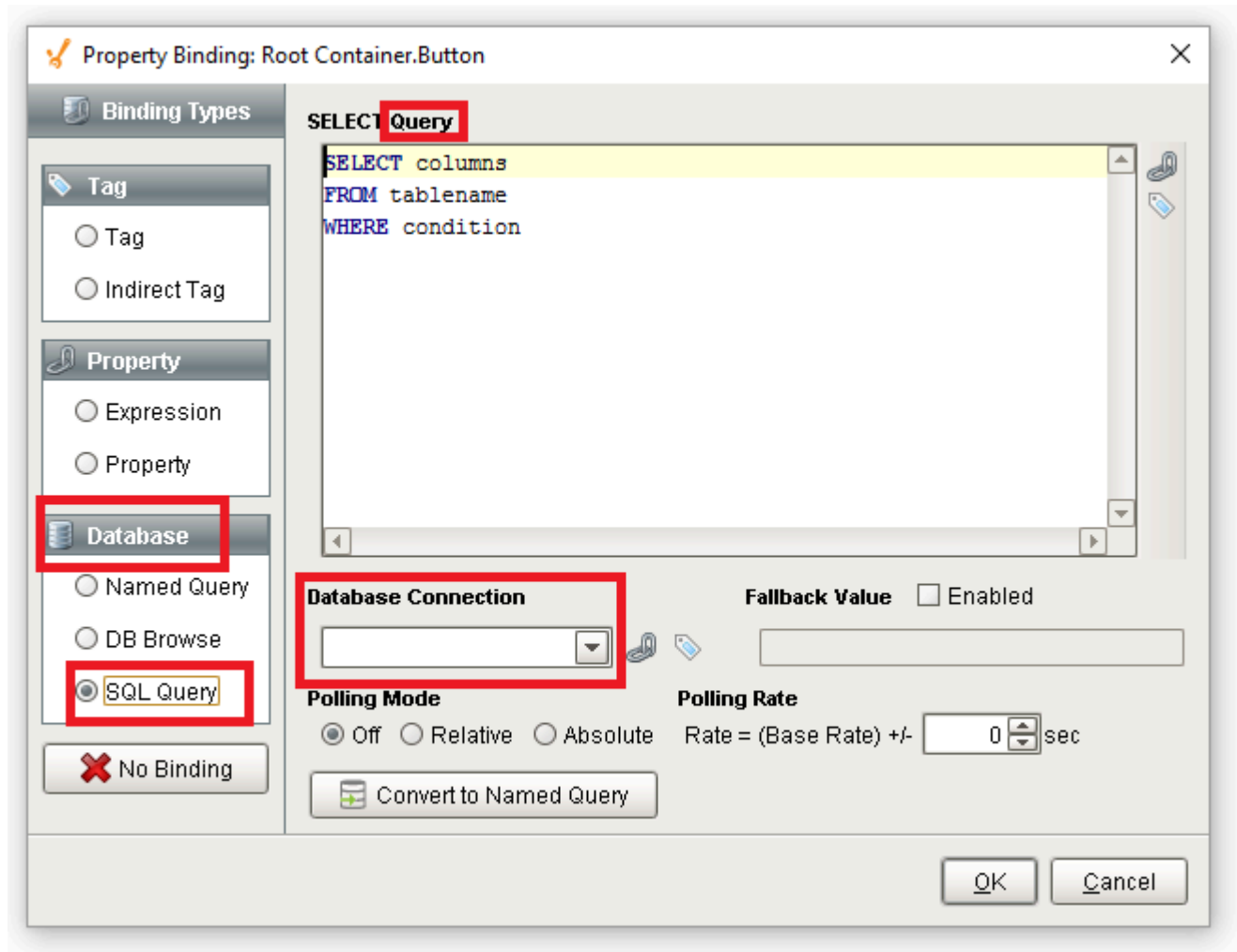
SQL Queries

Overview

The SQL language is used for selecting information from a database. It can be used in a variety of ways, but most of them will either make modifications to the database or return a set of values (with a few notable exceptions like Stored Procedures). The majority of users will be returning large chunks of data into a Table or Report in Ignition. This means a complete dataset will be returned based on a user selection, a time range, or any combination of factors.

How Can I Tell If I Should Be Using SQL Syntax?

Typically, the words **Query** and **Database** appear in the interface somewhere. Additionally, there is usually a way to specify a **Database Connection**.



Where Is SQL Used in Ignition?

Below is a reference of the most common areas in Ignition where SQL queries may be used.

* [SQL Query Bindings](#) * [Named Queries](#) * [Database Query Browser](#) * Reporting Data Source: * [SQL Query Data Source](#) * [Basic SQL Query](#) * [Query Tags](#) * Python Scripts - See the SQL in Python header below for more details

SQL in Python

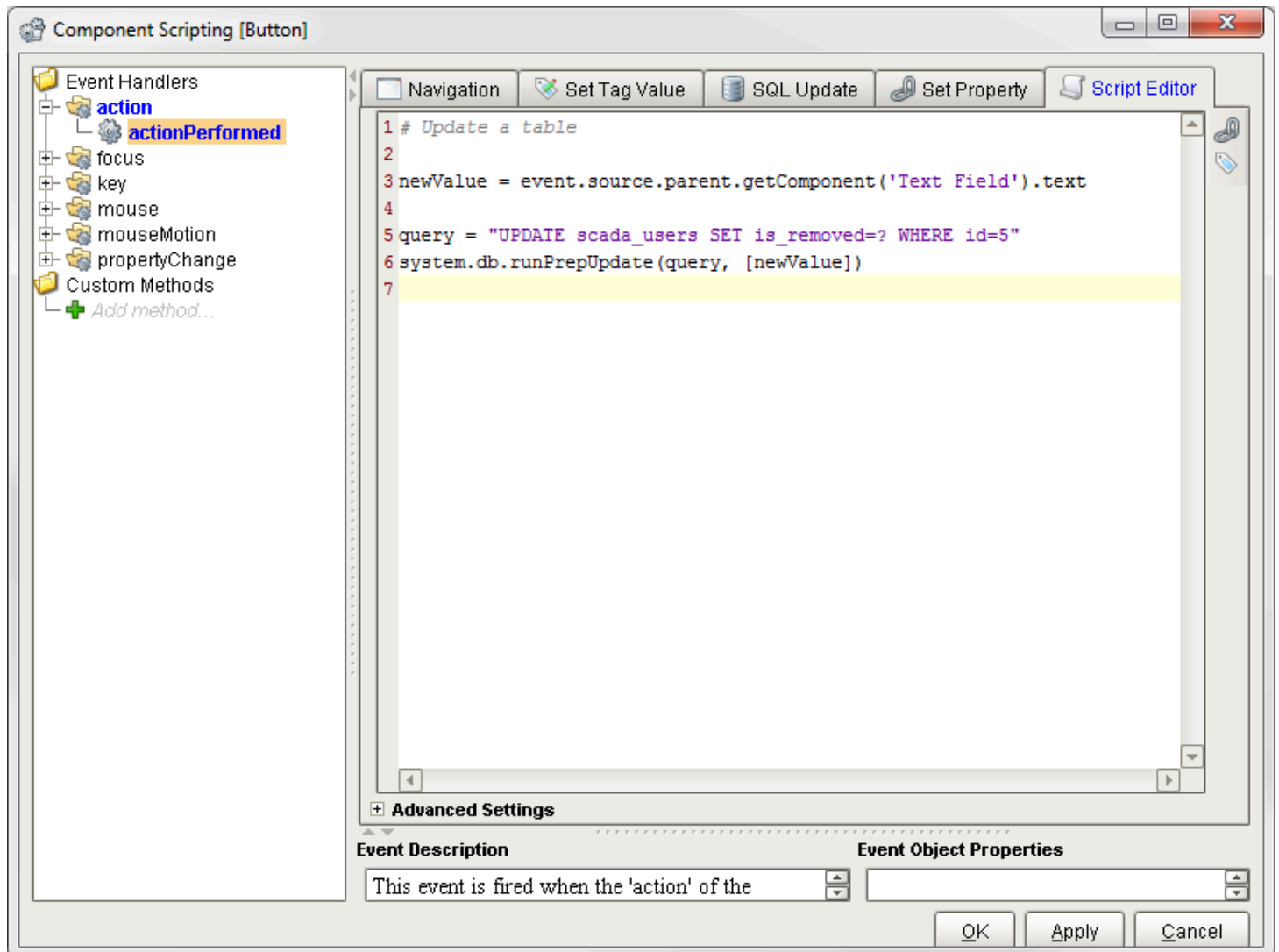
SQL queries can be called from a **Python** script. There are several system functions in Ignition that allow a script to run a query against the database, such as [system.db.runPrepQuery](#). This is a more advanced technique, as you need to adhere to both language's syntax. Furthermore, when typing a SQL query in a Python script interface, the syntax highlighting can not help with the SQL portions. The syntax highlighting in a Scripting Window is only looking for Python syntax, not SQL.

In cases where you plan on calling a SQL query from a Python script, it is highly recommended to write the query in the **Database Query Browser** first (substituting parameters with static values for testing purposes), and then move the query over to the script once the query executes successfully

on its own. This approach can save you some time troubleshooting, as there will be less ambiguity when an error occurs since you know the query runs.

Below we see an example of calling a SQL query in a script. Line 5 creates a variable called "query", and assigns it a string consisting of a prepared statement (using SQL). The query is then executed with the `system.db.runPrepUpdate` function.

For more examples of using a query in a Python Script, check out the `system.db.*` functions.



Expression Language

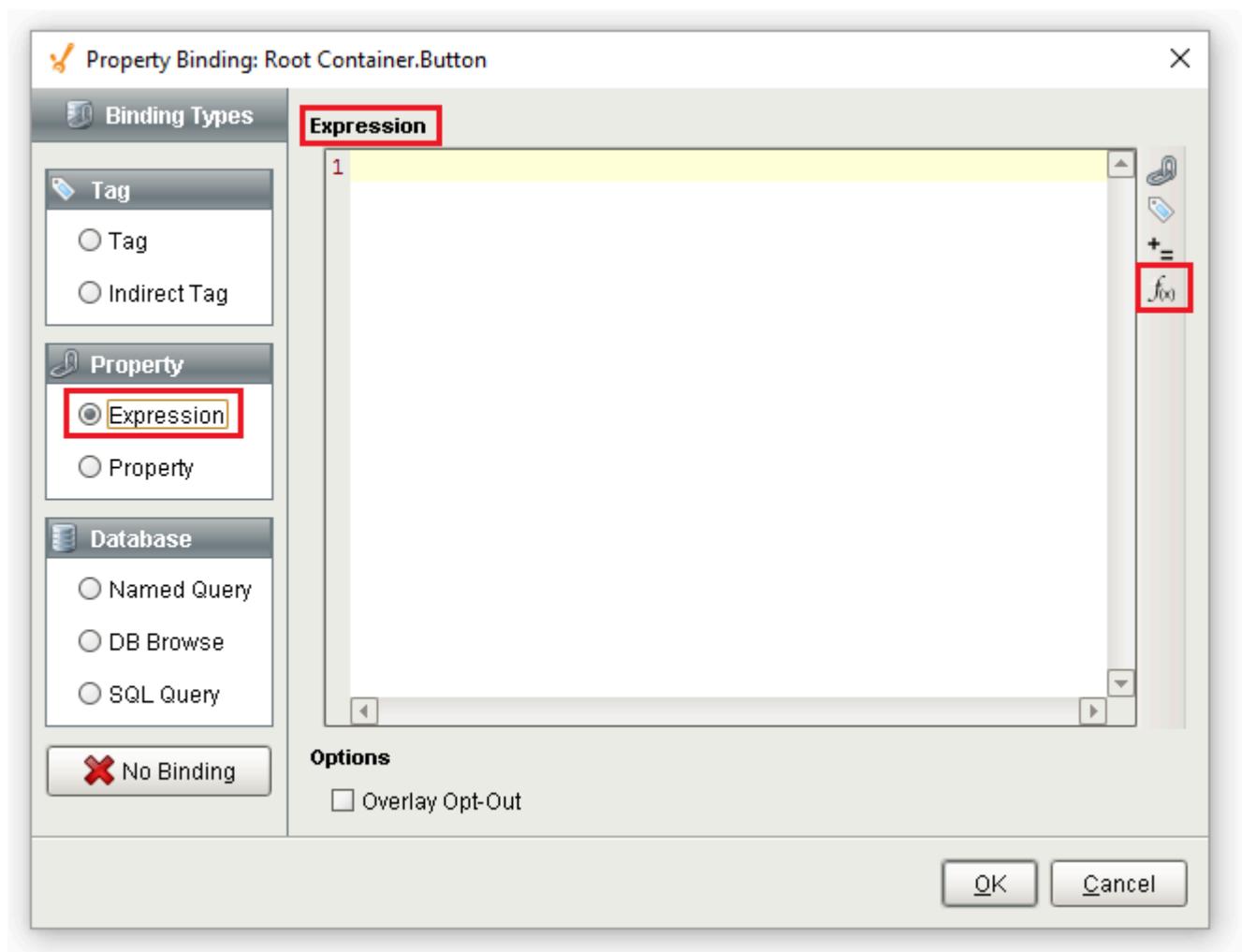
Overview

The Expression Language is a simple programming language that we invented (very similar to many other existing expression languages), and is different from the Python scripting you will find in Ignition. The expression language is a very simple kind of language where everything is an expression - which is a single piece of code that returns a value. This means that there are no statements, and no variables, just operators, literals, and functions.

The most common expression language that most people are familiar with is the one found in Microsoft Excel. You can have Excel calculate a cell's value dynamically by typing an expression like `=SUM(C5:C10)`. Our expression language has similar functionality, but different syntax. It is mainly used to define dynamic values for **Tags** and **component properties**. In Ignition's Expression Language, you will use component properties and functions like `if({Root Container.type}="Type C", True, False)`.

How Can I Tell If I'm Writing an Expression?

Typically interfaces that expect the Expression Language use the word **Expression**. Additionally, you'll commonly find the **Expression Function** button  towards the right side of the text area.



Where Is The Expression Language Used?

Below is a list of resources that commonly utilize the Expression Language

- * [Expression Bindings](#) on Property Bindings and [Alarm Bindings](#)
- * [Expression Tags](#) and [Derived Tags](#)
- * Several [Alarm Pipeline Blocks](#), including the [Expression Block](#), [Switch Block](#), and [Notification Blocks](#).
- * [Expression Items](#) in Transaction Groups
- * [Parameters on Reports](#)
- * [SFC Transitions](#)

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Common Scripting Tasks

This section contains examples for items we've identified as "common" tasks: undertakings that many users are looking to utilize when first starting out with a specific module or feature in Ignition.

Additionally, this section aims to demystify some of the more complex or abstract tasks that our users may encounter.

The examples in this section are self-contained explanations that may touch upon many other areas of Ignition. While they are typically focused on a single goal or end result, they can easily be expanded or modified after the fact. In essence, they serve as a great starting point for users new to Ignition, as well as experienced users that need to get acquainted with a new or unfamiliar feature.

Below is a list of common tasks related to this section of the manual.

Reading and Writing to Tags

There are simple interfaces in Ignition that allow you to easily write to a Tag on some Event, and Reading can be as simple as creating a Tag binding. Sometimes, however, the built-in approaches can be too simplistic or limiting. The [Reading and Writing to Tags](#) page details how to better interact with Tags from scripting.

Importing and Exporting a CSV

CSV files are used by many software programs to export data so that other systems may utilize the information contained within. The [Importing and Exporting a CSV](#) page demonstrates how to both import a CSV into Ignition, as well as export data from Ignition into a CSV.

Reading a Cell from a Table

Once data is populated into a Table component, it's useful to know how to read and extract a data from a cell in a Table, particularly if users can select a row in a Table. The [Read a Cell from a Table](#) page has some good examples for retrieving data from a single cell and multiple cells in a Table.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Reading and Writing to Tags



INDUCTIVE
UNIVERSITY

Reading and Writing Tags

[Watch the video](#)

In many cases, the binding system is the most appropriate way to display a Tag value on the screen. However, you may wish to access a Tag's value in a script. Using the system functions, you can read from a Tag and write to a Tag in Ignition.

Script Builder

If you simply need to write to value to a Tag from a script, you can use the Set Tag Value tab of the Script Editor. To learn more, refer to the sections on [Script Builders](#) for more information. However, if you need to do more than just send a single write, see the section below.

Know Your Scope

Whenever a script from a Shared resource (such an Alarm Pipeline) attempts to interact with a Tag, the script must specify the Tag Provider, otherwise, the script could return an exception. Tag Providers are always included at the start of the Tag Path and look like the following: `[tagProvider]`

When interacting with a Tag from the Project scope (such as a script on a component) you may optionally include the Tag provider. If omitted, the Project default Tag provider will be used.

Pseudocode - Tag Read Scope

Reading a Tag without the Tag Provider. You would not want to use this from the Shared scope.

```
system.tag.read("My/Tag/Path")
```

Reading from the same Tag, but specifying the Tag Provider. This format may be used safely in either scope.

```
system.tag.read("[default]My/Tag/Path")
```

Manual Tag Reads

Reading from a Single Tag

Reading a Tag from a script is accomplished with the `system.tag.read()` function, which requires the Tag path you wish to read from. Note, that this function returns a 'Qualified Value', this object is more than just the value. A Qualified Value is the Tag value that has three attributes; Value, Quality, and TimeStamp. If you are getting multiple attributes, make sure to read the Tag once and not once for each attribute.

Pseudocode - Reading Tag Attributes

get the tag value

```
value = system.tag.read("tagPath").value
```

get all three attributes

```
tag = system.tag.read("tagPath")
```

```
value = tag.value
```

```
quality = tag.quality
```

```
timestamp= tag.timestamp
```

Reading from Multiple Tags

In the event you need to read from a large number of Tags, it is more efficient to use `system.tag.readAll()` instead which reads a Python List of Tag Paths.

Python - Reading Multiple Tags and Printing the Values

```
# Create a List of Tag Paths to read
paths = ["Scripting/Tags/Alarm_1", "Scripting/Tags/Alarm_2",
"Scripting/Tags/Alarm_3"]

# Read the Tags, and store the complex results in a variable
values = system.tag.readAll(paths)

# For each Tag Path, iterate through our results...
for index in range(len(paths)):

    # ...and do something with the individual values
    print values[index].value
```

Relative Tag Paths

Like elsewhere in Ignition, **relative paths** may be used from within a script. This is especially useful when writing a Tag Event script inside of a UDT, as you can specify relative members in the same UDT with "[.]".

Pseudocode - Reading with Relative Tag Paths

```
# Assuming a UDT with two sub-members, a script from one can read from one member
to the other using the following code:
system.tag.read([.]otherMember).value
```

Manual Tag Writes

Writing to a Single Tag

Writing to a Tag with the **system.tag.write()** function requires the Tag path, as well as the value you wish to write to the Tag.

Pseudocode - Tag Write

```
system.tag.write("tagPath","Hello World")
```

Writing to Multiple Tags

Writing to multiple Tags from a single script should use the **system.tag.writeAll()** function, as it is more efficient than several `system.tag.write()` calls.

Python - Multiple Tag Writes

```
# Create a List of Tag Paths to write to
paths = ["Scripting/Tags/Alarm_Setpoint_1", "Scripting/Tags/Alarm_Setpoint_2"]

# Create a List of values to write
values = [72, 72]

# Send of the write requests
system.tag.writeAll(paths, values)
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Exporting and Importing a CSV

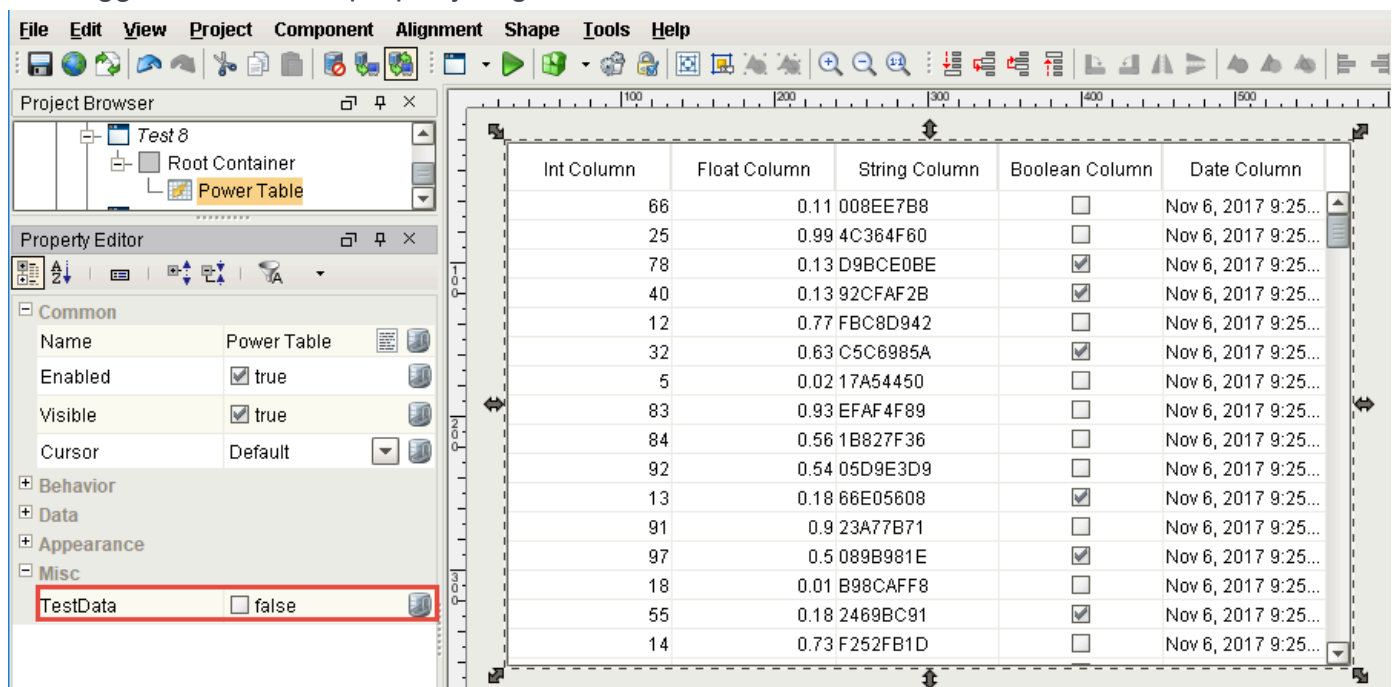
A CSV file, (comma separated values) is one of the most simple structured formats used for exporting and importing datasets. It is a convenient and flexible way to edit and share data across applications. Ignition has a built-in function to convert a dataset to CSV data called `system.dataset.toCSV`. You can even convert the contents of a CSV to a script and move it to an Ignition component, such as a Power Table.

This section contains examples for exporting and importing data to a CSV as well as converting the contents of a CSV to a script.

Exporting Data to a CSV

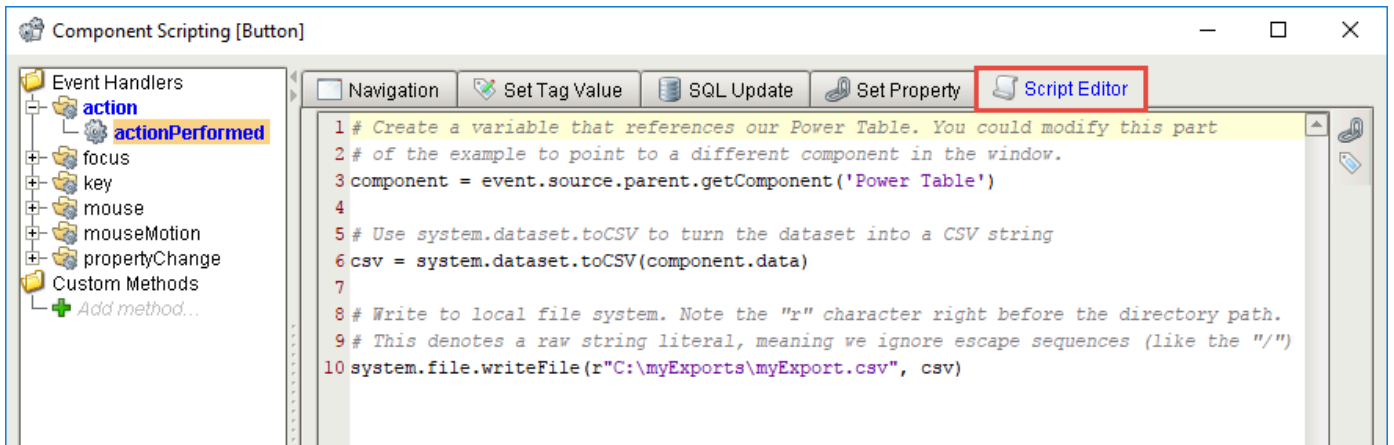
You can export a dataset from a query or table to a CSV file.

1. Identify the dataset that you want to export to a CSV file. In this case, we can generate some data on a Power Table and export that data. Drag a **Power Table** component on to your window and toggle its **TestData** property to generate some data.



2. Drag a **Button** component on the window, and double click on the Button to open the **Component Scripting** window.

- Next, let's add our script on the Button component's **actionPerformed** event. Select the **actionPerformed** event, and click on the **Script Editor** tab.
- Copy the contents from one of the examples below, and paste the contents to the **Script Editor**. Notice the **system.dataset.exportCSV** scripting function is used in the first example to export the dataset to a CSV file:



Python - Hard Coded Filepath

```
# Create a variable that references our Power Table. You could modify this part
# of the example to point to a different component in the window.
component = event.source.parent.getComponent('Power Table')

# Use system.dataset.toCSV to turn the dataset into a CSV string
csv = system.dataset.toCSV(component.data)

# Write to local file system. Note the "r" character right before the directory
# path.
# This denotes a raw string literal, meaning we ignore escape sequences (like the
# "/")
system.file.writeFile(r"C:\myExports\myExport.csv", csv)
```

- Instead of hardcoding the path as we did in the above example, we could ask the user to select a directory on the local system with **system.file.saveFile**:

Python - User Selected Directory

```
# Create a variable that references our Power Table. You could modify this part
# of the example to point to a different component in the window.
component = event.source.parent.getComponent('Power Table')

# Use system.dataset.toCSV to turn the dataset into a CSV string.
csv = system.dataset.toCSV(component.data)

# Use system.file.saveFile to have the user find a directory to write to.
filePath = system.file.saveFile("myExport.csv", "csv", "Comma Separated Values")

# We can check the value of filePath to make sure the user picked a path before
# attempting to write.
if filePath:
    system.file.writeFile(filePath, csv)
```

6. To test your scripts, put the Designer in **Preview Mode**, and press the **Button**. Open your **myExport.csv** file and check your data.

	A	B	C	D	E	F
10	84	0.56352293	1B827F36	FALSE	11/6/2017 9:25	
11	92	0.53782463	05D9E3D9	FALSE	11/6/2017 9:25	
12	13	0.17691666	66E05608	TRUE	11/6/2017 9:25	
13	91	0.90138406	23A77B71	FALSE	11/6/2017 9:25	
14	97	0.5005872	089B981E	TRUE	11/6/2017 9:25	
15	18	0.010637939	B98CAFF8	FALSE	11/6/2017 9:25	
16	55	0.18030751	2469BC91	TRUE	11/6/2017 9:25	
17	14	0.7256514	F252FB1D	FALSE	11/6/2017 9:25	
18	15	0.2902785	197496C1	TRUE	11/6/2017 9:25	
19	99	0.8383941	916B3310	FALSE	11/6/2017 9:25	
20	38	0.03875667	C8081AD1	FALSE	11/6/2017 9:25	

Importing Data From a CSV

There are several ways to import data from a CSV file. First, we could use **system.file.readFileAsString** to read the entire file as a string. Note, that this will read the file as is, meaning "\n" can be used to denote new lines.

Python-Using system.file.readFileAsString()

```
# Ask the user to find the CSV in the local file system.
path = system.file.openFile("csv")

# Use readFileAsString to read the contents of the file as a string.
# This string will be the parameter we pass to fromCSV below
stringData = system.file.readFileAsString(path)

# Split stringData into a List of strings, delimited by the new line character
stringData = stringData.split("\n")

# Iterate through the list, and do something with each line
for i in range(len(stringData)):

    # We're printing the row here, but you could do something more useful with the data.
    print stringData[i]
```

Alternatively, Python's CSV Library could be use to read in the contents of a CSV. In some cases, this is the easier approach, as the reader object is ready to be iterated over. Note, that this approach does read in each row as a List of strings:

Python - Using csv.reader()

```
# Import Python's built-in csv library
import csv

# Ask the user to find the CSV in the local file system.
path = system.file.openFile("csv")

# Create a reader object that will iterate over the lines of a CSV.
# We're using Python's built-in open() function to open the file.
csvData = csv.reader(open(path))

# Iterate through the reader object, and do something with each line
for row in csvData:

    # We're printing the row here, but you could do something more useful with the data.
    print row
```

Converting the Data into a Dataset

Once you've read in the contents of a CSV into a script, you may wish to move it elsewhere. It is not uncommon to move the data to a **Power Table**, or other components on the screen. The main difficulty with this is converting the CSV data into a dataset so that it fits into the **Power Table** component's **Data** property. We have a couple of approaches listed below.

Calling the `system.dataset.fromCSV` Function

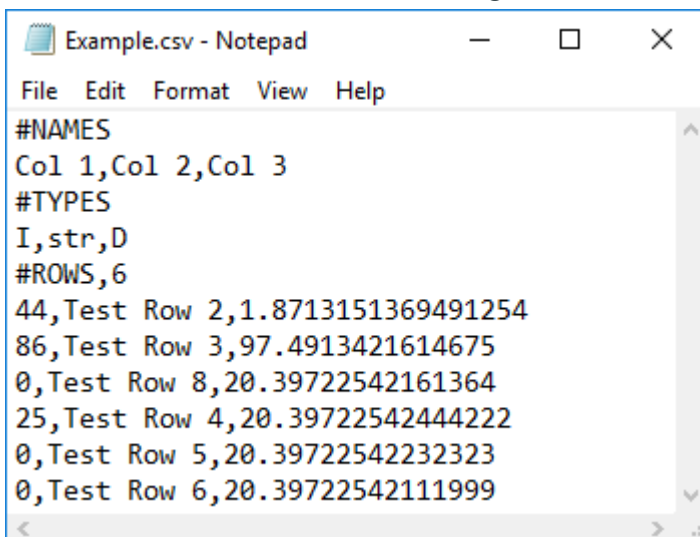
The `system.dataset.fromCSV` function can take a string and convert it to a dataset. Note, that the function expects a very specific format:

CSV File Content

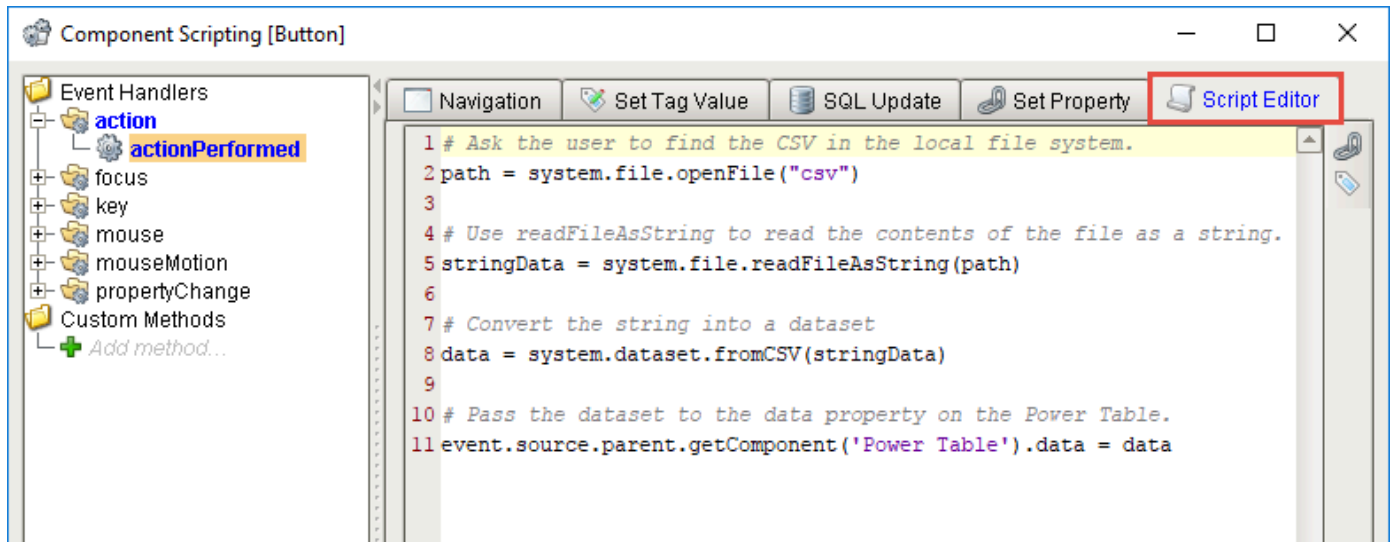
```
#NAMES
Col 1,Col 2,Col 3
#TYPES
I,str,D
#ROWS,6
44,Test Row 2,1.8713151369491254
86,Test Row 3,97.4913421614675
0,Test Row 8,20.39722542161364
25,Test Row 4,20.39722542444222
33,Test Row 5,20.39722542232323
62,Test Row 6,20.39722542111999
```

Example

1. Create a Text file on your local system named "**example.csv**."
2. Open the Text file, copy the contents of the "**CSV file Content**" box above, and paste the contents to the file. Save the changes to the **example.csv** file.



3. Let's move the contents of a CSV file to a Power Table. Add a **Power Table** and a **Button** component to your window. Double click on the **Button** component, and paste the following code into the **Script Editor** of the **actionPerformed** event:



Python - Using system.dataset.fromCSV()

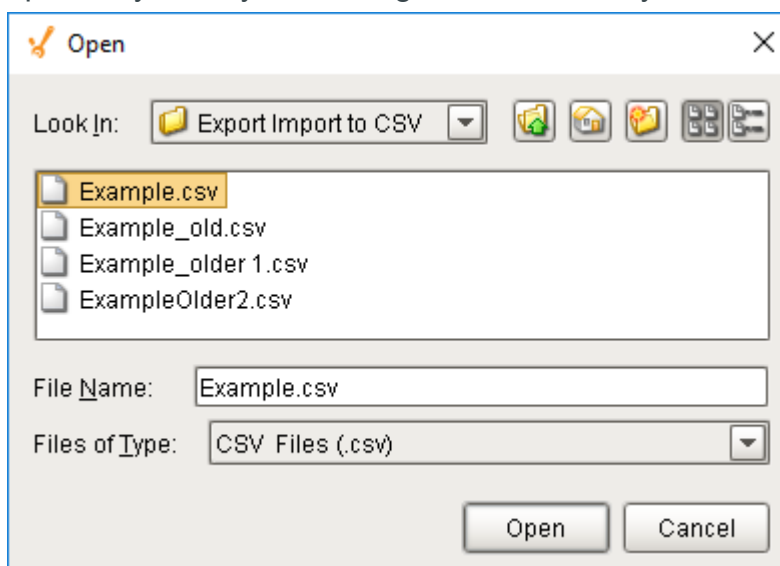
```
# Ask the user to find the CSV in the local file system.
path = system.file.openFile("csv")

# Use readFileAsString to read the contents of the file as a string.
stringData = system.file.readFileAsString(path)

# Convert the string into a dataset
data = system.dataset.fromCSV(stringData)

# Pass the dataset to the data property on the Power Table.
event.source.parent.getComponent('Power Table').data = data
```

4. To execute your script, put the Designer into **Preview Mode**, and press the **Button**. A window will open for you for you to navigate and choose your CSV file, then click Open.



5. Your data will be displayed in the Power Table as Shown below.

Col 1	Col 2	Col 3
44	Test Row 2	1.87
86	Test Row 3	97.49
0	Test Row 8	20.4
25	Test Row 4	20.4
33	Test Row 5	20.4
62	Test Row 6	20.4

Button

Calling the csv.reader Function

As mentioned, `system.dataset.fromCSV()` requires a specific format, which may not match the format of your file. In this case, we can use Python's CSV Library to parse the file and convert it to a dataset.

CSV File Content

```
Col 1,Col 2,Col 3
44,Test Row 2,1.8713151369491254
86,Test Row 3,97.4913421614675
0,Test Row 8,20.39722542161364
25,Test Row 4,20.39722542444222
33,Test Row 5,20.39722542232323
62,Test Row 6,20.39722542111999
```

Here is the code to import the above CSV data.

Python - Using Python's csv library

```
# Import Python's built-in csv library.
import csv

# Ask the user to find the CSV in the local file system.
path = system.file.openFile("csv")

# Create a reader object that will iterate over the lines of a CSV.
# We're using Python's built-in open() function to open the file.
csvData = csv.reader(open(path))

# Create a List of strings to use as a header for the dataset. Note that the
number
# of headers must match the number of columns in the CSV, otherwise an error will
occur.
# The simplest approach would be to use next() to read the first line in the file,
and
# store that as the header.
header = csvData.next()

# Create a dataset with the header and the rest of our CSV.
dataset = system.dataset.toDataSet(header ,list(csvData))

# Store it into the table.
event.source.parent.getComponent('Power Table').data = dataset
```

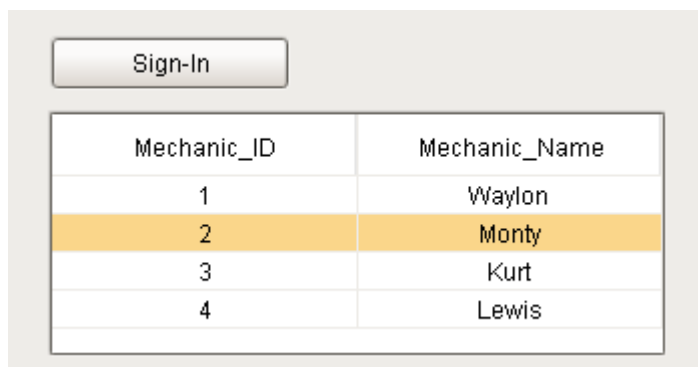
This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Read a Cell from a Table

After data has been populated in a **Table** or **Power Table** component, it may be useful to read or extract a particular cell from the Table, especially if users can select rows in the Table. On this page, we'll take a look at how to retrieve information from a particular cell on a Table.

The example on this page utilizes a simple **Power Table** and **Button** component. Users select a row from the Table to extract one of the cells from the highlighted row in the Power Table and press the Sign-In button.



Power Table Example

This example provides the code that you can use for selecting a single cell and multiple cells in a Table.

DATASETS IN TABLES

Before we get started, it is important to understand that a cell in a Table is actually a cell in a Dataset. Data in a Table is stored in a property on the component (the Data property), and the script needs to interact with that property.

Selecting a Single Cell

Retrieve a single cell requires that we reference the Power Table a couple of times in the same script. Because of this, we can create a variable that references the table, and use the variable later on. Be

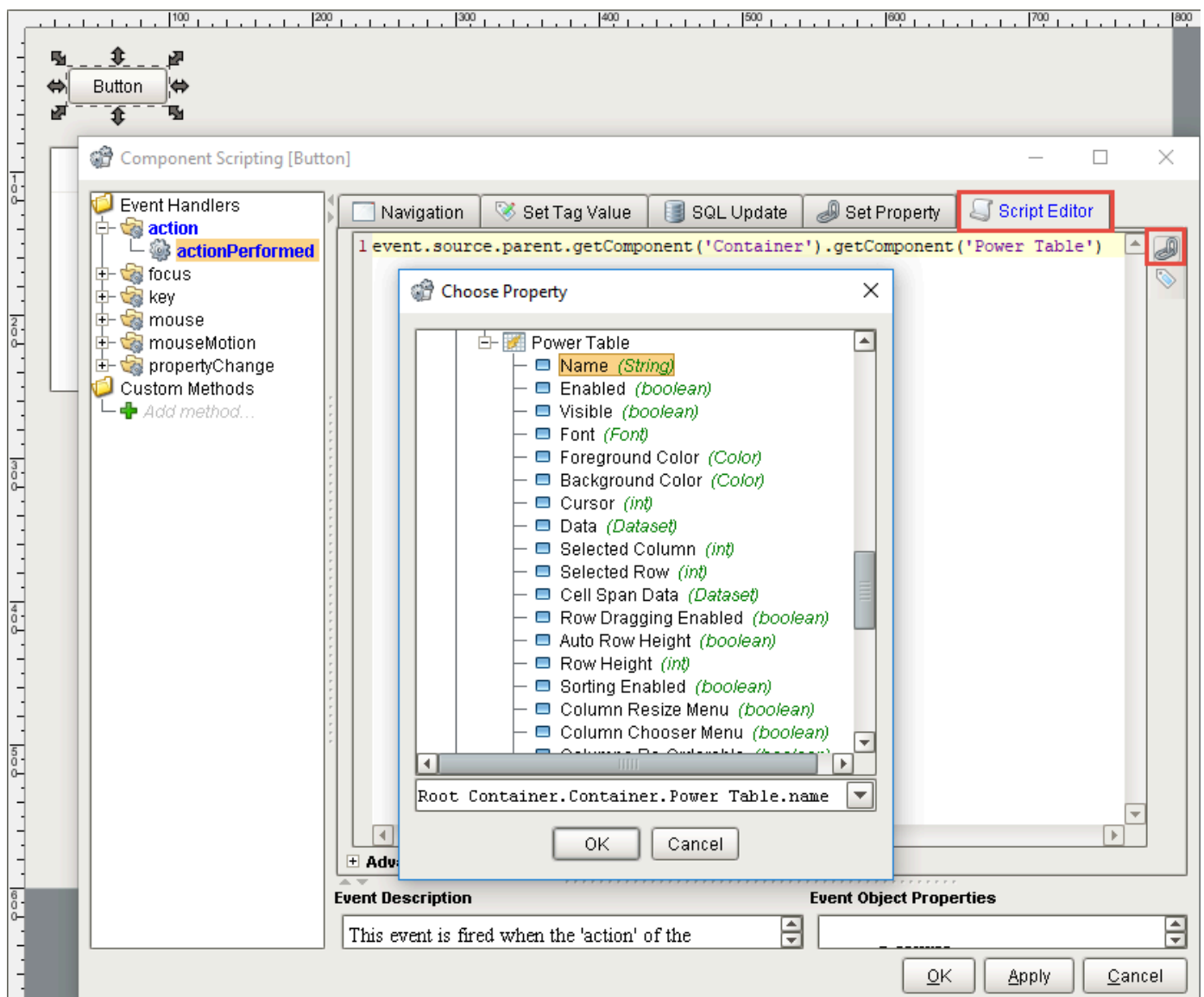
aware that a Button and Power Table can be in the same Container or separate Containers. If your Button and Power Table are in separate containers your path may differ.

For a Button and a Power Table that are in the same container, assuming that the Power Table has not been renamed, we can reference the Power Table component with the line of code below.

Python - Reference the Power Table

```
# Grab a reference to the Power Table.  
table = event.source.parent.getComponent('Power Table')
```

1. As mentioned above, if the Button and Table are in separate containers, the path will be different. Double click the Button component, and use the **Property Reference** button  to generate the path to a property, but not the component.
2. **Select** a property from the Power Table in the **Choose Property** window and click **OK**.
3. Remove the **".propertyName"** portion at the end of the script in the Script Editor, and click **OK**.



4. Next , we simply need to figure out which row contains the cell we want to read from. In our example, the user will select the row for us, so we simply need to know which row is selected

when the Button is pressed. Fortunately, the Power Table contains a **Selected Row** property that can be used to determine the row that is selected. Furthermore, we can use the [getValueAt\(\) function that is built into datasets](#).

Python - Reference the Name of the Column

```
# Here the "Mechanic_Name" argument references the name of the column.  
table.data.getValueAt(table.selectedRow, "Mechanic_Name")
```

5. Alternatively, we can use an integer as the last argument to specify the index of the column our cell is located in.

Python - Reference the Index of the Column

```
# Here the '1' references the index of the column in the Power Table's raw dataset  
(Data property).  
# Remember, indexes are zero-based, so this would retrieve the second column from  
the left.  
mechanicName = table.data.getValueAt(table.selectedRow, 1)
```

6. Lastly, we will need to account for scenarios where the user did not selected a row, otherwise, this will throw an exception. An if-statement can be used here to check for a -1 value on the Power Table's **Selected Row** property, and an else-clause can be used to notify the user that a row needs to be selected. Note that there is a property on the Power Table named **Selection Mode** that allows users to select multiple rows. By default it is set to only allow a single row to be selected. Change it to test your button.

Here is all the code that you'll need to read a single cell from a Power Table in the same container.

Python - Putting it all Together

```

# Grab a reference to the table.
table = event.source.parent.getComponent('Power Table')

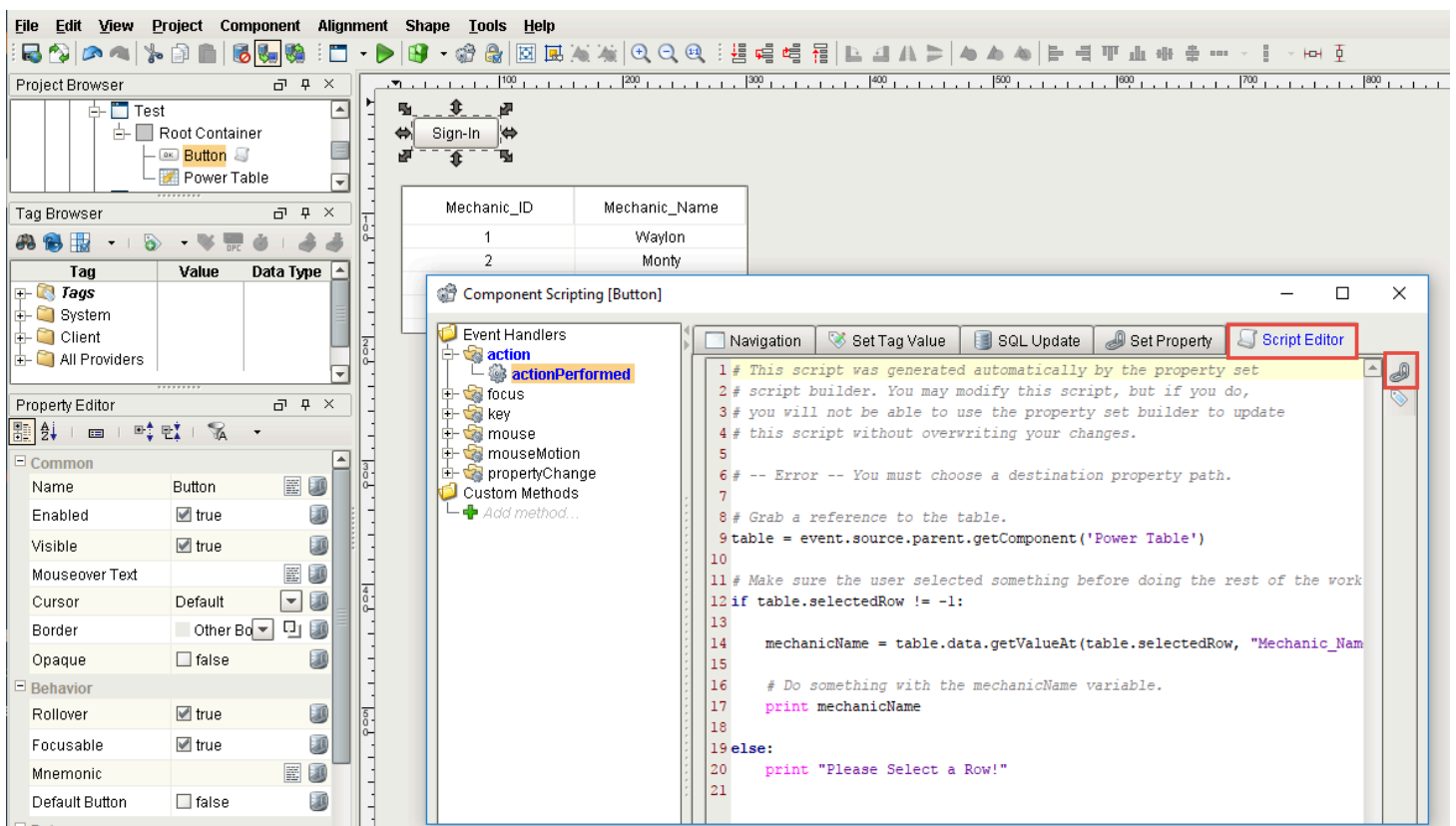
# Make sure the user selected something before doing the rest of the work.
if table.selectedRow != -1:

    mechanicName = table.data.getValueAt(table.selectedRow, "Mechanic_Name")

    # Do something with the mechanicName variable.
    print mechanicName

else:
    print "Please Select a Row!"

```



Selecting Multiple Cells - Same Column

By default, the Power Table allows for multiple rows to be selected. However, the Selected Row property only shows the row index for the first row selected. Fortunately, the **Power Table also has a built-in `getSelectedRows()` function** that can be used to return all of the indices. We simply need to iterate over each index. Note, we can still use the Power Table's **Selected Row** property to test if any rows are selected, but we could instead check the length of the object returned by `getSelectedRows()`:

Here is all the code you'll need to read multiple cells from a Table in the same container.

```
# Grab a reference to the table.
table = event.source.parent.getComponent('Power Table')

selectedIndices = table.getSelectedRows()

# If the length of selectedIndices is greater than 0, then at least one row is
selected.
if len(selectedIndices) > 0:

    for index in selectedIndices:

        mechanicName = table.data.getValueAt(index, "Mechanic_Name")

        # We can do something with the value as we iterate, or append to a list
and
        # do something with the entire list after the for-loop completes.
        print mechanicName

else:
    print "Please Select a Row!"
```

Test Your Script

Now you're ready to test your script.

1. Open the Output Console and select **Tools > Console** in the menubar.
2. In **Preview Mode**, select a row in the **Power Table**, and press the **Sign-In** button. You will see the selected `Mechanic_Name` displayed in the console.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Python Scripting

About Python

While it is entirely possible to create a complete and powerful project in Ignition without writing a line of script, many designers will find that in order to complete projects with specific requirements, they need to learn at least a little Python. In our experience, most industrial projects involve lots of very complex and specific requirements.

The good news is that learning **Python** is easy and **enjoyable**. Python is one of the most beautiful programming languages we've ever encountered. It is very easy to read - even if you don't know it at all, you will probably be able to understand a basic Python script. It is frequently called "executable pseudocode". If you find yourself doing a lot of scripting, you may want to pick up a basic **reference book** about Python, or go through a simple **online tutorial course**.

This section is a short tutorial specifically for python, which should help get you started. It goes over all of the core concepts you will need for scripting in Ignition but then next section (**Scripting in Ignition**) goes over using Python directly inside Ignition.

- **Python Variables, Datatypes, and Operators**: Learn what a variable is and how to create it, what the various datatypes are, and what operators you can use on them.
- **Conditions and Loops**: Learn the common if/else type of statements, as well as loops.
- **Functions**: Learn about the **built-in functions** that can do complex work with a simple command, as well as **user defined functions**, which help compartmentalize the code.
- **Libraries**: Learn about our built-in system functions, as well as pulling in outside libraries.

Python or Jython?

You'll often hear Python referred to as "Jython" by advanced users of Ignition. Python is the language, Jython is the implementation of the language that we use. Most users of Python use the implementation called "CPython" - they just don't realize it. See

[http://en.wikipedia.org/wiki/Python_\(programming_language\)#Implementations](http://en.wikipedia.org/wiki/Python_(programming_language)#Implementations).

One of the powerful things about using Jython is that your script has access to the entire Java standard library. In the Client, this will be Java 6 or above. When running under the Gateway, this will be Java 8 and above. For more information, see [Accessing Java](#).

Many scripting users are blown away by their script's speed. We can't take credit for this - the Jython engine hot-compiles (compiles the code when it is run) your Jython code to Java bytecode, which means it runs natively in the JVM, which in turn can hot-compile it to machine code. It's fast.

Which Version of Python is Used?

Ignition uses Jython 2.5. Jython is the Python programming language implemented over the Java Virtual Machine. When looking at outside documentation, such as on www.python.org, verify that you are looking at the correct version.

Jython 2.5 allows us to use the standard functions and tools in Python 2.5, so if you want to look up something in the Python docs, make sure to use version 2.5 (<https://docs.python.org/2/>).

Scripting Basics

Python is very easy to learn, and with some understanding of its basic syntax, you can get started making your own scripts.

Hello World

Let's get right to everyone's favorite example, "Hello World." The following script will print out "Hello World" to the [output console](#).

```
Python - Hello World
```

```
print "Hello World"
```

The print keyword is a handy tool in Python, allowing you to write text to the output console. This is useful for debugging your scripts. You can print multiple things by separating them with commas.

Variables

Variables are created by simply assigning a value to them. Variables do not need to be declared, because Python has a dynamic type system. That means Python figures out the type of the variable on the fly when the script is executed.

The following script would print out: 15

Python - Using Variables

```
x=5
y=3
print x*y
```

Quotations

Python makes limited distinction between single and double quotes. As long as they are used consistently then there are few times when the type of quotation mark you use matters. Some of the rules are shown here:

Python - Using Single and Double Quotes

```
print "This is my text"      # Using double quotation marks
print 'This is my text'      # Using single quotation marks
print "This is my text"      # This will not work because python does not allow
mixing the single and double quotation marks
print "My name is 'David'"    # This will print: My name is 'David'
print 'My name is "David"'    # This will print: My name is "David"
print 'My name is Seamus O\'Malley' # This will print: My name is Seamus O'Malley
```

Whitespace

Perhaps Python's most unique feature is logical blocks which are defined by an indentation in Python. A colon (:) starts a new block, and the next line must be indented (typically using a tab or 4 spaces). The block ends when the indentation level returns to the previous level. For example, the following will print out "5 4 3 2 1 Blast-off" with each value on a new line. The final print is not part of the while loop because it isn't indented.

Python - Logical Blocks / Indentation

```
countdown = 5
while countdown > 0:
    print countdown
    countdown = countdown - 1
print "Blast-off!"
```

Multi-Line Strings

You can start and end a section of comments with a triple quote (''' or ''').

Python - Multiple Lines of Comments Using a Triple Quote

```
'''
This is a lot of text
that you want to show as multiple lines of
comments.
Script written by Professor X.
Jan 5, 1990
'''

print 'Hello world'
```

Comments



INDUCTIVE
UNIVERSITY

Python Comments

[Watch the video](#)

Comments are a way to document your Python script. There are several ways to use comments, but the best advice we can give is to use them as much as possible! There are a few ways to make a comment in Python.

Individual Lines

You can start a line with a pound/hash (#) sign, or put one anywhere in a normal line of code.

Python - Document Scripts Using Comments

```
# this is a comment
print 'Hello world' # this is also a valid comment
```

Blocks of Lines

While Python doesn't explicitly have a way to block comment (comment out multiple lines), multi-line strings are functionally similar.

Comment Many Lines with a Keyboard Shortcut

In Ignition, you can use the Ctrl-/ keyboard shortcut to comment several lines of code at once. Just highlight one or more lines of code and hold the Ctrl key and press "/". This will prepend all of the selected lines of code with the pound/hash (#) sign. Press Ctrl-/ again to remove the pound/hash sign.

Control Flow

Control Flow statements, that is the ifs and loops, make the language do things differently based on the various conditions. Python has all of the basic control flow statements that you'd expect from a programming language.

If and Else

An if statement allows you to check if a condition is true or not true. Depending on the condition, you can either execute a block of code, or do something else. Many of these can be chained together to determine under what conditions should certain code execute.

Pseudocode - If Statement

```
if condition == True:  
    print value1
```

For and While

Looping can be done with a for, which executes a block of code a set number of times, or a while, which executes a block of code until a certain condition is met. Both can be incredibly useful.

Pseudocode - For Statement

```
for item in myList:  
    print item
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Variables, Datatypes, and Objects

Overview

Without incorporating any Ignition-specific objects, it is important to understand the basics of Python. This section seeks to introduce three main Python principles: Variables, Data Types, and Operators. Additionally, this page and the sub-pages herein will provide plenty of examples to get you started.

This section of the manual attempts to introduce these principles so that they may later be used in conjunction with Ignition. The information in this section is far from comprehensive, as [Python's official documentation](#) is a better reference for all things Python. However, this page serves as a great way to jump right into the action.

Variables

Variables are created by simply assigning a value to them. Variables do not need to be declared, because Python has a dynamic type system. That means, Python figures out the type of the variable on the fly when the script is executed.



INDUCTIVE
UNIVERSITY

Python Variables

[Watch the video](#)

The following script would print out: 15

Python - Declaring and Assigning Variables

```
x=5
y=3
print x*y
```

A space may be included on either side of the assignment operator ('='), but is not required. Thus, the following example would be functionality identical as the example above.

Python - Declaring and Assigning Variables with Spaces

```
x = 5
y = 3
print x * y
```

Built-in Datatypes

Python features several built-in datatypes. Below is an overview and links to appropriate pages where applicable.



INDUCTIVE
UNIVERSITY

Python Datatypes

[Watch the video](#)



INDUCTIVE
UNIVERSITY

Working with Different Datatypes

[Watch the video](#)

None

There is a special value in Python called `None` (with a capital N). This is simply a special value that means no value. This value is equivalent to Java's `null` value. `None` can be used to initialize a variable, but is best when checking to see if something exists before doing extra work:

Python - None

```
# If some value is not equal to None...
if something != None:
    # ...then do some work
    doWork()
```

Booleans

Python has two built-in values to represent true and false values: `True` and `False`, respectively (note the capital letters). These can be used when testing for truth, and are implicitly returned when the comparison operator is used:

Python - Booleans

```
# Prints True
print 1 == 1

# Prints False
print 1 == 0
```

When using Booleans, such as predicates in an if-statement, you typically don't have to use `True` or `False` directly. Instead, many other values are considered `True` or `False`. The following values are also considered `False`:

- `None`
- Numeric values of 0, such as 0
- Empty Sequences and Dictionaries, like `[]` or `{}`

Values aside from the ones mentioned above are considered `true`, so you can easily utilize the existence of a non-zero value as a `True`. Note that for integers both positive and negative numbers are `True`, only a value of 0 is `False`.

Strings

Literal **strings** can be typed in using either double quotes or single quotes. This can be handy when your string contains one quote or the other. You can also use the backslash character to escape special characters including these quotes. See the **Strings** page for more information.

Numeric Types

Numbers can just be typed in normally, like 42 or 3.14159 . Adding a decimal point differentiates an Integer from a Float. More information on Numeric types can be found on the **Numeric Types** page.

Colors

Working with colors in Python is remarkably easy. You can simply use any tuple of 3 or 4 integers to represent a color in RGB or RGBA. For example, to set a label's text color to red, you can simple do

something like this:

Python - Tuple as Color

```
label = event.source  
label.foreground = (255,0,0)  #(red,green,blue)
```

Additionally, the [system.gui.color](#) function allows you to pick a color in a similar fashion:

Python - System Function as Color

```
newColor = system.gui.color(255,0,0)
```

Lists and Tuples

Python offers a variety of sequence types: most notably Lists and Tuples. These are ordered collections, meaning they are indexed and the sorted order is maintained. More information on these types can be found on the [Lists and Tuples](#) page.

Python - System Function as Color

```
newList = [1,2]  
newTuple = (2,5,7)
```

Dictionaries

A Dictionary is a very useful type that holds a set of key-value pairs. Unlike sequences, they are not ordered, so there is no need to sort them. Instead you give each Value in the dictionary a Key, which handles as a reference to Value. You may have used these in other languages and know them as hashmaps, maps, associative memories, or associative arrays. More information on Dictionaries can be found on the [Dictionaries](#) page.

Python - System Function as Color

```
newDictionary = {"itemName":5}
```

JSON

JSON stands for JavaScript Object Notation. While it is not a datatype, it is a way of defining data in a human readable format, and is commonly used in many applications. It comes from the Javascript programming language, but it is language independent. Each JSON object contains lists and objects. A list is a series of ordered values separated by commas that is commonly used within Python. The object works like a **dictionary**, using any number of name/value pairs, where each value could be any basic datatype, a list, or even another object with its own name/value pairs. Because JSON is just a way of defining data, it can be used in many different programming languages, including Python. This makes it a useful tool for defining data in a way that humans can easily read.

Ignition also has two scripting functions that allow you to convert between a JSON string and a native Python object: **system.util.jsonEncode** and **system.util.jsonDecode**.

Official Documentation

For more information on JSON, see <http://www.json.org>.

Datasets

A **Dataset** is a multidimensional collection of values, stored in a manner similar to how values on a spreadsheet appear. Python does **not** natively have a Dataset type. Instead these datasets were created for use inside of Ignition. There are two types of datasets:

- Dataset: Sometimes called a "Standard Dataset", this type is commonly used on many Vision Components, such as the **Power Table** or **Chart**, to display multiple values simultaneously.
- PyDataset: Short for "Python Dataset", these datasets act in a manner very similar to a Python Sequence when it comes to accessing specific values, or iteration (see Lists and Tuples). Ignition's built-in system functions that interact with the database typically return a PyDataset. While there are two types of datasets, you can easily convert one type of dataset to the other. Additionally, you can easily create a dataset from a script.

Python - Creating a Dataset

```
header = ["The Only Column"]
rows = [[1],[2],[3]]

myDataset = system.dataset.toDataSet(header, rows)
```

Dates

Dates and times can be created in Python with the `datetime` and `time` libraries. However, the Ignition's built-in `system.date` functions can also be used instead without having to import either library.

Python - Show the Current Datetime

```
currentTime = system.date.now()

print "The current time is: %s" % currentTime
```

Using the In-Keyword

Python's `in` keyword can be used to check the contents of something for a specific instance of another object. One use is to check the contents of a string for a certain substring:

Python - Using In Keyword to Look for a Substring

```
myString = "Hello World"
subString = "World"

# Here we are using an if-statement to look for the substring. If the substring exists inside of myString, the expression returns True.
if subString in myString:
    # This will print out, because the substring "World" exists in the string "Hello World"
    print "The word 'World' appears in myString"
```

The `in` keyword can be used to look for a certain object inside of a sequence:

Python - Check for the Presence of an Object

```
myList = [1,2,3,4]

# If the integer 3 exists in myList, then this will print True, otherwise it will print False.
print 3 in myList
```

Python - Check for the Presence of an Object

```
# Check for the Administrator role.
if "Administrator" in system.security.getRoles():
    print "Administrator found"
```

Additionally, the `in` keyword can be used in an expression on a `while` loop:

Python - Check for the Presence of a Keyword in an Expression

```
myList = [1,2,3,4]

# As long as the integer 4 is in myList, the while loop will continue to iterate.
while 4 in myList:
    # pop(0) will remove the first element in the list
    myList.pop(0)
else:
    print "all done!"
```

Basic Operators

Python has many common operators as you would expect, or are at least familiar with if you've worked with other scripting languages.

These are just the basics. There are other operators, like bit shift operators and more. Read about them at: <http://docs.python.org/library/stdtypes.html>

Operator Reference

ARITHMETIC OPERATORS

In regards to Arithmetic Operators, the precision of the returned value depends on the data types of the arguments: including at least one **float** will return a **float**, otherwise, an **integer** is returned.

Operator	Meaning	Example	Output
+	Addition. Note that the data type of the returned object depends on the datatype of the arguments being used:	<pre>print 5 + 9 print 5.0 + 9</pre>	14 14.0
-	Subtraction	<pre>print 5 - 9</pre>	-4
*	Multiplication	<pre>print 5 * 9</pre>	45
/	Division	<pre>print 10 / 4.5</pre>	2.2222222222222223
//	Floored Division: will divide and return just the nearest integer value, even if dividing floats.	<pre>print 10 // 4.5</pre>	2.0
%	Modulo: returns just the remainder of the dividend (the left argument) after being divided by the right argument. In the example on the right, the 4.5 divides evenly twice, leaving 1.0 as a remainder.	<pre>print 10 % 4.5</pre>	1.0

Operator	Meaning	Example	Output
**	Power: raise the the number on the left to the power of the number on the right.	<pre>print 5 ** 3</pre>	125

Boolean Operators

Operator	Meaning	Example	Output
or	Returns True if either argument is True. If both are False, then returns False.	<pre>x = False y = True print x or y</pre>	True
and	Returns True only if both arguments are True. Otherwise, returns False.	<pre>x = False y = True print x and y</pre>	False
not	Returns a boolean value that represents that opposite value of the trailing expression: False becomes True, and True becomes False.	<pre>x = True print not x</pre>	False

Comparison Operators

Operator	Meaning	Example	Output
<	Less than	<pre>print 10 < 5 print 10 < 10</pre>	False False
<=	Less than or equal to	<pre>print 10 <= 5 print 10 <= 10</pre>	False True
>	Greater than	<pre>print 10 > 5 print 10 > 10</pre>	True False
>=	Greater than or equal to	<pre>print 10 >= 5 print 10 >= 10</pre>	True True
==	Equal	<pre>print 10 == 10 print 10 == 0</pre>	True False
!=	Not equal	<pre>print 10 != 10 print 10 != 0</pre>	False True
is	Returns True if both arguments are referring to the same object, otherwise, returns False.	<pre>x = 10 y = 5 print x is y</pre>	False

Operator	Meaning	Example	Output
is not	Returns True if both arguments are referring to different objects.	<pre>x = 10 y = 5 print x is not y</pre>	True

Order of Operations

Note that there is an order of operations for Arithmetic and Boolean operators that can be modified with parenthesis. For arithmetic they are grouped in tiers and evaluated left-to-right within a given tier:

- Parenthesis
- Exponents
- Multiplication, Division, Modulo
- Addition, Subtraction

Python - System Function as Color

```
6+10*2    # produces 6+(10*2) = 26
6*10+2    # produces (6*10)+2 = 62

10*6%2    # produces (10*6)%2 = 0
10%6*2    # produces (10%6)*2 = 8
```

For Boolean operators:

- Not
- And
- Or

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Numeric Types

The Differences Between the Types

Within Python, there are a few numeric types that we will often use within the scope of Ignition: integers, floats and booleans. **Integers** have at least 32 bits of precision and consist of non decimal values (1, 2, 50, 246). When working with larger numbers, Python also supports a **long** type that has unlimited precision. **Floats** consist of numbers with decimal values (1.0, 2.4, 50.7, 246.8734). **Booleans** are a unique subset of integers. They are either **True** or **False**, but can also be represented using a 1 or 0, respectively.

Numbers in Arithmetic

When using numeric values in expressions or equations, Python is very flexible in how it handles multiple types in the same operation. When using two of the same type in an expression, the outcome will always be of that type. Adding two integers together yields an integer, while multiplying two floats will yield a float. With division, using integers will always result in a floor division so that the answer remains an integer. Floor division will simply remove the decimal, and will not round up at any point.

When using different types, the more precise type will be used for the outcome. Doing an operation with one integer and one float will produce an outcome that is a float. Adding a float and an integer together will yield a float.

Below, we have some examples that show off some of the different ways that Python handles expressions. You can try these out in the interactive interpreter of the [Script Console](#).

Type of Arithmetic	Description	Example	Output
Addition and Subtraction	Adding/Subtracting integers will return an Integer. If at least one of the numbers is a float, then the resulting number will also be a float. Otherwise the result will be an integer	<code>4 + 5 # int + int = int</code> <code>4.0 + 5 # float + int = float</code> <code>4.0 + 5.3 # float + float = float</code>	9 9.0 9.3
Multiplication	Similar to Addition and Subtraction, multiplication will only return a float if at least one of the numbers is a float	<code>4 * 5 # int * int = int</code> <code>4 * 5.0 # int * float = float</code>	20 20.0
Division	Dividing an integer by another integer results in floor division: meaning the remainder of division is never returned. Note that this is not the same as rounding to the nearest integer. To include the remainder, at least one of the numbers needs to be a float	<code>4 / 5 # int / int = int</code> <code>4 / 5.0 # int / float = float</code>	0 0.8

Operators and Functions

Python offers a wide range of operators that can be used to perform calculations on numeric values. Additionally, there are many functions that can perform specific operations to numeric values. The tables below contain some of the most common operations and functions, but this is not an exhaustive list.

Operators

Operator	Description	Example	Types that use the Operator
+	Used to find the sum of two numeric values. Can also be used to concatenate string values.	$5 + 1 = 6$ "Hello " + "World"	Numeric Types, Strings
-	Used to find the difference of two values	$5 - 1 = 4$	Numeric Types
*	Used to find the product of two values	$5 * 2 = 10$	Numeric Types
/	Used to find the quotient of two values	$4 / 2 = 2$	Numeric Types
%	Modulus operator. When used with numeric types, returns the remainder of the quotient of two values The operator is also seen in string formatting.	$5 \% 2 = 1$	Numeric Types, Strings
**	Used to find a value to the power of another value	$5 ** 2 = 25$	Numeric Types

Functions

Function	Description	Example
int(x)	Converts x to an integer.	int(4.8) = 4
long(x)	Converts x to a long.	long(5.7) = 5
float(x)	Converts x to a float.	float(4) = 4.0
abs(x)	Absolute value of x.	abs(-4.6)=4.6
round(x[, n])	Rounds x to n digits. If n is not specified, default is 0.	round(5.674)=6.0 round(5.674, 2)=5.67

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Strings

What are Strings?

In Python, as well as most other programming languages, a string is a grouping or string of characters that are used literally instead of as names of variables or other objects. In Python, they can be enclosed in either single quotes or double quotes:

Python - Print Statements

```
# These two print statements will both print out a sentence that looks the same as the other.  
print 'These will both print the same!'  
  
print "These will both print the same!"
```

Strings are also considered sequences, which means they can be iterated through much like a list.

String Escape Character

Strings use a special character called an "escape" character to denote when something different should be happening within the string. In Python, the escape character is the backslash (\). The escape character is mainly used in a few different ways:

Comments

Here, we use the escape character to let Python know we want to ignore the second single quote and directly print it. This works with either single or double quotes.

Alternately, we can enclose the string in the opposite quotes that we are using in the string. Because we enclose the string in double quotes here, the single quote prints normally.

Both of the following examples will result in `doesn't`:

```
print 'doesn\'t'

print "doesn't"
```

This also works in reverse, where we can enclose the string in single quotes and use double quotes within. We can also use the escape character with double quotes enclosing the string.

For example, both of the following print statements will result in `"Test," he said.:`

```
print '"Test," he said.'
```

```
print "\"Test,\" he said."
```

New Lines and Tab Spacing

A new line may be specified with the escape character and `"n"`. Alternatively, a multi-line string may be used to add line breaks to a string literal.

```
print "First line\nSecond line"
```

Results in:

```
First line

Second line
```

Tab-spacing may be added to a string literal with the escape character and `"t"` `print "Hello \tWorld"` results in `Hello World`.

Specifying a Backslash

If you need to place a backslash inside of a string literal, simply use the escape character twice.

For example, `print "Folder\\file.txt"` results in `Folder\file.txt`.

Multi-Line Strings

If a string literal should contain multiple lines, you can use a multi-line string, which is created by using three quotation marks.

Example	Output
<pre>print """This is on multiple lines"""</pre>	<pre>This is on multiple lines</pre>

This is useful when using the `system.db` functions, as full SQL queries are typically passed to these functions, so using a multi-line string allows you to write the query in an easy to read format.

Python - Multi-line String

```
query = ''' SELECT * FROM myTable
          WHERE id < 10'''

system.db.runPrepQuery(query,[])
```

Raw Strings

Sometimes, it is necessary to print the raw string without allowing escaped characters. This is done by placing the letter "r" in front of the string.

Python - Raw String

```
print r'This string \n will print out directly as written.'
```

This is especially useful in cases where a file path needs to be hard coded as a string literal:

Python - File Path as a Raw String

```
# Specifying a Windows file path
myPath = r"C:\Folder\Another Folder\file.txt"
print myPath

# Specifying a Linux file path
myPath = r"/home/Directory/Another Directory/file.txt"
print myPath
```

Unicode Strings

Strings that contain characters beyond 7-bit ASCII, such as é or щ need to be marked as unicode strings by placing the letter u in front of the string.

Python - Unicode String

```
print u'été'
```

Combining Strings

Two different string type variables can actually be combined or concatenated using the plus (+) sign. It is important to understand that they are concatenated exactly.

Python - Concatenated String Type Variables

```
a = "this"  
b = "that"
```

```
# Will print 'thisthat' because there was no space at the end of a or the  
beginning of b.  
print a + b
```

```
# Will print 'this that' because we added a space between them.  
print a + " " + b
```

String Indexing

Strings in Python are actually indexed, with the first character having an index of 0. To grab a value at a specific index, you place a value within square brackets ([]) after the string variable.

Python - Indexed Strings

```
a = "Ignition"

# Will print out 'I', since it is in the zero position.
print a[0]

# Will print out 't', since it is in the fourth position.
print a[4]
```

You can also use a negative index, which will start from the right side of the string with the last character having a index value of -1. This is useful for getting the last character when you aren't sure how long the string is.

Python - Negative Indexed String

```
a = "Ignition"

# Will print out 'o', since it is the second to last character.
print a[-2]
```

Slicing Strings

Using the string index values, we can actually grab slices or parts of the string. Similar to grabbing an individual character, we place two values separated by a colon within square brackets. You can think of these numbers as the slices between characters, with 0 before the first character, 1 before the second, and so on. It should look like this: string[4:7] The first value is the start location, while the second value is the end location.

Python - Sliced Strings

```
a = "Inductive Automation"

# Will print out 'ctive Auto'. Note how the space is counted as a character,
# and the 14th location is before the m, which is not included.
print a[4:14]
```

If left blank, the first value will default to 0, while the second value will default to the length of the string. Additionally, negative values can be used as well, just like with the index.

Python - Negative Value Sliced Strings

```

a = "Inductive Automation"

# From index 0 to index 12.
print a[:12] # Will print out: Inductive Au

# From index 12 to the end of the string.
print a[12:] # Will print out: tomaton

# From the 4th character from the end to the 5th index. Note this prints nothing
because the start character is after the end character.
print a[-4:5] # Will print out:

# From the 8th character from the end to the 2nd character from the end.
print a[-8:-2] # Will print out: tomati

# From index 5 to the 5th character from the end.
print a[5:-5] # Will print out: tive Autom

```

You can use all of these together (and with integer variables or functions) to have a very flexible way to format strings.

String Formatting

Also known as string substitution, it allows you to enter in values into a string, similar to having a variable inside of the string. This is useful when doing a database query where you can build the query as a string, and then add in the query parameters when executing the query.

Doing this requires the use of the percent (%) sign followed by the parameter type within the string. After the string, you then use the percent sign followed by the parameter values within parentheses in order.

Python - String Substitution

```

a = "I have %i apples."

# Will print exactly as written, because we are not substituting any values in.
print a

# Will print with an 8 substituted in for the %i. I have 8 apples.
print a % (8)

```

Instead of using literal values, you can also use variables as parameter values instead. This is useful when pulling the value from a function. Each string can also have multiple values substituted, as long as the parameters are in the order that they are in the string.

Python - String Substitution using Variables as Parameters

```
apples = 10
oranges = 14
peaches = 5

a = "I have %i apples, %i oranges, and %i peaches."

# Will print with the variable values substituted in for the %i. I have 10 apples,
14 oranges, and 5 peaches.
print a % (apples, oranges, peaches)
```

There are a few different types used with string formatting, the most common of which are listed here. The character values are what go after the percent sign.

Type	Character
Signed Integer	i
Floating Point Decimal Format	f
Floating Point Exponential Format	e
String	s

String Search Example

You can bring all this together for a simple way to print out just the last word in a string.

Python - Find the Last Word of a String

```
# Find just the last word of a string.
myString = "Inductive Automation"

lastSpaceIndex = myString.rfind(" ")

# Add one to not include the space.
print myString[lastSpaceIndex+1:] # Will print out: Automation
```

String Functions

Python strings have many functions available that can manipulate the string or give information on the string. The most common are in the table below.

Function	Description	Example
len(string)	Returns the length of the string .	<pre>a = "Inductive Automation" print len(a)</pre>
x in string	Will return True if x is within the string, False if not. Can also be used to iterate through the string.	<pre>string = "Inductive Automation" if 'd' in string: print "There is a d in the string" for letter in string: print letter</pre>
string.find(x, [, start[, end]])	Returns the first location of the substring x from the string. Returns -1 if the substring is not found.	<pre>string = "Inductive Automation" print string.find(" ")</pre>
string.rfind(x, [, start[, end]])	Similar to find, but returns the last location of the substring x . Returns -1 if not found.	<pre>string = "Inductive Automation"</pre>
string.upper()	Returns a copy of the string with all characters uppercase. This is useful when comparing user input to a string value, as the user may use a different case for certain letters.	<pre>userInput = "administration"</pre>
string.lower()	Returns a copy of the string with all characters lowercase. This is useful when comparing user input to a string value, as	<pre>serInput = "ADminIStration"</pre>

Function	Description	Example
	the user may use a different case for certain letters.	
string.capitalize()	Returns a copy of the string with the first character capitalized and all other characters lowercase.	<pre>string = "here is my sentence"</pre>
string.title()	Returns a copy of the string with the first letter of each word capitalized and all other characters lowercase.	<pre>string = "here is my sentence"</pre>
string.strip([x])	Returns a copy of the string with leading and trailing characters removed, where x is the string of characters.. If x is omitted, then removes whitespace from the leading and trailing edges of the string. <code>lstrip()</code> and <code>rstrip()</code> may instead be used to strip characters from the leading or trailing edge	<pre>string = " This string has leading and trailing spaces" print string print string.strip() fencedString = "+++++My String+++++" print fencedString.strip("+") print fencedString.lstrip("+") print fencedString.rstrip("+")</pre>
string.count(x[, start[, end]])	Returns the number of occurrences of x in the string. A start and end can be specified that will limit the count to that area.	<pre>string = "Inductive Automatic Control" print string.count('i') print string.count('i', 4, 9)</pre>

Function	Description	Example
<code>string.split([delimiter[,maxsplit]])</code>	Returns a list of the words in the string. Optionally, specifying a delimiter will split the string on the delimiter string. Specifying a maxsplit will split the string a maximum number of times, with the remainder of the string as the final list object. The number of items in the list will not be more than <code>maxsplit + 1</code>	<pre>sentence = "This is an example sentence" print sentence.split() dashedSentence = "Yet-another sentence" print dashedSentence.split('_')</pre>
<code>string.rsplit([delimiter[,maxsplit]])</code>	Similar to <code>split</code>, but splitting is performed from right to left. Thus if maxsplit is smaller than the total number of delimiter in the string, only the rightmost words will be split off as separate items.	<pre>sentence = "This is an example sentence" print sentence.rsplit() dashedSentence = "Yet-another sentence" print dashedSentence.rsplit('_')</pre>
<code>string.join(x)</code>	Returns a copy of the string that is the concatenation of the strings in the iterable x. The string calling the join is what will separate the values of the iterable.	<pre>a = '-' b = 'abcdef' print a.join(b)</pre>
<code>string.replace(old, new[, count])</code>	Returns a copy of the string where occurrences of the old substring are replaced with the new substring. Optionally, if count is specified, will	<pre>string = "Ignition is good!" print string.replace("good", "great")</pre>

Function	Description	Example
	only replace the first count number of occurrences.	
...		

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Lists and Tuples

Sequences

Like **Strings**, Python has two other common sequence types: Lists and Tuples. While lists and tuples are similar to strings in that they share many of the same functionality, they are unique in that they are used to group other values together. Both lists and tuples define a number of values separated by commas, but lists are enclosed in square brackets `[ ]` and are mutable, meaning their contents can change, while tuples are enclosed in parentheses `( )` and are immutable, meaning their contents can't change.

Lists

As stated above, lists are groups of comma separated values enclosed in square brackets, but can utilize many of the features available to other sequences like strings.

Python - Lists

```
# Lists are very simple to create.
```

```
myList = [1, 2, 3]
```

```
print myList # Will print out: [1, 2, 3]
```

```
# Empty lists can also be created, to have items added to them later.
```

```
myList = []
```

```
# Lists are not confined to hold values of a single data type either.
```

```
myList = [1, "hello", 3.3]
```

```
# Lists can even hold other lists! In this case, myList would actually hold 5 elements:
```

```
# 1, a list, 4, 'that', and another list. The last list also contains a list as well.
```

```
myList = [1, ["this", 3.3], 4, 'that', [6, [7.7, 'other'], 9]]
```

List Concatenation

Like strings, lists also support concatenation and can be combined to form a single larger list.

Python - List Concatenation

```
a = [1, 2, 3]
b = ['four', 'five', 'six']

print a + b # Will print out [1, 2, 3, 'four', 'five', 'six']
```

List Indexing

Like strings, lists are also indexed, which allows you to grab single values or splice them to get ranges of values. Just like everything else in Python, lists start with 0.

Python - List Indexing

```
myList = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

print myList[3] # Will print out: d

print myList[-2] # Will print out: f

print myList[2:5] # Will print out: ['c', 'd', 'e']
```

When trying to index a nested list, we simply need to add a second index after the first.

Python - Index a Nested List

```
myList = [1, 2, [3, [4, 5, 6, 7], 8], 9, 10]

print myList[2][1][3] # Will print out: 7
```

Appending to Lists

You can add values directly to the end of a list with the `append()` function. You can add anything that is normally allowed in a list.

Python - Appending to a List

```
myList = []  
  
myList.append('Hello')  
myList.append('World')  
  
print myList # Will print out ['Hello', 'World']
```

List Functions

Below is a list of common list functions. Some of them are similar to other sequences like strings, while others are unique to lists because lists are mutable.

Function	Description	Example	Output
len(list)	Returns the length of the list .	<pre>list = [1, 2, 3, 4, 5] print len(list)</pre>	5
x in list	Will return True if x is within the list, False if not. Can also be used to iterate through the list.	<pre>list = [1, 2, 3, 4, 5] if 4 in list: print "There is a 4 in the list" for value in list: print value</pre>	There is 1 2 3 4 5
list.index(x)	Will return the index number of item x . Throws an error if item is not found.	<pre>list = [1, 2, 3, 4, 5] print list.index(3)</pre>	2
min(list)	Returns the smallest item of list .	<pre>list = [1, 2, 3, 4, 5] print min(list)</pre>	1
max(list)	Returns the largest item of list .	<pre>list = [1, 2, 3, 4, 5] print max(list)</pre>	5
list.count(x)	Will return the number of times x	<pre>list = [1, 2, 3, 4, 5, 4] print list.count(4)</pre>	2

Function	Description	Example	Output
	appears in the list.		
list.append(x)	Will add x to the end of the list.	<pre>list = [1, 2, 3, 4, 5] list.append(6) print list</pre>	<pre>[1, 2, 3, 4, 5, 6]</pre>
list.insert(i, x)	Will insert x at position i .	<pre>list = [1, 2, 3, 4, 5] list.insert(1, 5) print list</pre>	<pre>[1, 5, 2, 3, 4, 5]</pre>
list.remove(x)	Will remove the first x from the list.	<pre>list = [1, 2, 3, 2, 4, 5] list.remove(2) print list</pre>	<pre>[1, 3, 2, 4, 5]</pre>
list.pop([i])	Will remove the item at index i , and return it. If no index is specified, it will remove and return the last item in the list.	<pre>list = [1, 2, 3, 4, 5] list.pop(2) print list list.pop() print list</pre>	<pre>3 [1, 2, 4, 5] 5 [1, 2, 4]</pre>

Function	Description	Example	Output
<code>list.reverse()</code>	Will reverse the items in the list.	<pre>list = [1, 2, 3, 4, 5] list.reverse() print list</pre>	<code>[5, 4, 3]</code>

Tuples

Tuples look similar to lists in that they are defined as a group of comma separated values, but they are enclosed by parenthesis and they are immutable like strings, meaning they can't be altered. Besides that, they are sequences, like lists and strings. This means that tuples have the functionality that other sequences have, such as concatenation, indexing and slicing, and even nesting.

Python - Tuples

```
# Tuples are very simple to create.
myTuple = (1, 2, 3)

print myTuple # Will print out: (1, 2, 3)

# Empty tuples can also be created, to have items added to them later.
myTuple = ()

# Like lists, tuples are not confined to hold values of a single data type either.
myTuple = (1, "two", 3.3)

# Tuples can even hold other tuples!
myTuple = (1, ("two", 3.3), 4, 'five', (6, (7.7, 'eight'), 9))

a = (1, 2, 3)
b = (4, 5, 6)

# Combine two tuples to make a new tuple
print a + b # Will print out: (1, 2, 3, 4, 5, 6)

myTuple = ('a', 'b', 'c', 'd', 'e', 'f', 'g')

print myTuple[2:5] # Will print out: ('c', 'd', 'e')
```

NOTE

Lists and tuples can also be nested within each other!

Tuple Functions

All of the functions that work on tuples work on lists, so these should look familiar. However, because they are immutable, not all of the list functions work on tuples. Check out the common tuple functions below.

Function	Description	Example	Output
len(tuple)	Returns the length of the tuple .	<pre>tuple = [1, 2, 3, 4, 5] print len(tuple)</pre>	5
x in tuple	Will return True if x is within the tuple, False if not. Can also be used to iterate through the tuple.	<pre>tuple = [1, 2, 3, 4, 5] if 4 in tuple: print "There is a 4 in the tuple" for value in tuple: print value</pre>	There is 1 2 3 4 5
min(tuple)	Returns the smallest item of tuple .	<pre>tuple = [1, 2, 3, 4, 5] print min(tuple)</pre>	1
max(tuple)	Returns the largest item of tuple .	<pre>tuple = [1, 2, 3, 4, 5] print max(tuple)</pre>	5

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Dictionaries

Mapping Type

A Dictionary is a mapping object. Where sequences are indexed with a numeric index value, dictionaries are indexed using keys. These keys then have a matching value pair that is associated with a particular key. For example, with a list we can extract the object at index 0, which we may have decided is the name, whereas with dictionaries, I can instead extract the a value using a key "name". Because of how they work, dictionaries are sometimes known as associative arrays in other programming languages.

Dictionaries are created using braces `{ }`, where each key/value pair is separated by a comma `( , )` and keys are separated from their values using a colon `( : )`. In the example below, I created a dictionary with two keys: name, id.

Python - Creating a Dictionary

```
# In this dictionary, I associated the name John Smith to the key "name",  
# and the id number 12345 to the key "id".  
myDictionary = {"name": "John Smith", "id": 12345}
```

Using a Dictionary

The keys in a dictionary can be numbers, strings, or tuples, but typically a string is used to make a key that best describes the value. Any given key may only appear once in a dictionary, so trying to set another value for a key that already exists will overwrite the previous value for that key. Alternately, attempting to access the value of a key that does not exist will throw an error, while setting a value to a key that does not exist will create a new key/value pair within the dictionary.

To access a value in a dictionary works much like accessing a value in a list; simply place brackets containing the key after the dictionary object.

Python - Accessing Values in a Dictionary

```
# Creates a dictionary with three key/value pairs.
myDictionary = {'Bob': 89.9, 'Joe': 188.72, 'Sally': 21.44}

print myDictionary['Joe'] # Will print out: 188.72

# Adds a key for 'Amir', and alters the value associated with the key 'Sally'.
myDictionary['Amir'] = 45.89
myDictionary['Sally'] = 146.23

print myDictionary # Will print out the whole dictionary: {'Joe': 188.72, 'Amir': 45.89, 'Bob': 89.9, 'Sally': 146.23}
```

It is also easy to loop through all of the values of a dictionary using the `keys()` function. For example:

Python - Keys Function

```
# The keys() function provides us with a list of keys, which we can iterate through and print out in addition to using in the value lookup.
for key in myDict.keys():
    print key, myDict[key]
```

There are many use cases for dictionaries, but they are commonly used in Ignition when passing values into a **Message Handler** or **creating a dynamic roster** for alarms.

Dictionary Functions

Dictionaries have a few functions that allow for greater control over the dictionary object and the values contained within.

Function	Description	Example
dictionary.clear()	Remove all of the items in the dictionary.	<pre> myDictionary = {"name":"John Smith", "id":12345} myDictionary.clear() print myDictionary </pre>
dictionary.keys()	Returns a list of the dictionary's keys.	<pre> myDictionary = {"name":"John Smith", "id":12345} print myDictionary.keys() </pre>
dictionary.values()	Returns a list of the dictionary's values.	<pre> myDictionary = {"name":"John Smith", "id":12345} print myDictionary.values() </pre>
len(dictionary)	Returns the number of items in the dictionary.	<pre> myDictionary = {"name":"John Smith", "id":12345} print len (myDictionary) </pre>
del dictionary[key]	Will remove the named key.	<pre> myDictionary = {"name":"John Smith", "id":12345} del myDictionary["id"] print myDictionary </pre>
key in dictionary	Will return True if the dictionary has that key. Can also use "key not	<pre> myDictionary = {"name":"John Smith", "id":12345} if "name" in myDictionary: print myDictionary["name"] </pre>

Function	Description	Example
	in dictionary"	

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Datasets

Datasets and PyDatasets

A Dataset can be thought of as a two dimensional list, or rather a list where each object is another list of objects. Datasets are not normally native to Python, but are built into Ignition because of their usefulness when dealing with data from a database. It is very common to deal with datasets in scripting, as datasets power many of the interesting features in Ignition, like charts and tables.

The main confusion when dealing with datasets is the difference between the Dataset object and the PyDataset object. Dataset is the kind of object that Ignition uses internally to represent datasets. When you get the data property out of a component like a Table, you will get a Dataset. The PyDataset is a wrapper type that you can use to make Datasets more accessible in Python. The biggest differences are seen in how we access the data in the two different objects. However, you can easily convert between the two with [system.dataset.toDataSet](#) and [system.dataset.toPyDataSet](#), making it simple to use the object that you find easier to use.

Creating Datasets

Because Datasets are not native to Python, there is no way to naturally create them within scripting. Instead they must be created using the [system.dataset.toDataSet](#) function, which also allows you to convert a PyDataset to a Dataset. It requires a list of headers and a list of each row's data. Each data row list must be the same length as the length of the headers list.

Python - Creating a Dataset

```
# First create a list that contains the headers, in this case there are 4 headers.
headers = ["City", "Population", "Timezone", "GMTOffset"]

# Then create an empty list, this will house our data.
data = []

# Then add each row to the list. Note that each row is also a list object.
data.append(["New York", 8363710, "EST", -5])
data.append(["Los Angeles", 3833995, "PST", -8])
data.append(["Chicago", 2853114, "CST", -6])
data.append(["Houston", 2242193, "CST", -6])
data.append(["Phoenix", 1567924, "MST", -7])

# Finally, both the headers and data lists are used in the function to create a
Dataset object
cities = system.dataset.toDataSet(headers, data)
```

CAUTION

All code snippets on this page will reference the cities dataset we created above, so place that code at the beginning of every code snippet.

Accessing Data in a Dataset

To access the data inside of a dataset, each dataset has a few functions that can be called on to access different parts of the dataset. These are listed in the table below.

Function	Description	Example
data.getColumnCount()	Returns the number of columns in the dataset.	<pre>print cities.getColumnCount()</pre>
data.getColumnIndex(colName)	Returns the index of the column with the name colName.	<pre>print cities.getColumnIndex("Timezone")</pre>
data.getColumnName(colIndex)	Returns the name of the column at the index colIndex.	<pre>print cities.getColumnName(1)</pre>
data.getColumnNames()	Returns a list with the names of all the columns.	<pre>print cities.getColumnNames()</pre>
data.getColumnType(colIndex)	Returns the type of the column at the index colIndex.	<pre>print cities.getColumnType(3)</pre>
data.getColumnTypes()	Returns a list with the types of all the columns.	<pre>print cities.getColumnTypes()</pre>

Function	Description	Example
<code>data.getRowCount()</code>	Returns the number of rows in the dataset.	<pre>print cities.getRowCount()</pre>
<code>data.getValueAt(rowIndex, colIndex)</code>	Returns the value at the specified row and column indexes.	<pre>print cities.getValueAt(1, 2)</pre>
<code>data.getValueAt(rowIndex, colName)</code>	Returns the value at the specified row index and column name.	<pre>print cities.getValueAt(2, "Population")</pre>

Looping Through a Dataset

Oftentimes, you need to loop through the items in a dataset similar to how you would loop through a list of items. Using the functions above, this becomes possible to do.

```
# We use the same cities dataset from above. Using the range function, we can come up with a range of values that represents the number of columns.
for row in range(cities.getRowCount()):
    for col in range(cities.getColumnCount()):
        print cities.getValueAt(row, col) # Will print out every item in our cities dataset, starting on the first row and moving left to right.
```

Accessing Data in a PyDataset

New in 7.9.5

PyDatasets can now be accessed in the same ways that Datasets can. This means that all of the above functions (`getColumnCount()`, `getValueAt()`, etc) can be used with PyDatasets too.

PyDatasets are special, in that they can be handled similarly to other Python sequences. Any Dataset object can be converted to a PyDataset, using the function `system.dataset.toPyDataSet`. All of the functions listed above can be used on a PyDataset, but the data can also be accessed much easier, similar to how you would a list.

Python - Accessing Data in a PyDataset

```
# First convert the cities Dataset to a PyDataset.
pyData = system.dataset.toPyDataSet(cities)

# The data can then be access using two brackets at the end with row and column indexes. This will print "PST"
print pyData[1][2]
```

Looping Through a PyDataset

Looping through a PyDataset is also a bit easier to do, working similar to other sequences. The first for loop will pull out each row, which acts like a list and can be used in a second for loop to extract the values.

Python - Looping Through a PyDataset

```
# Convert to a PyDataset
pyData = system.dataset.toPyDataSet(cities)

# The for loop pulls out the whole row, so typically the variable row is used.
for row in pyData:
    # Now that we have a single row, we can loop through the columns just like a list.
    for value in row:
        print value
```

Additionally, a single column of data can be extracted by looping through the PyDataset.

Python - Extract a Column of Data by Looping Through a PyDataset

```
# Convert to a PyDataset
pyData = system.dataset.toPyDataSet(cities)

# Use a for loop to extract out a single row at a time
for row in pyData:
    # Use either the column index or the column name to extract a single value
    # from that row.
    city = row[0]
    population = row["Population"]
    print city, population
```

Altering a Dataset

Technically, you cannot alter a dataset. Datasets are immutable, meaning they cannot change. You can, however, create new datasets. So to change a dataset, you really create a new one and then replace the old one with the new one. There are system functions that are available that can alter or manipulate datasets in other ways. Any of the functions in the `system.dataset` section can be used on datasets, the most common ones have been listed below:

- `system.dataset.addRow`
- `system.dataset.deleteRow`
- `system.dataset.setValue`
- `system.dataset.updateRow`

The important thing to realize about all of these datasets is that, again, they do not actually alter the input dataset. They return a new dataset. You need to actually use that returned dataset to do anything useful.

For example, the following code is an example of the `setValue` function, and would change the population value for Los Angeles.

Python - Altering a Dataset Using the `setValue` Function

Create a new dataset with the new value.

```
newData = system.dataset.setValue(cities, 1, "Population", 5000000)
```

The cities dataset remains unchanged, and we can see this by looping through both datasets.

for row in range(cities.getRowCount()):

for row in range(cities.getRowCount()):

for col in range(cities.getColumnCount()):

print cities.getValueAt(row, col)

for row in range(newData.getRowCount()):

for col in range(newData.getColumnCount()):

print newData.getValueAt(row, col)

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Dates

Overview

Dates can normally be tricky since they generally require very specific formats. Furthermore, some functions/objects require a date object instead of a string. Fortunately, there are several ways to create and alter date objects with scripting in Ignition.

Python has some **built-in libraries** to create and manipulate dates and times. However, most users find both Ignition's built-in **system functions** and even Java's Calendar class easier to use. Regardless, this section will demonstrate some examples from each approach.

Ignition's System Functions

Ignition's **system.date** library has a large number of functions that provide easy access to datetime creation and manipulation. This page has just a few simple examples.

Creating Dates

New datetimes can be created by using either the **system.date.now** or **system.date.getDate**. The **system.date.getDate** function returns a datetime, but the time is set to midnight. However, we can use **system.date.setTime** to change the time.

Python - System Functions - Creating Dates

```
# Get the current datetime.
print system.date.now()

# Create a date. The time will be set to midnight.
newDate = system.date.getDate(2018, 10, 28)
print newDate

# Change the time on the new date to 11:30 am.
print system.date.setTime(newDate, 11, 30, 0)
```

Formatting Dates

When printed, datetimes default to a format like the following: **Sun Jan 1 00:00:00 TZ 2018**. However, this can be manipulated by using special characters in the `system.date.format` function:

Python - System Functions - Date Formatting

```
rightNow = system.date.now()

# Demonstrating the standard format.
print rightNow

# Demonstrating the modified format.
print system.date.format(rightNow, "yyyy-MM-dd HH:mm:ss")
```

Date Arithmetic

The `system.date.add*` functions can be used to add to or subtract some amount of time from a date. See the `system.date.add*` functions for more information.

Python - System Functions - Date Arithmetic

```
# Get the current datetime.
newDate = system.date.now()

# Change the time on the new date to 30 minutes ago.
print system.date.addMinutes(newDate, -30)
```

Date Formatting Characters

The following is a reference of date formatting characters that can be used by `system.date.format` or Java's `DateFormat` class. Additionally, there are many other non-scripting uses in Ignition (such as the `Calendar` component's `Format String` property) that can utilize this reference.

Symbol	Description	Presentation	Example	Other Notes
G	Era designator	Text	G=AD	
y	Year	Year	yyyy=1996; yy=96	Lowercase y is the most commonly used year symbol
Y	Week year	Year	YYYY=2009; YY=09	Capital Y gives the year based on weeks (ie. changes to the new year up to a week early)
M	Month in year	Month	MMMM=July; MMM=Jul; MM=07	
w	Week in year	Number	27	If Dec31 is mid-week, it will be in week 1 of the next year
W	Week in month	Number	2	
D	Day in year	Number	189	
d	Day in month	Number	10	
F	Day of week in month	Number	2	2nd Sunday of the month
E	Day name in week	Text	EEEE=Tuesday; E=Tue	
u	Day number of week	Number	1	(1 = Monday, ..., 7 = Sunday)

Symbol	Description	Presentation	Example	Other Notes
a	Am/Pm marker	Text	PM	
H	Hour in day (0-23)	Number	0	
h	Hour in am/pm (1-12)	Number	12	
k	Hour in day (1-24)	Number	24	
K	Hour in am/pm (0-11)	Number	0	
m	Minute in hour	Number	30	
s	Second in minute	Number	55	
S	Millisecond	Number	978	
z	Time zone	General time zone	zzzz=Pacific Standard Time; z=PST	
Z	Time zone	RFC 822 time zone	Z=-0800	
X	Time zone	ISO 8601 time zone	X=-08; XX=-0800; XXX=-08:00	

Java's Calendar Class

CAUTION

Ignition's System Functions vs Java's Calendar Class

While Java's Calendar class is useful, in many cases Ignition's built-in [system.date](#) functions are simpler to use. Furthermore, the system.date functions typically use the Calendar class to retrieve the current time, so you are not losing any functionality by using the system functions.

It is highly advisable to use Ignition's system functions to generate and manipulate dates. The information on this page pertaining to the Calendar class is maintained in the interest for legacy installations.

Creating Dates

To create an arbitrary date, you can use the `java.util.Calendar` class. It has various functions to alter the calendar fields, like `Calendar.HOUR`, `Calendar.MONTH`, and so on. After you're done manipulating the Calendar, you can use its `getTime()` function to retrieve the Date represented by the calendar. It also has a handy `set()` function that takes the common parameters of a Date. The one major "gotcha" here is that January is month zero, not month one. For example:

Python - Calendar Class - Creating Dates

```
from java.util import Calendar
cal = Calendar.getInstance()

# set year, month, day, hour, minute, second in one call
# This sets it to Feb 25th, 1:05:00 PM, 2010
cal.set(2010, 1, 25, 13, 5, 0)
myDate = cal.getTime()
```

Date Arithmetic

Often you'll have a Date object from a component like the [Popup Calendar](#) and want to alter it programmatically. Say, subtracting 8 hours from it, or something like that. The `java.util.Calendar` class is used for this as well. Following the example above, this code would subtract 8 hours from the variable `myDate`.

Python - Calendar Class - Date Arithmetic

```
from java.util import Calendar
cal = Calendar.getInstance()
cal.setTime(myDate)
cal.add(Calendar.HOUR, -8)
myNewDate = cal.getTime()
```

Python's Time and Datetime Libraries

⚠ IGNITION'S SYSTEM FUNCTIONS VS PYTHON'S LIBRARIES

Many components in Ignition that contain a Date property actually expect a Java calendar object. Creating a datetime object using Python's built-in libraries and passing them to a date property on a component will result in an exception.

As a result, it is highly recommended to use Ignition's built-in [system.date](#) functions.

Creating Dates - Python's Time Library

The time library can be used to return dates as well as time. Times are created as a tuple of integers. The integers represent the following values: year, month, day of the month, hour, minute, second, weekday, day of the year, daylight savings time)

Check out [Python's time library documentation](#) for more information.

Python - Python Library - time

```
import time

# Finds the current local time. The time is returned as a tuple of integers.
myTime = time.localtime()

# Print the time into a 24-character string with the following format: Sun Nov 20
12:00:00 2017
print time.asctime(myTime)

# Alternatively, we can reformat the time in a custom manner, then print it
print time.strftime('%H:%M:%S %b %d %Y', myTime)
```

Creating Dates - Python's Datetime Library

Python's datetime library offers a bit more flexibility since arithmetic can easily be applied. Additional information on the datetime library can be found in [Python's official documentation](#).

Note the double use of `datetime` in the example below. This is because the `datetime` library has a class named `datetime`.

Python - Python Library - datetime

```
import datetime

# Returns the current datetime.
print datetime.datetime.now()
```

However, we can clean up the above by importing in the datetime class from the library:

Python - Python Library - datetime

```
# Imports the class named 'datetime' from the 'datetime' library, so we don't have
to state it twice.
from datetime import datetime

# Returns the current datetime.
print datetime.now()
```

If you need to create a specific datetime, instead of just using the current, you can pass in the values directly when creating an instance of datetime:

Python - Python Library - Creating a New Time

```
from datetime import datetime

# Prints out the following datetime: 2018-01-02 03:04:05.000006
print datetime(2018,1,2,3,4,5,6)
```

Finding the difference between two datetime objects can easily be accomplished by using the '-' character

Python - Python Library - Date Difference

```
from datetime import datetime

rightNow = datetime.now()
someTime = datetime(2018,1,1,1,1,1,1)

# Find the difference between the two dates.
print someTime - rightNow
```

Date Arithmetic

With Python's built-in libraries, the `timedelta` class provides the simplest way to perform arithmetic on a date: It simply creates an object that effectively represents a duration. The duration can then be applied to a `datetime`.

Python - Python Library - Date Arithmetic

```
# We're including the timedelta class here
from datetime import datetime, timedelta

rightNow = datetime.now()

# Creating a timedelta object, and setting the hours to 8
offset = timedelta(hours = 8)

# Print the current time, and then print the time minus the offset.
print rightNow
print rightNow - offset
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Conditions and Loops

If-Statements

The **if** statement should be familiar to anyone with a passing knowledge of programming. The idea of an **if** is that you want your script to execute a block of statements only when a certain condition is true. Python's **if** is simple to use, and has some additional keywords to provide more flexibility.



INDUCTIVE
UNIVERSITY

Control Flow Logic

[Watch the video](#)

Simple If-Statement Example

The syntax for **if** is as follows:

Pseudocode - If Statement

```
# Note that 'if' uses lowercase characters.  
# Additionally, a colon is placed after the expression.  
if expression:  
  
    # The statements that should execute when the expression is true  
    # MUST be indented.  
    statement
```

Example:

```
x = 5
z = 15
if x < 10:
    # Since the condition "x < 10" is true,
    # the following line will execute
    print "'x' is less than 10"
if z < 10:
    # This condition "z < 10" is false,
    # so the following line will not execute
    print "this will never show"
```

Output:

```
'x' is less than 10
```

If and Else

You can use the **if...else** form of an **if** statement to do one thing if a condition is true, and something else if the condition is false.

Example:

```
x = 15
if x < 10:
    print "x is less than 10"
else:
    print "x is not less than 10"
```

Output:

```
x is not less than 10
```

Elif (Else If)

Lastly, you can use the **if...elif** form. This form combines multiple condition checks. `elif` stands for "else if". This form can optionally have a catch-all `else` clause at the end.

Example:

```
x = 3
if x == 1:
    print "one"
elif x == 2:
    print "two"
elif x == 3:
    print "three"
else:
    print "not 1-3"
```

Output:

```
three
```

You can use as many elif items as you want, and the else is not required at the end.

For-Loops and While-Loops



INDUCTIVE
UNIVERSITY

Control Flow Loops

[Watch the video](#)

For-Loop

Python's **for** loop may be a bit different than what you're used to if you've programmed any C. The **for** loop is specialized to iterate over the elements of any sequence, like a list. A **for** loop uses an iterator variable to reference each item as it steps through the sequence. This means it's very simple to write a loop!

Note that the syntax of the for-loop requires use of the in-keyword.

Pseudocode - For Loop

```
# In this example, "item" is a variable created specifically by the "for" loop to
act as an iterator.
# The name "item" is not a keyword, and a different variable name may be used.
# Additionally, note that "for" and "in" are lowercase, and a colon is present at
the end of the line.
for item in sequence:
    # All statements that should execute each iteration must be indented after the
    "for" statement
    statement
```

Example:

```
listOfFruit = ['Apples', 'Oranges', 'Bananas']
for fruit in listOfFruit:
    print fruit
```

Output:

```
Apples
Oranges
Bananas
```

You don't need to manually create a sequence to repeat a task several times in a for loop. Instead, the built-in function **range()** function can generate a variable-size list of integers starting at zero. For example, calling **range(4)** will return the list [0, 1, 2, 3].

Example:

```
# Even though this example isn't using the value of "x",
# the print statement will still be executed once for each item
# in the list returned by range().
for x in range(4):
    print "this will print 4 times"
```

Output:

```
this will print 4 times
this will print 4 times
this will print 4 times
this will print 4 times
```

While-Loop

A while loop will repeat a block of statements as long as a condition is true. This code will print out the contents of the items in the list.

Pseudocode - While Loop

```
# A while loop simply needs the keyword "while", the condition that  
# determines when we should stop iterating, and a colon at the end of the line.  
while condition:  
    # All statements that should be repeated each iteration must be indented after  
    the "while" statement  
    statement.
```

This code uses a function called **len()**, which is a built-in function that returns the length of a sequence.

Example:

```
listOfFruit = ['Apples', 'Oranges', 'Bananas']  
x = 0  
while x < len(listOfFruit):  
    print listOfFruit[x]  
    x = x + 1
```

Output:

```
Apples  
Oranges  
Bananas
```

The Break and Continue Statements

You can stop a loop from repeating in its tracks by using the **break** statement. This code will print out "Loop " exactly two times, and then print " Finished ".

Example:

```
for x in range(10):  
    if x >= 2:  
        break  
    print "Loop"  
print "Finished"
```

Output:

```
Loop  
Loop  
Finished
```

You can use the **continue** statement to make a loop stop executing its current iteration and skip to the beginning of the next iteration. The following code will print out the numbers 0-9, skipping 4.

Example:

```
for x in range(10):  
    if x == 4:  
        continue  
    print x
```

Output:

```
0  
1  
2  
3  
5  
6  
7  
8  
9
```

Infinite Loops

It is incredibly easy to create an infinite loop when using a **while** statement. Depending where the infinite loop was created, it could cause you to lose your work in the Designer, or create a large amount of overhead on the Gateway.

Python - Infinite Loop Created by While Statement

```
x = 0
while x < 10:
    x += 1 # Forgetting to add a way to increment "x" will cause an infinite loop
    print x
```

In many cases, a **for** loop could be used instead of a **while**, but this is not always possible. When using **while**, the best way to avoid an infinite loop is to make sure you always have a way to exit the loop: a simple approach involves using a counter that can eventually trigger a **break** statement, or add the counter as an additional condition to the **while**.

Python - Preventing Infinite Loops Using the Break Keyword

```
###Example 1: using the break keyword
# The counter variable will be used as a guaranteed way out of the While.
counter = 0

# Normally, using True as a condition in a While would be a quick
# way to generate an infinite loop, but the counter helps prevent that
while (True):

    # Increase the counter
    counter += 1

    # Check the value of the counter. If it's at the point where we can assume
    # we're going to be looping indefinitely...
    if counter >= 1000:

        # Break out of the loop
        break
```

Python - Preventing Infinite Loops Using an Additional Condition

```
###Example 2: using an additional condition
# Again, the counter variable will be used as a guaranteed way out of the While.
counter = 0

# Instead of using nested logic, we can simply add counter's value as an
# additional condition with "and"
while (True and counter < 1000):

    # Increase the counter. Once counter >= 1000, the while loop will be forced to
    # end.
    counter += 1
```

The Pass Keyword

When using conditional statements and loops, the pass keyword can be especially useful when writing a new script. When called, the pass keyword does nothing, which may seem useless. However it is great when you find yourself in a situation where you need a line of code to meet a syntax requirement, but don't want the code to do any additional work.

Python - Pass Keyword

```
myVar = system.tag.read(tagPath).value

if myVar == 0:
    firstFunction()
elif myVar == 1:
    secondFunction()
elif myVar == 2:
    # I haven't implemented the thirdFunction() yet, so I can use pass here as a placeholder
    pass
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Error Handling

What is Error Handling

The concept of error handling is recognizing when an error might occur in a block of code, and instead of throwing the error, handling it gracefully. This can involve giving the user a more distinct error message, letting the user know that their attempt to run the code failed, or even undoing something that your code set in motion so that it can be started again from the same starting point.

Error Handling in Python

Within Python, we can use the **try** and **except** blocks to handle errors. We would first use `try:` and write the code we would like to try indented underneath it. We then must have an `except:` with code that will run if there is an error.

Pseudocode - Error Handling

```
# With try, we can attempt any amount of code
try:
    some code
    more code
    even more code

# If any of lines in the try block were to throw an error, then we move down and
run the block under except.
# The except statement is NOT optional: you must define what your code should do
in the event an exception occurs.
except:
    failover code
```

When running the code above, there is a specific order to the way it executes.

1. The try block is run, and the code within is executed line by line. a. If an error occurs, the code in the try block will immediately stop and the except block will begin to execute. b. If no error occurs after all code in the try block has been executed, the try block will be finished and the code in the except block will be skipped.

2. After either outcome, any code after the except will then execute.

Because of the way the try and except blocks work, it is very useful on situations that require user input, where something may have been incorrectly entered which would usually result in an error.

The Pass Keyword

The **pass** keyword is unique in that it does nothing except to fill in a spot where code is required. This is useful if we want to handle the exception so that it doesn't throw an error message, but we don't actually want to do anything with it. In this case, we can use pass to tell the script to continue.

Pseudocode - Pass Keyword

```
try:
    some code
    more code
    even more code

except:
    # The error will bring the code to the exception, and then the exception will
simply do nothing.
    pass
```

Error Handling Example

An easy way to demonstrate how error handling works is with a division example, since it is easy to cause an error by dividing by 0. Take the code below:

When we run it, we get a printed value of 50. There was no error in the division, and the try block finished successfully. However, if we were to change the value of x to 0, we can see that "An error occurred!" is printed.

Python - Error Handling Division

```
# We start with a value, which could represent some user input.
x = 2

# We use that value in our try block, and have a simple print statement if there
is an error.
try:
    value = 100/x
    print value
except:
    print "An error occurred!"
```

Exception-Specific Failover

While each try block must be followed by at least one except block, there can be multiple except blocks to handle different types of errors. This is done by listing the error object after the except keyword and before the colon. Looking back at the example above, I know that my most common error is going to be a divide by zero error. So I can make an exception that is specific to divide by zero errors.

Python - Exception-Specific Failover

```
# We start with a value, which could represent some user input.
x = 0

# Use the user input in division in our try block.
try:
    value = 100/x
    print value

# If the exception that would occur is of type ZeroDivisionError, we can run a
specific block of code
except ZeroDivisionError:
    print "Dividing by zero is not allowed. Please stop trying to divide by zero"
# We can then have a second exception without a specific error. This except acts
as a catch-all;
# if the user caused an exception we didn't account for above, we can failover to
the block of code below.
except:
    print "An error occurred!"
```

The except blocks in the code above cover all possible errors as represented in the table below. Now, we have a more tailored approach to how we handle errors while still catching all of them that may occur.

Inputs (x value)	Output
2	50
0	Dividing by zero is not allowed. Please stop trying to divide by zero
'a'	An error occurred!

Each try block can have many except blocks, and each except block can also name multiple types of errors as shown below. However, an error that happens within a try block will only ever trigger a single except block.

Pseudocode - Try Block with Except Blocks

```
try:
    some code
except (ZeroDivisionError, RuntimeError, TypeError):
    failover code
```

Determining the Error Object

To determine the name of the error object that will be thrown from certain errors, we can take a look at the error to figure that out. We already mentioned that dividing by zero gives a `ZeroDivisionError`, but what about when we divide by a string? If I divide 100 by the letter a without error handling, this is the error I get:

```
Traceback (most recent call last):
File "buffer", line 3, in module
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

The last line of that error gives the name of the error object "TypeError" followed by the cause of that error. This makes sense, because the string 'a' is the wrong type to be using in division. However, not all errors are so simple and may require a bit more to find the name of the error. For a list of python error names check out this page in the python docs:

<https://docs.python.org/2.7/library/exceptions.html#Exception>

Additionally, some exceptions may be returned by Java. In these cases, Oracle's documentation on the Exception class is more useful:

<https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html> A list of common exceptions are listed below.

Exception	Description	Exception Demonstration
ArrayIndexOutOfBoundsException	This typically occurs when a line of code attempts to access an index in a collection, but the index specified doesn't exist. This exception is unique to datasets in Ignition. Lists, and other built-in Python objects will return the <code>IndexError</code> below.	<pre>myDataset = system.dataset.newDataset('myDataset') # This will fail # so trying to read something # read something print myDataset.get(100, 'value')</pre>
<code>AttributeError</code>	Typically seen when a script tries to access a property that doesn't exist. Example: a script tries to get the value of a property on a component, but the property name is misspelled.	<pre>myVar = 10 # integers do not have a .name property # there isn't an attribute named .name print myVar.name</pre>
<code>IndexError</code>	Similar to <code>ArrayIndexOutOfBoundsException</code> above, but occurs when Python specific object, such as a list.	<pre>myList = [1,2,3] # There isn't an element at index 4 print myList[4]</pre>
<code>NameError</code>	Occurs when the local or global name is not found. Typically happens when referencing a variable that hasn't been assigned a value.	<pre># We haven't given a value to myVar print "The value of myVar is: " + myVar</pre>
<code>TypeError</code>	A <code>TypeError</code> occurs when a function or similar operation is applied to another object of an incorrect type.	<pre>myList = [1,2,3] # The first argument to pop() must be a string # Passing a string to pop() print myList.pop(1)</pre>

Exception	Description	Exception Demonstration
ValueError	A ValueError is returned when the value of an argument is inappropriate. Typically the exact issue is returned in the returned exception.	<pre>myVar = "Hello" # Strings can be # coerced to integers # Because "Hello" is not an integer print int(myVar)</pre>

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Built-In Functions

Python Built-In Functions

Functions are code that can be called repeatedly from other places. Functions can have parameters passed into them and may return a resulting value. Some functions, like `len()`, are built-in. Some functions, like `system.gui.messageBox()`, are part of the [scripting libraries](#) provided by Ignition. Some functions, like `math.sqrt()`, are provided by the Python Standard Library.

Functions are invoked by using their name followed by an argument list surrounded in parentheses. If there are no arguments, you still need an open and close parenthesis.

This section details several useful Built-in Functions, along with some simple examples. See the [official docs](#) for more information.

Type Casting Functions

Python has many functions to convert between data types. Some common type casting functions are listed below.

Function	Notes	Example	Output
bool()	When casting a numeric value to a boolean, a zero value is false, while all non-zero numbers are True When casting a String or Unicode value to a boolean, an empty string is False, any other string is True.	<pre># Results in False print bool("") # Results in True print bool("Test")</pre>	False True
int() and long()	When casting a float, rounding will not occur automatically to the decimal value. Instead, the round() function should be called. When casting a String or Unicode value, the string literal needs to be a	<pre># Float to Integer print int(123.8) # Float to Long print long(321.8) # String to Integer print int("400") # ValueError: the value is not base 10 print int("400.5")</pre>	123 321 400 ValueError

Function	Notes	Example	Output
	valid integer or long: decimal values contained in the string will result in a ValueError. Integers have at least 32 bits of precision, while Longs have unlimited precision.		
float()	When casting a string literal as a float, non-numeric characters in the string will result in an exception, except for a decimal point (".").	<pre># Integer to Float print float(123) # String to Float print float("400.5")</pre>	<pre>123.0 400.5</pre>
str() and unicode()	Most objects can be cast as a string representation of some sort, including sequences.	<pre>print "First line:" + str(80) # Even sequences can be cast as a string, # making for easy concatenation myList = [1,2,3] print str(myList)</pre>	<pre>80 [1, 2, 3]</pre>

Checking an Object's Type

Checking the data type of an object can easily be done with both the `type()` and `isinstance()` functions.

Function	Notes	Example	Output
<code>type(object)</code>	When passed a single parameter, this function returns the type of the <code>object</code> .	<pre>var = 10 print type(var) print type(str(var))</pre>	<code><type 'int'></code> <code><type 'str'></code>
<code>isinstance(object, classinfo)</code>	Returns True if the <code>object</code> is an instance or subclass of <code>classinfo</code>, otherwise, returns false. If checking for a string or unicode type, a <code>classinfo</code> of "basestring", which is the base class for both strings and unicode types, would return True.	<pre>var = 10 print isinstance(var,int) strVar = "string" print isinstance(strVar,basestring)</pre>	<code>True</code> <code>True</code>

```

## type() Example
# This example attempts to validate the type of a variable. As written, this will
# evaluate as True, and thus the print statement would execute.
var = "My String"
if type(var) == type(""):
    print "Variable 'var' is a string"

## isinstance() Example
# The isinstance() function can offer the same functionality as above.
var = "My String"
if isinstance(var, str): # Note the lack of quotation marks around the classinfo
# parameter. We want to reference the class str, not the string "str".
    print "Variable 'var' is a string"

```

Generating a Range of Values

In some cases, it is useful to generate a range of integers for iteration. Python's `range()` function will return a list of integers.

Function	Notes	Example	Output
range([start,] stop[, step])	Returns a list of progressively greater integers. <code>start</code> - Integer value denoting the initial value in the list. If omitted, defaults to 0. This parameter is inclusive. <code>stop</code> - Integer value, determines when to cease generating integers. This parameter is exclusive. <code>step</code> - Integer value to increment each new integer by. If omitted, step defaults to 1. If <code>step</code> is positive, integers will be generated as long as (<code>start + i * step < stop</code>) is true. If <code>step</code> is negative, integers will be generated as long as (<code>start + i * step > stop</code>) is true.	<pre> print range(5) print range(1, 5) print range(1, 10, 3) print range(15, 0, -3) </pre>	<pre> [0, 1, 2, 3, 4] [1, 2, 3, 4] [1, 4, 7] [15, 12, 9, 6, 3] </pre>

Assume we need to read from five separate Tags with a nearly identical Tag path in a single script:

Pseudocode - Tag Path

```
[Provider]Folder/Sub_Folder_1/Tag  
[Provider]Folder/Sub_Folder_2/Tag  
[Provider]Folder/Sub_Folder_3/Tag  
[Provider]Folder/Sub_Folder_4/Tag  
[Provider]Folder/Sub_Folder_5/Tag
```

Instead of manually typing each path, we could use `range()` in a for loop that would write the paths automatically.

Python - Range in a For Loop

```
# Initialize an empty list that will ultimately hold all the Tag paths.  
tagPaths = []  
  
# Use range to repeatedly append 5 tag paths to the tagPaths list: starting with a  
value of 1, and ending with a value of 5.  
for num in range(1,6):  
    # Use String Formatting to create a Tag path with the iterator's (num) value.  
    tagPaths.append("[Provider]Folder/Sub_Folder_%i/Tag" % num)  
  
# Now that tagPaths contains all our tag paths, we can use the list to interact  
with the tag, such as by reading their values simultaneously.  
tagValues = system.tag.readAll(tagPaths).value
```

Rounding Numbers

You can round numbers inside Python with a few simple functions.

Function	Notes	Example	Output
round(number[, digits])	When passed a single parameter, this function returns a rounded integer value of a number. If the decimal is greater than or equal to .5, the it rounds up, less than .5 rounds down. If the optional digits argument is used, then it rounds to that many decimal places.	<pre>var = 10.236981 print round(var) print round(var,3)</pre>	10 10.237
math.floor(number)	Returns a truncated integer from the number given. The largest integer value less than or equal to <i>number</i>. Note that the example needs to import that math library before being able to call floor().	<pre>import math var = 100.938 print math.floor(var)</pre>	100.0
math.ceil(number)	Returns the ceiling integer from the number given. The smallest integer value greater than or equal to <i>number</i>. Note that the example needs to import that math library before being able to call ceil().	<pre>import math var = 100.138 print math.ceil(var)</pre>	101.0

```
stringVar = "40"
```

Without using int(), this line would cause an exception. However int() is able to cast the type of stringVar's value to an integer.

```
print 20 + int(stringVar)
```

Type casting is also useful in cases where a string parameter is required, but a numerical value should be given,

such as the message parameter in system.gui.messageBox().

```
intVar = 60
```

Note that this could also be accomplished with String Formatting instead of using str().

```
system.gui.messageBox(str(intVar))
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Libraries

The System Library

Ignition comes with a group of system functions, called the System Library. Using a system function is simple. For example, the following code will access the value of a Tag.

Pseudocode - Reading a Tag Value

```
value = system.tag.read("tagPath").value
```

The [scripting appendix](#) is full of built-in functions such as this.



INDUCTIVE
UNIVERSITY

System Library

[Watch the video](#)

Python Libraries

Python Libraries are packages of extra functions that expand the functionality of the code and can be imported into a script. We do this by using the import keyword:

Pseudocode - Import a Library

```
# This pseudo code will import a library, and then call a function of that library.
```

```
import myLibrary
```

```
myLibrary.specialFunction()
```

The import keyword imports that entire library and allows you to use all of the functions inside of it by calling them off of the imported library. You can also import a piece of a library:

Pseudocode - Import a Function of a Library

```
# This pseudo code will import a function from a library, and then call that function.  
from myLibrary import specialFunction  
  
specialFunction()
```

Note, that since we are directly importing in the function, we can directly call it instead of having to call it off of the library.

Python Standard Library

Python has an extensive standard library that provides a host of new functionality to the scripting language. The python documentation goes over all of the libraries in its standard library as well as how to use them here: <https://docs.python.org/2/library/index.html>

Let's take a look at an example of using a common library:

Python - Accessing Files in a Python Standard Library

```
# The csv library provides an easy way to read csv files, regardless of how they are formatted.  
import csv  
  
# We first grab our filepath, and feed it into the open function, which opens the file.  
filepath = "C:\\\\test.csv"  
csvFile = open(filepath, 'r')  
  
# We then pass our opened csv file object into the csv.reader function, which will read the file.  
# This can be looped through in a for loop to print every row of the csv.  
reader = csv.reader(csvFile)  
for row in reader:  
    print row
```

Importing 3rd Party Libraries

In addition to the standard libraries, 3rd party libraries can also be imported into Ignition's scripting environment. A Python Library or individual module file will consist of a python file (.py) that contains the code that implements the functions of the library. You can often find python libraries built by other

users on the web, or can even create your own. These files can then be placed into a folder within your Ignition server.

- **Windows folder:** C:\Program Files\Inductive Automation\Ignition\user-lib\pylib
- **Linux folder:** /var/lib/ignition/user-lib/pylib
- **Mac OS X folder:** /usr/local/ignition/user-lib/pylib Once the python file is in that folder, you can then import the library into a script just like any of the standard libraries.

CAUTION

Ignition uses Python version 2.5. This means that any imported libraries must be compatible with Python 2.5

Accessing Java

Scripting in Ignition executes in the java based implementation of Python called Jython. (See [Python or Jython?](#)). While this doesn't have any great effect on the Python language itself, one of the great side benefits is that your Python code can seamlessly interact with Java code as if it were Python code. This means that your Python code has access to the entire Java standard library.

To use Java classes, you simply import them as if they were Python modules. For example, the following code will print out all of the files in the user's home directory. This code uses the Java classes `java.lang.System` and `java.io.File` to look up the user's home directory and to list the files. Notice that we can even use the Python-style for loop to iterate over a Java sequence.

Python - Accessing Java

```
# Importing the appropriate java libraries.
from java.lang import System
from java.io import File

# Used to look up the files in the users home directory.
homePath = System.getProperty("user.home")
homeDir = File(homePath)

# Loops through the list of files and prints them.
for filename in homeDir.list():
    print filename
```

You can find the reference documentation for the Java standard class library (also known as, the "JavaDocs") at: <http://docs.oracle.com/javase/8/docs/api/>

Subclassing Java

You can also create Python classes that implement Java interfaces. You do need some understanding of Java and object-oriented programming concepts, which are outside the scope of this manual. To create a Python class that implements a Java interface, you simply use the interface as a superclass for your Python class. For example, we could augment the example above to use the overload `java.io.File.list(FilenameFilter)`. To do this, we'll need to create a `FilenameFilter`, which is an interface in Java that defines a single function:

```
boolean accept(File dir, String name)
```

To implement this interface, we create a Python class that has `java.io.FilenameFilter` as its superclass, and implements that Java-style function in a Python-esque way.

Python - Implementing Java Interfaces

```
# Importing the appropriate java libraries.
from java.lang import System
from java.io import *

# This sets up an extension filter that can check the file extension. Txt is the default.
class ExtensionFilter(FilenameFilter):
    def __init__(self, extension=".txt"):
        self.extension=extension.lower()

    def accept(self, directory, name):
        # make sure that the filename ends in the right extension
        return name.lower().endswith(self.extension)

# Used to look up the files in the users home directory.
homePath = System.getProperty("user.home")
homeDir = File(homePath)

# Prints out all .txt files. Txt is provided if nothing is specified.
for filename in homeDir.list(ExtensionFilter()):
    print filename

# Prints out all .pdf files.
for filename in homeDir.list(ExtensionFilter(".pdf")):
    print filename
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

User Defined Functions

Functions



INDUCTIVE
UNIVERSITY

Functions

[Watch the video](#)

A function is code that can be called repeatedly from other places. Functions can have parameters passed into them, and may return a resulting value. Some functions, like `len`, are built-in. Some functions, like `system.gui.messageBox()`, are part of the **scripting libraries** provided by Ignition. Some functions, like `math.sqrt()`, are provided by the **Python standard library**. However, functions can also be defined in a script that can be used later on in the script. In these user defined functions, you give the function a name and some code that will run when the function is called. Then later on in the script, you can call the function by its name and it will run the code specified in the function. This is useful, because it allows you to run a segment of code many times without having to repeat it within the script.

Functions are invoked by using their name followed by an argument list surrounded in parentheses. If there are no arguments, you still need an open and close parenthesis.

Defining Functions

Functions are defined using the `def` keyword. A function needs a name and a list of the arguments that it can be passed. For example, this code defines a function that prints "Hello World!".

Python - Defining a Function

```
# First we define our function.
def printHW():
    print "Hello World!"

# We can then call our function.
printHW()
```

Functions Arguments

When a function accepts arguments, the names of those arguments become variables in the function's namespace. Whatever value was passed to the function when it was invoked becomes the value of those variables. Arguments can have default values, which makes them optional. If an argument is omitted, then its default value will be used. The following code defines a function called `cap`, which will take a number, and make sure it is within an upper and lower limit. The limits default to 0 and 100.

Python - Defining Arguments

```
# We first define our function. Notice that we have 3 different arguments.
# x, min, and max. The min and the max are set to equal 0 and 100 respectively.
def cap(x, min=0, max=100):

    # Check if x is less than the min, return the min if true.
    if x < min:
        return min

    # Check if x is greater than the max, return the max if true.
    elif x > max:
        return max

    # Return the value if it is within the bounds.
    else:
        return x

# We can then see the outcome by running our function with a few different
parameters.
# This will print out a 40, since it is within the bounds.
print cap(40)

# This will print out "0", since it is less than the min of 0.
print cap(-1)

# This will print out "100", since it is greater than the max of 100.
print cap(150)

# This will print out "150", because it uses a max of 200 instead of the default
100.
print cap(150, 0, 200)
```

Keyword Arguments

In Ignition, some complicated script functions are designed to take keyword arguments instead of normal parameters. In the description for those functions, you may see the following info box in this User Manual:



TIP

This function accepts keyword arguments.

Arguments can also be specified by keyword instead of by position. In the example above, the only way someone would know that the 200 in the last call to cap specified the max is by its position. This can lead to hard-to-read function invocations for functions with lots of optional arguments. You can

use keyword-style invocation to improve readability. The following code is equivalent to the last line above, using 200 for the max and the default for the `min`.

Python - Using Keyword Arguments

```
print cap(150, max=200)
```

Because we used a keyword to specify that 200 was the max, we were able to omit the min argument altogether, using its default. However, using a keyword argument before a non-keyword or positional argument is not allowed.

Python - Non-keyword Argument

```
# This would fail, because the function isn't sure what 150 is being used for.  
print cap(max=200, 150)
```

Functions Are Objects

Perhaps one of the most foreign concepts for new Python users is that in Python, functions are first-class objects. This means that functions can be passed around to other functions (this concept is similar to the idea of function pointers in C or C++).

Suppose we wanted a general way to filter a list. Maybe sometimes we want the odd entries, while other times we want even ones. We can define a function called `extract` that takes a list and another function, and returns only entries that "pass" through the other function.

Python - Functions Passed to Other Functions

```

# We define a function that checks if the value passed in is odd.
def isOdd(num):
    return num % 2 == 1

# We define a function that checks if the value passed in is even.
def isEven(num):
    return num % 2 == 0

# We define a function that inserts our list into the appropriate function and
returns valid values.
def extract(filterFunction, list):
    newList = []
    for entry in list:
        if filterFunction(entry):
            newList.append(entry)
    return newList

# Prints out [0, 2, 4, 6, 8]
# Notice that isEven is not invoked, but passed to the filter function.
print extract(isEven, range(10))

```

Where Can Functions Be Defined

User Defined Functions can be defined anywhere that a script is used. As stated before, they are useful to run segments of code multiple times without having to repeat it. They are also used extensively in **project and shared script modules** where multiple functions can be defined in a single script module. Finally, some special Ignition System functions like **system.gui.createPopupMenu** or the **runScript** Expression function use functions as arguments.

Function Scope

The concept of scope is very important in all programming, and Python is no exception. Scope defines what names are directly accessible without any qualifiers. Another way to put this is that the scope determines what variables are defined. In Python, a variable is defined at the time that it is assigned. What scope it belongs to is also defined by where the assignment occurs.

Pseudocode - Defining a Function for Scope

```
# On this line, there is no variable 'x' in scope.
doSomeWork()

# Now 'x' is defined in our scope, because we've assigned a value to it
x = 5

# This will work because x is in scope.
print x
```

When you define a function, that function gets its own scope. Variables that are assigned within that function body will not be available outside of the function.

Python - Variables Defined within a Function Not Available Outside Scope

```
# x is local to myFunction() because this is where it is defined.
def myFunction():
    x = 15
    print x

# This will fail, because x is not available in the outer scope
y = x + 10
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Scripting in Ignition

Where Is Scripting Used?

Python is used in many places in Ignition. Each location has its own events that trigger your scripts to run, and add functionality to your projects in different ways. The most apparent place is in **event handlers** on components and other objects in Ignition Clients. Project and Gateway **event scripts** are another major place where Python is used.



INDUCTIVE
UNIVERSITY

Scripting in Ignition

[Watch the video](#)

Script Scope

One important thing to keep in mind before scripting in Ignition, is to understand the concept of scope. Within Ignition, there are two different scopes: the Gateway Scope and the Client Scope. Where a script is run from determines what scope it is running in. This is important because it determines what system functions can be run, what resources the script can interact with, and where the output will be written to. For example, running a script on a Tag is run in the Gateway Scope and the output is sent to the Gateway console (i.e., wrapper.log file) because Tags are stored in the Gateway. This means that the script will not be able to access any client level resources such as windows or components that you may have open in the Client. Additionally, some of the system functions like `system.gui.errorBox()` only work in the "Client Scope," so you will not be able to use them in the script on the Tag.

"Client Scope" scripts, however, execute on the running client (and also in Designer when testing, but only in Preview Mode). For example, if a component on a window is running a script, its values are isolated to the client, and the output will be displayed on the Designer/Client output console.

System Functions

Ignition comes with a group of system functions, called the System Library. Using a system function is simple. For example, the following code will access the value of a Tag.



INDUCTIVE
UNIVERSITY

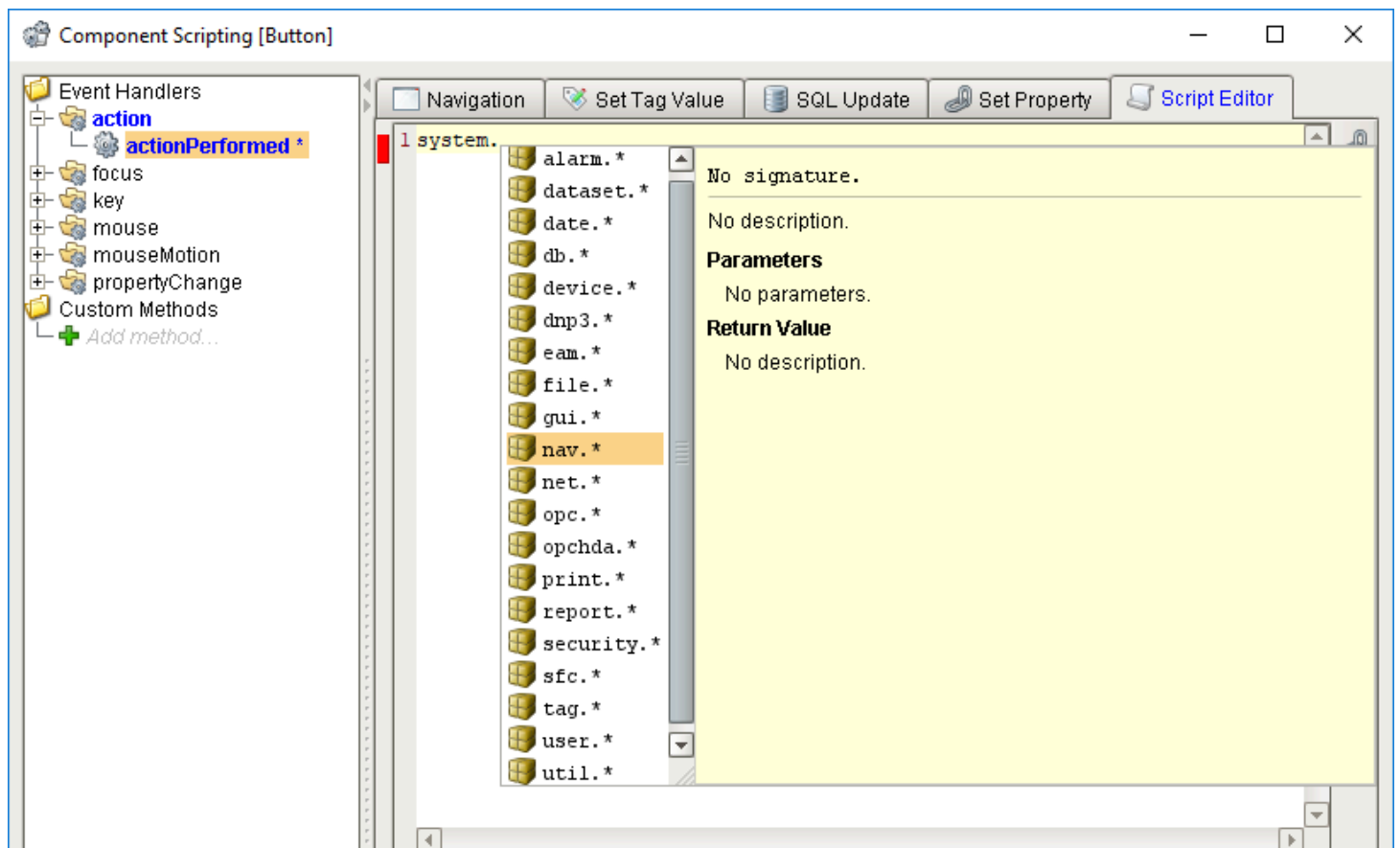
System Library

[Watch the video](#)

Python - Simple Script Using a System Function

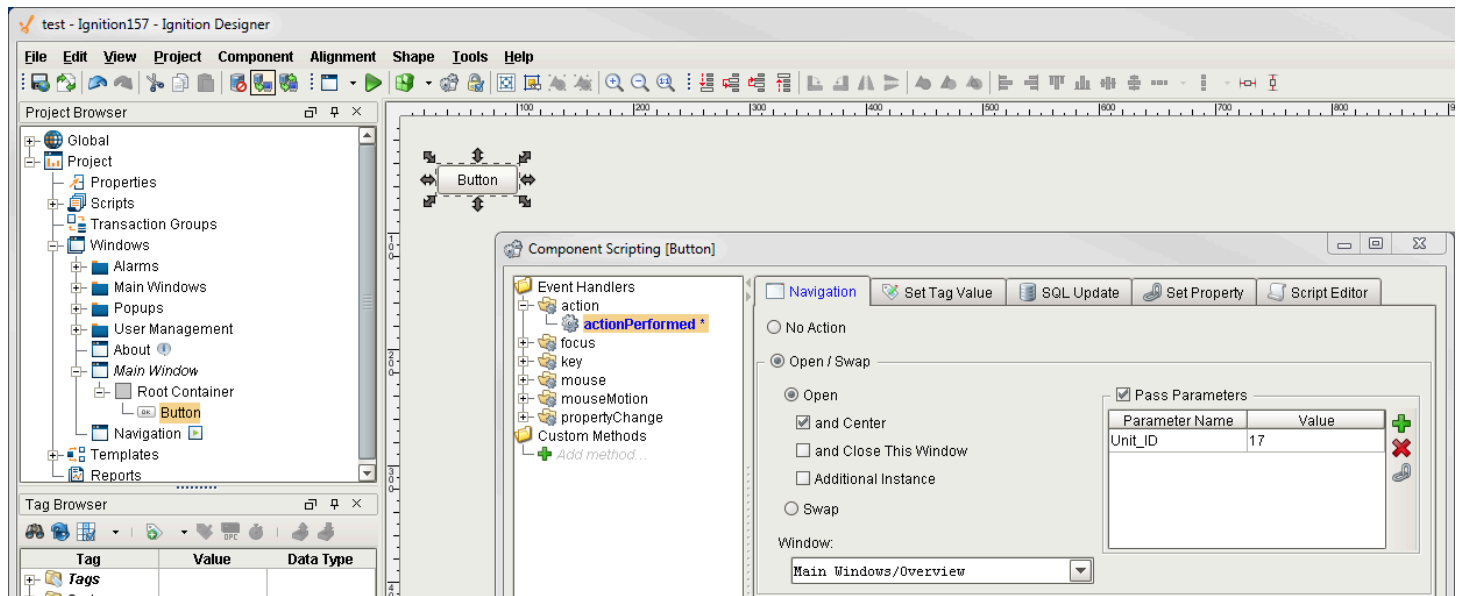
```
value = system.tag.read("tagPath").value
```

A complete list of these functions (with their definitions) is available wherever you can add a script. Just type **system.** and then press **Ctrl+Space** to get a list of all the functions available. If you keep typing, the list will even be automatically narrowed down for you! Additionally, the [Scripting Functions](#) page in the appendix contains complete documentation for the built-in system functions.



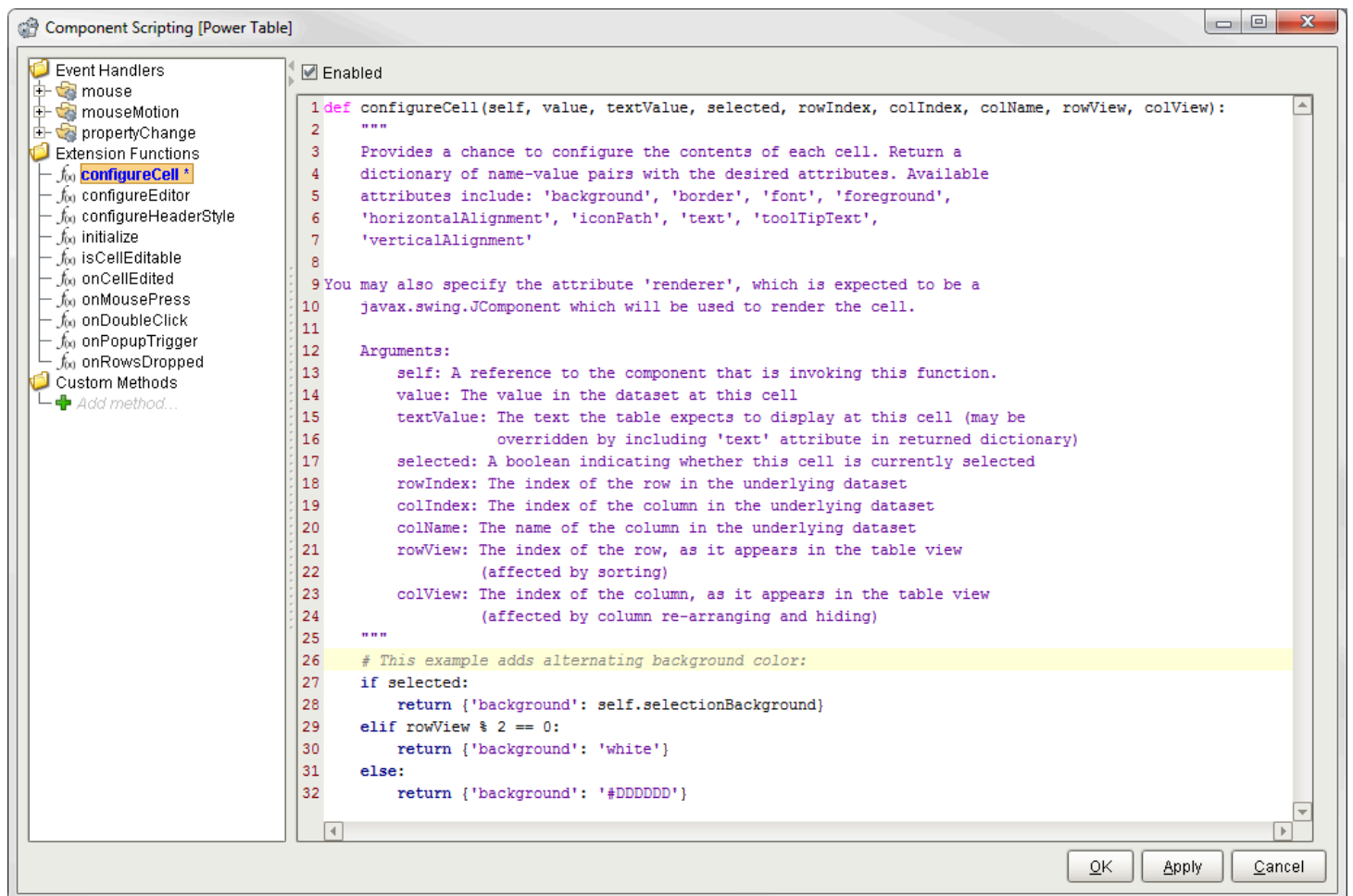
Components

Every component has "**events**" associated with them such as when a mouse is clicked, a key is pressed, or a value changes. Each type of component has a slightly different set of events based on what you do with that component. For example, all of them have the Property Change event, but only components that can be activated have the focus events. Because these events are on components, they are "Client Scoped."



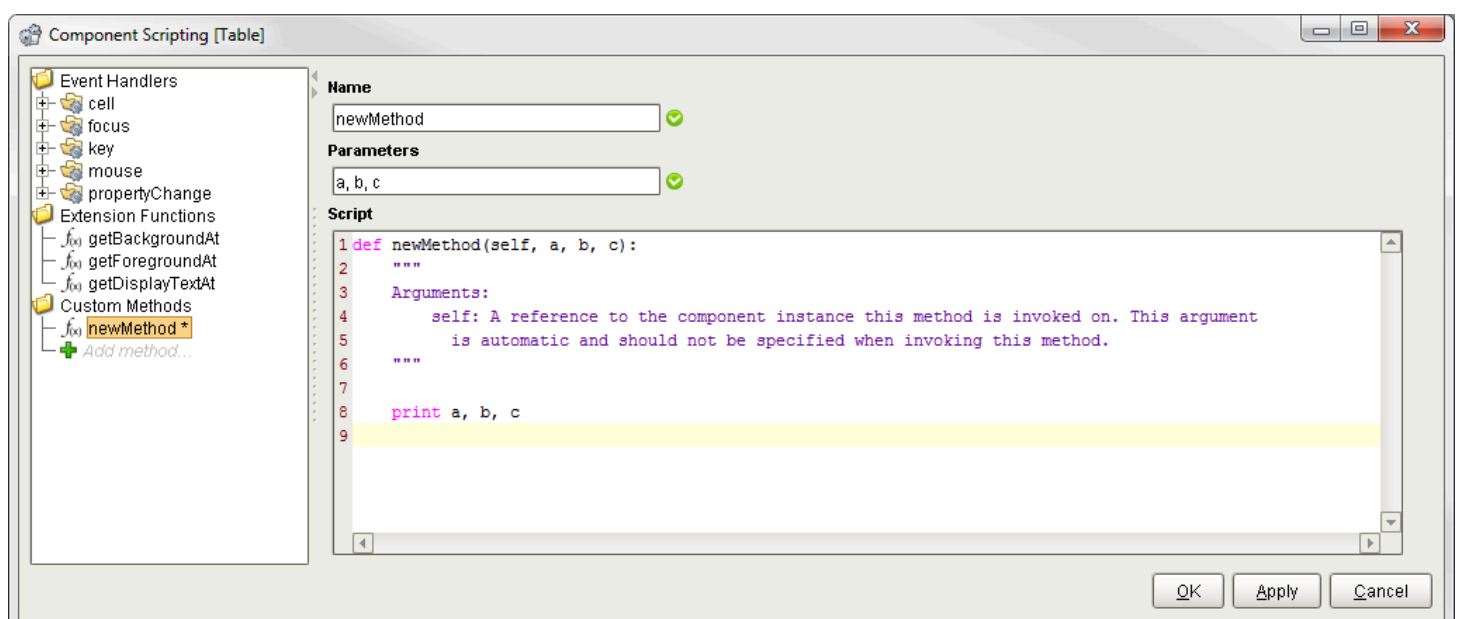
Extension Functions

Extension Functions are chunks of code that allow you to interact with some of the basic parts of components that would otherwise be hidden. They vary for each component and not all components have them because they are so specialized. These functions allow you to do things like **configure a cell or create a right-click menu** in a table, or **change the way the Easy Chart** works. Because these events are on components, they are Client Scoped.



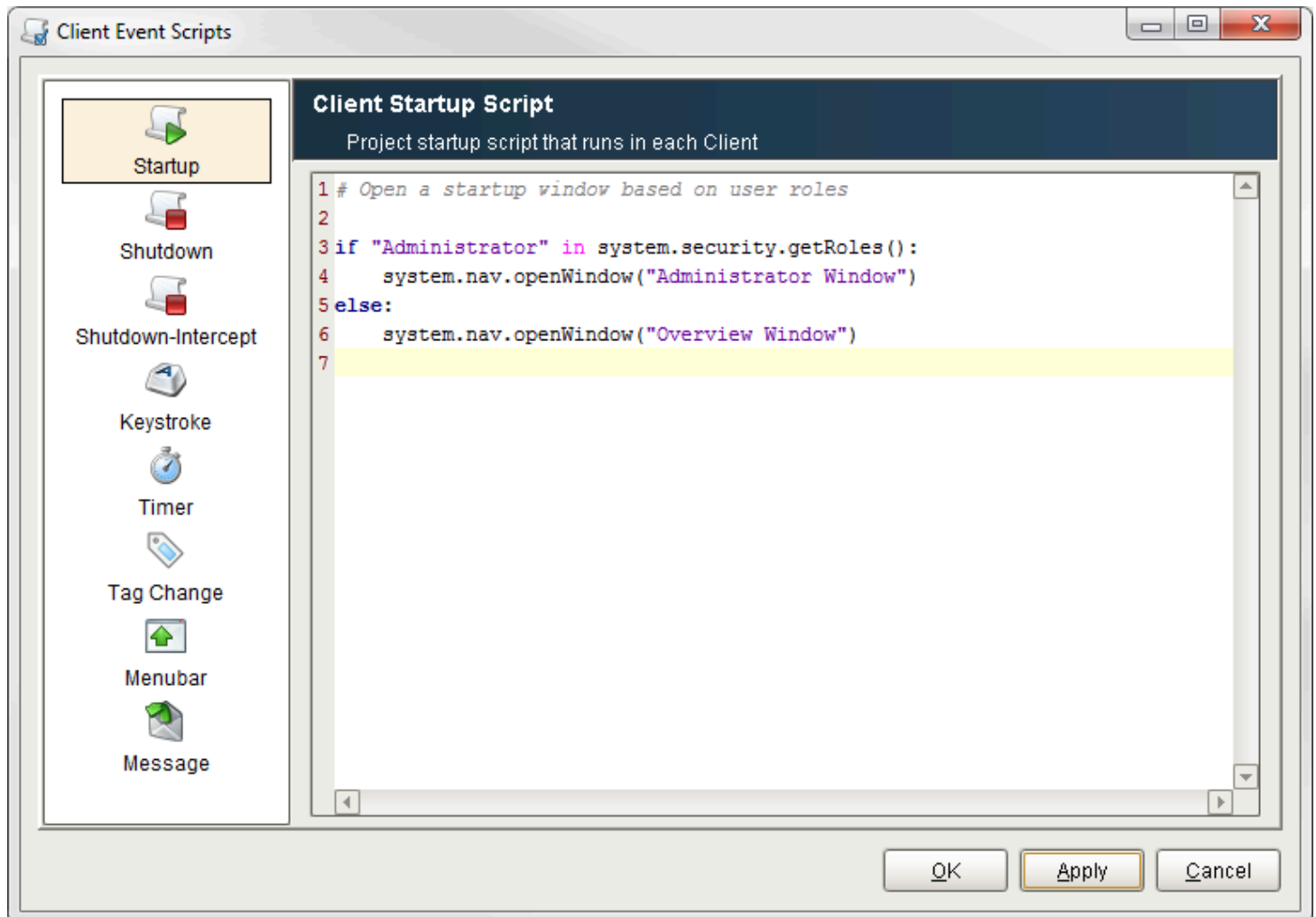
Custom Methods

You might want to have a table contain all the scripting to edit it's contents, or a button that does a complicated check before continuing. Using **Custom Methods** allow you to add functions just like the Script Modules, but attach them directly to a specific component. This means that if you have a template or component used in several places, you can set up common functions that will stay with that component. Because these events are on components, they are Client Scoped.



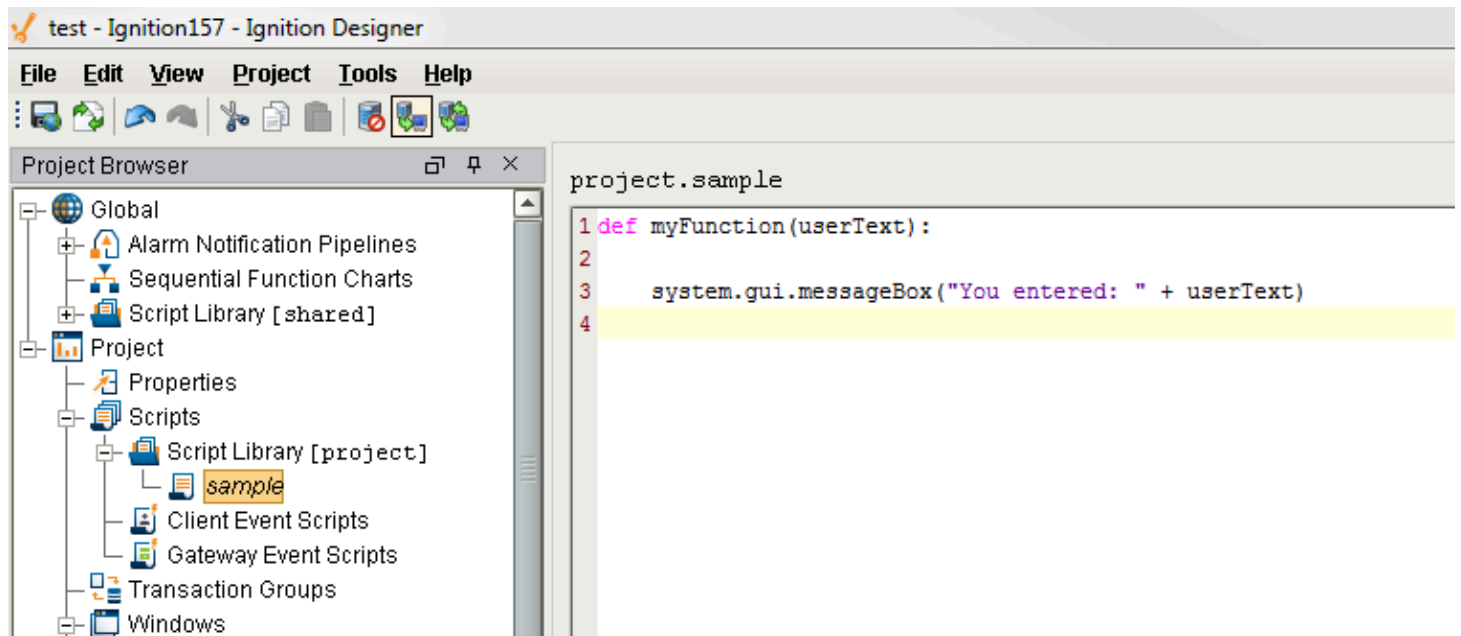
Client and Gateway Event Scripts

Scripts can be set to activate on specific events in the Client, or in the Gateway. These scripts may run every 5 seconds, on a Tag change, or when the Client or Gateway start up or shut down. There is even a messaging system that allows you to create functions that will run on demand when called by another script. These scripts vary in scope, depending on whether you set up a Client Event Script or a Gateway Event Script.



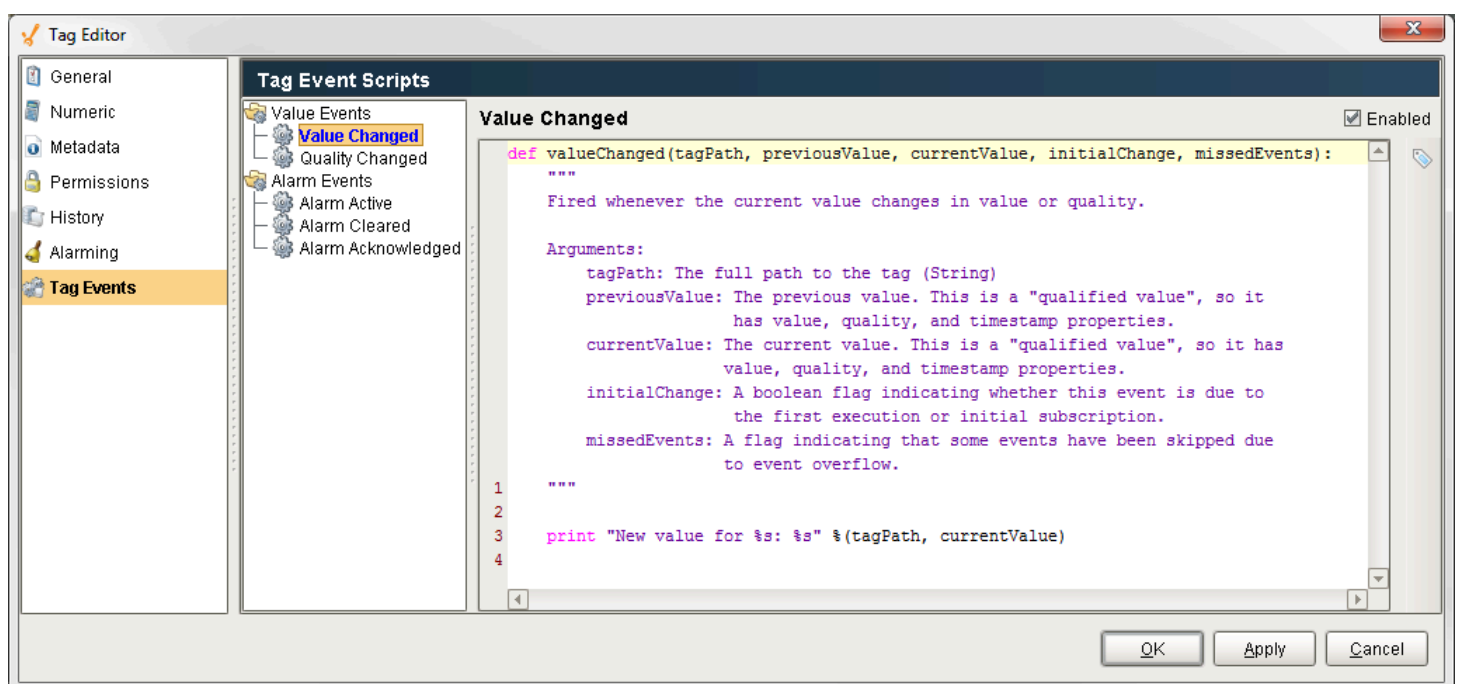
Scripting Modules

You can create your own reusable blocks of code in the Script Library. You can set up these **Script Modules** that exist in just one Project, or ones that are shared between all projects on a Gateway. Once completed, these functions can be called from anywhere in Ignition just like our **system.*** functions. The Script Modules vary in scope depending on where you call them. They act similar to the system functions except you create them. Calling a Script Module in a Tag Script will be in the Gateway Scope, while calling one from a component will be in the Client Scope. However, a Script Module made in the Project Script Library can only be called in places with the Project Scope, whereas Script Modules made in the Shared Script Library can be called from anywhere.



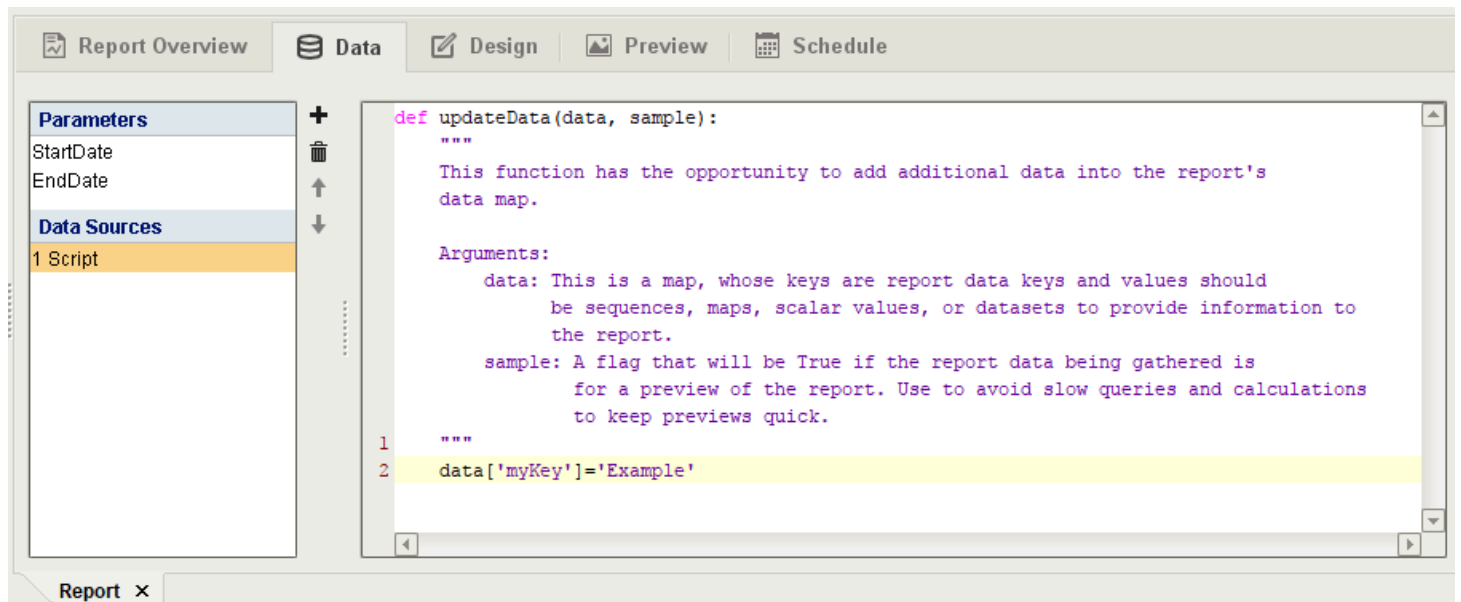
Tag Scripts

Once Enabled, these scripts are fired whenever a **Tag value changes** or an alarm event happens. You can use them for additional diagnostics, to set additional Tags, or to react to an alarm event. Because these events are on Tags, they are Gateway Scoped.



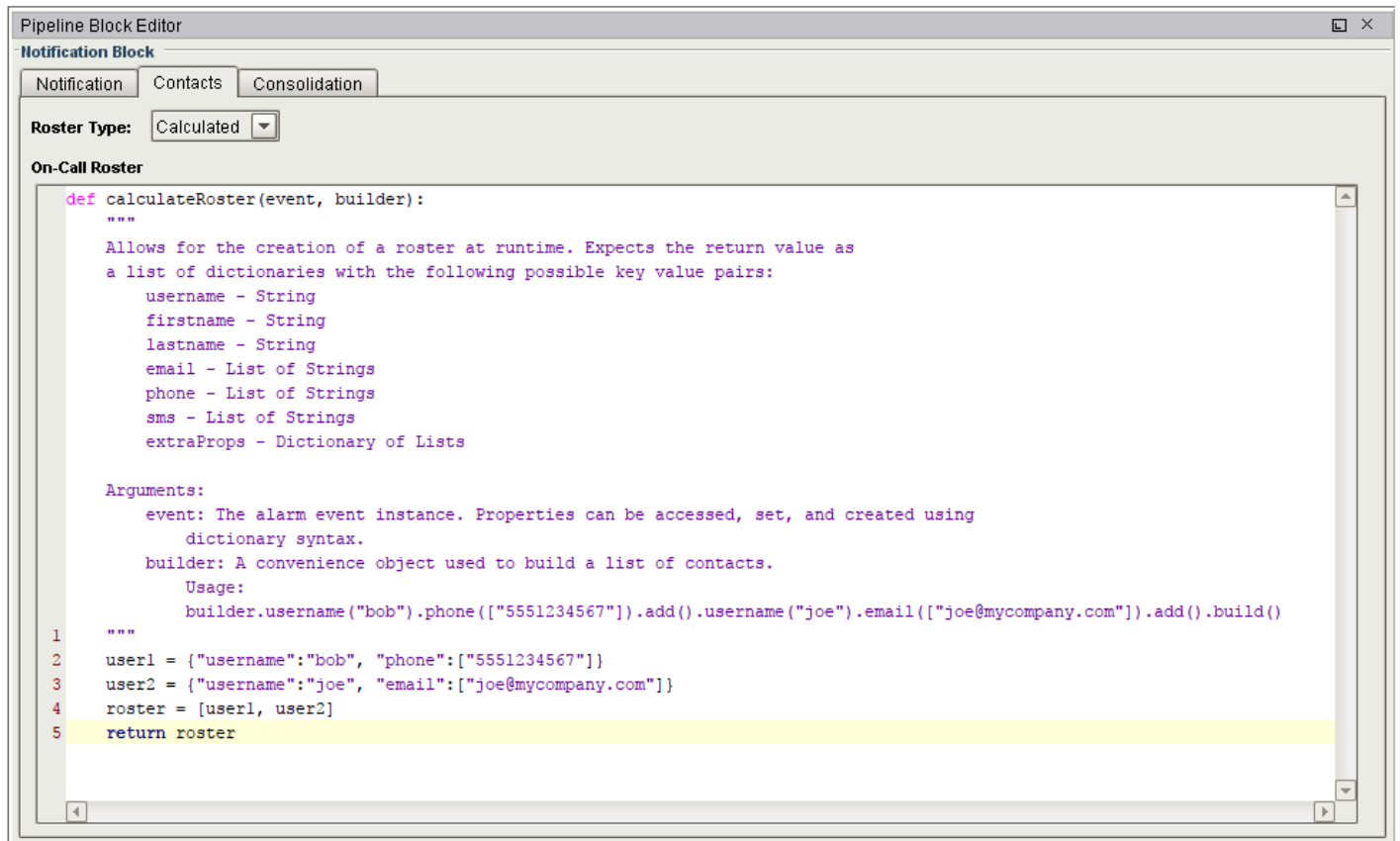
Reporting

Reporting uses scripting in many different ways to help increase the effectiveness of the report. Scripting in Reports is used to **create and modify data sources**, **manipulate charts**, and set up a script as a **scheduled report** action.



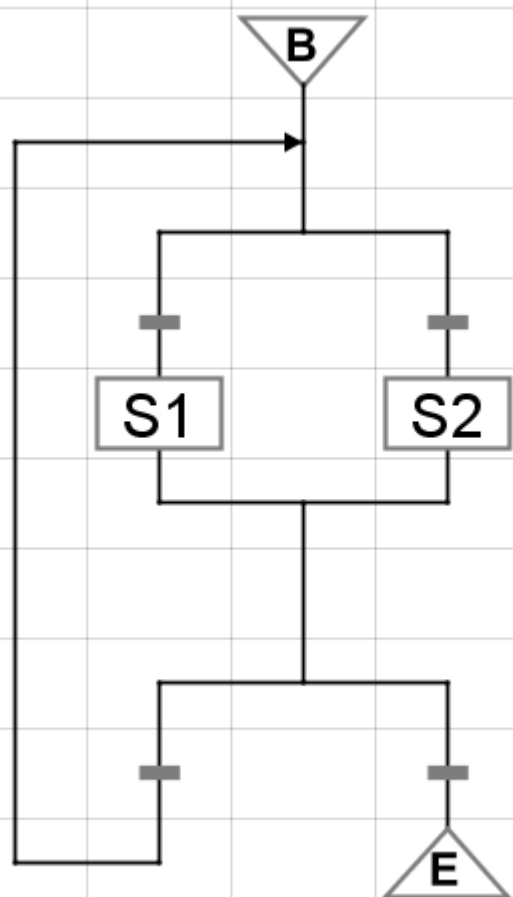
Alarming

The Alarm Notification system can also use scripting to great effect. A **script block** allows a script to be run within the pipeline, allowing data to be manipulated as the alarm event travels through the pipeline. Additionally, scripting can be used to generate a **custom roster** of users at runtime, giving full customization to who gets notified by the alarm event.



Sequential Function Charts

Sequential Function Charts (SFCs) are a flowchart of blocks that run scripts. They are executed in a specific sequential order along with some logic to potentially loop or call other charts. The scripts here can interact with the Gateway, and provide greater control when each step needs to complete before the next one can begin in multi-step processes.



This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Getting Started with Scripting in Ignition

title: Getting Started with Scripting in Ignition
sidebar_position: 1

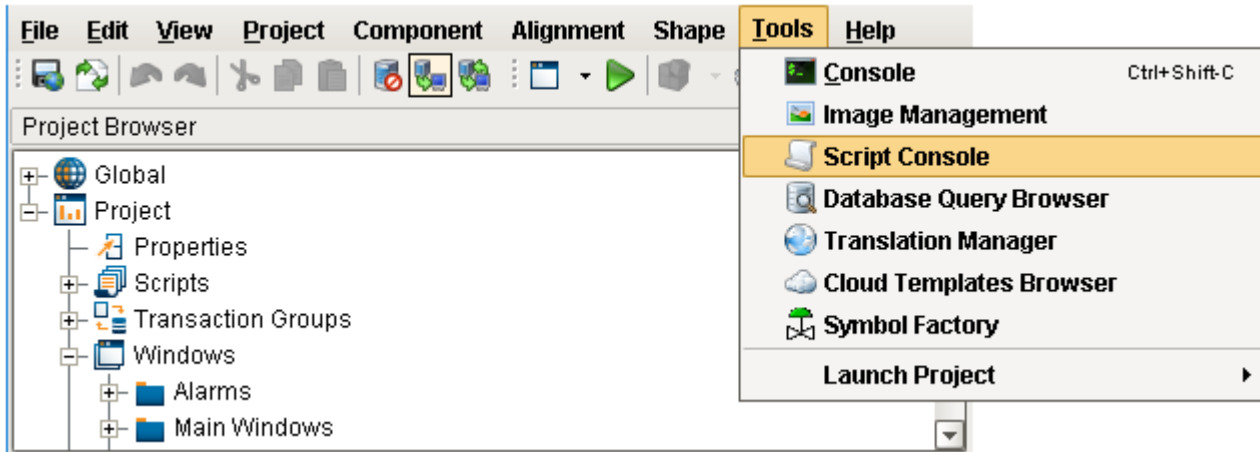
Overview

The best way to get started with scripting in Ignition is to write some very simple scripts and print them within Ignition. Let's take the phrase, "Hello World," and print it within Ignition. To show how flexible Ignition is, there are actually a few ways to easily do this. One way is to use the Scripting Console, and another way is to use a Button component. Both examples are shown below.

Example - "Hello World" Using the Script Console

When testing or writing a new script, the **Scripting Console** is very useful since you can immediately get some feedback on the results of the script. When learning Python, it is a great place to start since you don't have to create a window or component before you begin writing your code.

1. In the **Designer**, open the Script Console from the Menu Bar: **Tools > Script Console**.

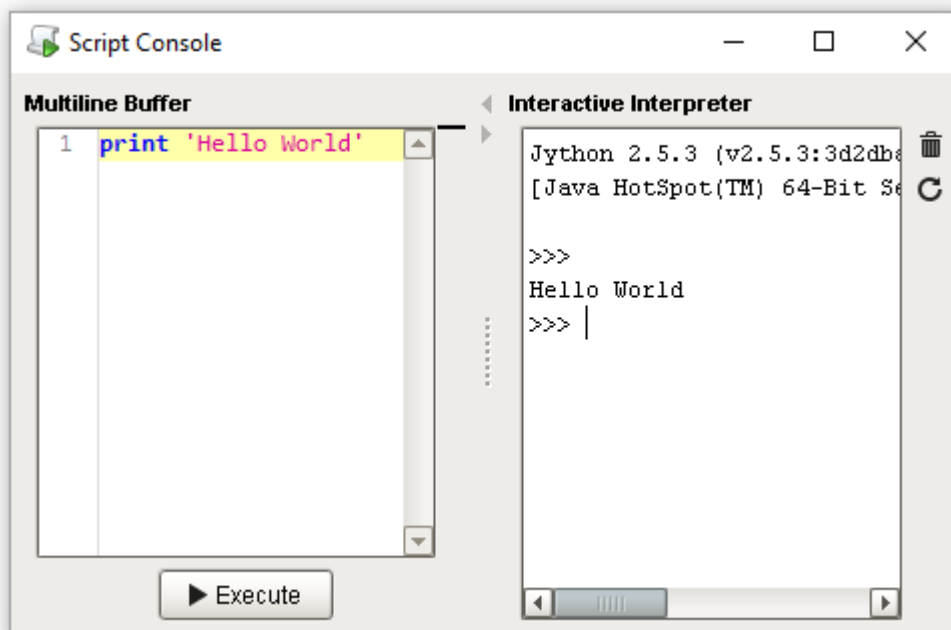


2. The Script Console will appear. Type the following code, or simply copy and paste it into the text area in the Multiline Buffer on the left side of the Script Console:

Python - Simple Print

```
print 'Hello World'
```

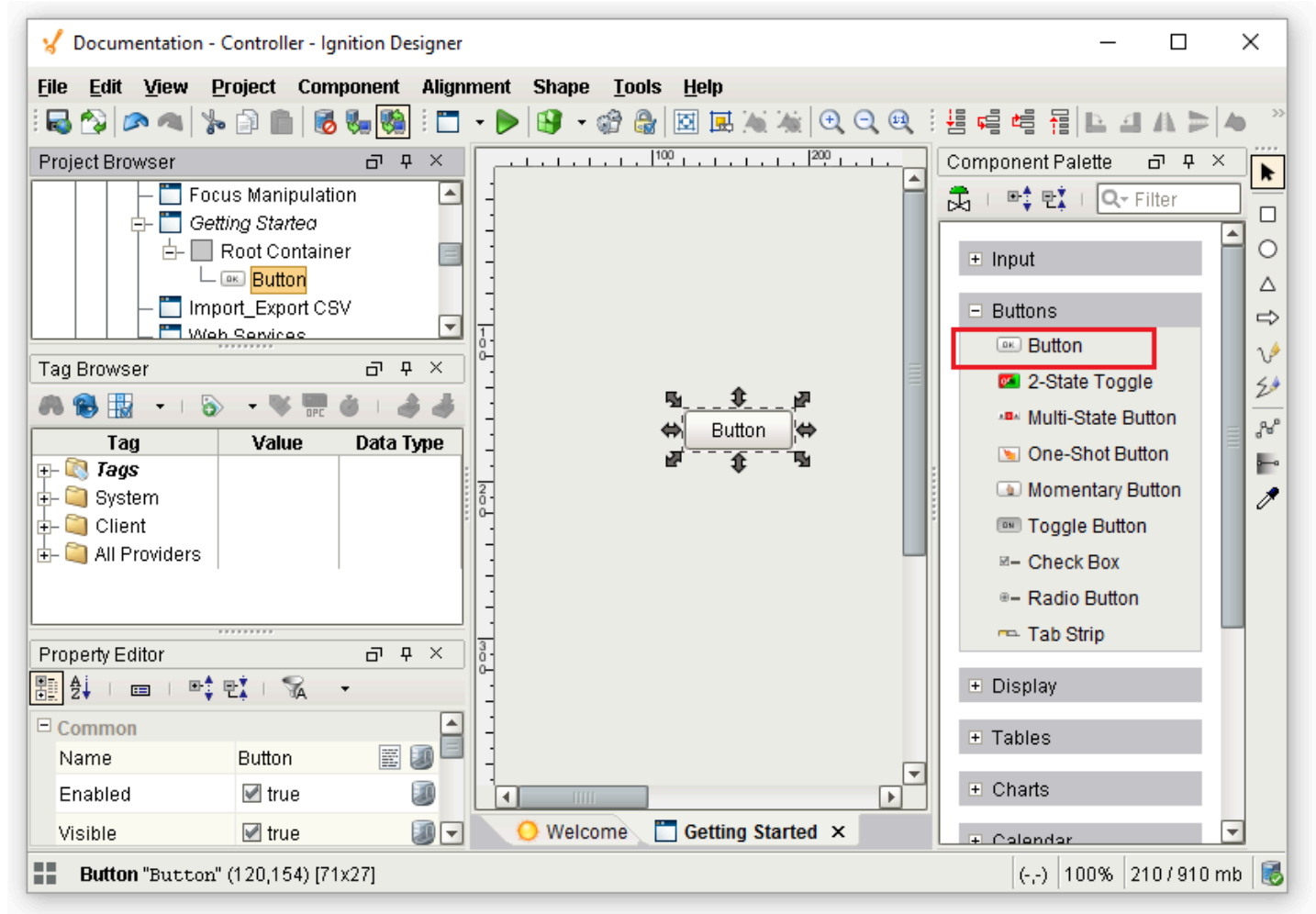
3. Click the **Execute** button at the bottom of the Script Console. You should see the message "Hello World" appear in the Interactive Interpreter on the right side of the Script Console.



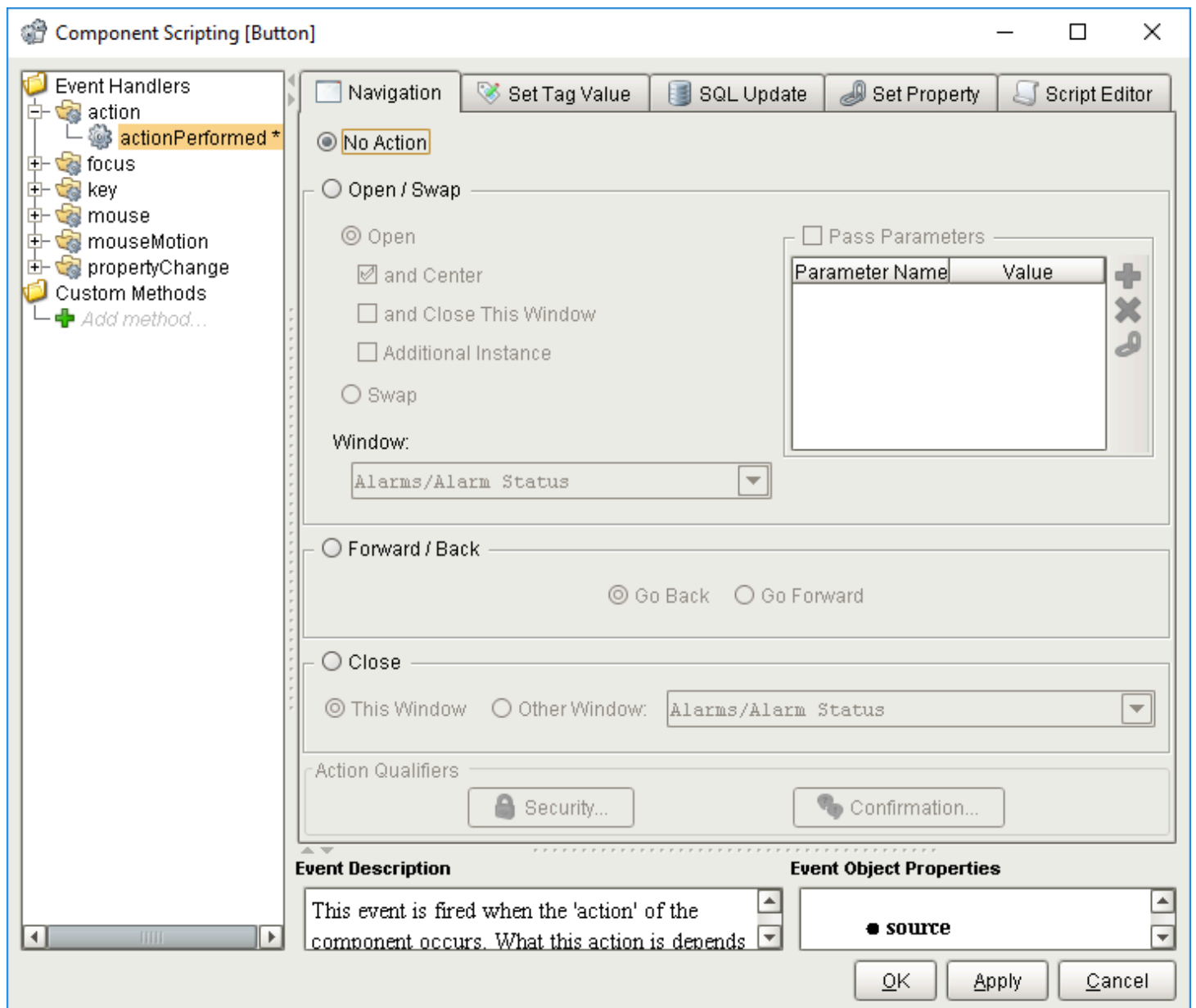
Example - "Hello World" on a Button Component

Scripts are commonly located on components in a window. In this example, let's add a script on a Button component, and print out "Hello World" when the Button is pressed.

1. In the **Designer**, create a new **Main Window**. Give it a meaningful name if you like. (We won't need to reference the name of the window in any of these examples).
2. Drag a standard **Button** component to your window.



3. Let's add a script that triggers when the Button is pressed. With the Button component selected, right click on **Scripting** to open the **Component Scripting** window. Alternatively, you can double-click the Button to open the scripting window.
4. Under Event Handlers, select the **action > actionPerformed** option, and click the **Script Editor** tab. Make sure the **actionPerformed** event is highlighted. If an event is not selected, all of the remaining features in this window will be disabled, so we will not be able to write a script. Additionally, if the wrong event is selected, then our script will not trigger when we expect it to.

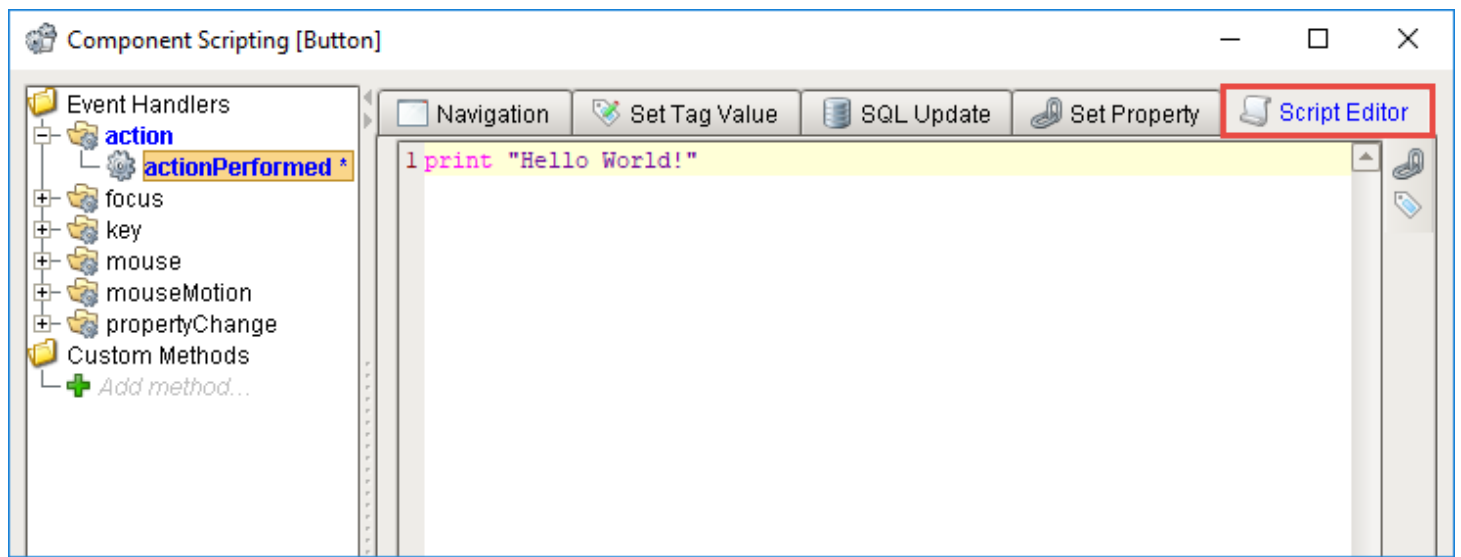


5. In the Script Editor, you will see a large editable Text Area. This is where we will type our script.

6. Let's generate a message that shows "Hello World!" Use the following code to get started. Make sure the word "print" lines up exactly to the left edge. Indention in Python means something, so we need to avoid starting our lines with a space or tab unless we're denoting a block of code under something like an **if-statement** or **function definition**. Click **OK** to close the **Component Scripting** window.

Python - Simple Print

```
print "Hello World!"
```

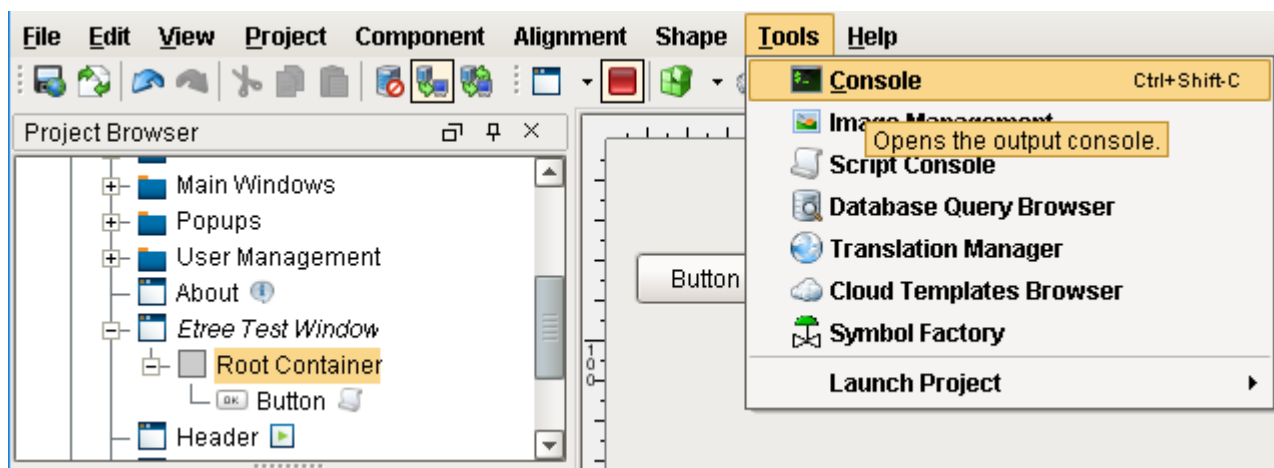


Notice the **actionPerformed** event is blue and bold. This means there is a script on this event. This is useful to know in situations where a component has scripts on multiple events.

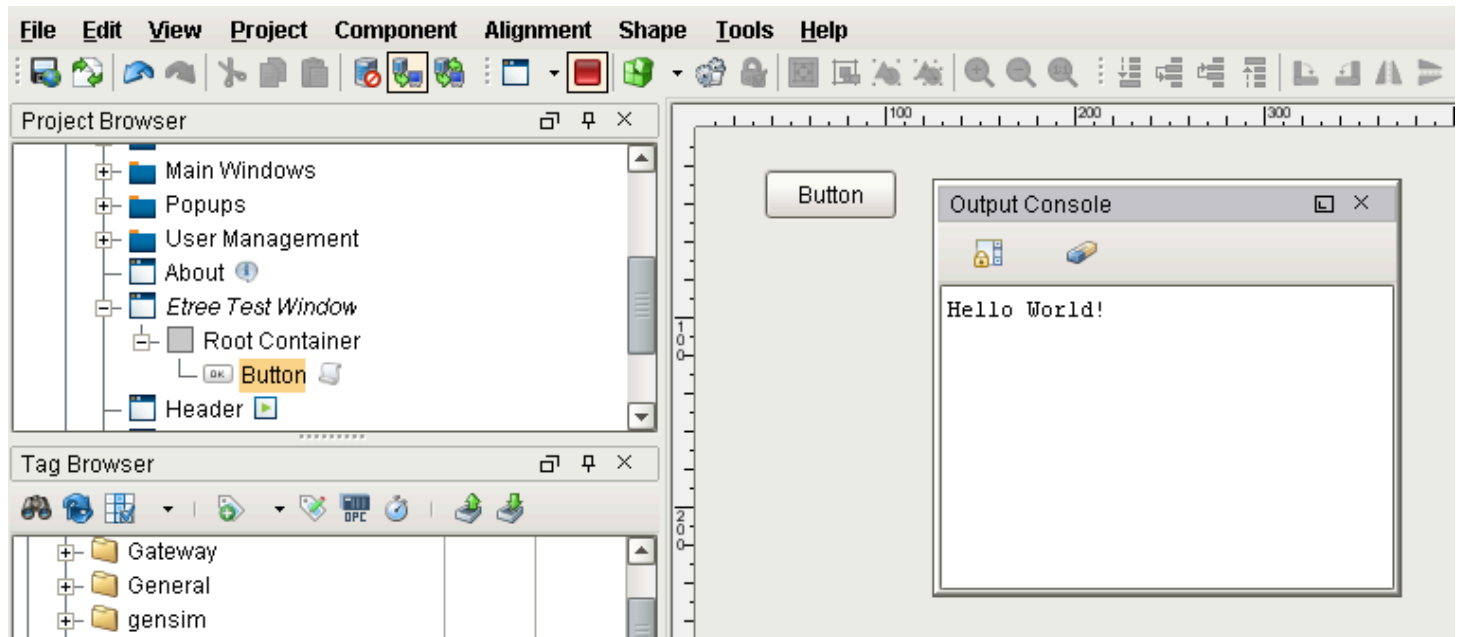
Additionally, an asterisk character (*) is next to the event. This means you have not applied/saved the changes to the script. The asterisk will disappear when you press either **OK** or **Apply**, and reappear whenever you make a change to the script. If you see this, then it means you may want to save any changes you made.

The **Script Editor** tab also has a blue color, denoting where the script is. An event will only ever have a script located on a single tab at any time. If a new tab is selected and configured, it will wipe out the work on the prior tab, for example, writing a script on **Script Editor** and then configuring the **Navigation** tab will erase the script on the **Script Editor** tab.

7. Now we can test the script. Place the **Designer** into **Preview Mode**, and press the Button. If everything is working as intended, then it should appear as if nothing happened. This is because we used print in our script, which always outputs to a console, as opposed to popping up at the user. This means we need to open the Designer's **Console** to see the results of our script. At the Designer's menu bar, select **Tools > Console**. This will make the console appear.



8. There may be a large amount of text in the Output Console. The Designer logs many different types of activities and events here, including polling events from components on other windows that are currently open in the Designer. However, the most recent events should be towards the bottom. As a quick tip, you can also click the eraser icon  to clear out the console, and then press your **Button** again to generate a new entry like we did in this example.



We can now see where print statements go when called from a component, but this isn't too useful, as we don't want our users to open the console to see messages. Print statements are very useful when troubleshooting a problematic script, or even when testing a new script. Keep this in mind as you start to delve into scripting more.

Example - Using a Message Box

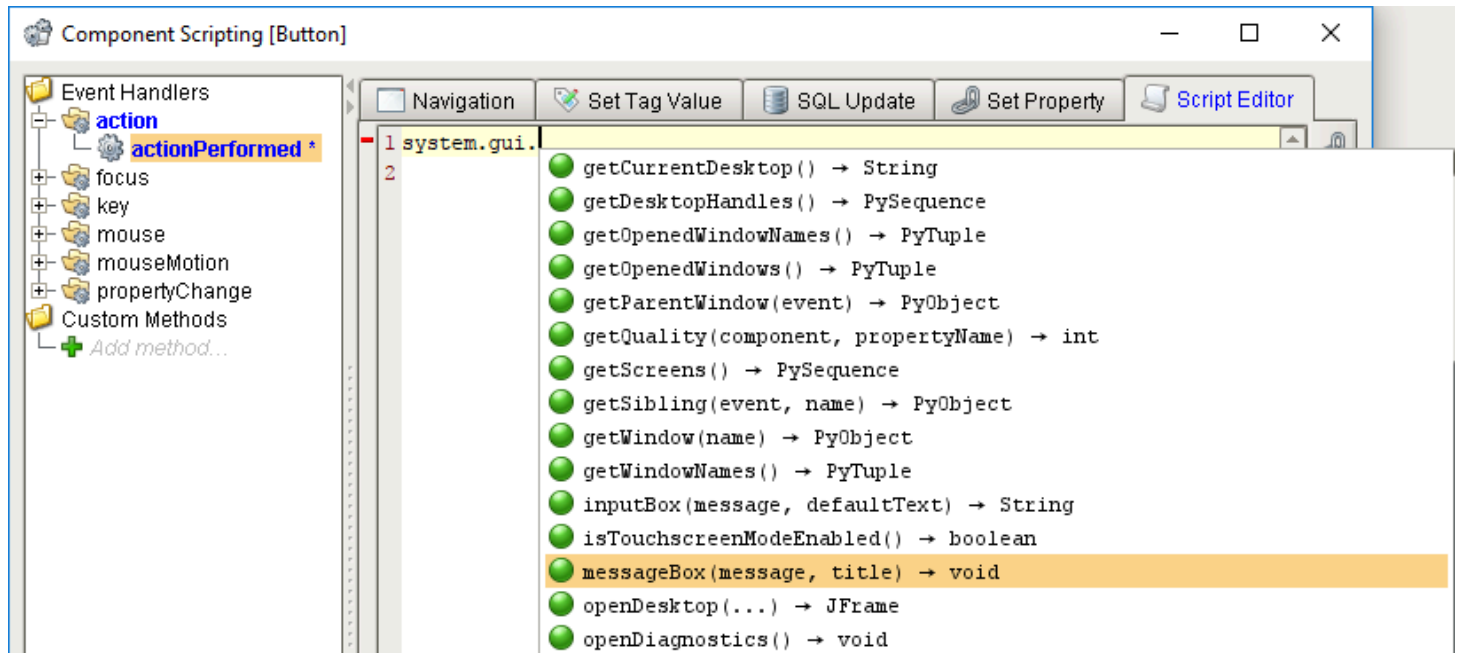
Now that we've seen how to print to the Output Console, lets make our message appear when we call it. This time, we will modify the script on the Button component to bring up a window that the user will see.

1. Open the **Component Scripting** window again, and find the script on the Button.
2. We will use one of Ignition's built-in functions, called [system.gui.messageBox](#), to display the message. This will make a message box appear, which is a modal window that we can pass a string to. Remove your old code on the Button, as we will be replacing it. Start by typing the following:

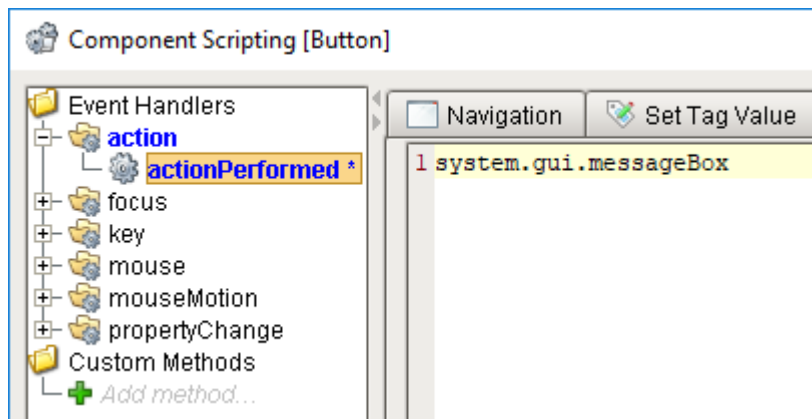
Pseudocode - Built-in Functions Begin with system.

```
system.
```

3. With the Text Cursor just to the right of the ".", hold The **Ctrl/Cmd** key and press the **Spacebar**. This will make the Autocompletion popup appear. This lists all of the available system functions. Start by clicking on **gui**, and then **messageBox**. Note, that you can still type while the popup is open to filter the results in the list.



4. Once you've selected **messageBox**, click on it, and it should start some of the code for you automatically.

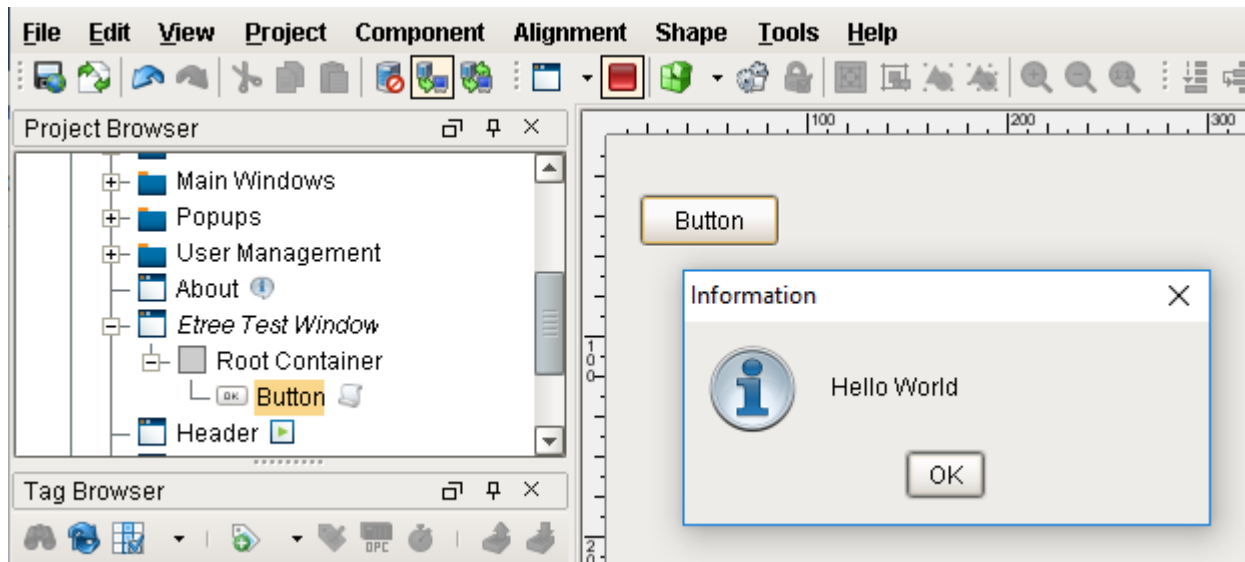


5. Complete the code by placing some parentheses and a message as a string. Alternatively, you can copy the example below.

Python - Using a System Function to Write to a Message Box

```
system.gui.messageBox('Hello World')
```

6. Place the **Designer** into **Preview Mode**, and click your Button. You will see a Message Box appear displaying the message we passed to the function.



This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Script Console



INDUCTIVE
UNIVERSITY

Scripting Console

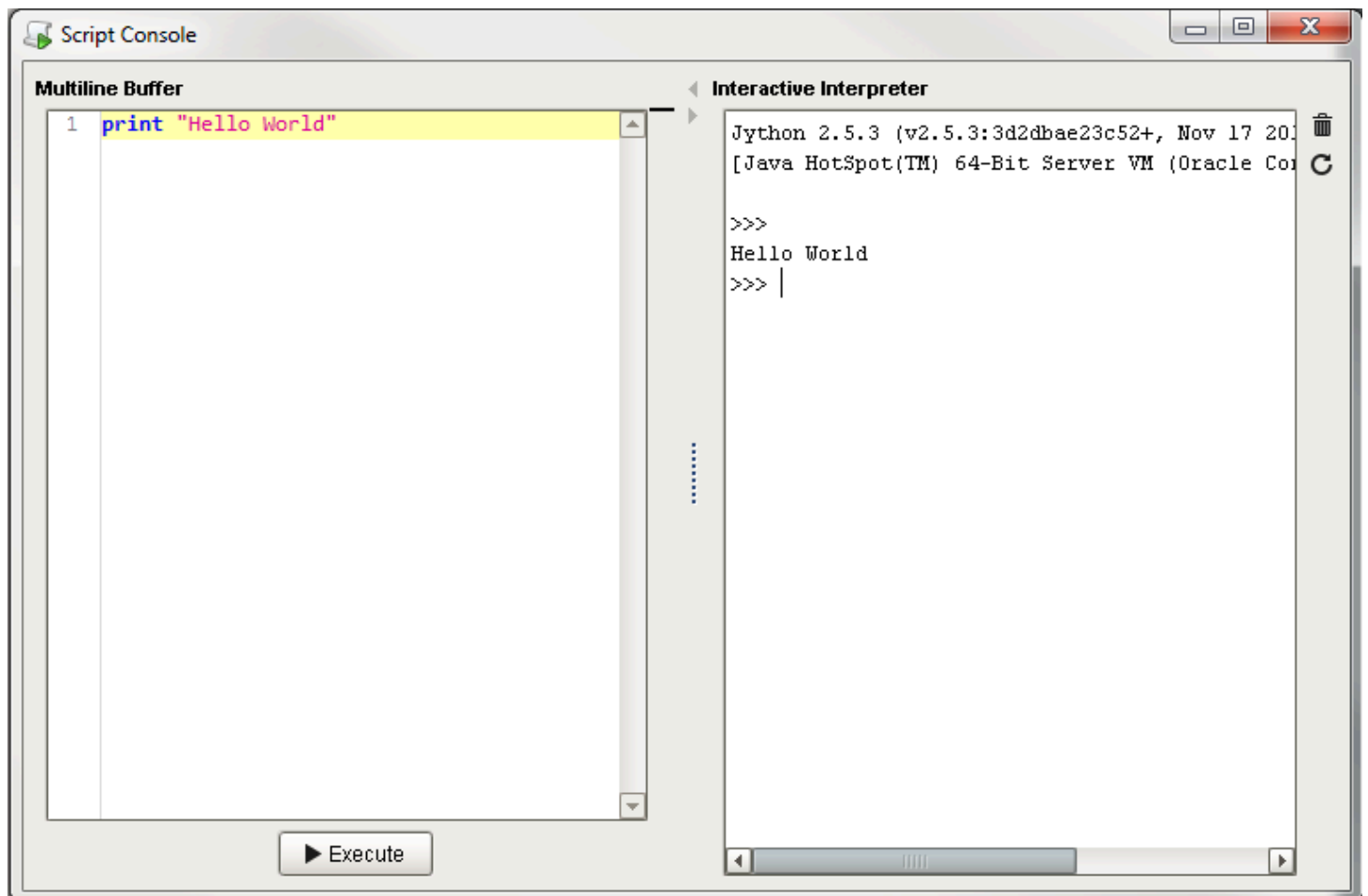
[Watch the video](#)

The Script Console is a live python terminal that is only accessible in the Designer. It is a great way to very quickly test a script as it does not rest on a scripting event or specific component. The Script Console can be opened via the **Tools > Script Console** menu. It consists of two parts: a **Multiline Buffer**, and an **Interactive Interpreter**. Code can be typed into both sides.

Due to how "[scope](#)" works, the Script Console can not interact with components on a window, but it can call Project and Shared scripts. If a Project or Shared script was recently added, then the console will need to be reset before it can be called. This can be accomplished by clicking on the Reset () button in the upper right.

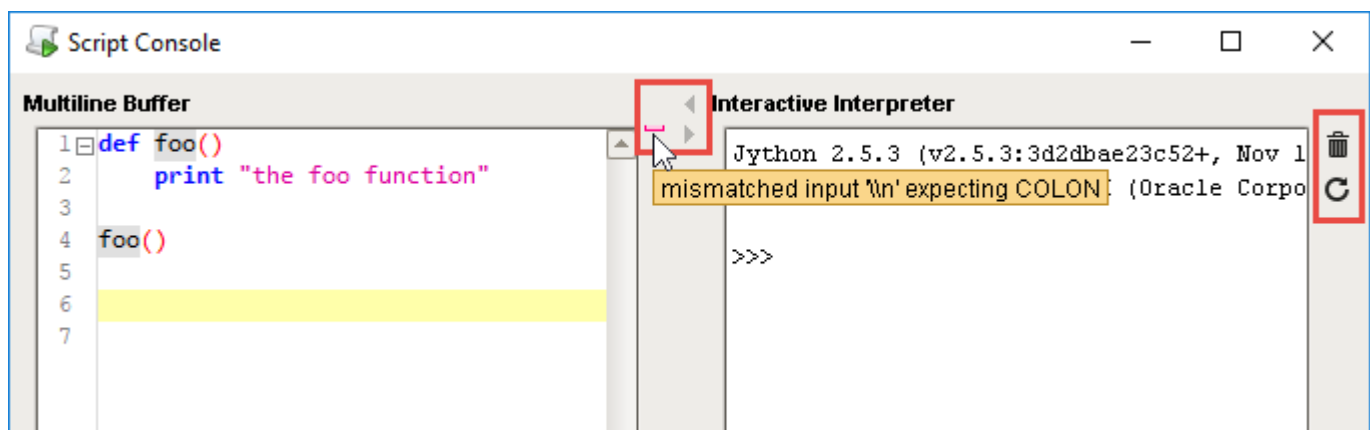
CAUTION

Gateway scoped information will not appear in either the Script Console or [Output Console](#). Instead, the output will be sent to the wrapper.log file. Alternatively, [system.util.getLogger\(\)](#) will send messages to the Gateway Console, and is a preferred method of troubleshooting Gateway scoped scripts.



Features

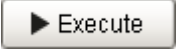
There are several icons and UI elements located on the Script Console window.



A reference of the icons can be found in the table below:

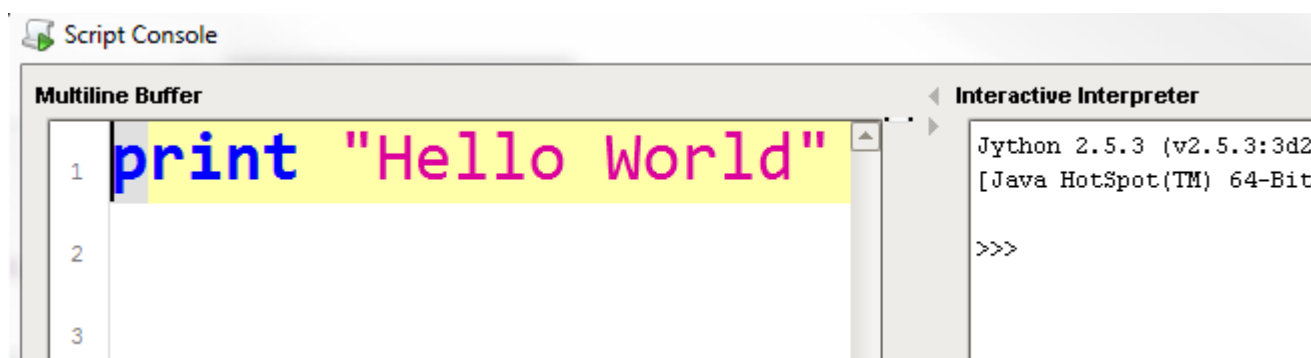
Icon	Name	Description
	Clear	Clears the text from the Interactive Interpreter.
	Reset	Clears and resets the text, and deletes all user defined objects (variables and functions) from the Interactive Interpreter.
	Expand/Collapse	Expands/Collapses Multiline Buffer and Interactive Interpreter.
	Syntax Error Highlight	Shows up next to a line in the Multiline Buffer identifying an error. Hover over the Error Symbol to see information on the exception.

Multiline Buffer

The Multiline Buffer, located on the left side of the Script Console, allows for multiple lines of code to be entered and then executed by clicking on the  button. All **print** statements will output to the Interactive Interpreter. It also supports code folding for function definitions and comments.

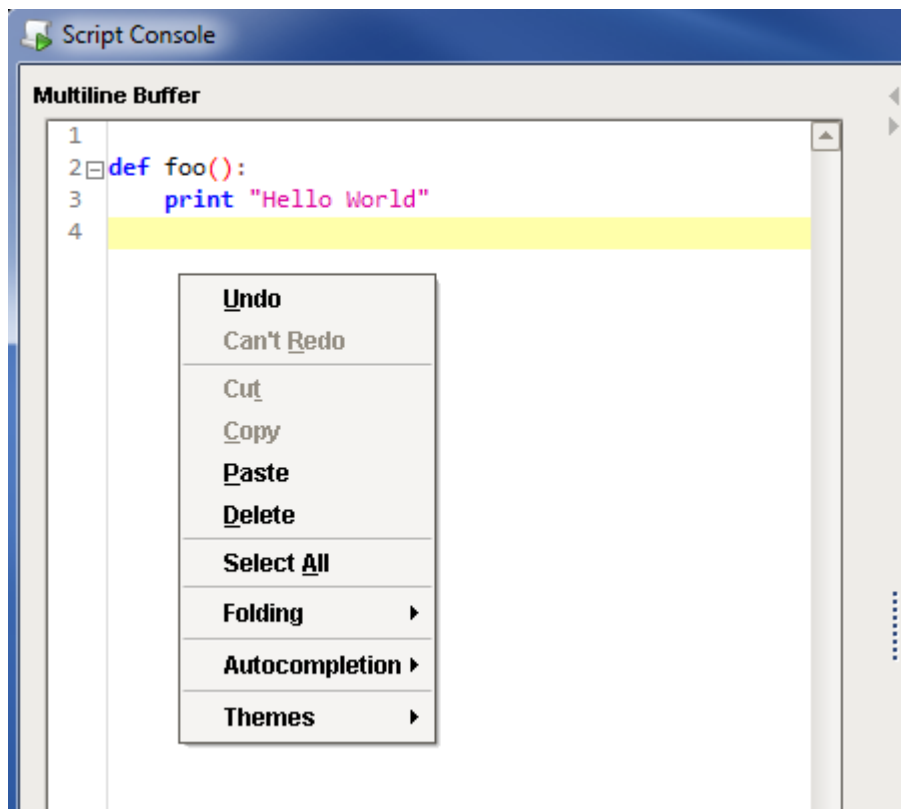
Font Size Adjustment

Font size in the Multiline Buffer can be adjusted by holding **Ctrl** and scrolling the mouse wheel.



Right Click Menu

Right-clicking on the Multiline Buffer opens a menu. The menu options are described in the table below.



Main Menu Options

Name	Description
Undo	Undoes the last action.
Redo	Gets rid of the last undo action.
Cut	Cuts the selected text.
Copy	Copies the selected text.
Paste	Pastes the selected text.
Delete	Deletes the selected text.
Select All	Selects all text in the window.

Folding

Name	Description
Toggle Current Fold	Expand or collapse the fold where the text cursor is located.
Collapse All Comments	Collapse all instances of multi-line comments. Only contiguous comments are collapsible.
Collapse All Folds	Collapse all expanded folds.
Expand All Folds	Expand all collapsed folds.

Autocompletion

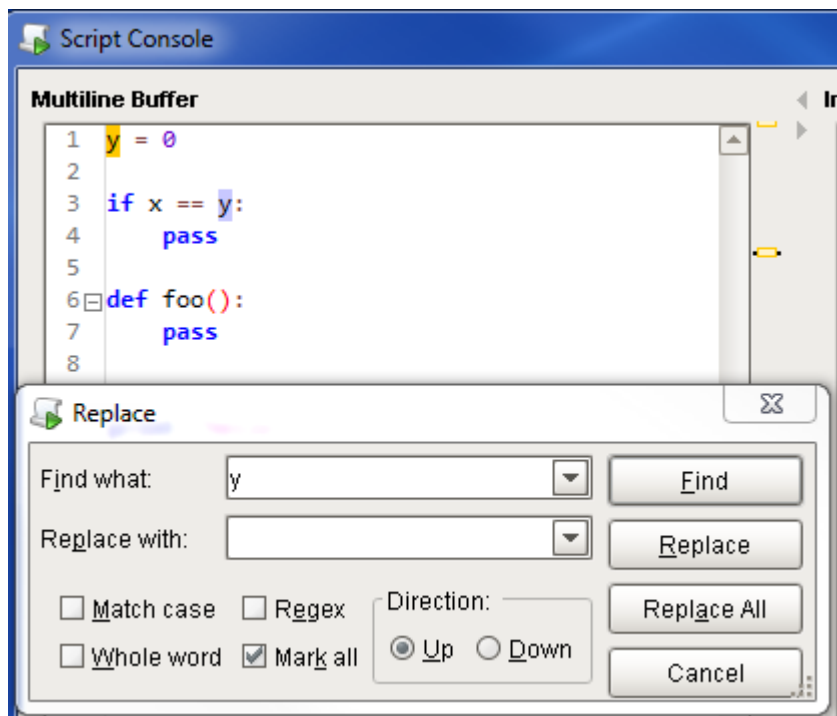
Name	Description
Automatic Activation	Determines if the Autocompletion window to automatically appear. When set to true, the window will appear after a second of inactivity once at least "system." has been typed. When set to false, the window can still be brought up manually by pressing Ctrl+Space .
Show Description Pane	Determines if the Description pane should appear in the Autocompletion window.

Themes

Name	Description
Default	Uses the default theme with light background, dark characters.
Dark	Uses a dark theme with dark background, light characters.
Visual Studio	Uses a theme reminiscent of Visual Studio. Light background, dark characters.

Find/Replace

Pressing **Ctrl+R** while the text cursor is in the Multiline Buffer opens a Find and Replace window. This will search for instances of text throughout the Multiline Buffer, and allows the user to replace all or some instances with new text.



Keyboard Shortcuts

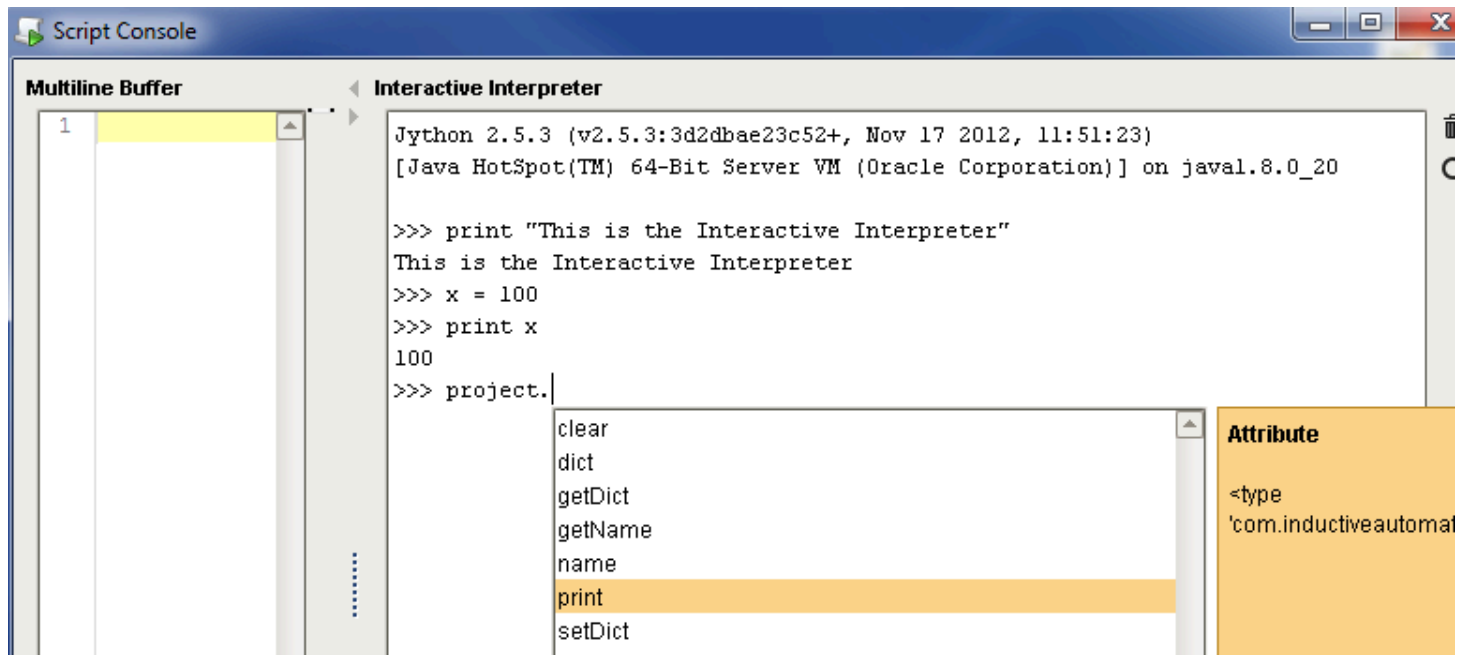
The following shortcuts apply only to the Multiline Buffer.

Keys	Description
Ctrl + </> (on the number pad)	Collapse all folds.
Ctrl + <*> (on the number pad)	Expand all folds.
Ctrl + <-> (on the number pad)	Collapse the fold on the same line as the text cursor.
Ctrl + <+> (on the number pad)	Expand the fold on the same line as the text cursor.
Ctrl + <Space>	Open Autocompletion window. By default, the window will automatically open once "system." has been typed.
Ctrl + <R>	Open Find/Replace window.
Ctrl + <Mouse Wheel Scroll>	Increase and decrease the font size.

Interactive Interpreter

The Interactive Interpreter is located on the right side of the Script Console, and allows you to run a single line of code at a time. Code is executed from the Interactive Interpreter by pressing the **Enter** key. **Print** statements from both sides of the Script Console will appear in the Interactive Interpreter.

The Autocompletion window, available in the Interactive Interpreter, has access to the current working environment so items such as Project and Shared scripts will automatically appear. They can also be typed in manually.



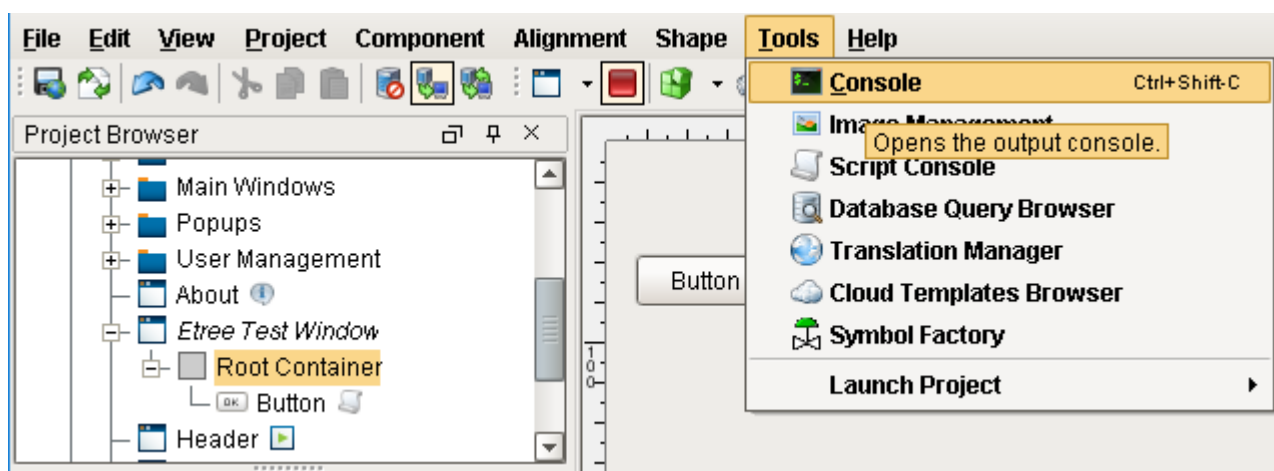
Keyboard Shortcuts

The following shortcuts apply only to the Interactive Interpreter

Keys	Description
Ctrl + <L>	Clear the Interactive Interpreter. Functionally the same as clicking the Clear button.
Ctrl + <R>	Reset the Interactive Interpreter. Functionally the same as clicking the Reset button.
Up arrow	Cycle backward through command history.
Down arrow	Cycle forward through command history.
Ctrl + <C>	Keyboard interrupt.
Ctrl + <Space>	Open Autocompletion window. By default, the window will automatically open once an " object ." has been typed such as "system" or "project," and a Project script has already been defined.
Ctrl + <A>	Move the text cursor to the start of the line. Similar to pressing the Home key.

Output Console

The **Output Console** is the script-writer's best friend. It is a dockable panel, and can be opened via the **Tools > Console** menu or the **Ctrl-Shift-C** keyboard shortcut.



The Output Console is most frequently used to test and debug Python scripts on components in Ignition. By using the **print** keyword in your script, you can observe the inner workings of your script as it executes. For example, if you executed the following script:

Python - Using the Output Console to Test and Debug Python Scripts

```
# A function that intercepts tag writes, printing out the previous value first
def writeToTag(path, value):
    prevValue = system.tag.getTagValue(path)
    print "Writing value '%s' to %s, was previously '%s'" % (value, path,
prevValue)
    system.tag.writeToTag(path, value)

writeToTag("Compressor/HOA", 2)
```

It would print the following to the console:

```
Writing value '2' to Compressor/HOA, was previously '0'
```

NOTE

The Output Console is also available in the Vision Client from **Help > Diagnostics** and selecting the **Console** tab.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Component and Window Scripting

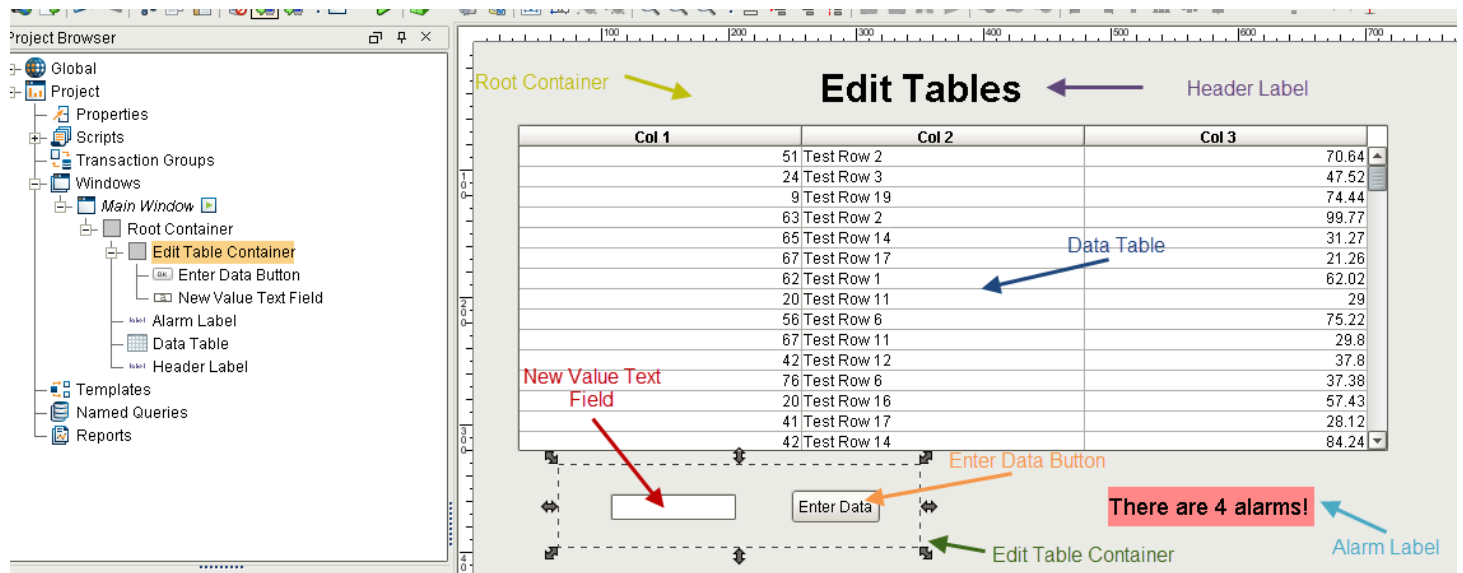
A lot of the scripting that happens in Ignition is either located on components and windows, or manipulates component and window values. Many times, the property selector () button is available to allow you to easily add in the path to another component or property from your script. However, there are also places where the property selector is not available, but you may still want to access specific components or properties. So it is important to understand how the component hierarchy of a window works as well as how to properly access components and properties in a script anywhere in a project.

Component Hierarchy

Every window in a project has a hierarchy to it, with components and containers arranged in a tree structure, with a single parent up at the top, and many children down at the bottom. While small windows with only a few components can have a simple hierarchy, windows with many containers can get much more complex. Since the components are arranged like a tree, this means that we can only move up and down through the tree structure and not sideways. To get from one component to a sibling component, we must first navigate up towards the common parent, and then back down to the desired child. Let's take a look at an example of how this works. Here we have a simple window, with just a few components:

- Root Container
 - Edit Table Container
 - New Value Text Field
 - Enter Data Button
 - Alarm Label
 - Data Table
 - Header Label

We can flip the tree around to get a better understanding of exactly how the tree works:



Here we can get a better idea of why we can only move up and down through the tree structure, and how getting to a component can be very different depending on where you start. Let's say we want to go to the New Value Text Field so that we can grab its value.

Start From	Path To New Value Text Field
Edit Table Container	1. Down to New Value Text Field
Enter Data Button	1. Up to Edit Table Container 2. Down to New Value Text Field
Data Table	1. Up to Root Container 2. Down to Edit Table Container 3. Down to New Value Text Field

To move up or down within the hierarchy, there are two special commands we can use on component objects: **parent** and **getComponent()**. The **parent** property allows us to grab a reference to whatever is directly above the component in the hierarchy, which in most cases is the Root Container. We could then access any component on the Root Container using **getComponent("Component Name")** and placing the name of the component we want to access within the parenthesis.

Pseudocode - Component Hierarchy

```
# This pseudo code shows how to grab the parent of a component
component.parent
```

```
# This pseudo code shows how to grab a child of a component
component.getComponent("Text Field")
```

Both **parent** and **getComponent()** can be used as many times as necessary to reach the desired component, drilling up or down through layers of containers or grouped components. Once you have a component reference, you can then access any one of that component's attributes by using the name of the property, just like when accessing a property on the source component.

! INFO

There is one exception to the pattern of using `.parent` to go up the hierarchy and using `.getComponent(name)` to go down. The parent of a root container is not the window, and a reference to the window does not have a `.getComponent(name)` function. To get a reference to a window, simply use [`system.gui.getParentWindow`](#) with any component's `event` object as the parameter. Once you have a reference to a window, you can use its `.rootContainer` property to get to the root of the component hierarchy, and from here you can follow the rules laid out above.

Accessing Component Properties



INDUCTIVE
UNIVERSITY

Accessing Component Properties

[Watch the video](#)

To access a property within the component, we simply need to use the scripting name of the property. The scripting name can be found in the description of each property, either by enabling the description field or hovering over the property until the mouseover text appears. The scripting names for every property on every component can also be found in the appendix. For a text field, the scripting name of the text property is just `text`, so we would need to call that on the text field which has the text property we want to access.

Pseudocode - Component Properties

```
# This pseudo code will access the text property of a component and assign it to  
value.  
value = component.text
```

Accessing a Component

Now that we have an understanding of how the component hierarchy works, we can apply that knowledge to accessing a component from anywhere within the project. While moving up and down

within the hierarchy remains the same, accessing our initial component can differ depending on where we start our script from.

From an Event Handler

Event Handlers get a special **event** object that has special properties depending on the type of event. Regardless of the event, all event objects have a source property, which gives the component that fired the event. When accessing a component from an event handler, we can first use `event.source` to get a component on the window to start at. From there, we can use `parent` or `getComponent()` as needed to get to the component we need to access.

Python - Accessing a Component from an Event Handler

```
# This would access the text property of a Text Field component.  
print event.source.parent.getComponent('Text Field').text
```

! INFO

When accessing a component from an event on the window, there will be a different path to the component than normal. If you already have a window object, you can use the function `getComponentForPath()`. This allows you to enter in the path to the component as a string (similar to expression bindings), and will end up looking something like this:

Python - Accessing a Component from an Event on a Window

```
system.gui.getParentWindow(event).getComponentForPath('Root Container.Text  
Field').text
```

You can also get the Root Container directly using the `getRootContainer()` function. This link of code works the same as the one above:

Python - Accessing a Component from an Event on a Window

```
system.gui.getParentWindow(event).getRootContainer().getComponent('Text  
Field').text
```

From an Extension Function

Extension Functions get a special **self** object which is actually a direct reference to the component that the extension function is on. This provides a direct reference point from which to access other components within the component hierarchy.

Python - Accessing a Property of a Component from an Extension Function

```
# This would access the text property of the component running the extension function script.
print self.text

# This would access the text property of the component named 'Text Field' if it is in the same container.
print self.parent.getComponent('Text Field').text
```

From a Client Event Script

Client Event Scripts are special because they don't start with a direct reference to anything on a particular window. So, we have to use another means of finding a starting point on the window. The **system.gui.getWindow()** function allows us to get a reference to a window which we can use to navigate to the root container with `.getRootContainer()`, and then to any component on that window. However, this will only work if the window is currently opened. If the window is closed, it will throw an error, which can be handled with normal exception handling.

Python - Accessing a Component from a Client Event Script

```
# Start the try block in case the window is not open.
try:
    # Grab the window reference and assign it to the variable window.
    window = system.gui.getWindow("Other Window")

    # Use the window reference to get the text property off of a text field.
    print window.getRootContainer().getComponent("Text Field").text

# Handle the exception by opening an informative error.
except ValueError:
    system.gui.errorBox("The window is not open!", "Error")
```

From a Project or Shared Script

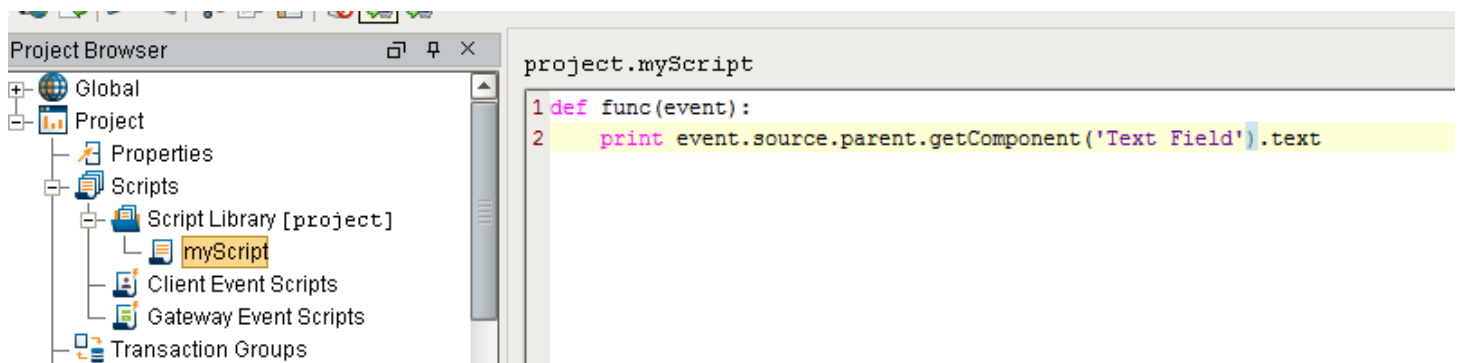
Project and Shared Scripts are unique in that how they access components can vary depending on where the Script Module is being called from and what is being passed to it. If the script module is being called from an event handler or an extension function, it is possible to pass in the event or self objects and use them within the script module.

Python - Accessing a Component from a Project Script

```
# This code would go in a project script. We are defining our function that takes an event object  
# and uses it to find the value of the text property on the text field in the same container.  
def func(event):  
    print event.source.parent.getComponent('Text Field').text
```

Python - Calling a Function from the actionPerformed Button

```
# On the action performed of a button on our window, we could then use this to call our function.  
project.myScript.func(event)
```



However, this may not always be the case. In these instances, it is possible to instead use the same method that Client Event Scripts use and grab the window object instead.

Accessing Components on Other Windows



INDUCTIVE
UNIVERSITY

Finding Components on Other Windows

[Watch the video](#)

You can also grab properties from components on other open windows from anywhere in the project using the same method used in Client Event Scripts. This allows you to grab properties on a main window from an event handler on a popup window.

CAUTION

Remember, you can only grab a property from another window if the other window is open.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Component Events



INDUCTIVE
UNIVERSITY

Component Event Handlers

[Watch the video](#)

Event Handlers

When running scripts on a component, we typically don't want it to be constantly running, but instead want the script to trigger when the user does something on screen such as clicks with the mouse. That something the user does is called an *Event* and can range from a simple mouse click, keypress to a window opening, or a component property change. When certain events happen, they trigger Event Handlers, which use a script to *handle* what happens when the event occurs.

This page lists out all of the event handlers that are on Ignition's standard modules. Any third party modules may add new components which may potentially have new event handlers.

Event Object

Every Event Handler contains an **event** object, which allows you to interact with the component and the entire window hierarchy within your script. While each **event** object has different properties depending on what Event Handler it resides in, each **event** object contains a **source** property, which is a reference to the component that fired the event. Using `event.source` not only gives us access to all of the properties available on that particular component, such as the text property of a text field,

Pseudocode - Event Handler Source Component Properties

```
# Here we start with the event object, then use source to go to the component that
fired the event,
# and then use the name of the property to access its value. In this case, we
accessed the text property.
text = event.source.text
```

but it also provides us with a way to navigate to other components within the hierarchy. For example, here we have a script on a button that references a text field.

Pseudocode - Event Handler Other Components

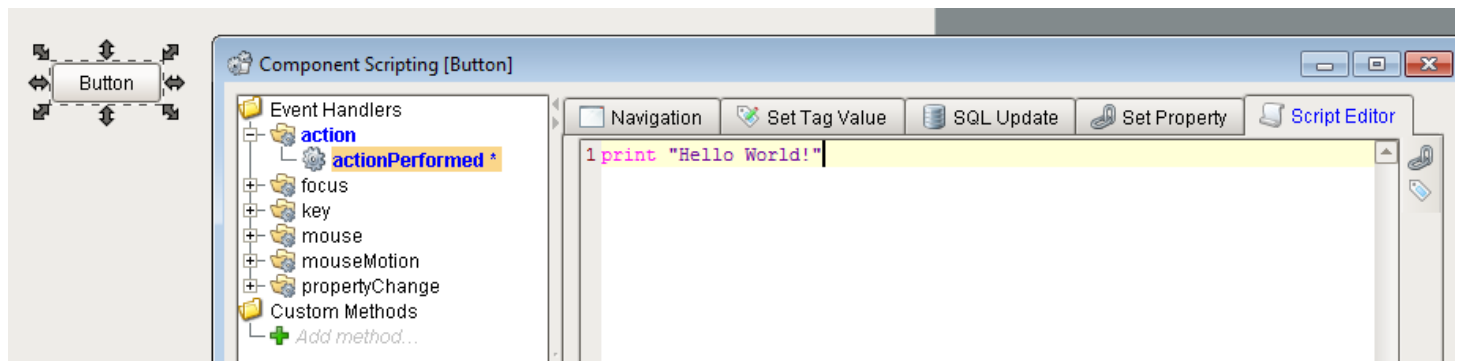
```
# Here again we start with event.source to get to the component that fired the
event, but now we use
# parent to go up to the root container, and then getComponent to navigate back
down to a different component.
text = event.source.parent.getComponent("Text Field")
```

! INFO

Even when components are disabled, most scripting events can still occur. For example, a mouse click can still happen on a disabled component, which is why we recommend using the action performed event when placing a script on a button.

Action Event Handlers

The Action category of event handlers pertains to components being "used" from the client, such as a button being pressed or a checkbox component being selected.



Events

Events	Description
actionPerformed	This event is fired when the 'action' of the component occurs. What this action is depends on the type of the component. Most commonly, this is used with buttons, where the action is that the button was pushed, via a mouse click or a key press. See the component reference for details on what the action means for other components. It is recommended to use this event over mouseClicked whenever possible.

Event Object Properties

Properties	Description
source	The component that fired this event.

Python - Button Action Performed

```
# On the actionPerformed of a button, this will print Hello World! to the console
each time the button is pressed.
print "Hello World!"
```

Property Event Handlers

Property event handlers typically trigger based on the property of a component.

Events

Events	Description
propertyChange	Fires whenever a bindable property of the source component changes. This works for standard and custom (dynamic) properties.

Event Object Properties

Properties	Description
source	The component that fired this event.
newValue	The new value that this property changed to.
oldValue	The value that this property was before it changed. Note that not all components include an accurate oldValue in their events.
propertyName	The name of the property that changed. NOTE: remember to always filter out these events for the property that you are looking for! Components often have many properties that change.

Python - Printing the Changing Property

```
# On the propertyChange of a component, this script will print out the name of the  
property that is changing.  
print event.propertyName
```

Python - Looking for a Specific Property

```
# It is common to use propertyName to look for specific properties to change. This  
is a great way to restrict how often your scripts execute  
if event.propertyName == 'text':  
    print 'The Text property changed'
```

Mouse Event Handlers

The mouse events all correspond to the clicking and movement of the mouse. They are triggered in the client by an operator interacting with a mouse. Touchscreen monitors will trigger these events when a user touches the screen, but not all touchscreens will fire the mouseEntered and mouseExited events.

Events

Events	Description
mouseClicked	This event signifies a mouse click on the source component. A mouse click is the combination of a mouse press and a mouse release, both of which must have occurred over the source component. Note that this event fires after the pressed and released events have fired.
mouseEntered	This event fires when the mouse enters the space over the source component.
mouseExited	This event fires when the mouse leaves the space over the source component.
mousePressed	This event fires when the mouse presses down on the source component.
mouseReleased	This event fires when a mouse button is released, if that mouse button's press happened over this component.

Event Object Properties

Properties	Description
source	The component that fired this event.
button	The code for the button that caused this event to fire. Use the constants <code>event.BUTTON1</code> , <code>event.BUTTON2</code> , and <code>event.BUTTON3</code> .
clickCount	The number of mouse clicks associated with this event.
x	The x-coordinate (with respect to the source component) of this mouse event.
y	The y-coordinate (with respect to the source component) of this mouse event.
popupTrigger	Returns True (1) if this mouse event is a popup trigger. What constitutes a popup trigger is operating system dependent, which is why this abstraction exists.
altDown	True (1) if the Alt key was held down during this event, false (0) otherwise.
controlDown	True (1) if the Control key was held down during this event, false (0) otherwise.
shiftDown	True (1) if the Shift key was held down during this event, false (0) otherwise.

Python - Printing on a Mouse Enter

```
# On the mouseEntered event of a component, this script will only fire if the
mouse enters the bounds of the component.
print "The mouse is inside the component space!"
```

MouseMotion Event Handlers

The mouseMotion events deal with the motion of the mouse over a component. Not all touchscreen monitors will fire these events.

 CAUTION

mouseMotion events will not trigger when the project is viewed from a mobile project as these gestures are used by the browser/device to zoom or pan.

Events

Events	Description
mouseDragged	Fires when a mouse button is clicked on a component and held, and the pointer is then dragged around.
mouseMoved	Fires when the mouse moves over a component, but no buttons are being held.

Event Object Properties

Properties	Description
source	The component that fired this event.
button	The code for the button that caused this event to fire. Use the constants <code>event.BUTTON1</code> , <code>event.BUTTON2</code> , and <code>event.BUTTON3</code> .
clickCount	The number of mouse clicks associated with this event.
x	The x-coordinate (with respect to the source component) of this mouse event.
y	The y-coordinate (with respect to the source component) of this mouse event.
popupTrigger	Returns True (1) if this mouse event is a popup trigger. What constitutes a popup trigger is operating system dependent, which is why this abstraction exists.
altDown	True (1) if the Alt key was held down during this event, false (0) otherwise.
controlDown	True (1) if the Control key was held down during this event, false (0) otherwise.
shiftDown	True (1) if the Shift key was held down during this event, false (0) otherwise.

Python - Printing on a Mouse Movement

```
# From the mouseMotion event on a component, this will print each time the mouse
moves when it is over the component.
print "The mouse is moving over the component!"
```

Key Event Handlers

The key events all have to do with the user pressing a key on the keyboard.

Events

Events	Description
keyPressed	Fires when a key is pressed and the source component has the input focus. Works for all characters, including non-printable ones, such as SHIFT and F3.
keyReleased	Fires when a key is released and the source component has the input focus. Works for all characters, including non-printable ones, such as SHIFT and F3.
keyTyped	Fires when a key is pressed and then released when source component has the input focus. Only works for characters that can be printed on the screen.

Event Object Properties

Properties	Description
source	The component that fired this event.
keyCode	The key code for this event. Used with the keyPressed and keyReleased events. Uses the standard Java key codes, see below for more information.
keyChar	The character that was typed. Used with the keyTyped event.
keyLocation	Returns the location of the key that originated this key event. Some keys occur more than once on a keyboard, e.g. the left and right shift keys. Additionally, some keys occur on the numeric keypad. This provides a way of distinguishing such keys. The keyTyped event always has a location of KEY_LOCATION_UNKNOWN. Uses the standard Java key locations, see below for more information.
altDown	True (1) if the Alt key was held down during this event, false (0) otherwise.
controlDown	True (1) if the Control key was held down during this event, false (0) otherwise.
shiftDown	True (1) if the Shift key was held down during this event, false (0) otherwise.

Python - Printing the Key Released

```
# On the keyReleased event of a component, this will print out the key code of the
key that was hit on the keyboard,
# but only on release of the key, and only when the component has focus.
print event.keyCode
```

Java Keys

The key Event Handlers use the Java KeyEvent class, which has unique identifiers for both keys and locations on the keyboard to help differentiate which key is actually being pressed on the keyboard. The numeric codes for each unique location and character can be called from the event object using a constant. For example, the letter "a" has the constant name VK_A. This can then be used to compare against the keyCode value like this:

```
if event.keyCode == event.VK_A:  
    print "The key press was a"
```

We listed the locations and some common codes below, but the full list of codes can be accessed by going to <https://docs.oracle.com/javase/8/docs/api/java/awt/event/KeyEvent.html>.

CAUTION

Some Operating Systems reserve certain keys for certain function, and will capture the key press or release before it gets sent to the Client. For example, many Operating Systems use the TAB key to shift focus to the next field.

Key Code Constants

VK_0 - VK_9	VK_END	VK_PAGE_UP	VK_DOWN
VK_A - VK_Z	VK_ENTER	VK_RIGHT	VK_PAGE_DOWN
VK_F1 - VK_F24	VK_HOME	VK_SHIFT	VK_UP
VK_ALT	VK_INSERT	VK_SPACE	VK_ESCAPE

Key Location Constants

KEY_LOCATION_LEFT	KEY_LOCATION_RIGHT	KEY_LOCATION_NUMPAD
KEY_LOCATION_STANDARD	KEY_LOCATION_UNKNOWN	

Focus Event Handlers

Focus events deal with focus moving between different components on a window. Opening windows, using tab to move around the screen, or clicking on components will trigger these events. Note that not all components can hold focus.

Events

Events	Description
focusGained	This event occurs when a component that can receive input, such as a text box, receives the input focus. This usually occurs when a user clicks on the component or tabs over to it.
focusLost	This event occurs when a component that had the input focus lost it to another component.

Event Object Properties

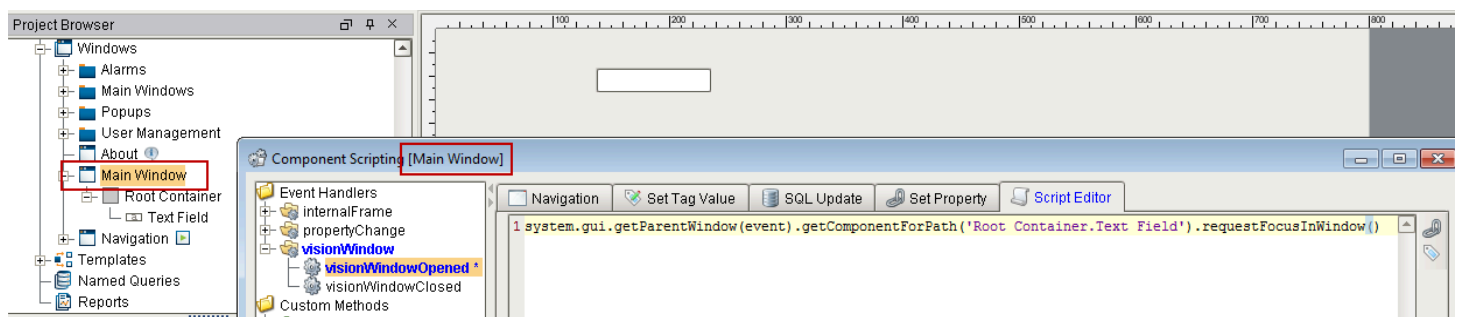
Properties	Description
source	The component that fired this event.
oppositeComponent	The other component involved in this focus change. That is, the component that lost focus in order for this one to gain it, or vice versa.

Python - Printing on Focus Gained

```
# On the focusGained event of a few different components, this script can print  
out when the component has gained focus.  
print "The component name now has focus!"
```

VisionWindow Event Handlers

The visionWindow events are specific to windows and not available elsewhere. Right-Click on the window name in the Project Browser pane and select Scripting to get access to these events. These are triggered by a window opening or closing.



Events

Events	Description
visionWindowOpened	This event is fired each time the window is opened and before any bindings are evaluated.
visionWindowClosed	This event is fired each time the window is closed.

Event Object Properties

Properties	Description
source	The vision window that fired this event.

Python - Grabbing Focus on Window Opened in Two Different Ways

From a visionWindowOpened event on a window, you can request the focus of components in the window, to start the focus on a component other than the upper left most component.

Here we grab the reference to the component using the property selector on the upper right side of the script editor.

```
system.gui.getParentWindow(event).getComponentForPath('Root Container.Text Field').requestFocusInWindow()
```

Here we can manually enter in the path to the component using our knowledge of the component hierarchy and

the getRootContainer function. Both of these functions work in the same way.

```
system.gui.getParentWindow(event).getRootContainer().getComponent("Text Field").requestFocusInWindow()
```

InternalFrame Event Handlers

The internalFrame events are fired by windows: windows are known as "internal frames" in the underlying Java windowing system that the Vision component uses. Note that the source of these events is the window itself, just like the visionWindow events above.

Events

Events	Description
internalFrameActivated	Fires whenever the window is shown or focused. If you want a script to fire every time a window is opened, use this event.
internalFrameClosed	Fires when a window is closed.
internalFrameClosing	Fires right before a window is closed.
internalFrameDeactivated	Fires when a window loses focus.
internalFrameOpened	Fires the first time a window is opened. Note that when windows are closed and cached, next time they are opened this event will not be fired. Use internalFrameActivated instead.

Event Object Properties

Properties	Description
source	The window that fired this event. Use source.rootContainer to get the root container.

Python - Printing on Frame Activation

```
# From the internalFrameActivated event on a window, this will fire each time the
window is focused, so clicking between two different windows will trigger it.
print "This window is now in focus!"
```

Cell Event Handlers

The cell event is unique in that it only appears on the **Table** component. It will trigger when something within a cell changes, and once for each cell changed.

Events

Events	Description
cellEdited	This event is fired when one of the cells in a table component has been modified.

Event Object Properties

Properties	Description
source	The table component that fired this event.
oldValue	The old value in the cell that changed.
newValue	The new value in the cell that changed.
row	The row of the dataset this cell represents.
column	The column of the dataset this cell represents.

Pseudocode - Updating a Database Table

```
# From the cellEdited event of a table component, this script can update our database table with any new data that is entered in the table

# Get the id of the row we edited and the headers
id = event.source.data.getValueAt(event.row, 'id')
headers = system.dataset.getColumnHeaders(event.source.data)

# Build our Update query.
query = "UPDATE User SET %s = ? WHERE id = ?" % (headers[event.column])
args = [event.newValue, id]

# Run the query with the specified arguments.
system.db.runPrepUpdate(query, args)
```

Item Event Handlers

The item event is unique in that it only appears on components that can be "on" or "off", such as with **Radio Buttons**, **Check Boxes**, and **Toggle Buttons**.

Events

Events	Description
itemStateChanged	This event fires when the state of the component changed. This event is the best way to respond to the state of that component changing.

Event Object Properties

Properties	Description
source	The component that fired this event.
stateChange	An integer that indicates whether the state was changed to "Selected" (on) or "Deselected" (off). Compare this to the event object's constants to determine what the new state is.
SELECTED	The constant that the stateChange property will be equal to if this event represents a selection.
DESELECTED	The constant that the stateChange property will be equal to if this event represents a de-selection.

Python - Printing on a Radio Button Selection

```
# On the itemStateChanged event of a radio button, this will print when this
specific radio button is selected.
if event.stateChange == event.SELECTED:
    print "This radio button is selected!"
```

Paint Event Handlers

The paint event is only found on the **Paintable Canvas** component, and is used to customize how the component gets painted. This event requires a heavy knowledge of programming using the Java 2D drawing tools, but there is code for a pump shape each time you add a new Paintable Canvas to a window.

Events

Events	Description
repaint	This event will fire whenever the component needs to repaint itself. It will repaint when any of its custom properties change, or when <code>.repaint()</code> is called on it. When a Paintable Canvas is first dragged onto the screen, the repaint Event Handler will be filled with an example that draws out a pump.

Event Object Properties

Properties	Description
source	The Paintable Canvas component that fired this event.
graphics	An instance of <code>java.awt.Graphics2D</code> that can be used to paint this component. The point (0,0) is located at the upper left of the component.
width	The width of the paintable area of the component. This takes into account the component's border.
height	The height of the paintable area of the component. This takes into account the component's border.

Python - Painting a Circle

On the repaint event of a paintable canvas component, this will create a circle with a gradient background color of orange and white.

```
from java.awt import Color
from java.awt import GradientPaint
from java.awt.geom import Ellipse2D

g = event.graphics

# Body
innerBody = Ellipse2D.Float(8,8,72,72)

#### Scale graphics to actual component size
dX = (event.width-1)/100.0
dY = (event.height-1)/100.0
g.scale(dX,dY)

# Paint body
g.setPaint(GradientPaint(0,40,Color.WHITE, 0,100, Color.ORANGE,1))
g.fill(innerBody)
g.setColor(Color.ORANGE)
g.draw(innerBody)
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Navigation Strategies



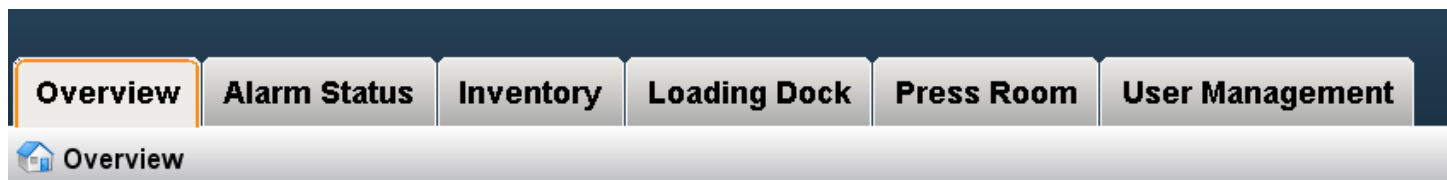
INDUCTIVE
UNIVERSITY

Navigation Functions

[Watch the video](#)

Tab Strip Navigation

Using the **Tab Strip** to navigate between windows is probably the most common way to navigate to different windows in a project. It can provide a set of tabs that can be clicked to navigate to the window that corresponds to that tab. The Tab Strip has a built in customizer, so it is easy to put together a tab strip that has a tab for every window of a project. It is best used on a docked window.



Navigation on Components

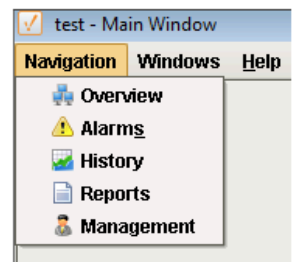
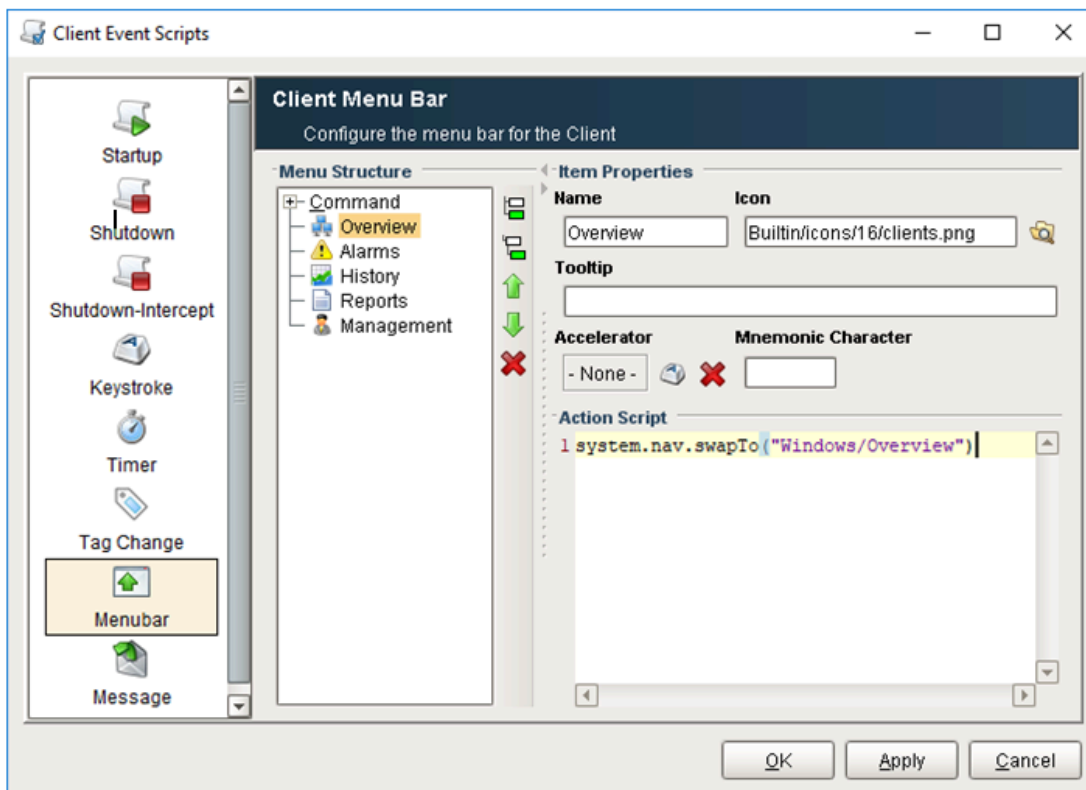
The **Script Builder** can be used to setup simple navigation options on buttons or other components. This allows you to better customize how your navigation works, instead of using a simple set of tabs. While the Navigation Script Builder is great for simple navigation, it can be limited in what it can do to navigate around your project. We can actually use the same navigation functions that drive the Script Builder to create a more complex approach to navigation, such as one button navigating to different places depending on what roles the user has.

Python - Opening Window based on User Role

```
# We first check if the user has the administrator role. That role determines
which window will open.
if "administrator" in system.security.getRoles():
    system.nav.openWindowInstance("Popups/Administrator_Popup")
else:
    system.nav.openWindowInstance("Popups/Operator_Popup")
```

Menubar Navigation

You can setup a special menu within the **Menubar** that allows you to navigate throughout the project using the scripting functions. They can be simple, like swapping to a window, or be more complex in how they navigate around the project. The benefit of using the Menubar for navigation is that it keeps navigation tucked away instead of using up valuable screen space. You can find the screen below in the **Client Event Scripts** space.



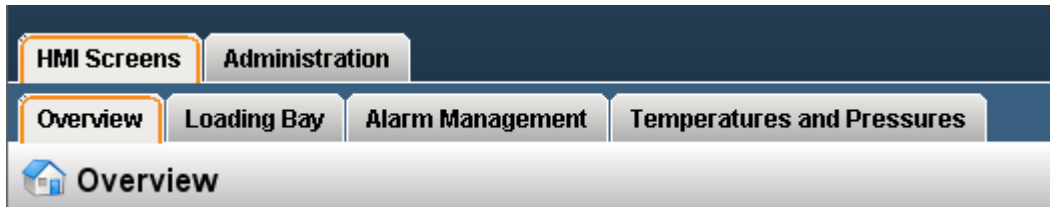
Project Templates

Each of the **project templates** that you can apply when creating a project uses a unique form of navigation. These are a great resource that can help give ideas on how to tackle navigation, or can be modified to work in your own project. These come in three types:

- **Single-tier navigation:** There is a single level of buttons, each taking you to a different window.

- **Double-tier navigation:** There are two levels of buttons. The first set controls which set is displayed below, and the second set takes you directly to those windows.
- **Multi-tier navigation:** There is a single tree-view style display that lists all available windows and can be grouped into folders.

You can select one of the project templates when a new project is created. If you'd like to use them in an existing project, you could just create a new project with the template you want and copy the pages you need to your project.



Retargeting



INDUCTIVE
UNIVERSITY

Retargeting

[Watch the video](#)

Retargeting is a special form of navigation which involves navigating to an entirely different project. Retargeting is accomplished through scripting, usually as a response to a button press or other component event. The `system.util.retarget()` function allows you to 'retarget' the Client to a different project. You can have it switch to another project on the same Gateway, or another Gateway entirely, even across a WAN. This feature makes the vision of a seamless, enterprise-wide SCADA application a reality.

The retarget feature will attempt to transfer the current user credentials over to the new project / Gateway. If the credentials fail on that project, the user will be prompted for a valid username and password. Once valid authentication has been achieved, the currently running project is shut down, and the new project is loaded.

You can pass any information to the other project through the parameters dictionary. All entries in this dictionary will be set in the global scripting namespace in the other project. Even if you don't specify any parameters, the system will set the variable `_RETARGET_FROM_PROJECT` to the name of the current project and `_RETARGET_FROM_Gateway` to the address of the current Gateway.

Retargeting can be as simple as 1 line of code, just make sure you are using the project name (no spaces allowed), and not the title. See the [retarget](#) function in the appendix for more information.

```
# Retarget to another project on the same gateway.  
# This script can be run from any button in a project.  
system.util.retarget("My_Other_Project")
```

Local Client Fallback

Local client fallback is not a navigation strategy in the normal sense (you cannot cause it to happen), but it is very useful when you want to ensure a user can continue work when the network goes out. Click the links to learn more about [Local Client Fallback](#) and about [Ignition Edge](#).

Testing Local Fallback

Testing local fallback is highly recommended before you start to depend on it in a production setting. The easiest way to test fallback is to simply unplug the network cable to the Client machine, or disable the network card on the machine. If the **Fallback Project** button is not visible on the **Gateway Connection Lost** window, check your local Gateway console and verify that the message **Started Fallback Socket on port 6501** is present in the console. Any other error message related to the **Fallback Socket Controller** indicates that some other problem has occurred (most likely the port cannot be reserved) and local fallback is not available to Clients.

Reconnect to Central

In many circumstances, the communication loss to the central Gateway is only a temporary event. To minimize the amount of time that you need to run the local Client, you can add some functionality to the Client that allows you to check on the status of the central Gateway. One way to do this is to add a **timer script** to your local Client. The script calls the `system.util.getGatewayStatus()` function at regular intervals and updates an item such as a Client Tag with the Gateway status. Then, you can bind an indicator component to the Client Tag and get ongoing visual feedback on the central Gateway status. As soon as you can confirm that the central Gateway is running again, you can call the `system.util.retarget()` function in a button to seamlessly direct the Client back to the central Gateway.

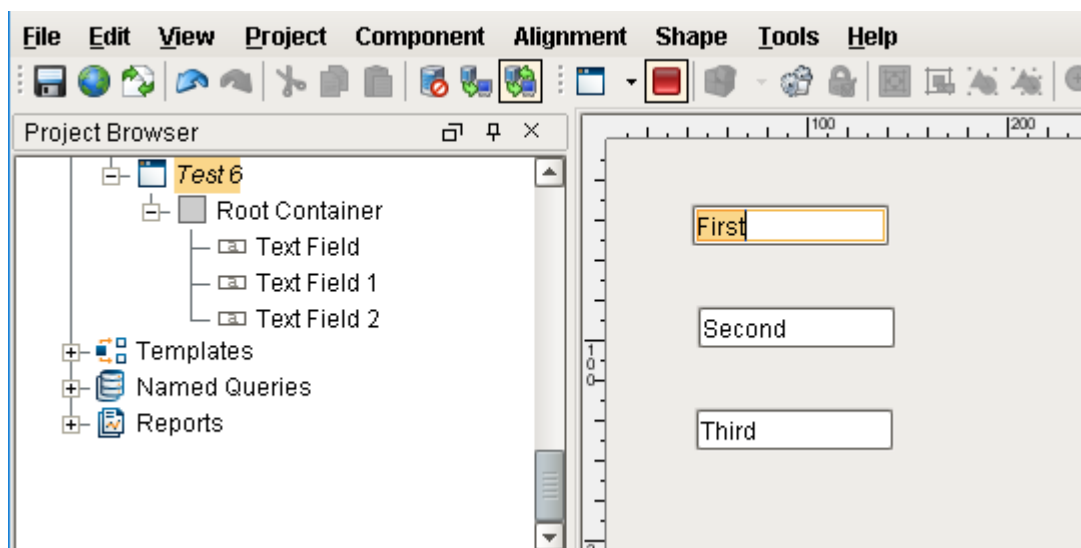
This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Focus Manipulation

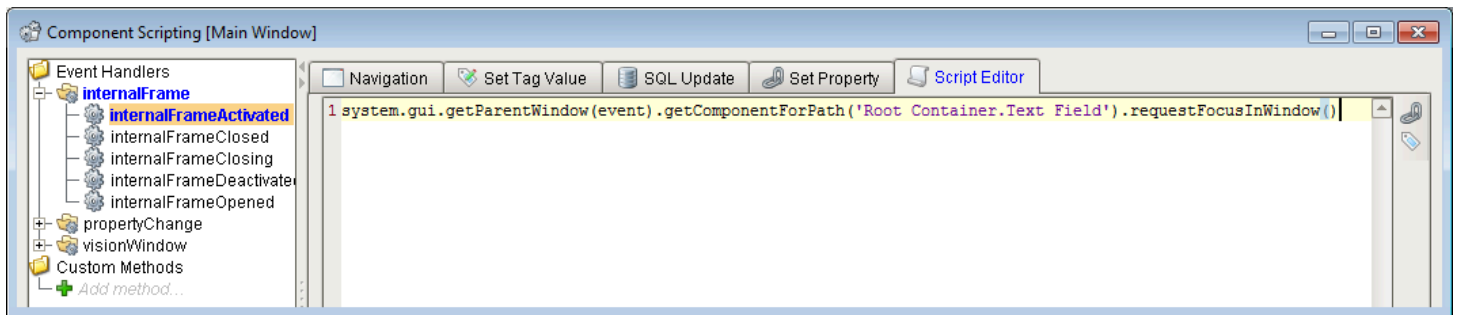
Focus Order

How components are laid out on a window determines the focus order. When tabbing through components on a window, the focus moves from left to right, then top to bottom. Focus will then cycle back to the first component. When determining order, the top left corner of the component is used. To see for yourself, drag several components into your window. Go to Preview Mode, and tab through the components.



Requesting Focus in the Window

You can programmatically request that focus be given to a component by calling the function `requestFocusInWindow()` on that function. This function is called on a component and will pull the focus to that component so it is selected and ready to use. It is best used with an input component, such as a Text Field, so the user can immediately begin typing into the component. You can use it on the `internalFrameActivated` event to bring focus to the component right when the window opens.



The getComponentForPath Function

The example above references the function **getComponentForPath**. This function can be called from a window object, and allows you to specify the full path to a component inside of the window as a single string, using the following format:

Pseudocode - Get Component For Path

```
getComponentForPath('Root Container.ComponentName')
```

When referencing a component from a window event handler, such as `internalFrameActivated`, clicking the property reference button  will use the `getComponentForPath` function. While this function is useful, you never have to use the `getComponentForPath` function. Instead you can use the component pathing that's seen from event handlers on other components. Below is a comparison of using both `getComponentForPath`, as well as the more traditional

Pseudocode - Component Hierarchy from a Window

```
# Both lines below will return a reference to the root container in the window that the event originates from.
```

```
print system.gui.getParentWindow(event).getComponentForPath('Root Container')
print system.gui.getParentWindow(event).getRootContainer()
```

```
# These lines will both reference the text property on a Label that is nested in a Group window.
```

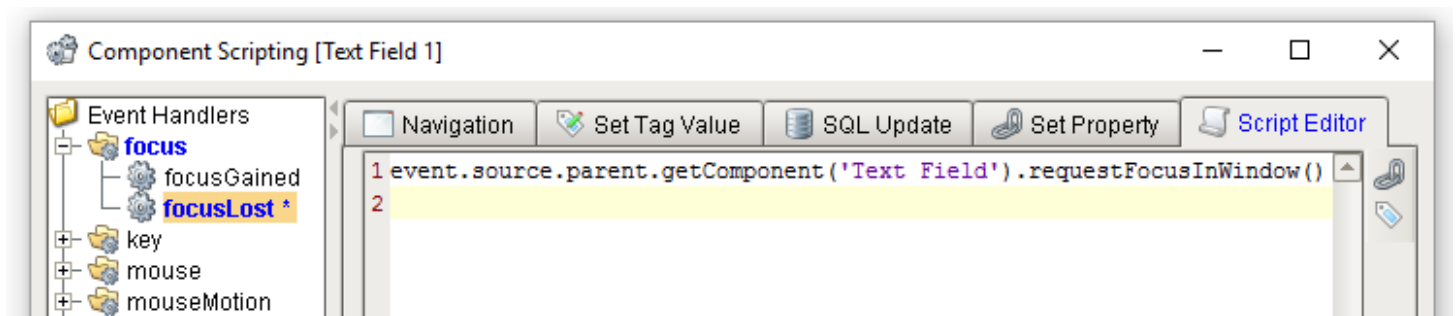
```
print system.gui.getParentWindow(event).getComponentForPath('Root Container.Group.Label')
print
system.gui.getParentWindow(event).getRootContainer().getComponent('Group').getComponent('Label')
```

Python - Requesting Focus on Frame Activation

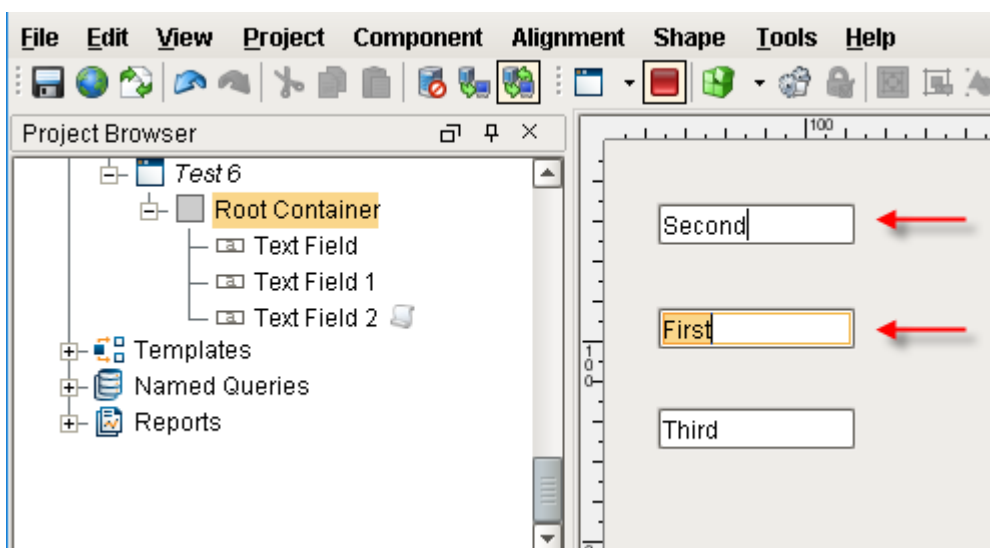
```
system.gui.getParentWindow(event).getComponentForPath('Root Container.Text Field').requestFocusInWindow()
```

Controlling Focus Between Components in a Window

In some cases, you may wish to control which component gains focus when the user clicks on a different component, or tabs away, instead of using the default focus order. You can call `requestFocusInWindow()` from the **focusLost** event to control which component should gain focus next.



However, calling `requestFocusInWindow()` may cause some irregular behavior as shown below. Notice below how the "first" and "second" Text Fields both have a text cursor in the component, but only the "second" text field has focus. This is because `requestFocusInWindow()` is being called on the `focusLost` event, which runs when one of our components loses focus. This means that while focus is being pulled to one component (the "second" Text Field), our script changes focus again to a different component.



The solution to the problem above is to have the `requestFocusInWindow()` call occur as the last part of the event trigger. This can be accomplished in one of two ways: using **Invoke Later** under

Advanced Settings, or the **invokeLater()** system function.

About Invoke Later

The concept of invoking some code later leads to a broader discussion on event handling and timing, which deviates from the purpose of this page: focus manipulation. The concept of "Invoke Later" simply means to wait for the current event to finish processing before running our focusLost script should run. In the scenario above, clicking from one component to another (or tabbing to a different component) natively calls a focus change event. A script on the focusLost event handler that uses requestFocusInWindow will also cause a focus change event, except it does so mid-execution of the native focus change event.

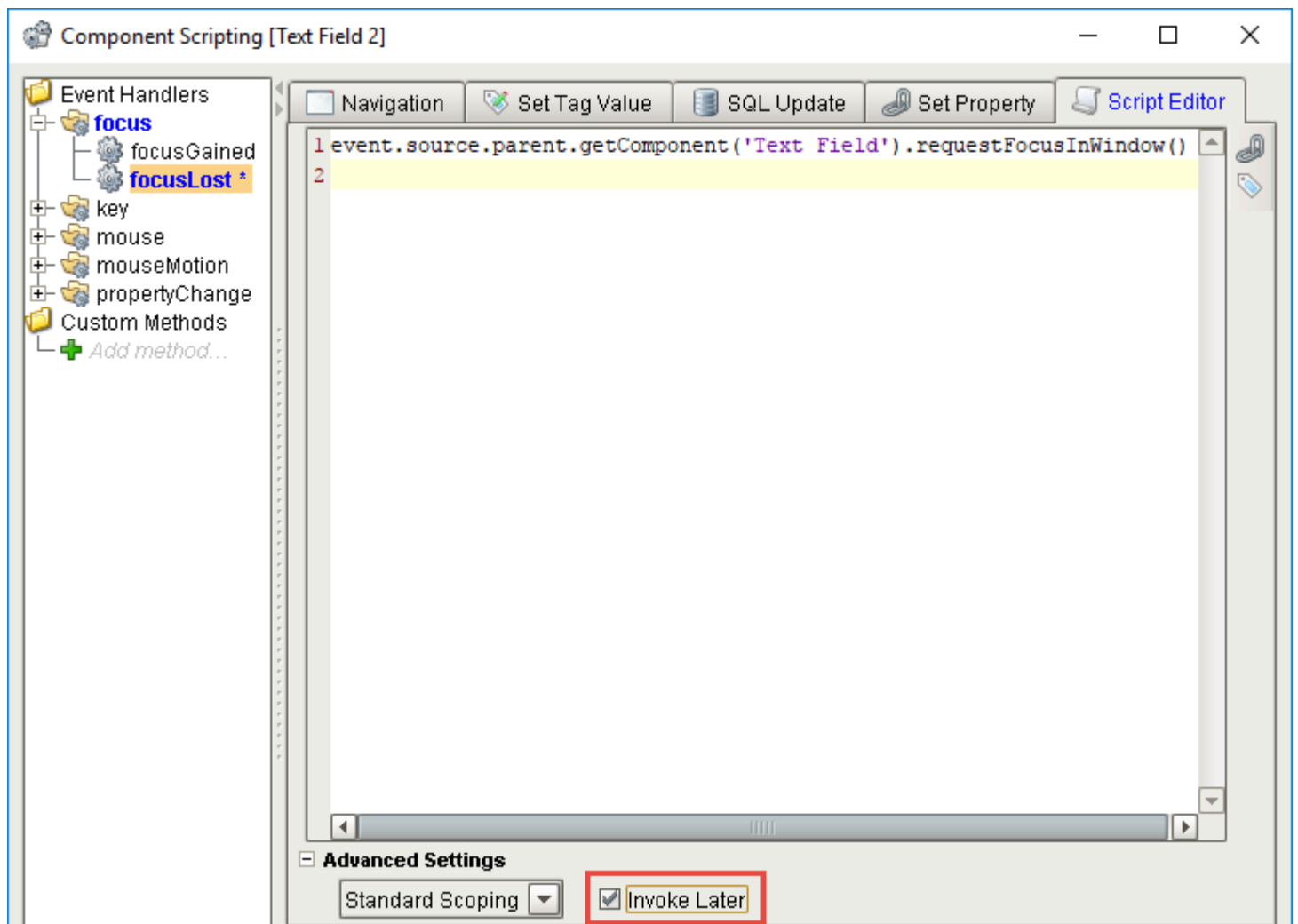
The main issue then is that focus is being moved to two components simultaneously from within the same event stack. The solution then is to have them occur in sequential order instead, with our custom focusLost script occurring last, hence using Invoke Later.

Running a script with one of the invoke later approaches mentioned here is not required on most scripts in Ignition, but it is common when the script is attempting to set or interact with focus on a window. On a related matter, invoking a script later is not the same as adding a delay mid execution.

To resolve the focus manipulation issue mentioned above, we can take one of two approaches:

Script Editor Advanced Settings

The simplest approach would be to use the **Invoke Later** option under Advanced Settings in the Script Editor.



Using the invokeLater System Function

Alternatively, we could address this by using `system.util.invokeLater` to request focus at the end of the event.

Python - Requesting Focus Later

```
# Create a function for invokeLater that requests focus.
def focus():
    event.source.parent.getComponent('Text Field').requestFocusInWindow()

# Call the function once the event has executed.
system.util.invokeLater(focus)
```

Text Areas and Focus Requests

Text Area components are slightly more complex when it comes to focus requests. Calling `requestFocusInWindow()` directly on the component will not allow the user to immediately start

typing into the component. However, we can accomplish this by calling `getViewport()`, and then calling `getView()` first as shown in the code below.

Python - Requesting Focus from a Text Area

Create a reference to the Text Area. This step could be skipped and combined with the line below, but is segregated in this example for clarity.

```
textArea = event.source.parent.getComponent('Text Area')
```

This line demonstrates how to request focus on a Text Area

```
textArea.getViewport().getView().requestFocusInWindow()
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Extension Functions



INDUCTIVE
UNIVERSITY

**Component Extension
Functions**

[Watch the video](#)

What Are Extension Functions

Extension Functions are found on the component scripting window of certain components, and they allow for more advanced customization of the component using scripting. These functions are generally more advanced and require a better understanding of Python. Unlike [Event Handlers](#), Extension Functions are not driven by a specific event, but are instead called by the component itself for a specific purpose when appropriate. This may be when the component first loads in the window, or whenever the function receives new input.

From an object-oriented point of view, Extension Functions create a custom "subclass" of the base component type. Your subclass can then override and implement parts of the functionality of the component itself, in Python. Following Python object-oriented methodology, each extension function's first argument is called `self`. That is because these are methods that belong to the component's class itself, instance methods. The value of `self` will always be the component itself. Notice that this is different than Event Handler scripts where you are given an `event` object in your scope and the component that fired the event is under `event.source`. When you write an Extension Function, there is no `event` object so the component is given to you as the `self` object instead.

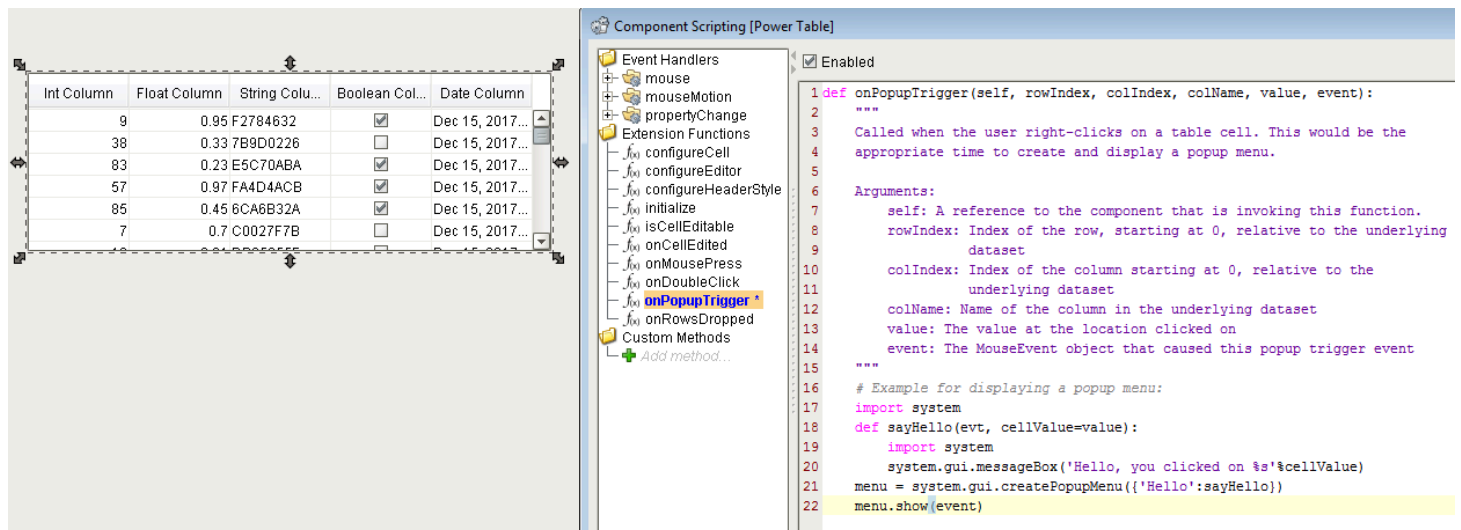
Each component Extension Function comes with its own documentation built-into the function's default implementation using a standard Python "doc-string". You will find that you are unable to edit the function's signature or docstring. Changing the method's signature (arguments or function name) would prevent the component from calling it correctly. Changing the docstring could be misleading or confusing as you'd lose the documentation for how your implementation of the function should work.

Using Extension Functions

Using an Extension Function works much like using an Event Handler. First select and **Enable** the Extension Function within the component scripting window, and then add in a script. The script will then automatically run when called.

Example - Power Table Component

The **Power Table** component has several extension functions on it that change the way the table looks or behaves. One in particular, called `onPopupTrigger()`, makes it easy to implement a right click popup menu as it is called each time a user right clicks on a cell of the table. It can be used in conjunction with the `system.gui.createPopupMenu` function to create your own custom popup menu. When the Power Table is first created in a window, it will have a pre-built example commented out in the extension function. Simply uncomment the lines of code, and see it in action! It helps to set the **Test Data** property for the Power Table so you have some data to test on.



Clicking on the "Hello" menu option will have a message box appear that has the value of the cell that was right clicked.

Python - Power Table Popup Trigger

```
import system
def sayHello(evt, cellValue=value):
    import system
    system.gui.messageBox('Hello, you clicked on %s'%cellValue)
menu = system.gui.createPopupMenu({'Hello':sayHello})
menu.show(event)
```

Int Column	Float Column	String Colu...	Boolean Col...	Date Column
9	0.95	F2784632	☒	Dec 15, 2017...
38	0.33	7B9D0226	☐	Dec 15, 2017...
83	0.23	E5C70A...	☒	Dec 15, 2017...
57	0.97	FA4D4A...	☒	Dec 15, 2017...
85	0.45	6CA6B32A	☒	Dec 15, 2017...
7	0.7	C0027F7B	☐	Dec 15, 2017...
10	0.01	5B858555	☐	Dec 15, 2017...

Example - Table Component

The **Table** component exposes an Extension Function called `getBackgroundAt()`. By implementing this function, you can control the background color of each cell of the table component using scripting. Starting with the Table component, we can add in some test data using the `TestData` property, and then add the following script to the `getBackgroundAt()` Extension Function.

The screenshot shows a Table component with 3 columns: Col 1, Col 2, and Col 3. The table contains 20 rows of test data. The Component Scripting window for the Table component is open, showing the `getBackgroundAt` function being implemented. The function checks the value of the cell in Col 1 for each row. If the value is greater than 50, it returns a light grey color. Otherwise, it returns a white color.

```

1 def getBackgroundAt(self, row, col, isSelected, value, defaultColor):
2     """
3     Called for each cell, returns the appropriate background color. Do not
4     block, sleep, or execute any I/O; called on painting thread.
5
6     Arguments:
7         self: A reference to the component that is invoking this function.
8         row: The row index of the cell
9         col: The column index of the cell
10        isSelected: A boolean representing if this cell is currently selected
11        value: The value in the table's dataset at index [row,col]
12        defaultColor: The color the table would have chosen if this function
13                     was not implemented
14    """
15    # For each cell, we check the value of the cell in the same row but in Col 1.
16    if self.data.getValueAt(row, "Col 1") > 50:
17        # If the value is greater than 50, we return a light grey color.
18        return system.gui.color("lightgrey")
19    else:
20        # Otherwise, we return a white color.
21        return system.gui.color("white")

```

Python - Table Row Color

```

# For each cell, we check the value of the cell in the same row but in Col 1.
if self.data.getValueAt(row, "Col 1") > 50:
    # If the value is greater than 50, we return a light grey color.
    return system.gui.color("lightgrey")
else:
    # Otherwise, we return a white color.
    return system.gui.color("white")

```

The script allows us to change the background color of the component based on the value of the cell in Col 1 for that row. After clicking the OK button to save our script, you should see the script run automatically, and the background color of the table will change.

Col 1	Col 2	Col 3
62	Test Row 11	77.41
56	Test Row 3	66.94
56	Test Row 11	87.83
84	Test Row 18	68.28
80	Test Row 14	99.92
50	Test Row 1	46.59
64	Test Row 2	50.9
81	Test Row 0	42.25
49	Test Row 17	65.5
43	Test Row 1	65.14
25	Test Row 1	14.48
11	Test Row 8	55.9
25	Test Row 14	59.7
7	Test Row 8	49.66
99	Test Row 12	31.52
17	Test Row 14	81.75
36	Test Row 19	60.05
50	Test Row 2	39.99
90	Test Row 17	60.99
90	Test Row 15	86.01
59	Test Row 4	58.83
89	Test Row 9	39.37

Example - User Management Component

The **User Management** component has many Extension Functions that provide a way to customize how the component works. The `filterUser()` extension function is useful for filtering out users you don't want to see in the user source, preventing users from editing that user in the client. We can add a simple script to the `filterUser()` Extension Function that will hide the user from the list if they have the Administrator role.

Users

Username	Roles	Schedule
admin	Administrator	Always
arthur	Administrator	Always
jane	Manager	Always
john	Operator	Always
mary	Administrator, Manager, Operator	Always
susan	Operator	Always

Roles

Role name	# of Members
Administrator	3
Manager	2
Operator	3

Component Scripting [User Management]

☒ Enabled

Event Handlers

- mouse
- mouseMotion
- propertyChange

Extension Functions

- filterUser**
- filterRole
- filterSchedule
- onCreateUser
- onDeleteUser
- onSaveUser
- onCreateRole
- onDeleteRole
- onSaveRole

Custom Methods

- Add method...

```

1 def filterUser(self, user):
2     """
3     Called for each user loaded into the management table. Return false to
4     hide this user from the management table. This code is executed in a
5     background thread.
6
7     Arguments:
8         self: A reference to the component that is invoking this function.
9         user: The user object itself. Call user.get('propertyName') to
10            inspect. Common properties: 'username', 'schedule', 'language'. Call
11            user.getRoles() for a list of rolenames.
12
13     """
14     # Check to see if the user has the Administrator role.
15     if "Administrator" in user.getRoles():
16         # Return 0 to hide them if they do.
17         return 0
18     else:
19         # Otherwise, show the user in the table.
20         return 1

```

OK Apply Cancel

```
# Check to see if the user has the Administrator role.
if "Administrator" in user.getRoles():
    # Return 0 to hide them if they do.
    return 0
else:
    # Otherwise, show the user in the table.
    return 1
```

By enabling this script, we now only see the users without the Administrator role in the list of users.

Users

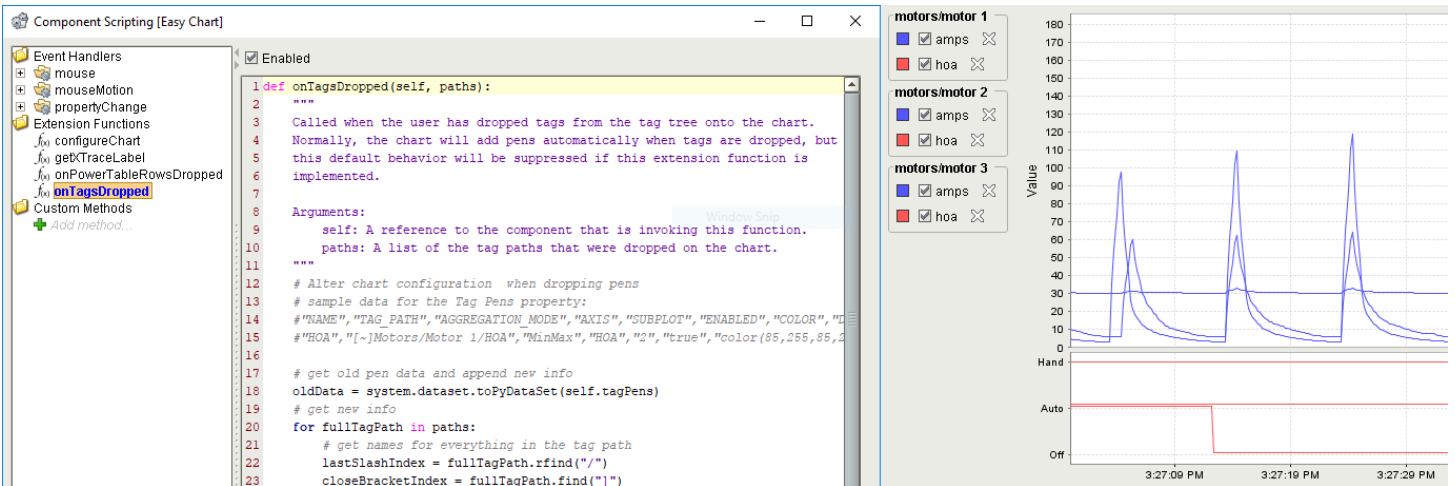
Username	Roles	Schedule
jane	Manager	Always
john	Operator	Always
susan	Operator	Always

Roles

Role name	# of Members
Administrator	3
Manager	2
Operator	3

Example - Ad Hoc Charting

The Easy Chart component has an Extension Function to allow scripting when a Tag is dropped onto it (see [Ad Hoc Charting](#)). There is a lot of customization possible in the designer but any client side changes to the Tag Pens dataset must be done here. Generally, people want to change what axis and subplot a pen goes into based on some other information. Below is a simple example that uses the Tag's name to determine this. For this example to work you need to have two subplots and a second axis named "HOA".



Python - Drop Tags on Easy Chart

```

# Alter chart configuration when dropping pens
# sample data for the Tag Pens property:
#"NAME", "TAG_PATH", "AGGREGATION_MODE", "AXIS", "SUBPLOT", "ENABLED", "COLOR", "DASH_PATTERN", "HOA", "[~]Motors/Motor 1/HOA", "MinMax", "HOA", "2", "true", "color(85,255,85,255)", "", "1"

# get old pen data and append new info
oldData = system.dataset.toPyDataSet(self.tagPens)
# get new info
for fullTagPath in paths:
    # get names for everything in the tag path
    lastSlashIndex = fullTagPath.rfind("/")
    closeBracketIndex = fullTagPath.find("]")
    tagName = fullTagPath[lastSlashIndex+1:]
    tagPath = fullTagPath[closeBracketIndex+1:]
    groupName = fullTagPath[closeBracketIndex+1:lastSlashIndex]

    # find which tags are named "hoa" and put them in the HOA subplot.
    if tagName.lower() == "hoa":
        axis = "HOA"
        subplot = 2
        color = "color(255,85,85,255)" #red
        digital = "true"
    else:
        axis = "Default Axis"
        subplot = 1
        color = "color(85,85,255,255)" #blue
        digital = "false"

    # append to the old pen data
    newData = system.dataset.addRow(oldData, [tagName,tagPath,"MinMax",axis,subplot,"",color,digital])

    # push new pens back to the tagPens property
    self.tagPens = newData
  
```


This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Custom Component Methods



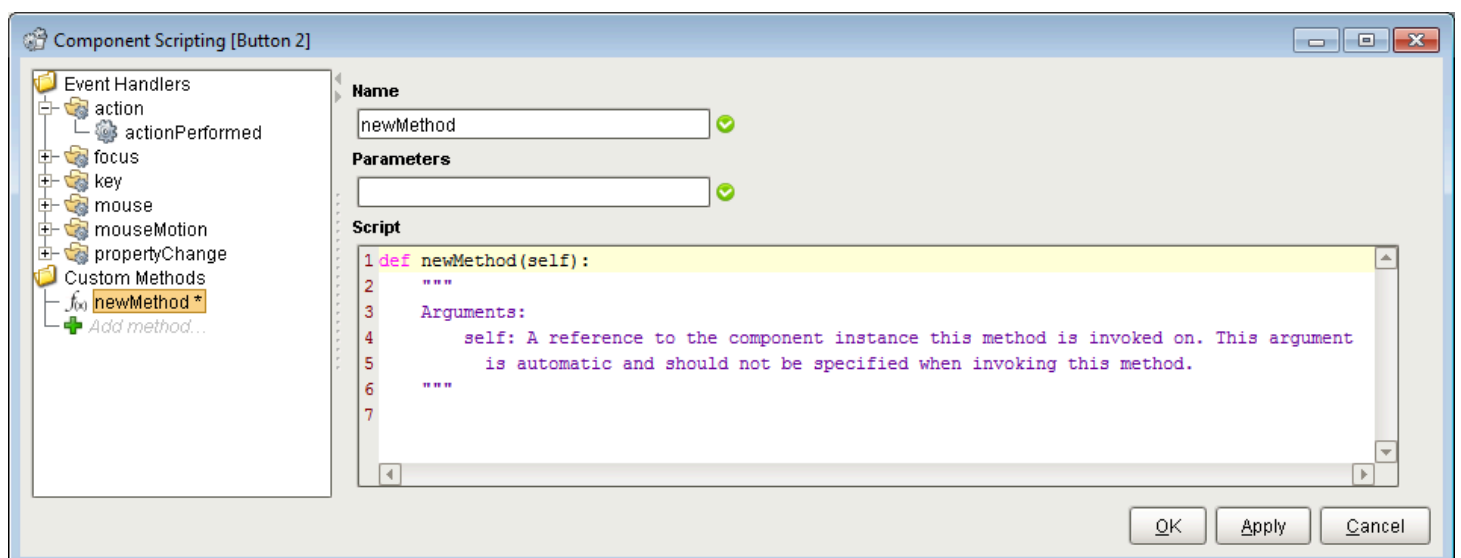
INDUCTIVE
UNIVERSITY

Component Custom Methods

[Watch the video](#)

Custom Methods

Custom Methods function much like **Project or Shared Scripts** in that you write a script and call it from somewhere else. However, Custom Methods are written on a component instead of a separate scripting section, and are also automatically passed the value of `self`, just like an **Extension Function**. The `self` object provides the script with easy access to everything within the window. In addition to `self`, Parameters can be added that you can use to pass in other objects into your Custom Method.



Custom Methods can then be called from the same component or from other components. Custom Methods are called just like any other method on a component: `.methodName()`. So if I had a custom method on a text field, and I wanted to call it from the `actionPerformed` event on a button in the same container, I would use:

Python - Custom Method

```
# The name of the custom method in this instance is myCustomMethod.  
event.source.parent.getComponent("Text Field").myCustomMethod()
```

Templates are another good use for custom methods. By adding a custom method directly to a template, all the components that make up the template can call the custom method from the template itself. Another advantage of templates is, in the event you want to share your template with another project, your custom methods (scripts) would not have to be exported separately like they would as if they were in a Script Library. When you export the template, the custom methods are included in the export automatically.

Sample Custom Method

A great use for Custom Methods is checking for valid input on a form with a lot of text fields. Instead of checking every text field all within a script on a button press, we can instead build a value check script on each text field that is unique to that Text Field's specific type of input. This keeps each script organized on the appropriate Text Field component. Take this sample code that can go into a custom method on a text field which checks for a valid email address using :

Python - Text Field Input Validation

```
# First, we need to import the Regular Expression library  
import re  
  
# We need to grab the value of the text field.  
text = self.text  
  
# Here, we put together our regular expression for email addresses.  
validAddress = re.compile(r"^[a-zA-Z0-9_+-.]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-]+.$")  
  
# Check the text against the regular expression.  
# If the text does not fit in with the regular expression, it returns a value of  
None, which we use to show an error box.  
if validAddress.match(text) is None:  
    system.gui.errorBox("Please enter a valid email address!", "Email Not Valid")
```

A similar script can be repeated for each Text Field component, but modified to fit the expected input. Then, the another script can be on a button that simply calls each of the Text Field's Custom Methods.

Python - Validate Input on Button Press

```
event.source.parent.getComponent("Email Field").validInputCheck()
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Script Builders

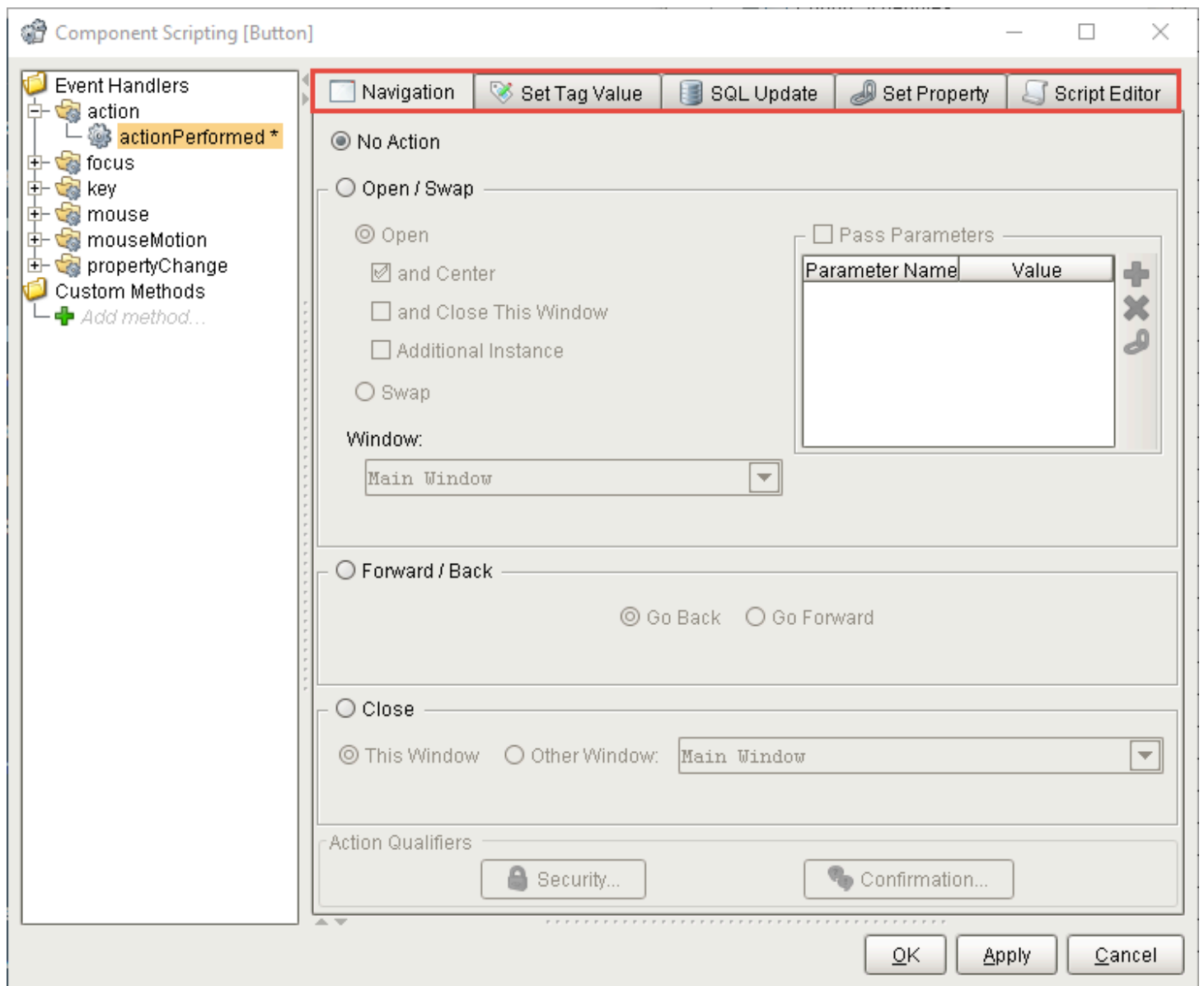


INDUCTIVE
UNIVERSITY

Script Builders

[Watch the video](#)

When creating an Event Handler on a component, you can use one of the handy **Script Builders** instead of writing your own script. In the Component Scripting window, the Script Builders are accessible as tabs along the top. Each tab represents a different kind of action that users can associate with an event. The last tab, **Script Editor**, lets you write your own event handler. You can also use it to view the script that was generated by the **Script Builder**, which is a good way to get started learning how to write your own event handlers. The examples in this section use a Button component. If you would like to follow along, drag a Button component on to a window and follow the steps to create the event handler for any of the Script Builders below.



⚠ ACTIONS GET OVERWRITTEN WHEN CHOOSING ANOTHER SCRIPT BUILDER

Only one Script Builder can be used at a time. If you previously picked another action using a different Script Builder, it will get overwritten by enabling another Script Builder. If you need to do more than one action at a time, use the Script Editor to combine scripts, or create your own script.

Navigation Script Builder

The Navigation Script Builder has various functions that deal with opening and closing windows.

Open / Swap

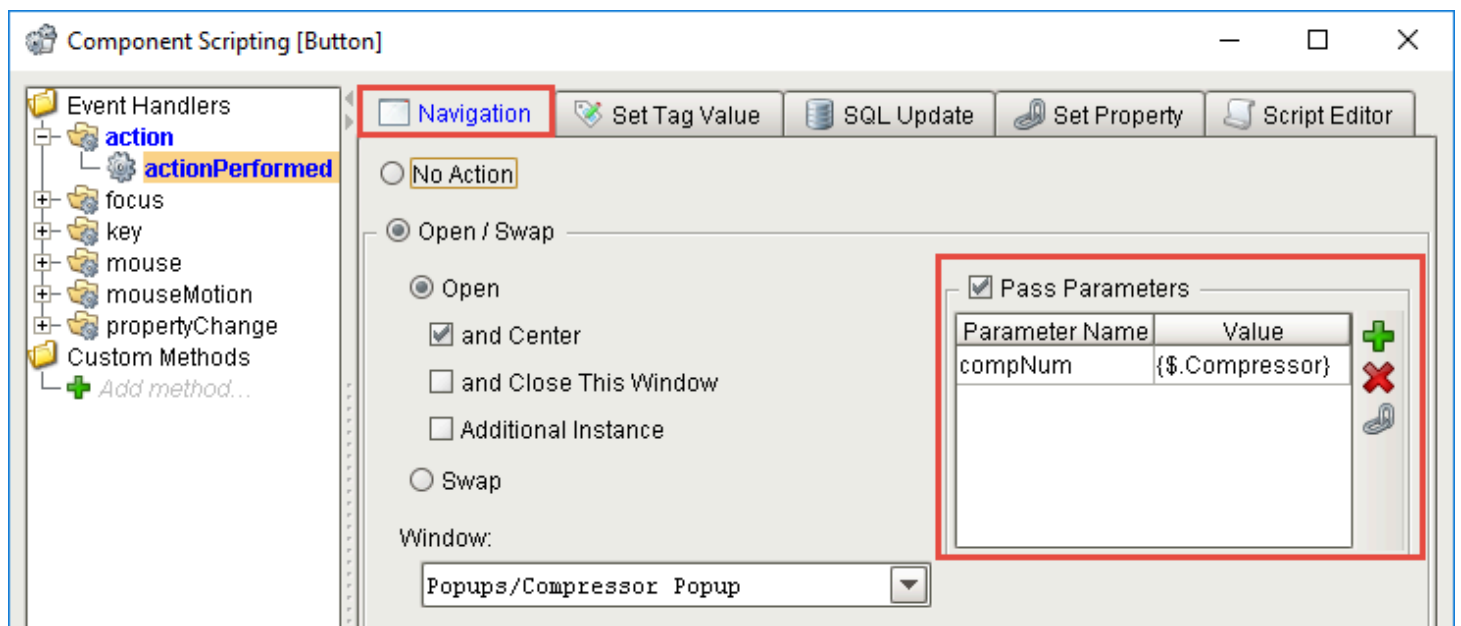
Opening is a very straight-forward operation, it simply opens the specified window at the same size it was in the Designer. Simply click on the Open / Swap button, and select a Window from the dropdown list that you want to open. There are options to center that window within the Client, and to close the

window that the event was fired from. The opened window can also be opened as an additional instance, meaning there can be multiple copies of the same window. This is useful when opening dynamic popups, so that a couple of popups can be opened, each with different values.

Swapping is the practice of opening another window in the same size, location, and state as the current window, and closing the current window. This gives the appearance of one window simply swapping into another, seamlessly. The Navigation Builder uses the **swapWindow** version of swapping, but most "by hand" script authors will use the **swapTo** version. This last version relies on the fact that the windows being swapped are both maximized windows. See the typical [navigation strategy](#) section for more information.

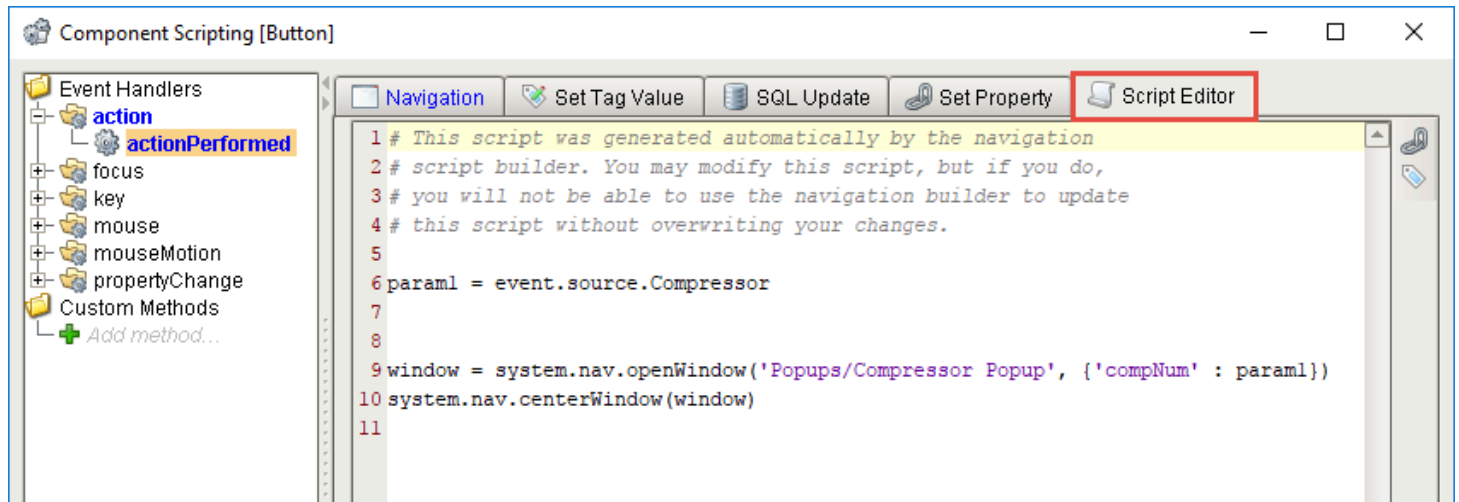
You can also **pass parameters** to the opened or swapped-to window. The names of these parameters must match names of custom properties on the root container of the target window. The values can either be literals or values of other properties from the source window. Passing in properties from the source window use two special characters: '\$' and '*'. The '\$**' character represents the source component while the '' character represents the parent. You can use these to construct a path to the property on the window. You can also use the property selector () to the right. To pass parameters in a navigation window, follow the steps below.

1. Check the Pass Parameters box, and click the green plus  button to add a row.
2. Highlight the empty cell in the Value column of the parameter table, and press the Insert Property  button. Select the property you want and press **OK**.
3. Choose a custom property from the "Parameter Name" dropdown list. This will be filled with custom properties on the root container of the selected window. You can also type a name in directly.
4. Press **OK** on the Component Scripting window to commit the change. For more information on this script, click the link below the image.



Refer to the [parameterized windows](#) section for more information.

The image below is the actual Navigation script that was generated by the Script Editor. If you compare the settings in the Navigation tab with the documented code in the Script Editor, you'll notice the Compressor popup window will be opened and centered in the window, and the value of the "compNum" parameter passed.



Forward / Back

The **Forward / Back** actions give you a simple way of implementing *browser*-style forward/back buttons in your client. Note, that you must be swapping between windows for this to work, because these functions rely on calls to `system.nav.swapTo` in order to keep track of what the sequence of recent windows has been.

Closing Windows

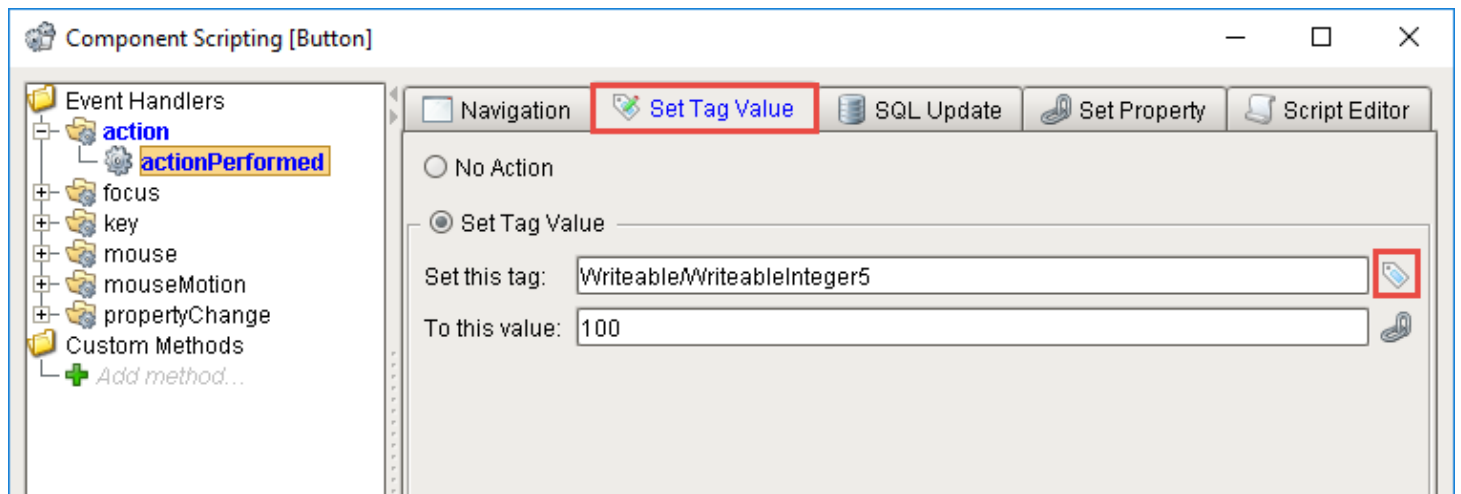
The Closing windows action allow for an easy way to have an event handler close the window that it is a part of, or any other window.

Set Tag Value Script Builder

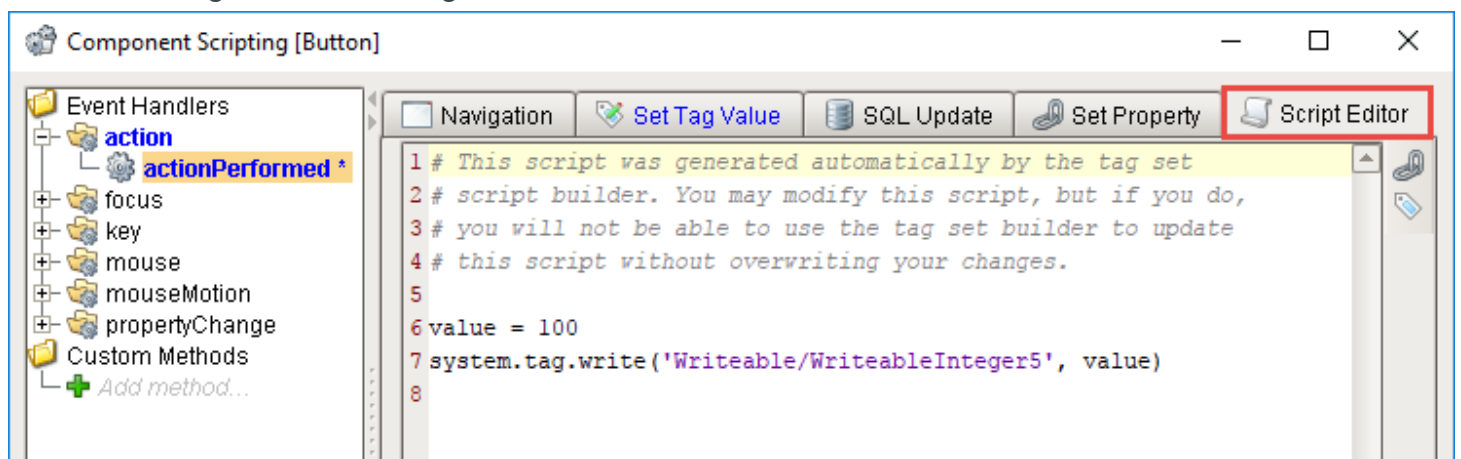
The Set Tag Value Script Builder responds to an event by setting the value of a Tag. You can set the Tag to either a literal value directly typed in, or you can use the Insert Property () button to have the event handler use the value of another property from the same window. Use the steps below to create a Set Tag Value.

1. Under the **Set Tag Value** tab, click the **Set Tag Value** radio button.
2. Click the **Set this Tag**  button and choose a **Tag** from the Tag browser list to write to (i.e., WriteableInteger5).

3. In the **To this Value** field, enter a number (i.e., 100) or click the property selector icon  to search the window for a property.
4. Press **OK** on the Component Scripting window to commit the change. For more information on this script, click the link below the image.



The Set Tag Value script shown below was generated by the Script Editor. Compare the settings in the Set Tag Value tab with the documented code in the Script Editor and you'll see the Tag is set to "WritableInteger5" and the Tag value is set to "100."

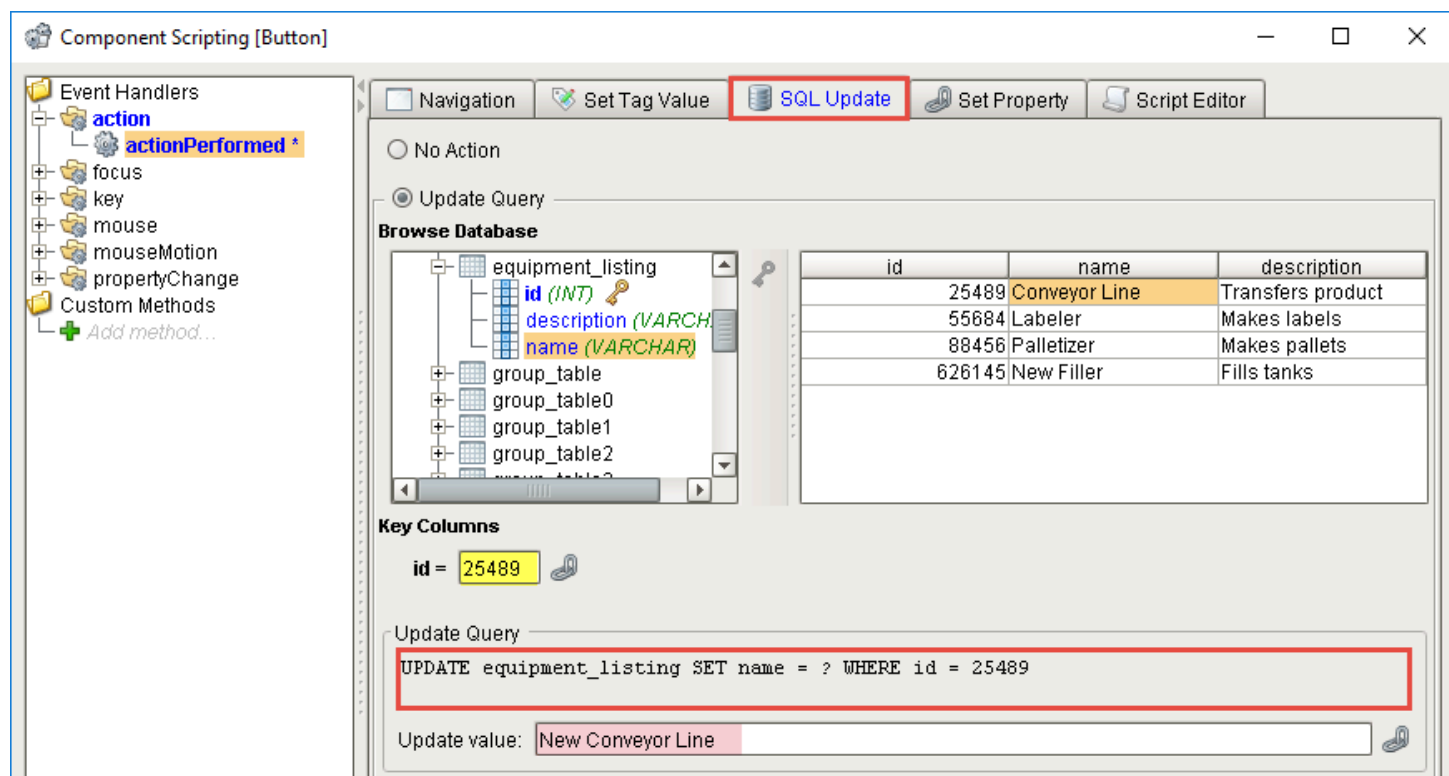


SQL Update Script Builder

The SQL Update Script Builder helps you build an update query using a database browsing interface. Choose a database and area in your target database, and the update query will be built for you. By setting columns as key columns, you can have the filter correctly filter to the right row. You may use either literal values or property values by using the Insert Property () button next to the Update Value text box. You can try this out using one of your existing databases.

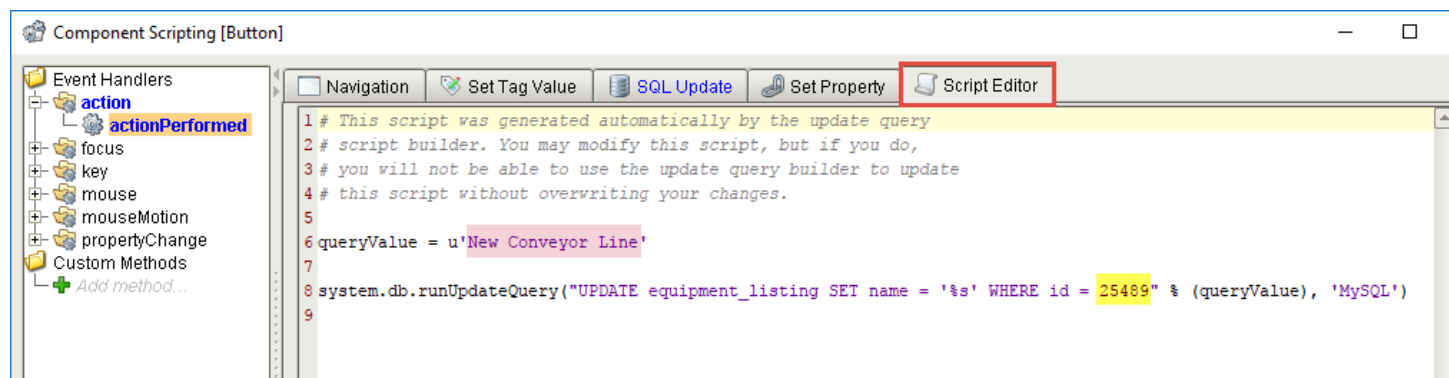
1. Under the **SQL Update** tab, click the **Update Query** radio button.
2. Select a database and choose a table in your database (i.e., **equipment_listing**). Choose a value in the table on the right (Conveyor Line), and the Script Builder will filter to the correct row in the table (i.e., 25489). Notice, an update query was immediately created for you below.

3. To change a value in the database, enter a new value in the Update Value field (i.e., "New Conveyor Line"), and it will replace the previous value (i.e., Conveyor Line) when you execute the action.
4. Press **OK** on the Component Scripting window to commit the change. For more information on this script, click the link below the image.



This will normally be used with a dynamic Update Value, and you can change what value(s) are your keys using the add/remove key icon.

The image below shows the SQL Update script that was generated by the Script Editor. If you compare the settings in the SQL Update tab with the documented code in the Script Editor, you'll see your database, table, and column name, including the row id of the searched value. You'll also see the new update value (i.e., New Conveyor Line) that will replace the existing value (i.e., Conveyor Line) when the action is executed. It will replace the name "Conveyor Line" with "New Conveyor Line."



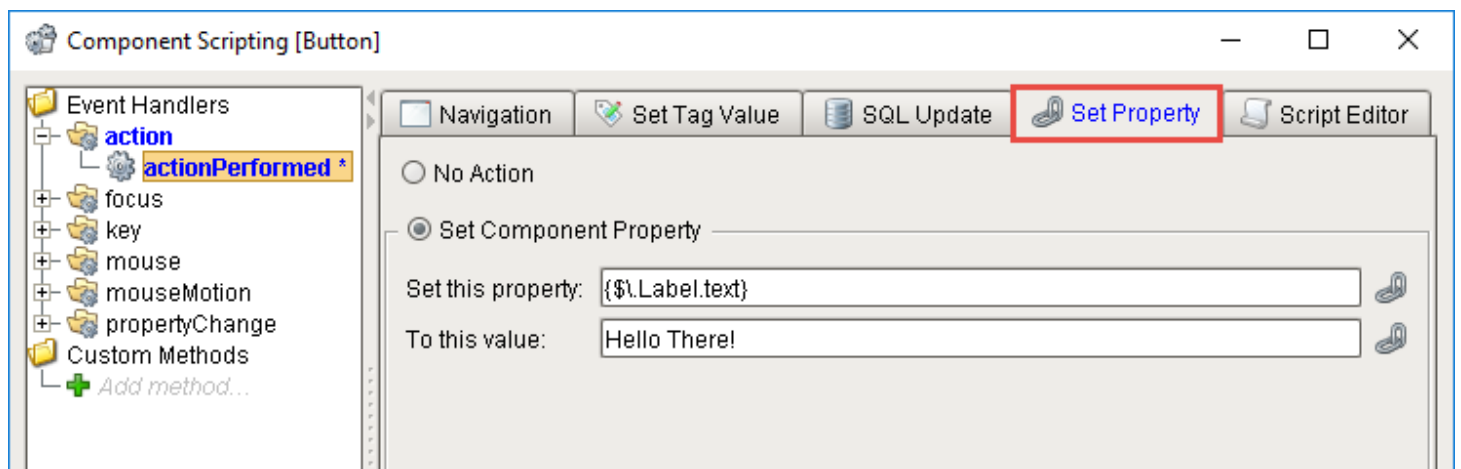
Set Property Script Builder

The Set Property Script Builder will respond to an event by altering a property in the window. You must choose the property to alter, and the value that you wish to assign to it. The new value can be a literal value or the value of any other property on the window by using the Insert Property (🔗) button.

⚠ CAUTION

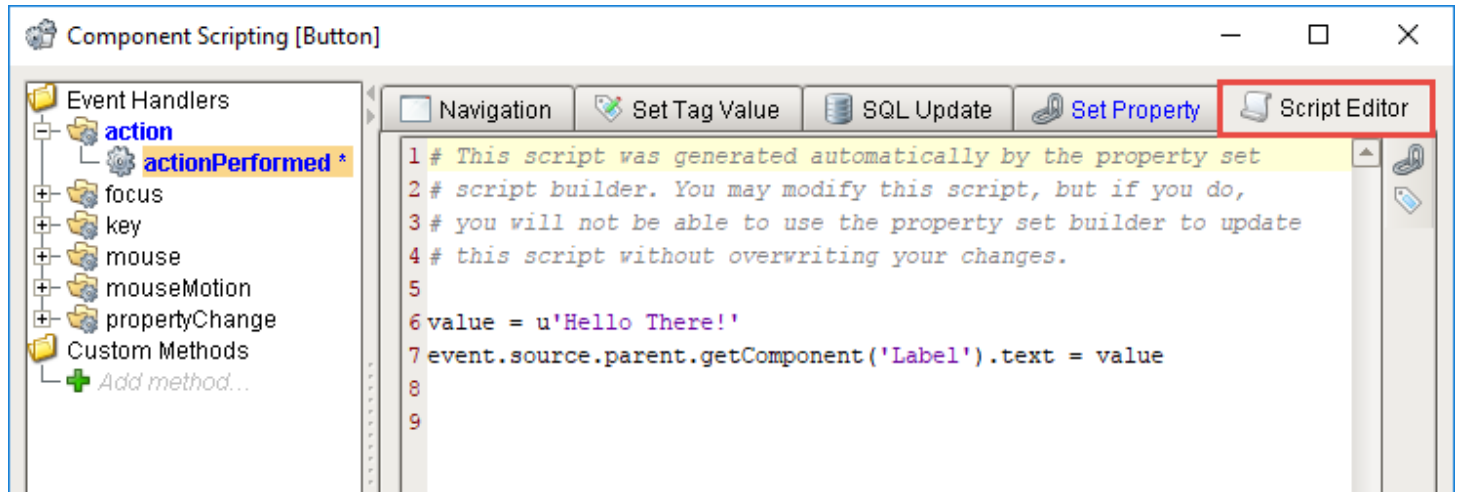
The property references here use the same two special characters ('\$' and '"') as passing parameters in the navigation section.

1. Drag a Label components to your window to set the property to.
2. Open the scripting window on your button component.
3. In the **Set Property** tab on your Button, click the **Set Component Property** radio button.
4. On the **Set this property:** field, click on the **Insert Property** (🔗) button to find your new label component, and then select a property (i.e. the **Text** property) and click OK.
5. Type something into the **To this value:** field (i.e., Hello There!), or select another property on the window.
6. Press **OK** on the Component Scripting window to commit the change.



This Set Property script was generated by the Script Editor. Compare the settings in the Set Property tab with the documented code in the Script Editor and you'll see the value string "Hello There!" will be

written to the Label component when the action executes.



Script Editor Builder

The Script Editor allows you to add more complexity to existing scripts, combine scripts, and even write your own custom scripts. For example, if you need to perform two or more actions at once, (i.e., set a Tag and navigate to another window), you can update the script by combining the two actions in the Script Editor. The Script Editor even gives you the flexibility to create your own code for any action you want to perform on an event handler.

As seen elsewhere on this page, the other builders ultimately generate a script that can be modified from the from the Script Editor Builder.

Advanced Settings

- **Scoping Dropdown** - Allows you to specify the scoping of the script. The scoping of event handlers in older versions of Ignition work differently that modern versions. This settings was added as a way to provide backwards compatibility when upgrading. All new scripts should ideally leave the scope set to Standard Scoping, as there is no reason for new scripts to use the Legacy Scoping option.
- **Invoke Later** - Provides an opportunity to allow the script to run after other events have finished processing. Most scripts will leave this setting disabled, however if can be useful in some scenarios:
 - **Controlling the focus order in a window**, since requesting focus from a component in the middle of an event being processed can cause undesirable results.
 - When writing a script on a window's **visionWindowOpened** event, you may wish to have your script run after processing.

Action Qualifiers

All of the Script Builders allow you to put Security and/or Confirmation qualifiers onto an event handler. These Action Qualifiers are optional.

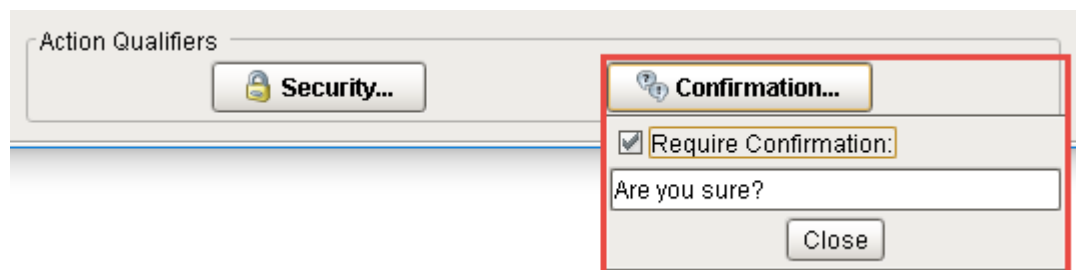
Security Qualifier

The Security Qualifier lets you restrict the event handler from running if the current user does not have one of the required roles highlighted in the Security qualifier dialog box. The roles listed will be all of the roles within the project's default user source. To setup the required roles for an event handler, select one or use **CTRL+click** to select multiple roles. Once the roles are selected, they will be highlighted. To deselect roles, use **CTRL+click** again.



Confirmation Qualifier

The Confirmation Qualifier prompts the user with a popup **Yes/No** dialogue box confirming you want to perform the action. The action will only be executed if the user chooses Yes. There is a default message, but you can delete it and enter your own message. Check the **Require Confirmation** box to enable the Confirmation qualifier.



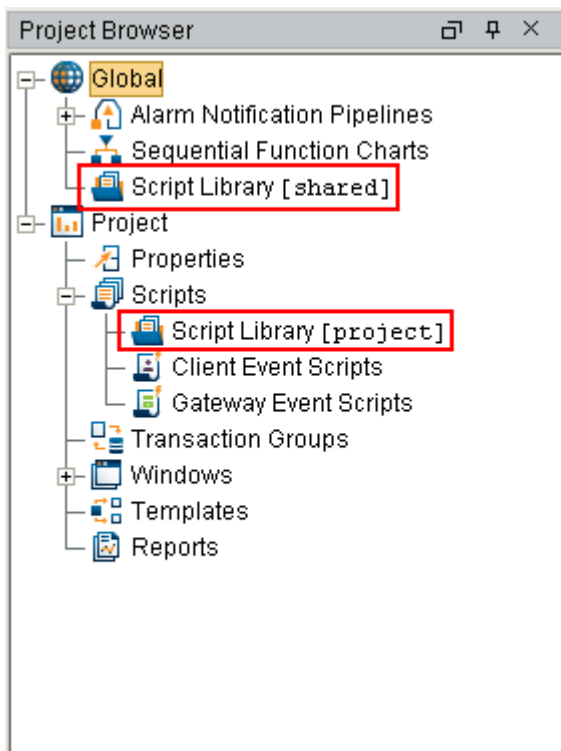
This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Project and Shared Scripts

Script Modules

Ignition's **Shared** and **Project Script Modules** are global libraries of scripts that can be called from anywhere within their scope. Shared Script Modules are available to the whole Gateway including every project, and Project Script Modules are available anywhere in the project that they were created in. These scripts are organized as named modules that all live under the **shared** or **project** Script Library. To open the Script Module Editor, go to the **Project Browser** and double click on the **shared** or **project** Script Library.



INDUCTIVE
UNIVERSITY

Shared Scripts

[Watch the video](#)



INDUCTIVE
UNIVERSITY

Project Scripts

[Watch the video](#)

💡 RULE OF THUMB: NEVER BLINDLY COPY-AND-PASTE

If you're unsure of when to put scripts in a script module vs. writing the script directly in an event handler, follow this simple rule: If you ever find yourself copying a script from one event handler to another, stop and refactor the script into a global script module! Then simply call your new module from the event handler. This rule will help prevent code duplication across your project which can be a major maintenance liability. Consolidating scripts into one location makes them easier to manage.

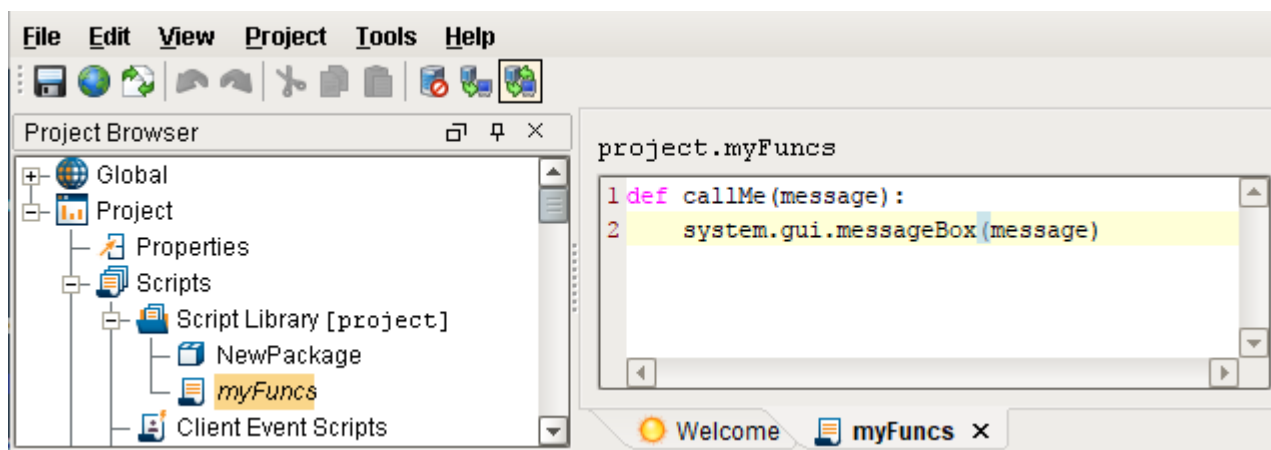
How to Use Script Modules

To add a script module, simply right click the **Script Library [project]** package and click the **New Script** option. Each module is a python script that can define many **functions**. You can also keep your scripting modules organized in Packages by creating additional nested subpackages. Packages act a little like folders except you can still add scripts to them. To create a Package, click on the Script Library **[project]** and click the **New Package** option.

For example, let's suppose you added the following script module named `myFuncs`, whose body is shown below.

Python - Project Script

```
def callMe(message):  
    system.gui.messageBox(message)
```

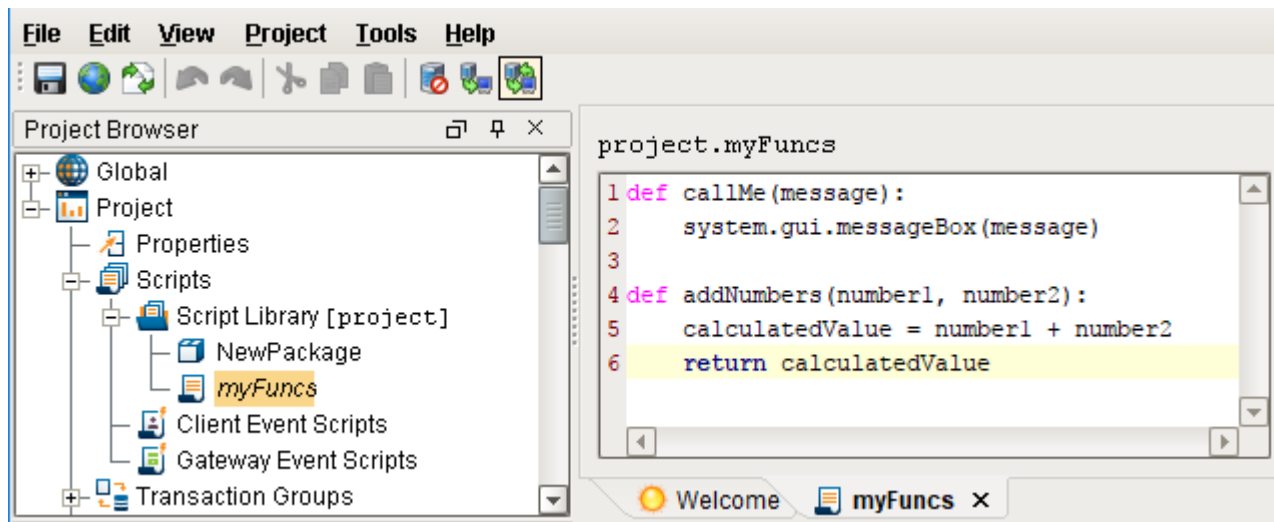


Now, anywhere in your project you can call this function.

Python - Calling the Project Script

```
project.myFuncs.callMe('Hello World')
```

Because each module can hold many functions, you can add any functions you need to this Script Module.



Just like before, you can call this function using:

Python - Calling Different Functions

```
result = project.myFuncs.addNumbers(14, 87)
```

Note, that the **addNumbers** function also has a return value. This will be returned to the script that called the function, allowing you to use it within that script.

System Import

In Ignition, certain libraries are automatically imported into any scripts you create to make everything easier. For example, the System Library is automatically imported, giving you access to all of the "system. functions" without having to add in "import system" at the beginning of your script. The Project and Shared Libraries are also imported, giving you access to any of the Script Modules you may have written. However, prior to version 7.7.0, Script Modules did not automatically import the System Library. This means that if using a version older than version 7.7.0, you will need to add "import system" at the beginning of your script to be able to use the System Library.

Project Scripts vs. Shared Scripts

There are a few differences between Project and Shared Script Modules. The first difference is the way each script type is called. Shared Scripts are called with "**shared.**" preceding the event handler function, while Project Scripts are called with "**project.**" The other difference is where Shared and Project Script Modules can be used. Project Script Modules are limited to the Project Scope, and can only be used within the project they are located in. This is useful when you are using a script many times within a project, but do not want it to be used by any other project or elsewhere in your Gateway.

Shared Script Modules, on the other hand, are not limited by scope and can be used anywhere within any project or Gateway. However, this means the Shared scripts can be used by projects that possibly shouldn't have access to them. Since Shared Script Modules are stored on the Gateway, they are not part of a normal project export. So moving a project to a new Gateway will require that Shared scripts also be included in the transfer.

A Simple Example

The Script Modules are very easy to get started with, and we can quickly put together a simple example to that demonstrates how they work.

1. Navigate to the **Script Library [project]**, right click and select New Script. Name the new script **Example**.
2. In the script area, add this script:

Python - Message Box Input

```
def myMessage(message):  
  
    # Will take the message string that we pass in and display it in a message  
    box.  
    system.gui.messageBox(message, "This is the message that was passed in.")
```

3. On a new window, drag a **Button** and **Text Field** components onto the screen.
4. Right click the button and navigate to Scripting.
 - a. On the `actionPerformed` Event Handler, add this script:

Python - Calling the Script with the Input

```
# Will call the project script that was just created, passing in the value of the  
text field for the message.
```

```
project.Example.myMessage(event.source.parent.getComponent('Text Field').text)
```

5. Save the project. The project script that we created in step 2 will not be available to use until it is saved.
6. Try it out! Type a message into the text field and then click the button. You should see a message box popup with your message inside.



Filtered Dataset Example

In this example, we can use a shared script to filter a dataset in any project. The script takes in a dataset, a column name, and a filter string. It will then remove any of the rows that do not contain the filter string in them.

Python - Dataset Filter

```
def datasetFilter(dataset, colName, filterValue):
    # Start with the full dataset, the name of the column we are searching
    # through, and the value to filter with.
    # Create an empty list to store row indices that will be removed.
    rowsToBeRemoved = []

    # Loop through the dataset, so we can check every row.
    for rowID in range(dataset.getRowCount()):

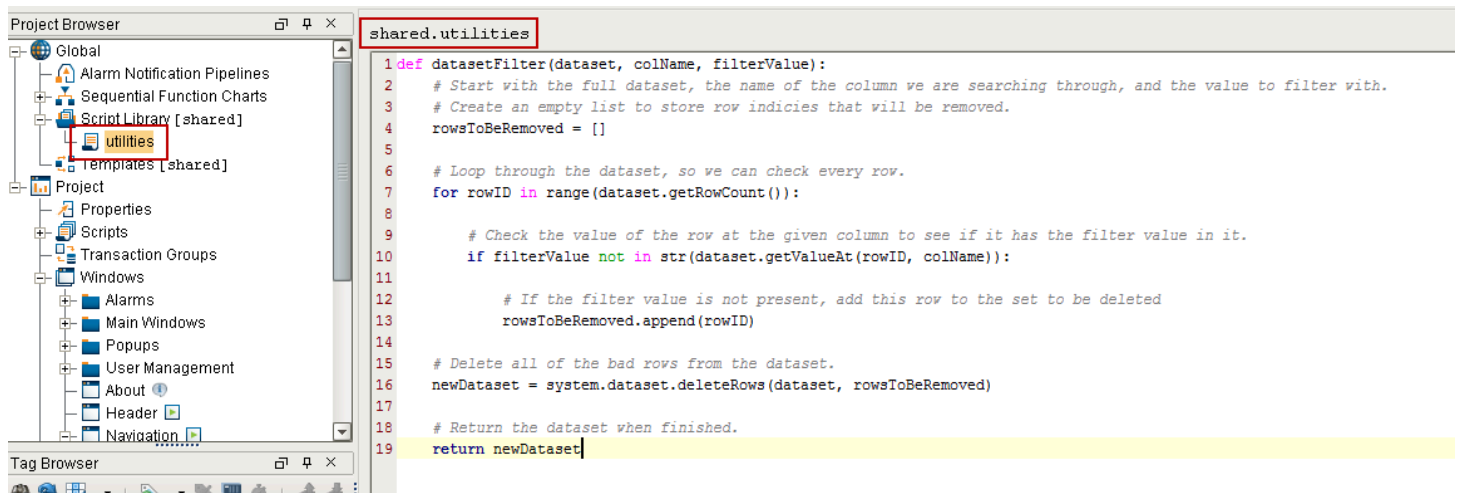
        # Check the value of the row at the given column to see if it has the
        # filter value in it.
        if filterValue not in str(dataset.getValueAt(rowID, colName)):

            # If the filter value is not present, add this row to the set to be
            # deleted
            rowsToBeRemoved.append(rowID)

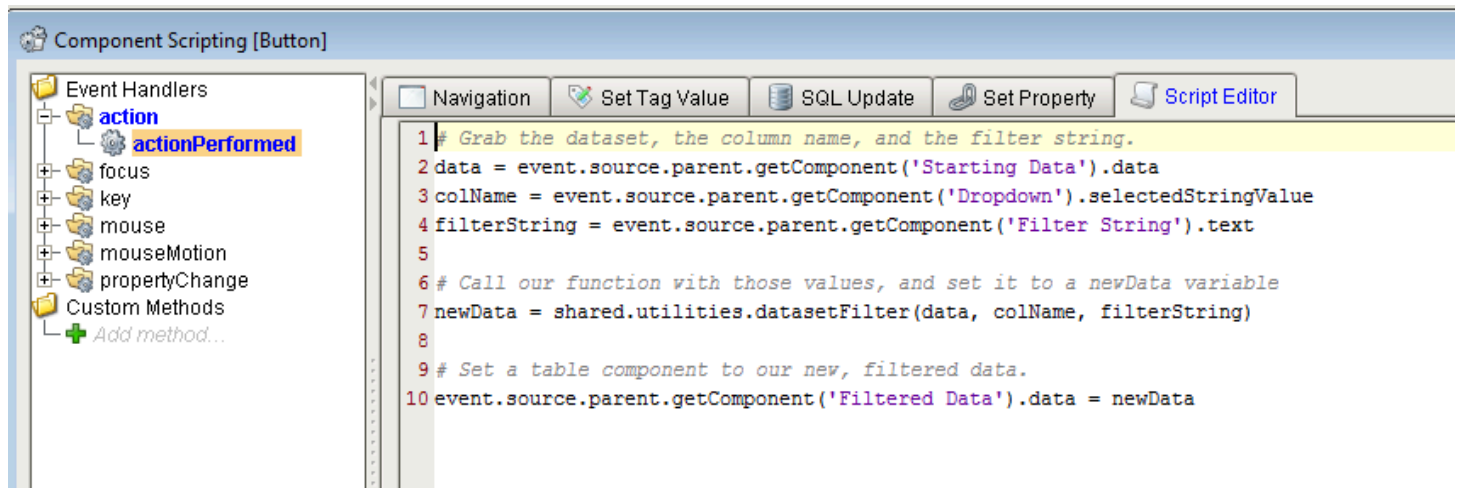
    # Delete all of the bad rows from the dataset.
    newDataset = system.dataset.deleteRows(dataset, rowsToBeRemoved)

    # Return the dataset when finished.
    return newDataset
```

By placing the script in a shared script called utilities, it can be used by any project, allowing us to filter a dataset no matter what the data is.



It can then be called in a project on something like a button component using `shared.utilities.datasetFilter()`.



Calculate OEE Example

In our projects, we use a fairly simple script that calculates OEE, and writes it to a Tag as shown below.

Python - Calculate OEE

```
def caculateOEE(lineNum):
    #Build the Tagpaths with the dynamic line numbers.
    paths = ["[default]Metrics/Line_%f/Quality" % lineNum,
             "[default]Metrics/Line_%f/Availability" % lineNum,
             "[default]Metrics/Line_%f/Performance" % lineNum]

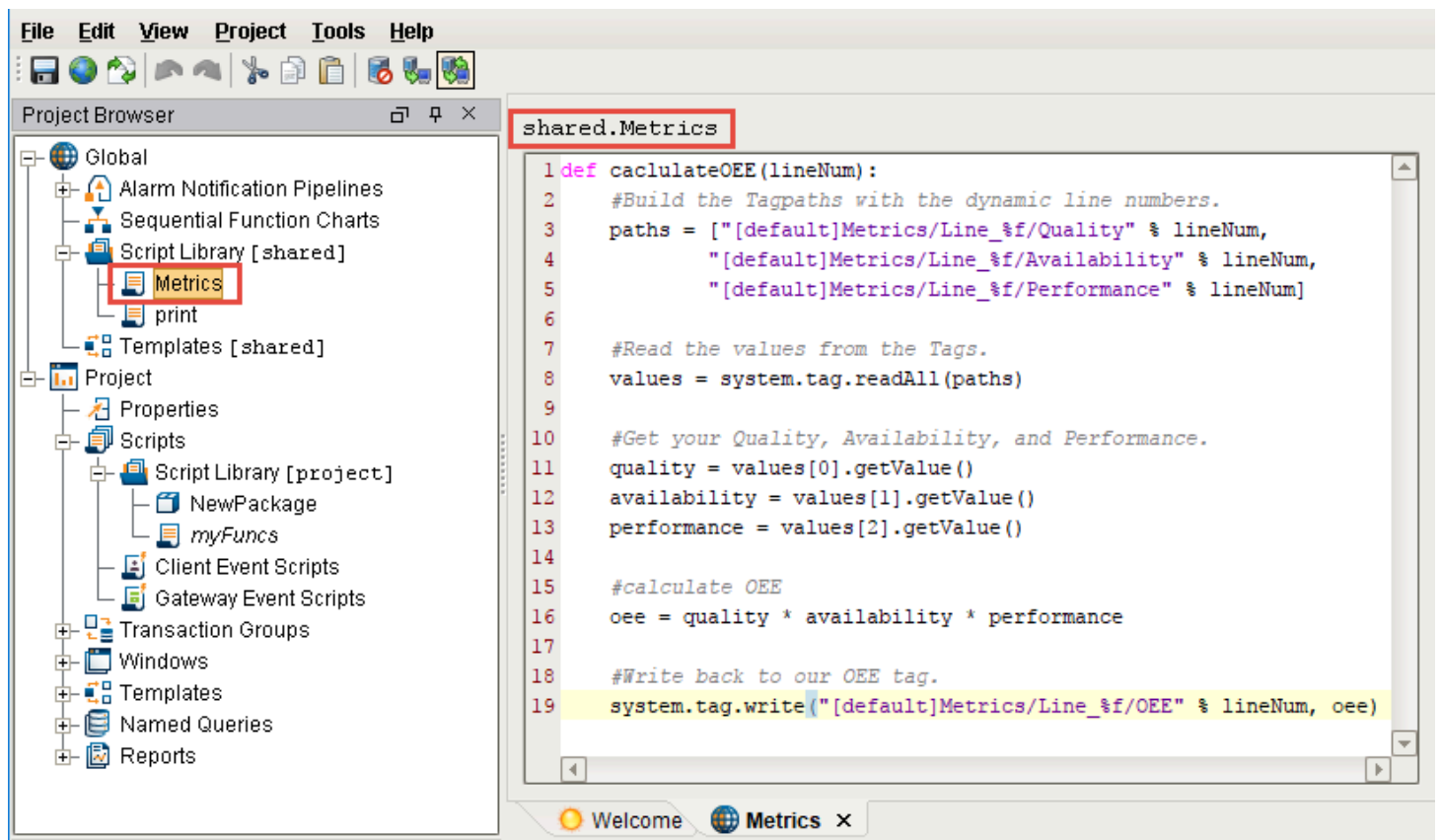
    #Read the values from the Tags.
    values = system.tag.readAll(paths)

    #Get your Quality, Availability, and Performance.
    quality = values[0].getValue()
    availability = values[1].getValue()
    performance = values[2].getValue()

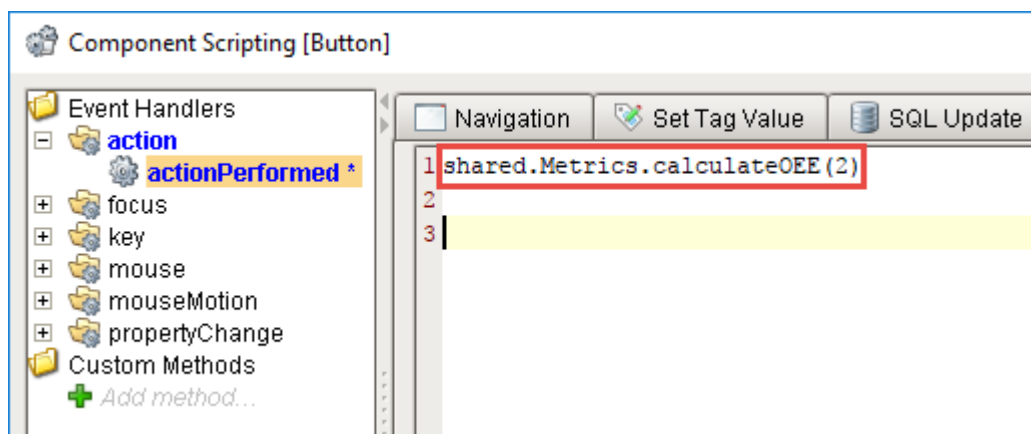
    #calculate OEE
    oee = quality * availability * performance

    #Write back to our OEE tag.
    system.tag.write("[default]Metrics/Line_%f/OEE" % (lineNum), oee)
```

Instead of copying this script between projects and throughout all of the windows that need it, a Shared Script Module was created in the Shared Script Library called "Metrics." The code was copied and pasted into the scripting area in the Shared Script Library.



This could then be called from a script in any project by calling **shared.metrics.calculateOEE()**:



This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Web Services, SUDS, and REST

Web Services Overview

Web services are software solutions that allow for interacting with machines residing on a network. In short, web services are nothing more than web pages for machines. They provide a standard way for a third party to request and receive data from a piece of hardware on the network without having to know anything about how that machine works.

Protocols

There are two common approaches to Web Services in Ignition: making a HTTP (HyperText Transfer Protocol) method call, or making a SOAP (Simple Object Access Protocol) call.

- HTTP methods, such as GET and POST, are called using the built-in system functions, such as [system.net.httpGet\(\)](#). More information on HTTP calls can be found on the [HTTP Methods page](#).
- SOAP is XML based, and typically requires a third party Python library. There are [many third party Python libraries that utilize SOAP](#), however, documenting them goes outside the scope of this manual (as do all third party libraries).

The approach you choose typically depends on the server you're trying to make calls to: more specifically, the protocol(s) it supports.

WHAT ABOUT THE SUDS LIBRARY?

The SUDS library, a library that used to come included with the Python Standard Library, offered SOAP based functionality. However, SUDS development has been halted, and is no longer included in the standard library.

In the interest of posterity, the legacy SUDS documentation has been condensed and can be found on the [SUDS - Library Overview page](#). Note that the legacy documentation should be considered deprecated.

Can Ignition Make RESTful Calls?

Yes it can! However, it is important to understand that REST is an architecture, NOT a protocol. Instead, REST utilizes and describes how a protocol should be used. Thus, a RESTful architecture could use both of the protocols mentioned on this page, although HTTP is far more common.

Common Web Services Workflow

While all Web Services follow the same standards, they all do different things. They wouldn't be worth anything if you didn't get the information you need, or if they contained a lot of excess data. If you are unfamiliar with a particular Web Service, there are a few things that you can do to figure out what data is available and how to get it.

1. Identify a Web Service that you will be using. Usually the Web Service has an API somewhere documenting how requests should be made.
2. Write a script to pull some information from the Web Service. If using HTTP, this could mean starting with a GET call, where as SOAP would involve retrieving the WSDL (Web Services Description Language). In both cases, you may need to find a way to authenticate against the server (usually with some user credentials or an auth token, the API for the service would have more details).
3. Once you have the results from the GET/WSDL, identify the information or functions you want to use.
4. Write a script to use that function and return your values.
5. Parse the results and use them. This can be for display, saving to a database, or anything else you need.

NOTE

Web Services sometimes take a lot of time to return results, especially the first time they are called. If you put your Web Services script in a button, the client will freeze until the call is complete (this is because the event handlers are run on the GUI thread). It's a good idea to use [`system.util.invokeAsynchronous\(\)`](#) or add a waiting image to your screen to let the user know Ignition is working as expected.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

HTTP Methods

Overview

Web services calls typically require some protocol to make requests. HTTP is an incredibly common protocol, so this page will introduce how to incorporate these calls in a python script. Note that all of the examples on this page can be easily called with the Script Console, but can be utilized through some other means (the actionPerformed event on a Button).

Finding an Endpoint

Ignition doesn't natively expose an endpoint for web services calls, so we'll have to utilize a service of some sort for the examples on this page. Fortunately, there are many public services we can utilize, such as Yahoo Weather. From here we can generate an endpoint to use. At the time of this writing, we're using the following endpoint to retrieve the sunset time in Maui,HI:

Pseudocode - Finding a Resource

```
https://query.yahooapis.com/v1/public/yql?  
q=select%20astronomy.sunset%20from%20weather.forecast%20where%20woeid%20in%20(select%
```

Making the Call

To retrieve the results of this information in Ignition, we can use `system.net.httpGet()` to fetch the results of this call. We can try the following script in the Scripting Console:

Python - Creates a variable to Store the Endpoint and Retrieves Results

```
# Create a variable to store the endpoint.
myEndpoint = "https://query.yahooapis.com/v1/public/yql?
q=select%20astronomy.sunset%20from%20weather."
myEndpoint +=
"forecast%20where%20woeid%20in%20(select%20woeid%20from%20geo.places(1)%20where%20tex
myEndpoint +=
"%22maui%2C%20hi%22)&format=json&env=store%3A%2F%2Fdatatables.org%2Falltableswithkeys

print system.net.httpGet(myEndpoint)
```

Printing this results in the following Python string:

Python - Results

```
{"query":{"count":1,"created":"2017-10-31T16:44:19Z","lang":"en-US","results":
{"channel":{"astronomy":{"sunset":"5:50 pm"}}}}
```

Parsing the Results

If we wanted to extract a single value out of the results, we have a number of approaches. One useful approach would be to turn this JSON string into a Python Dictionary. This way we can single out a key instead of using regex or looking for substrings (both valid approaches in their own regard).

When presented with a JSON string, we can call [system.util.jsonDecode\(\)](#) to turn a JSON string into a native Python Object. Thus, we can modify our code to the following:

Python - Parsing the Results Code

```

# Create a variable to store the endpoint.
myEndpoint = "https://query.yahooapis.com/v1/public/yql?
q=select%20astronomy.sunset%20from%20weather."
myEndpoint +=
"forecast%20where%20woeid%20in%20(select%20woeid%20from%20geo.places(1)%20where%20tex
myEndpoint +=
"%22maui%2C%20hi%22)&format=json&env=store%3A%2F%2Fdatatables.org%2Falltableswithkeys

# Make the GET call.
results = system.net.httpGet(myEndpoint)

# Convert the JSON string into a Python object. In this case, it results in a Dictio
decodedDict = system.util.jsonDecode(results)

# Now we can treat the results like a nested dictionary, thus we can specify the "que
key,
# and then the nested "created" key to return just the date.
print decodedDict.get("query").get("created")

```

Now we can easily retrieve a single value by specifying key names on the results.

Make the Results Human Readable

Now that we know how to extract the results, we should clean up the output of the GET call. The JSON string returned by the endpoint could potentially be long and cumbersome to read through for a human, but we can use Python's built-in **pprint** library to pretty print the results.

Python - Now with Pretty Print

```

# Import the pprint library
import pprint

# We'll instantiate an instance of PrettyPrinter, and store it in a variable named pp
pp = pprint.PrettyPrinter(indent=4)

# Create a variable to store the endpoint.
myEndpoint = "https://query.yahooapis.com/v1/public/yql?
q=select%20astronomy.sunset%20from%20weather."
myEndpoint +=
"forecast%20where%20woeid%20in%20(select%20woeid%20from%20geo.places(1)%20where%20tex
myEndpoint +=
"%22maui%2C%20hi%22)&format=json&env=store%3A%2F%2Fdatatables.org%2Falltableswithkeys

# Make the GET call.
results = system.net.httpGet(myEndpoint)

# Convert the JSON string into a Python object. In this case, it results in a Dictio
decodedDict = system.util.jsonDecode(results)

# Print out the dictionary in an easy to read format.
print pp.pprint(decodedDict)

```

The resulting output, which is much easier to read, looks like the following:

Python - Results

```

{   u'query': {   u'count': 1,
                  u'created': '2017-10-31T16:44:19Z',
                  u'lang': 'en-US',
                  u'results': {   u'channel': {   u'astronomy': {   u'sunset':
'5:50 pm'}}}}}}

```

From here we can see all of the keys that lead to our final value:

Interactive Interpreter

```

Jython 2.5.3 (v2.5.3:3d2dbae23c52+, Nov 17 2012, 11:51:23)
[Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)] on javal.8.0_131

>>>
{   u'query': {   u'count': 1,
                  u'created': '2017-10-31T16:44:19Z',
                  u'lang': 'en-US',
                  u'results': {   u'channel': {   u'astronomy': {   u'sunset': '5:50 pm'}}}}}}
None
>>> |

```

To get to the value for the sunset key, we simply need to address each key along the way:

Python - To Get Value for the Sunset Key

```
print
decodedDict.get("query").get("results").get("channel").get("astronomy").get("sunset")
```

Troubleshooting HTTP Methods

When making HTTP calls, it is helpful to be familiar with the status codes that are returned by errors. To demonstrate, we could modify an earlier example:

Python - Returns Error Status Codes

```
# Create a variable to store the endpoint.
myEndpoint = "https://query.yahooapis.com/v1/public/yql?
q=select%20astronomy.sunset%20from%20weather."

### Note that the following two lines are commented out, meaning we're about to make
call with a bad/incomplete address.
# myEndpoint +=
"forecast%20where%20woeid%20in%20(select%20woeid%20from%20geo.places(1)%20where%20tex
# myEndpoint +=
"%22maui%2C%20hi%22)&format=json&env=store%3A%2F%2Fdatatables.org%2Falltableswithkeys

print system.net.httpGet(myEndpoint)
```

This will return an error, which looks like the following:

Python - IOError: Response Code 400

```
Traceback (most recent call last):
  File "<buffer>", line 8, in <module>
IOError: Server returned HTTP response code: 400 for URL:
https://query.yahooapis.com/v1/public/yql?
q=select%20astronomy.sunset%20from%20weather.
```

Note that HTTP response code **400**, which means bad request, was referenced. This error code is correct because we intentionally used an incomplete address!

HTTP Response Codes

The World Wide Web Consortium has [a page dedicated to HTTP response codes](#), which details all possible error codes. However, several common codes are listed below:

Response Code	Description
400	Bad Request - The server could not understand the request due to malformed syntax.
401	Unauthorized - The request requires some sort of authentication. Potentially some user credentials or an auth token of some kind.
403	Forbidden - The server understood what you requested, but is intentionally refusing the request. In some cases, the error message may include a reason why the request was not fulfilled (but not always). Typically, if the server doesn't include a reason, they'll use a 404 error code instead.
404	Not Found - Your syntax was correct, but the server could not find the resource you were asking for. This could mean a typo, or missing portion of the URL you are using. In this case, double check the address you're specifying. Depending on the configuration, this could also mean that the server does actually have the resource you requested, but doesn't want to confirm its existence (typically due to a security policy. See error code 403 above).

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

SUDS - Library Overview

The SUDS Library

The SUDS library is a SOAP-based web services client developed for Python. It is extremely simple to use and practically eliminates the need for the user to understand or even view the WSDL of a web service.

DISCLAIMER

The SUDS library used to be included in the Python Standard Library. However it has since been removed, meaning you may not have access to it when performing a fresh install of Ignition. Additionally, development on the library has mostly ceased, so any copies you find online may be drastically outdated.

The information on this page will be maintained for legacy users that need to be familiar with the old SUDS library. As a result, this page and its contents should be considered deprecated.

The SUDS library interprets the WSDL file for you and through a couple simple function calls allows you to get a list of the available methods provided to you by the web service. You can then invoke these methods in Ignition through event scripting to send and receive data in your projects. You will have to familiarize yourself with the SUDS library in order to make use of it.

A Simple Example

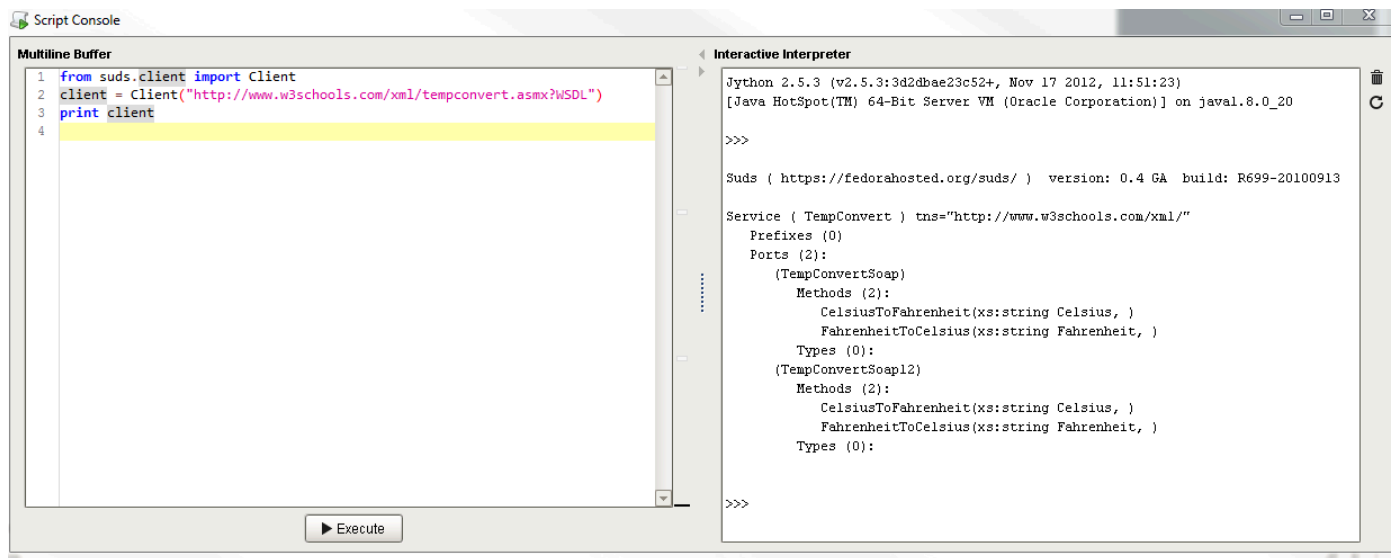
If you read through the SUDS documentation, you'll see that the Client object is the primary interface for most users. It is extremely simple using this object and a few print statements to view a list of available methods provided by the web service you are connecting to. This example will illustrate how to make an initial connect to a web service, print out the list of available methods, and then call one of these methods and display the resulting value in a label on an Ignition window at the push of a button. The below example uses a public web service and is available for anyone to test against.

1. First, we can use the script playground to test out some scripting calls and see the output. The below example shows how to get a reference to a client object. By printing this client object to the

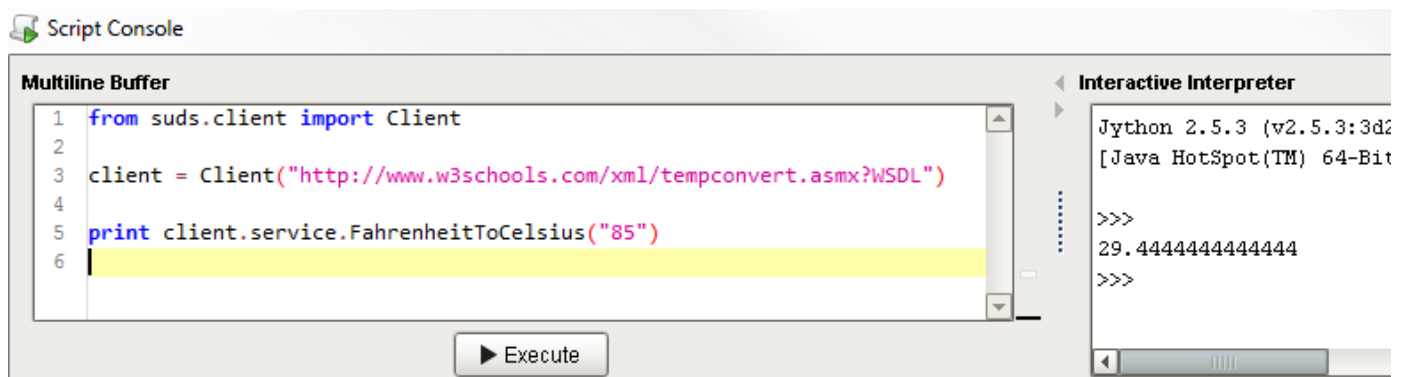
console we get an output of all the available methods, types, and other information about the web service as defined in the WSDL file.

Python - W3Schools WSDL

```
from suds.client import Client
client = Client("http://www.w3schools.com/xml/tempconvert.asmx?WSDL")
print client
```



This WSDL defines two functions: `CelsiusToFahrenheit(string tempCelsius)` and `FahrenheitToCelsius(string tempFahrenheit)`. These are the functions that this web service makes available to you. Don't worry about the fact that the methods are listed twice. This is just because the WSDL has two definitions of the functions that are formatted for different SOAP version standards. To call these functions in Ignition scripting, you have to make use of the "service" member of the client object. You can see printing the returned results shows the conversion:



2. To make a simple conversion window in an Ignition project you can add a button, a numeric textbox, and a label to a window. Then on the button to calculate a Fahrenheit to Celsius calculation, you would place something like the following:

Python - Fahrenheit to Celsius

```
from suds.client import Client
client = Client("http://www.w3schools.com/xml/tempconvert.asmx?WSDL")

far_value = event.source.parent.getComponent('Numeric Text Field').floatValue
cels_value = client.service.FahrenheitToCelsius(far_value)

event.source.parent.getComponent('Label').text = cels_value
```



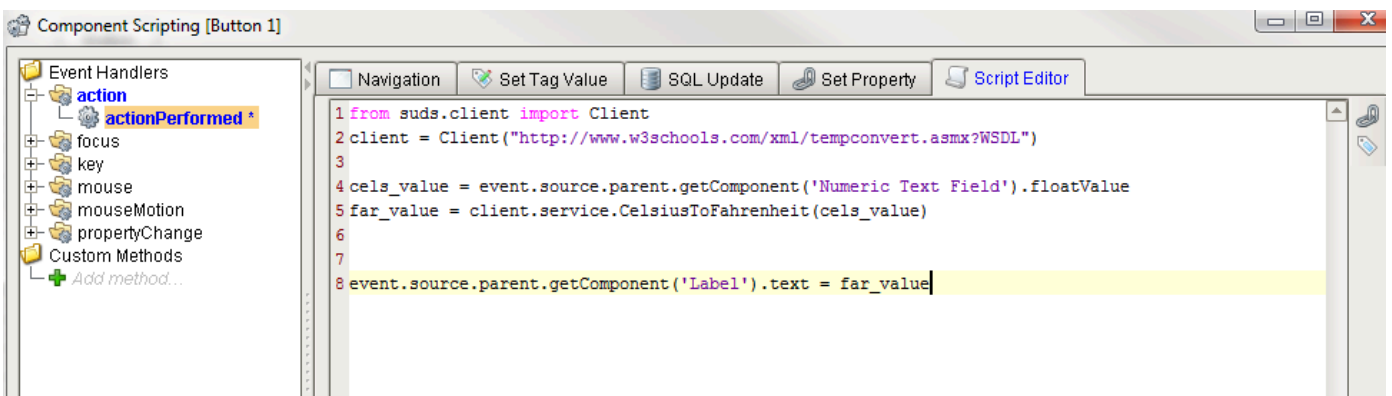
3. Then you can make a second button to do the opposite: calculate Celsius to Fahrenheit.

Python - Celsius to Fahrenheit

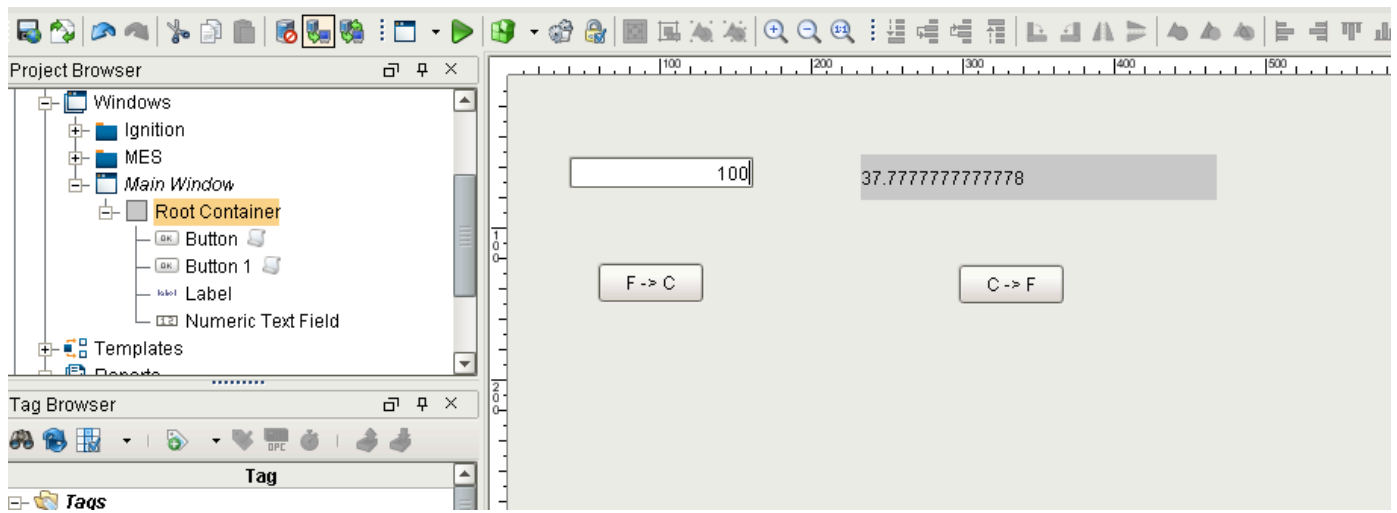
```
from suds.client import Client
client = Client("http://www.w3schools.com/xml/tempconvert.asmx?WSDL")

cels_value = event.source.parent.getComponent('Numeric Text Field').floatValue
far_value = client.service.CelsiusToFahrenheit(cels_value)

event.source.parent.getComponent('Label').text = far_value
```



4. With your scripts in place your window should now function as a simple temperature conversion tool!



Beyond the Example

While the example is relatively simple, it can easily be expanded upon. However, always keep the general workflow in mind when using the SUDS library:

Pseudocode - WSDL Workflow

```
#Import the SUDS Client object
from suds.client import Client

#Instantiate a new Client Object
client = Client("url_to_your_wsd1")

#Call the desired method using the service instance variable
client.service.MyMethod(myArgument)
```

Complex Arguments

In the overview, the methods provided by the web service were very simple and took simple argument types. Sometimes however, the web service will describe complex types and allow you create instances of these types that can then be added to the system/machine that the web service is providing an interface for.

A simple, hypothetical example of this would be a system that stores contact information of clients and can be used as an address book of sorts by other systems on the network. It may provide not only a way to pull contact information for a certain individual out, but also a way to insert new contacts. We'll keep the example simple and say that contacts have only a name and a phone number.

This example is completely hypothetical. It is intended to give insight into complex types. It does not make use of an actual functional web service.

For example, say we create and print the client object we associated with our web service in the following manner:

Pseudocode - Client Object

```
from suds.client import Client
url = 'http://localhost:7575/webservices/hypothetical_webservice?wsdl'
client = Client(url)
print client
```

And the resulting output is the following:

Python - Results

```
Suds ( https://fedorahosted.org/suds/ ) version: 0.4 GA build: R699-20100913

Service (hypothetical_webservice)
  Prefixes (0):
  Ports (1):
    (Soap)
    Methods:
      addContact(Contact contact, )
      getContactList(xs:string str, xs:int length, )
      getContactByName(Name name, )
  Types (3):
    Contact
    Name
    Phone
```

Here you can see that, while not too complicated, the web service defines more than just methods that take simple type arguments and return the same. Under the Types section, you can see there are three "complex" types. These are basically just objects like one creates in an object oriented programming language like java. The SUDS Client object has an instance variable called "factory" that allows you to create these complex types so you can use them to invoke methods defined by your web service that take complex arguments.

If we wanted to add a contact using the addContact() method, we have to create a contact object first:

Pseudocode - Using a Method

```
contact = client.factory.create('Contact')
print contact
```

The create function creates a new contact object that knows its own structure. We can view this structure by calling print on this new object and see that it prints the following:

Python - Structure

```
(Contact)=
{
  phone = []
  age = NONE
  name(Name) =
    {
      last = NONE
      first = NONE
    }
}
```

By examining the Contact type object, we can see its structure and know what we need to create in order to have a valid Contact to add to the address book. We could then do the following to supply the necessary information for the Contact object and then call our addContact function.

Pseudocode - Adding a Contact

```
contact = client.factory.create('Contact')

phone= client.factory.create('Phone')
phone.areacode = '916'
phone.number = '5557777'

name = client.factory.create('Name')
name.first = 'John'
name.last = 'Doe'

contact.name = name
contact.phone = phone
contact.age = 30

client.service.addContact(contact)
```

After execution a new contact will have been added via the web service!

Steps to remember when using complex types:

Pseudocode - Complex Type Reminders

```
#Create a new type object using the factory instance variable of the Client object  
my_type = client.factory.create('MyType')
```

```
#If you don't know the structure of the newly created object then print it to the console  
print my_type
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Web Dev

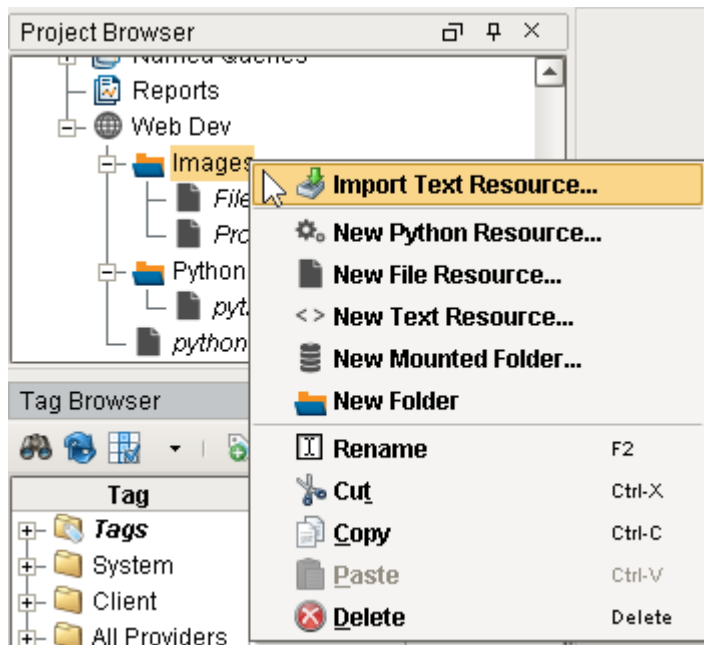
The Web Dev module allows users to directly program against the web server inside the Ignition Gateway. Webpages can be built by hand using a combination of python programming and static web resources such as images, CSS files, JavaScript files, and HTML files. Likewise, this module allows you to build RESTful web service APIs that allow external systems to interact with the Ignition server. This module follows the normal installation process.

⚠️ DISCLAIMER

The Web Dev module requires specialized web-programming knowledge. The Inductive Automation support team is unable to provide detailed advice about creating a particular site. Furthermore, they are unable to provide troubleshooting beyond the basic functionality of the module.

Basic Usage

Each type of resource may specify its **content type**. It is important to specify the correct content type for the contents of the resource. When the Web Dev module is installed, a new kind of project resource heading will appear in the Designer's project browser called "Web Dev". Right-clicking on this heading will allow the creation of several new types of project resources:



Python Resource

Python resources are dynamic web resources. Each time a user browses to the URL associated with a python resource, the script will run and generate a response in the form of a python dictionary. See [Return Value](#) for more details on formatting this response.

File Resource

A file resource is a static resource, usually a binary resource such as an image. Note that you will need to re-import a file resource if it has been changed since adding it to Web Dev.

Text Resource

A text resource is a static resource much like a file resource, except that its contents may be directly edited from within the Ignition Designer. These are useful for static HTML, CSS, and JavaScript files.

Mounted Folder

Mounted Folders are a way to expose a folder from the gateway's hard drive as a resource endpoint. For example, if the Ignition server had some directories that look liked the following:

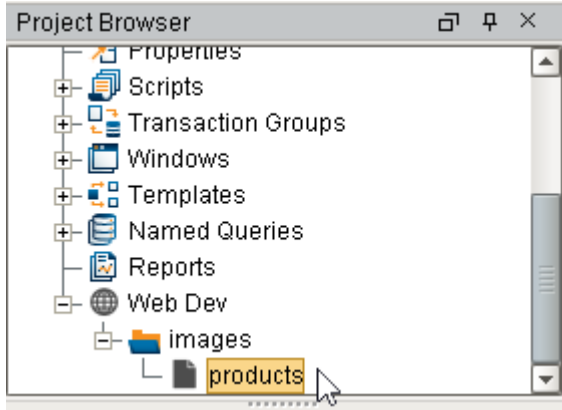
```
/opt/public/a.jpg  
/opt/public/info.html
```

You can create a Mounted Folder named "assets", and point it to `/opt/public/`, and then you can access those assets via:

```
http://host:port/main/system/webdev/projectname/assets/a.jpg
```

Project Resources Folders

If a Mounted Folder is placed in a Project Browser folder, then the endpoint must also include the folder name. For example, if the Mounted Folder named "products" is located in the "images" folder:



The files would be accessible via:

```
http://host:port/main/system/webdev/projectname/images/products/a.jpg
```

URL

Each resource will be directly accessible over HTTP and mounted beneath the /main/system/webdev path.

For example, if you created a Text Resource directly beneath the "Web Dev", it would be mounted at:

Pseudocode - Project Resource

```
http://host:port/main/system/webdev/project/resource_name
```

Notice that the project name and resource name are part of the path. If your resource is nested inside a folder, it will be part of the path too. For example:

Pseudocode - Folder Resource

```
http://host:port/main/system/webdev/project/folder_name/resource_name
```

Web Dev resources may have periods in their name. This means that if you upload an image file, you may include its extension directly in its name so that its path is more natural. For example, you might name an image resource "my_image.png" so that its URL is:

Pseudocode - Image Resource

```
http://host:port/main/system/webdev/project/my_image.png
```

The Web Dev module respects the published vs. staging system of Ignition. If your project is set up to have separate staging and published versions, you may access your staging version resources via the URL:

Pseudocode - Staging Version Resource

```
http://host:port/main/system/webdev_staging/project/resource_name
```

Requests to the root of your project will attempt to load a resource named "index.html". If no such resource exists, a 404 response code will be returned instead.

Pseudocode - Request to Root Project

```
http://host:port/main/system/project
```

Python Resources

Python resources are the heart of the functionality of the Web Dev module. These resources are completely dynamic, and can handle all parts of the HTTP protocol, formulating any type of response.

Each time an incoming HTTP request arrives at a URL where a python resource is mounted, the corresponding python script will be run. Each python resource may have up to seven scripts, one for each HTTP method (GET, POST, HEAD, TRACE, etc.). In practice, most python resources will probably only implement one or two of these, usually doGet or doPost at a minimum.

Parameters

Each `do*` method receives the same two parameters: 'request' and 'session'.

Request Parameter

The request parameter is a dictionary with a bunch of information about the incoming web request.

request['params']

Any URL parameters such as the following:

Pseudocode - Request Parameter

```
/main/system/webdev/project/foo?param1=value&param2=other_value
```

This will be contained in a dictionary accessible via `request['params']`. For the given example, `request['params'] = {'param1':'value', 'param2':'other_value'}`

request['remainingPath']

This provides the remaining text after a file resource. If you have a resource called 'foo', and a request is made to:

Pseudocode - Remaining Path

```
/main/system/webdev/project/foo/bar
```

`request['remainingPath']` will be `"/bar"`. Remaining path will be `"None"` if nothing is found after the resource name.

request['postData']

This parameter is only present for the `doPost` method, and its value is different based upon the value of the incoming HTTP request's Content-Type header. If the content type is `'application/json'`, then the `request['postData']` will be a python dictionary structure which is the result of parsing the json document that was posted. If the content type starts with `'text/'`, then the value of `request['postData']` will be the text which was posted, as a string.

request['headers']

The HTTP **request headers** will be in a dictionary under `request['headers']`. For example, you could read the User-Agent string with `request['headers']['User-Agent']`.

`request['scheme']`

The scheme is available via `request['scheme']`. The value will be 'http' or 'https'.

`request['remoteAddr']` / `request['remoteHost']`

These two parameters give the remote IP address and host of the entity that made the web request. Note, that these are from the perspective of the web server, and so may not be what you expect due to the effects of things like NAT-ing routers.

`request['servletRequest']` / `request['servletResponse']`

These two parameters give you direct access to the underlying Java servlet request and response objects. `request['context']`. This provides direct access to the Ignition GatewayContext. More information about this object may be found in the [Ignition SDK developer guide](#) and associated JavaDocs.

`request['context']`

This provides direct access to the Ignition GatewayContext. More information about this object may be found in the [Ignition SDK Programmer's Guide](#) and associated JavaDocs.

Session Parameter

The session parameter is a Python dictionary which may be used to hold information which persists across multiple different calls to your Web Dev Python Resource, or across multiple Python Resources. Session management is handled automatically, and this dictionary is created for each new client session. (The client must have cookies enabled for sessions to work). You may place any key-value pairs in the session dictionary you'd like, just make sure that the values are things that can be serialized. All normal python types (scalar, dictionary, lists, and tuples) are serializable.

Return Value

Each `do*` function on a python resource must generate some sort of HTTP response. This is done by returning a value from the function. The return value should always be a dictionary, in which the following keys are recognized. The keys are listed here in the order they are evaluated. For example, if you have both file and bytes, only file will take effect. The exception is the contentType key, which may be included with any of the other keys to override the default content type.

File: The value should be a string, which should be the path to a file on the server's filesystem. If no contentType is specified, then the system will attempt to probe the content type from the operating

system using `java.nio.Files.probeContentType`. If the `file` key is present, but the value points to a file that doesn't exist, an HTTP 404 will be returned.

Bytes: The value should be a byte array. The default content type is `application/octet-stream`, but you probably want to specify your own.

HTML: The value should be a string, which should be the source of an HTML document. Content type is assumed to be `text/html`.

JSON: The value is assumed to either be a string (which should be valid json) or to be a python object, which will then be encoded into a json string. Content type will be `application/json`

Response: If none of the other keys are present, the system will look for the key `response` which will be stringified and then returned with the content type `text/plain`.

If your implementation of the `do*` function returns a dictionary with none of the above keys, an HTTP 500 error will be returned. However, if you return `None`, no HTTP 500 error will be returned. In this case, it is assumed that you used the `request[ 'servletResponse' ]` parameter to directly formulate your HTTP response.

Security Settings

Each python resource can configure its own optional security settings.

Require HTTPS

If this is checked, then the resource will only be accessible via an SSL connection. If a non-secure HTTP transport is used, the browser will be sent a redirect to the the Gateway's SSL port. The Gateway must have SSL enabled, of course.

Require Authentication

If this is checked, the resource will require authentication before it executes. This uses HTTP BASIC auth, and so should really be combined with the Require HTTPS option so that the credentials are encrypted. The username/password combination sent through the HTTP BASIC authentication headers will then be passed through the chosen User Source. If roles are specified, the user must have at least one of the roles. Specify multiple acceptable roles using a comma separated list. If the credentials are missing, an HTTP 401 will be returned with the `WWW-Authenticate` header. If the credentials are present but incorrect, an HTTP 403 will be returned.

If the credentials succeed, the python resource will execute. In addition, the authenticated user object returned by the User Source will be accessible inside the session object as `session['user']`. Since the user is stored in session, if the client has cookies enabled, then further requests against the same session will use the stored user object and will not require additional authentication.

httpPost Example

This example is to allow Ignition to receive data from an external source. It uses a button to send JSON data through an httpPost command and a Python Resource in the Web Dev module to receive the post and do something with the data. This button example is for testing purposes only, the common use-case for posting data is to use another program to post data.

1. Open the Designer and right-click on the Web Dev object in the Project Browser. Create a Python Resource named **postjson**.
2. Select the **doPost** HTTP method from the dropdown in the upper left. Copy this code into the doPost function.

"postjson" Python Resource (Web Dev Section)

```
# take in some JSON data and print it
# expecting 'names' and 'values' that are of the same length

# get the incoming parameters
data = request['postData']
names = data['names']
values = data['values']
# this will print to the wrapper.Log file
print names, values

# format the string into HTML
formattedString = "<html><body>"
# loop through and add names and values
for i in range(len(names)):
    formattedString += "%s: %s, " %(names[i], values[i])
# remove the last ', ' and add closing html
formattedString = formattedString[:-2]+ "</body></html>"
# this will print to the wrapper.Log file
print formattedString

# return the value string
return {'html': formattedString}
```

3. Create a button on a window to test the above code. Copy the code below into your button. Make sure to change the **ProjectName** variable to the name of your project. If you used any name other than "postjson" for step 1, change the **doPostName** variable as well.

Call Web Service (Button component on a window)

```
# post data to the web service in a json format
# this allows you to use the 'postData' object in the Python Resource

# create url to post to
projectName = "MyProject"
doPostName = "postjson"
url = "http://localhost:8088/main/system/webdev/%s/%s" %(projectName,
doPostName)
# create the dictionary of parameters to pass in
params = {}
params['names'] = ['String','Integer']
params['values'] = ['Hello World', 42]
# encode dictionary to JSON
jsonParams = system.util.jsonEncode(params)

# post to Ignition
postReturn = system.net.httpPost(url,'application/json',jsonParams)
# print return value
print postReturn
```

4. Now test your button. Make sure to open the console to see the print out, or the wrapper.log file to see any errors caused by the doPost function.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Client and Gateway Event Scripts

Client and Gateway Event Scripts



INDUCTIVE
UNIVERSITY

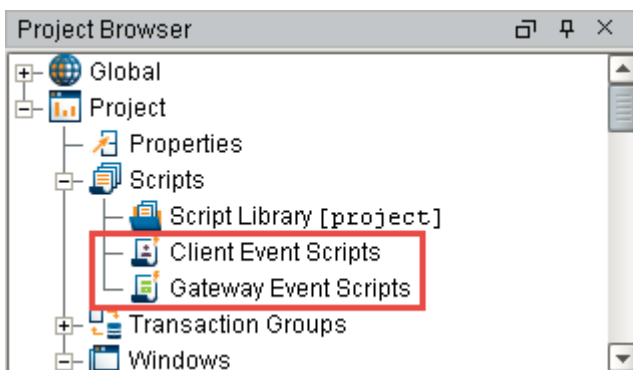
Gateway vs Client Event
Scripts

[Watch the video](#)

The Client and Gateway Event scripting workspaces are located in the Project menu of the Designer underneath Properties, or in the Project Browser under **Project > Scripts**. Each of the two **Scopes** have their own set of events but there are several repeated events. Even with the same names, different things may trigger the different Event Scripts.

NOTE

These scripts are stored inside a project, and will be copied if a project is copied.



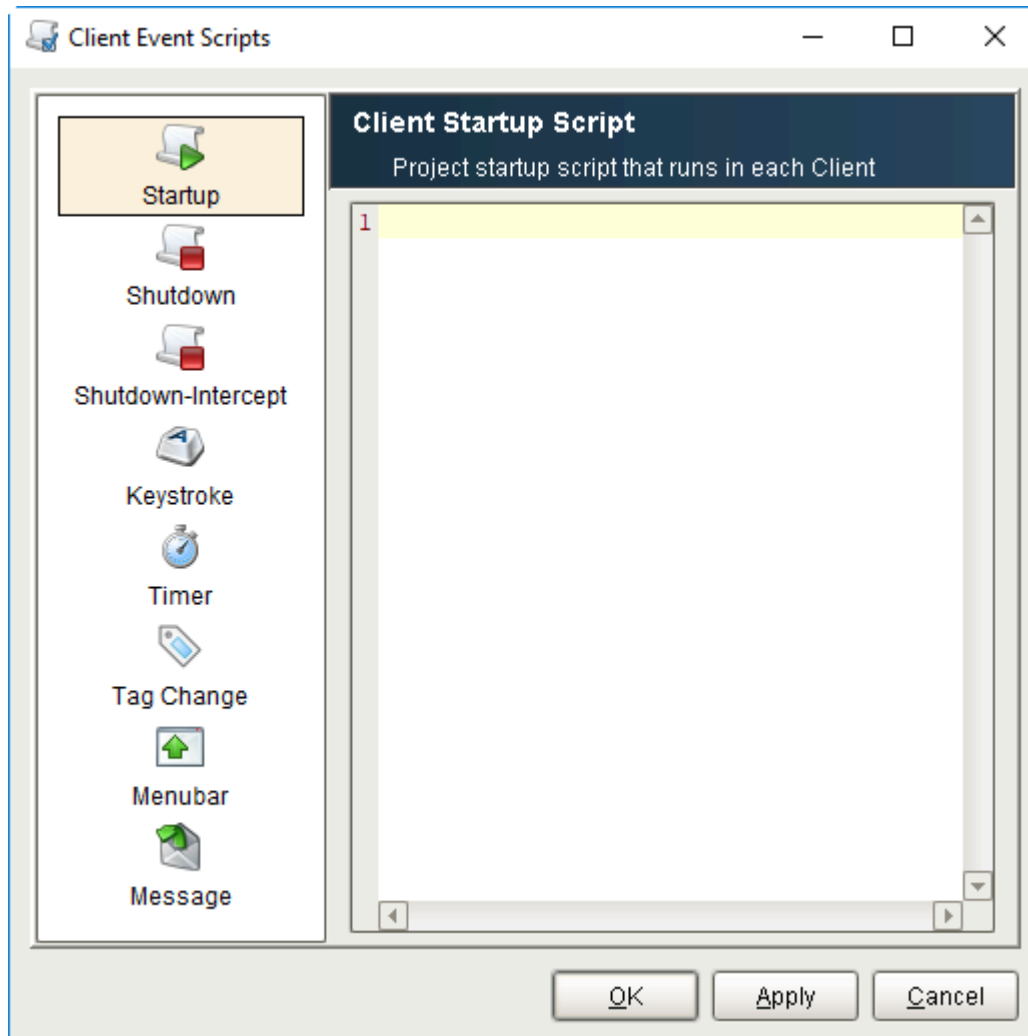
Client Event Scripts

Client Event Scripts execute in the running Client, which means that they may never execute if no clients are running, or they may execute many times if multiple clients are running. Client Event

Scripts will also execute in the Designer, but only in Preview Mode. Because Clients are full-fledged applications, Client Event Scripts run on the computer running the Client, not on the Gateway's host server computer. This means that they don't have access to the Gateway's file system, and so on.

The Client Event Scripts are:

- Startup (client)
- Shutdown (client)
- Shutdown-Intercept (client)
- Keystroke
- Timer
- Tag Change
- Menubar
- Message



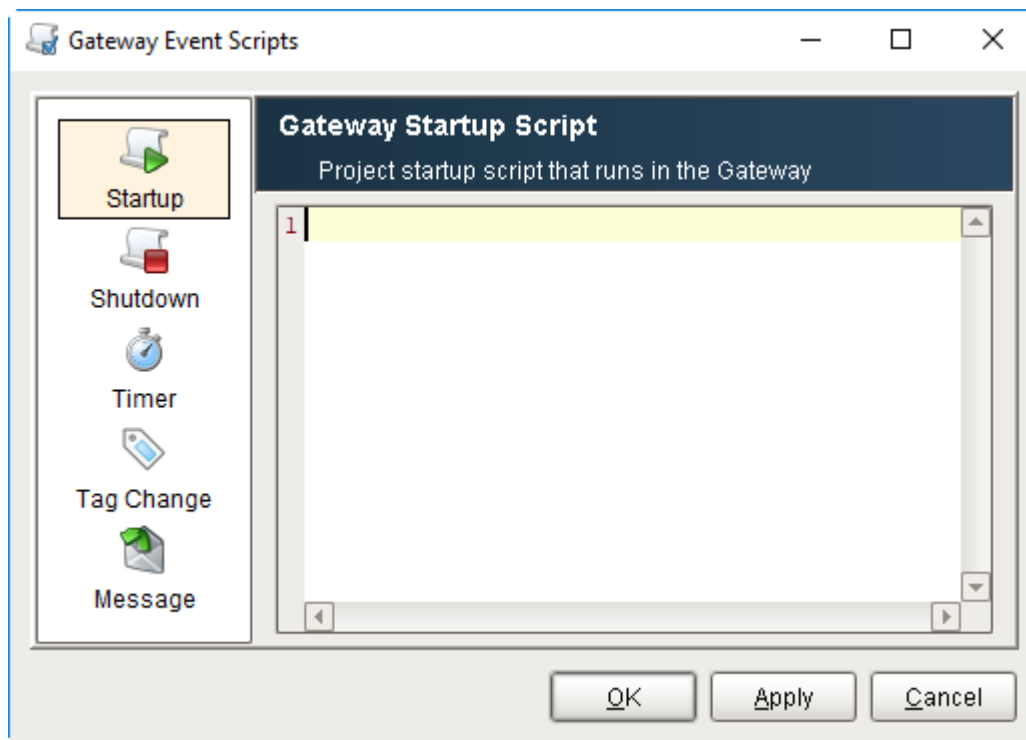
Gateway Event Scripts

Gateway Event Scripts execute on the Ignition Gateway service. This means that they will always execute as long as Ignition is running, even if there are no clients open. If there are multiple clients

open, these scripts will still only run once. For example, if you have a script that writes to the Gateway when a Tag changes, it belongs in the Gateway scope so you don't get duplicate records when multiple clients are open.

The Gateway Event Scripts are:

- Startup (Gateway)
- Shutdown (Gateway)
- Timer
- Tag Change
- Message



CAUTION

System functions are available for both Client Event Scripts and Gateway Event Scripts. Some system functions are available for both Client and Gateway Event Scripts, but some system functions are specific to either one or the other. When you're writing event scripts, it's important to remember the scope of where you writing the script: Client or Gateway. You can check [Scripting Functions](#) in the Appendix to see list of all system functions, their descriptions, and what scope they run in.

Startup Script



INDUCTIVE
UNIVERSITY

Startup Scripts

[Watch the video](#)

The Startup Script event runs at the startup of either the Client or the Gateway. This allows you to run a script at startup, giving you the chance to configure certain things dynamically.

Gateway Startup Behavior

In the Gateway scripting scope, this means that the script will run when the Gateway starts up and whenever the scripting configuration changes via a Designer save action. This means that while designing, the startup and shutdown events may happen frequently.

There is a specific order to when the various startup scripts are run. When troubleshooting your Gateway startup times, consider the following:

1. **Gateway starts** - The Gateway will start as an OS service, and start the context. No startup scripts can run before this is complete.
2. **Projects are started** - This includes all of the Gateway scoped items in the projects such as Transaction Groups, SFCs, etc. This does not refer to launching clients, and no clients can be automatically launched at this time. All **Gateway Startup Scripts** are run at this time for each project. Note: if you copied a project, always check for Gateway scoped events such as these. You generally don't want a Gateway Startup Script to run twice because it is in two projects.

Client Startup Behavior

In the Client scripting scope, Client Event Startup scripts run after a user successfully logs in to the client, but before any "Open on Startup" windows are opened. Client Event Scripts can be used to open up a custom set of windows based on who logged in.

Shutdown Script



INDUCTIVE
UNIVERSITY

Shutdown Scripts

[Watch the video](#)

The Shutdown Script event runs at the shutdown of either the Client or Gateway. It allows you to run a piece of code as the shutdown is occurring. After the script is run, the shutdown will finish.

Note: if the computer power is lost abruptly (power outage, hard restart, etc) this shutdown script will not run.

Gateway Shutdown Behavior

The Gateway Shutdown script will run as the Gateway shuts down, such as from the Gateway Control Utility.

Client Shutdown Behavior

The Client Shutdown script will run as the Client shuts down or logs out, such as with a script, or by clicking the close window "X" at the top of the OS window which is typical in most Operating Systems.

Shutdown-Intercept Script

Client Shutdown-Intercept Behavior (Not Available in Gateway Scripts)

The Shutdown-Intercept Script is unique in that it runs when a user attempts to shutdown a client, but before actual shutdown occurs. This means it will actually run before the Shutdown Script, and there is only a client version of it. There is a special event object that you can set a cancel property to prevent shutdown by using the code "event.cancel = 1." Doing this will cancel the shutdown event and leave the user at the spot they were last. This allows you to set special restrictions when the client is actually allowed to shut down, such as having a certain role, as seen in the example below:

Python - Cancel Application Exit

```
# Check to see if the user has a certain role.
if "SuperUser" not in system.security.getRoles():
    # If the role is not present, it will warn the user and cancel the shutdown process.
    system.gui.warningBox("Exit not allowed for non-admin user.")
    event.cancel = 1
```

Keystroke Scripts



INDUCTIVE
UNIVERSITY

Keystroke Scripts

[Watch the video](#)

Client Keystroke Behavior (Not Available in Gateway Scripts)

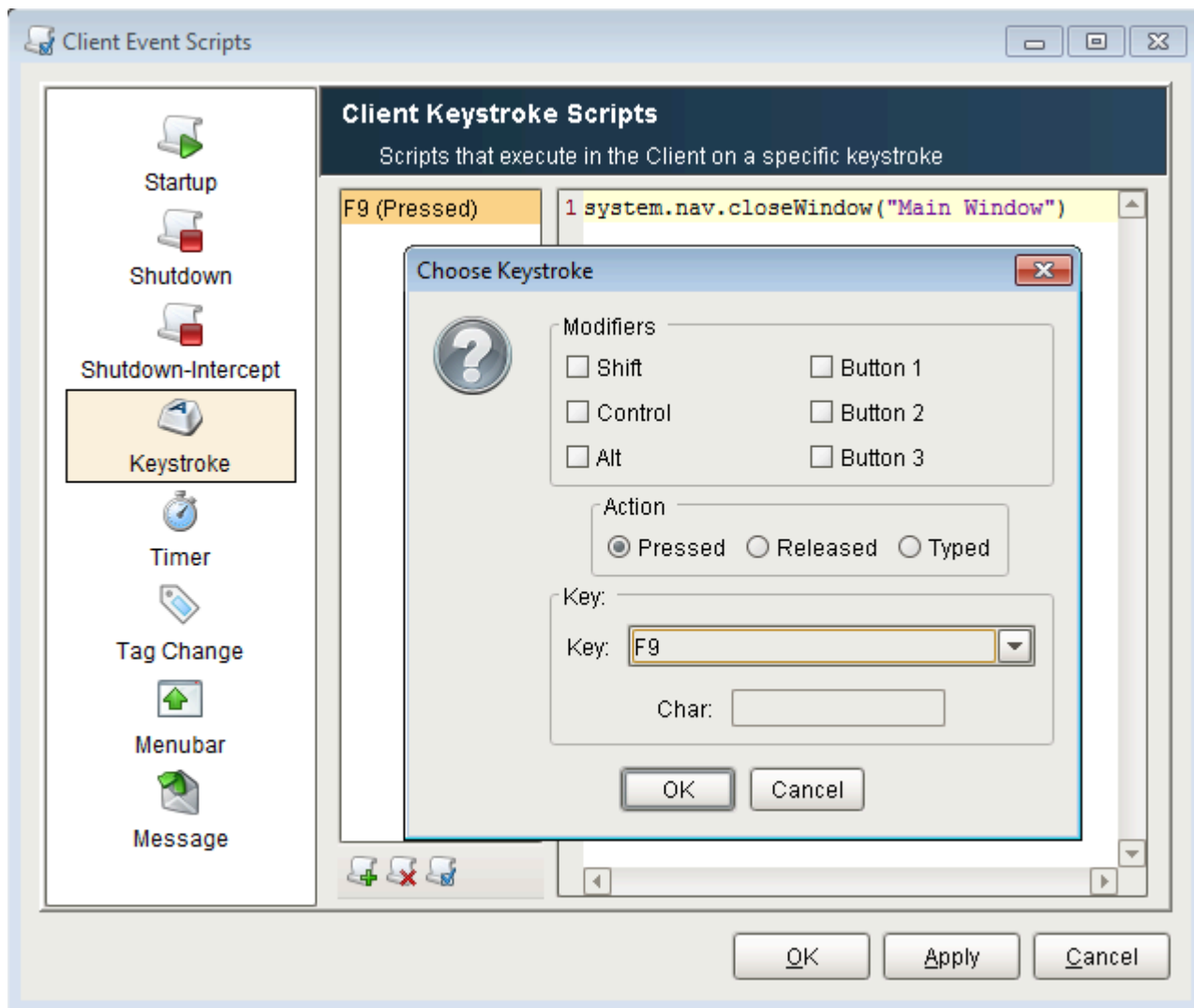
The Keystroke Scripts let you create different events that will activate on certain key combinations. They only run in the Client, but you may add as many Keystroke Script events as you'd like, as long as each one has a unique key combination.

Since multiple Keystroke Scripts can be added, there are separate buttons that allow you to add () a Keystroke Script, or edit () and delete () the currently selected Keystroke Script.

When choosing a keystroke, you may choose any number of modifiers, which are keys or mouse buttons that must be down to activate the keystroke. You can also choose whether or not the keystroke is on the pressed or released event of a keyboard key, or upon the typing of a character. Special keys like the Function keys (F1) or ESC key are only available in the pressed and released actions. The example below shows the F9 key when pressed, closes the window called "Main Window."

CAUTION

Some Operating Systems reserve certain keys for certain function, and will capture the key press or release before it gets sent to the Client. For example, many Operating Systems use the TAB key to shift focus to the next field.



Timer Scripts



INDUCTIVE
UNIVERSITY

Timer Scripts

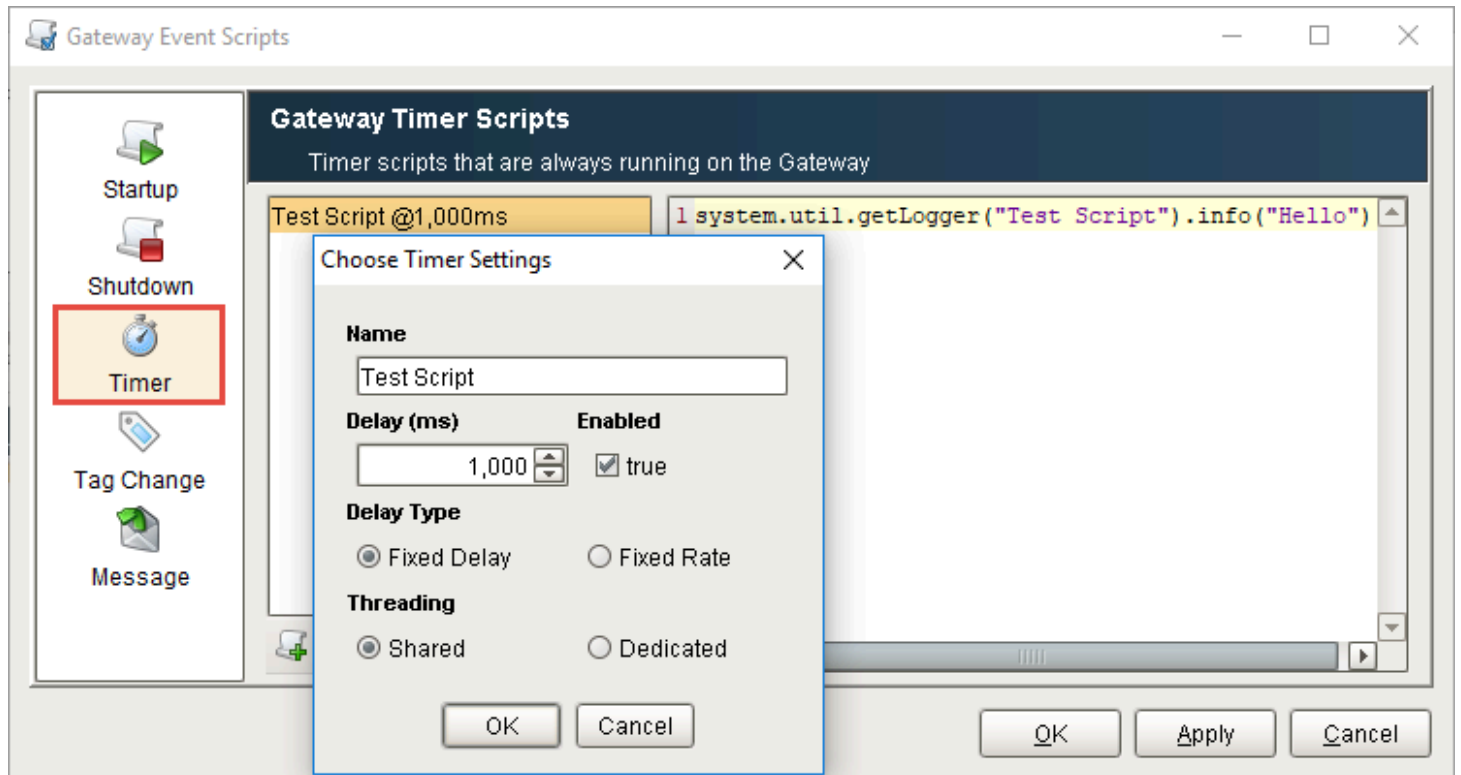
[Watch the video](#)

The Timer Scripts execute periodically on a timer at a fixed delay or rate. This allows you to set up a sort of heartbeat that can run in the Client or the Gateway Scope. Since multiple Timer Scripts can be added, there are separate buttons that allow you to add () a Timer Script, or edit () and delete () the currently selected Timer Script. The Timer Script has three major settings: Delay in milliseconds, Delay Type, and Threading.

A **Fixed Delay** timer script (the default) waits for the given **Delay** between each script invocation. This means that the script's rate will actually be the delay plus the amount of time it takes to execute the script. This is the safest option since it prevents a script from mistakenly running continuously because it takes longer to execute the script than the delay.

Fixed Rate scripts attempt to run the script at a fixed rate relative to the first execution. If the script takes too long, or there is too much background process, this may not be possible. See the documentation for [java.util.Timer.scheduleAtFixedRate\(\)](#) for more details.

In addition, all timer scripts for a given project that choose to run in a **Shared** thread will all execute in the same thread. This is usually desirable, to prevent creating lots of unnecessary threads. However, if your script takes a long time to run, it will block other timer tasks on the shared thread. In this case, you can have the Timer Script run in a **Dedicated** thread so that it runs independently. The rule of thumb here is that quick-running tasks should run in the shared thread, and long-running tasks should get their own dedicated thread.



Gateway Timer Behavior

This script will execute in the Gateway, meaning that it will begin running when the Gateway starts, and will continue running at the rate specified until the Gateway is stopped.

Client Timer Behavior

Since this will execute in a Client, it is important to remember that Client Timer Scripts may never execute (if no clients are open) or may execute many times (once per open client). They start when the client has a successful login.

Tag Change Scripts



INDUCTIVE
UNIVERSITY

Tag Change Scripts

[Watch the video](#)

The Tag Change Script event occurs when either the Value, Quality, or Timestamp of a Tag or list of Tags changes. They will also get an initial execution whenever the scripting system starts up. You can set the Tag Change Script to run in either the Client or the Gateway Scope. Multiple Tag Change Scripts can be created, using the add () button to add a script, and the remove () button to remove the selected script. To specify multiple Tags for a given script, enter them one per line in the Tag paths text area. To quickly add many Tags, you can open the Tag Browser () and drag-and-drop Tags from the window onto this text area. The actual script is then entered in a different tab. When executing, each Tag Change Script runs in a separate thread. This prevents long running scripts from blocking the execution of other Tag Change Script.

These scripts receive three special variables in their namespace when they are run: **event**, **initialChange**, and **newValue**. Both the event and newValue objects refer to the Tag that changed to initiate the event.

- **initialChange** - a variable that is a flag (0 or 1) which indicates whether or not the event is due to the initial subscription or not. This is useful as you can filter out the event that is the initial subscription, preventing a script from running when the values haven't actually changed.
- **event** - a **TagChangeEvent** object which contains the properties: tag, tagPath, and tagProperty, all of which are also complex objects.
 - **event.tag** - a **Tag** object which contains basic information about the Tag such as name, value, and type that can be accessed via `*event.getTag().getName()`.
 - **event.tagPath** - a **TagPath** object which contains information pertaining to the Tag path such as the parent and child paths via `event.getTagPath().getParentPath()`. You can also get the path of the Tag by calling `event.getTagPath().toString()`.
 - **event.tagProperty** - a **TagProperty** object, which can be used to create an enum of a property, which can be used in conjunction with the Tag object to access properties of the Tag such as OPC Item Path or Scale mode via `event.getTag().getAttribute(event.getTagProperty().getProp("OPCItemPath"))`. The possible property names are listed here: [Tag Properties](#)
- **newValue** - a **newValue** object which contains a value, quality, and a timestamp of the Tag. They can be called via `newValue.getValue()`.

Gateway Tag Change Behavior

Having the Tag Change Scripts run in the Gateway Scope means that the scripts will trigger when a Tag in their Tag list changes as long as the Gateway is running. The script will be able to interact with the Gateway and other Tags, but not the project.

Client Tag Change Behavior

Having the Tag Change Scripts run in the Client Scope means that the scripts will trigger when a Tag in their Tag list changes, but only if a Client of that project is open. This means that if no Clients are open, the script will not fire, and if many clients are open, the script will fire once for each open client. Additionally, because this is run from the Client Scope, it will have access to Client resources such as Client Tags and can trigger things in the Client such as opening a window.

Menubar Scripts



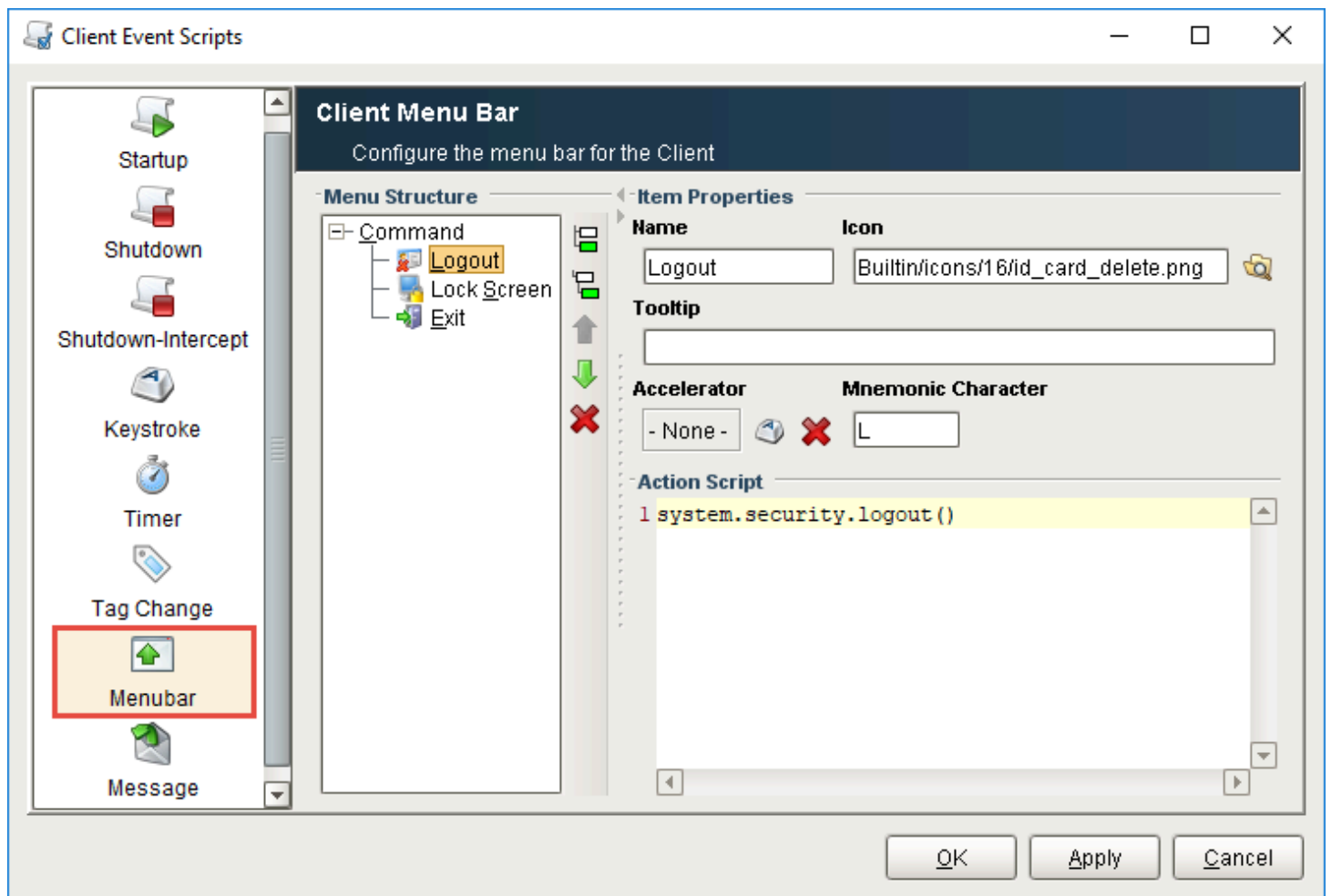
INDUCTIVE
UNIVERSITY

Menu Bar

[Watch the video](#)

Client Menubar Behavior (Not Available in Gateway Scripts)

The Client Menubar Scripts create and control the options available in the menubar of the Client. As such, these scripts are only available in the Client Scope. By default, a Client will have three menus: Command, Windows, and Help. The Windows and Help Menu are separate, and controlled through the project properties, but the Command menu is actually created in the Client Menubar Scripts. The default menu structure gives you an idea of how to create a menu.



The top level nodes will show up in the menubar. Each node can have an action script unless it has children. In the image above, the Command object will have a grayed out action script field because it has three children. This allows the Client to click into the menu, and look at what is inside. Children of objects can themselves have children, allowing you to create submenus within your menu. The structure is easy to create, using the buttons to the right to add peer () or child () nodes. You can also move the nodes up () or down () in the hierarchy, as that will affect the order they appear in the Client. You can also delete () nodes that you don't want.

With a node selected you can add to or edit its behavior. Each node can be given a name that it will display in the Client, an icon which will be displayed at the front of the name, and a tooltip that will display when hovering over the menu option. These help to identify what the node is and what it does.

Accelerators and Mnemonics

Each node can also be given an accelerator and a mnemonic character. The accelerator is a key or key combination that can be pressed at any time in the client to initiate that nodes event. It will display next to the name of the node in the client, and works much like many commonly known accelerators such as Control + S to Save. The mnemonic character is a key that can be pressed when currently in the menu to initiate the node event. If the character chosen is in the name of the node, then that character will be underlined.

Finally, you can set an action script on the node which will determine what happens when the node is selected. Typically this is used for navigation, such as swapping to a new window, or logging out and exiting, like the default nodes. However, you can enter in any kind of script you want.

Message Scripts

Message Handlers allow you to write a script that will run in either the Client or Gateway that they are located in, but they can be invoked by making a call from other projects or even other Gateways. They can be called using three different scripting functions: **system.util.sendMessage**, **system.util.sendRequest**, and **system.util.sendRequestAsync**.



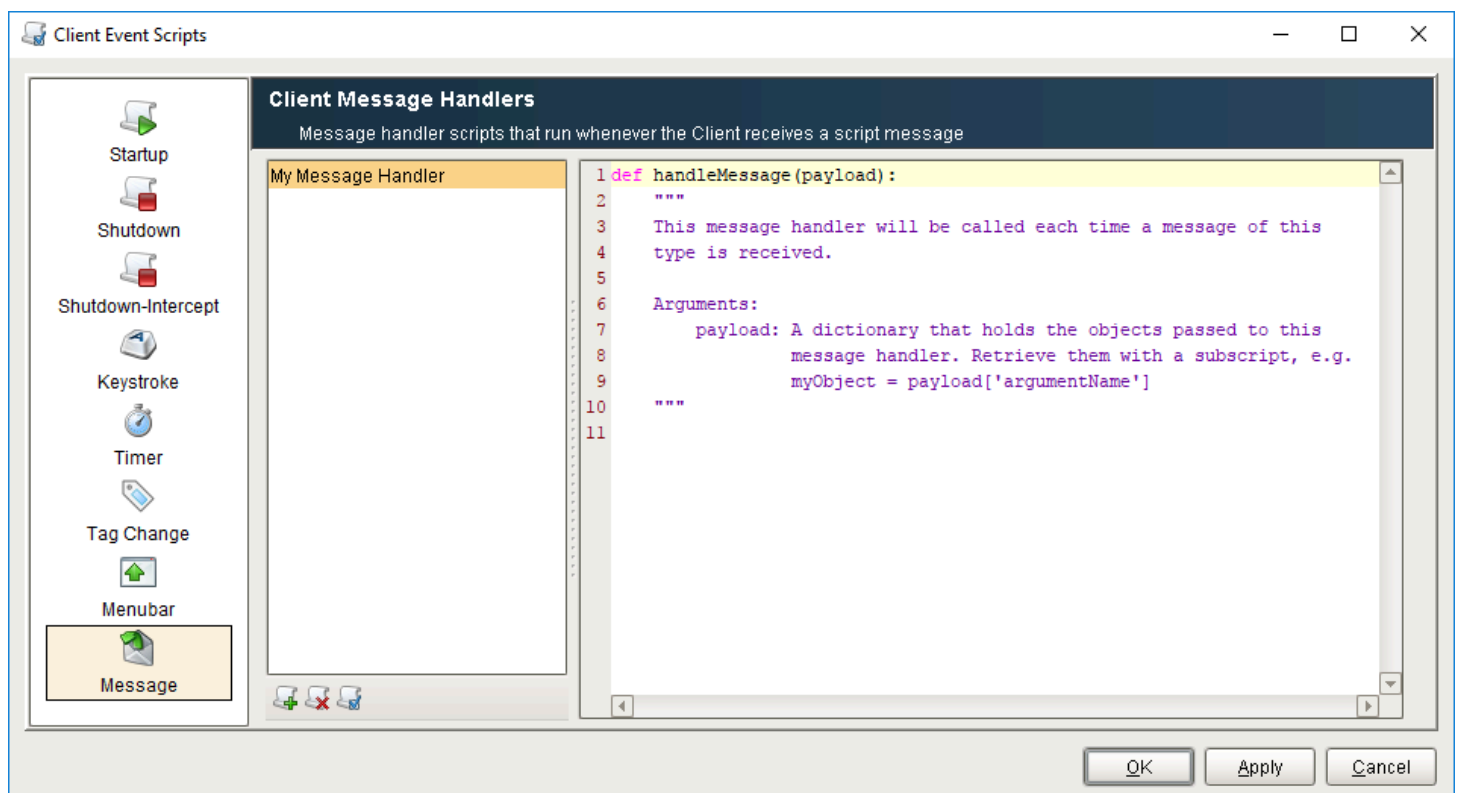
INDUCTIVE
UNIVERSITY

Script Messaging

[Watch the video](#)

Client Message Handlers

Located in the Message section of Client Event Scripts, client message handlers will execute in the client. This means if you have five clients open for a project with a message handler that gets called, the message handler will run in each client. You can easily create and manage all of your client message handlers in the Client Event Scripts window. Clicking on one of your message handlers will bring up its script on the right.

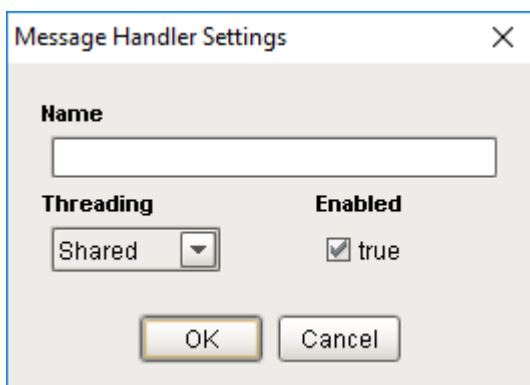


Under the list of handlers, three small buttons allow you to add, remove and manage your handlers.

-  **Add Message Handler** - Will add a message handler.
-  **Remove Message Handler** - Will delete the highlighted message handler.
-  **Modify Message Handler** - Will modify the settings for the highlighted message handler.

When adding or modifying a message handler, a small settings window will popup. In it, you can give your message handler a name or modify its existing name, enable or disable the message handler, and even select how it executes under the **Threading** dropdown. There are three different options that dictate how the message handler will execute:

- **Shared** - The default way of running a message handler. Will execute the handler on a shared pool of threads, in the order that they are invoked. If too many message handlers are called all at once and they take long periods of time to execute, there may be delays before each message handler gets to execute.
- **Dedicated** - The message handler will run on its own dedicated thread. This is useful when a message handler will take a long time to execute, so that it does not hinder the execution of other message handlers.
- **EDT** - This will run the message handler on the Event Dispatch Thread (EDT) which also updates the GUI. If a message handler were to take a long time to execute, it would block the GUI from running which may lock up your client. This is helpful when your message handler will be interacting with the GUI in some way, as the GUI will not be able to update until the message handler finishes.



The image shows a 'Message Handler Settings' dialog box. It has a title bar with a close button (X). Inside, there is a 'Name' label followed by a text input field. Below that, there are two sections: 'Threading' with a dropdown menu currently set to 'Shared', and 'Enabled' with a checked checkbox labeled 'true'. At the bottom, there are 'OK' and 'Cancel' buttons.

Inside the message handler is your script. The script will have a single object available to it, the payload. The payload is a dictionary containing the objects that were passed into it. Each object in the payload dictionary can be accessed by calling their corresponding key. For example:

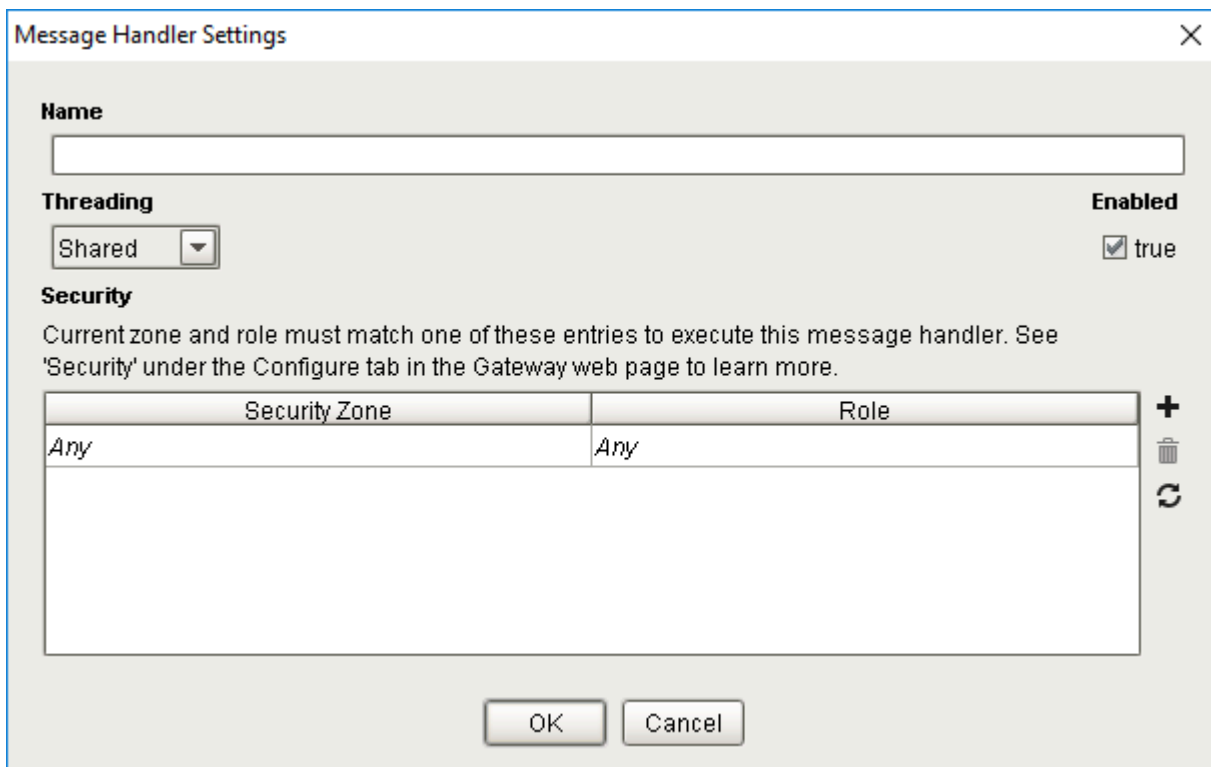
Pseudocode - Payload Values

```
value1 = payload["MyFirstValue"] # "MyFirstValue" is the key that is associated
with a value. We are taking the value associated with MyFirstValue, and assigning
it to value1.
value2 = payload["MySecondValue"] # Similarly, we are taking the value associated
with MySecondValue and assigning it to value2.
```

Gateway Message Handlers

Gateway Message Handlers are setup and function similarly to client message handlers. However, there are two major differences:

- They are setup in the Message section of the Gateway Event Scripts. As such, they are executed on the Gateway.
- **New in 7.9.4** They have an added layer of security. In the Message Handler Settings, it is possible to configure Security Zone/User Role pairs. The user invoking the message handler must match one of the combinations of Security Zone/User Role listed or else the message handler will not execute.



The screenshot shows the 'Message Handler Settings' dialog box. It has a title bar with a close button (X). The dialog is divided into several sections:

- Name:** A text input field.
- Threading:** A dropdown menu currently set to 'Shared'.
- Enabled:** A checkbox that is checked, with the label 'true' next to it.
- Security:** A section with a descriptive text: 'Current zone and role must match one of these entries to execute this message handler. See 'Security' under the Configure tab in the Gateway web page to learn more.'
- Table:** A table with two columns: 'Security Zone' and 'Role'. It contains one row with the value 'Any' in both columns. To the right of the table are three icons: a plus sign (+), a trash can, and a refresh/circular arrow icon.
- Buttons:** 'OK' and 'Cancel' buttons at the bottom.

Using Message Handlers

Once you have your message handlers created, you can then call them from a script using one of three scripting functions: **system.util.sendMessage**, **system.util.sendRequest**, and **system.util.sendRequestAsync**.. These functions allow you to call a message handler in any project, even if the project that the message handler resides on is different from the one you are

calling it from. The message handler will then execute in the scope in which it was created, and will use any parameters that you pass in through the payload.

Pseudocode - Calling a Message Handler

```
project="test"
messageHandler="My Message Handler"
myDict = {'MyFirstValue': "Hello", 'MySecondValue': "World"}
results=system.util.sendMessage(project, messageHandler, myDict)
```

Troubleshooting Gateway and Client Scripts

For both Gateway and Client scripts, Ignition gives you the tools to quickly check the status, troubleshoot, and diagnose problems with your scripts.

Gateway Scripts

The Gateway has a special section (**Gateway webpage - Status > Systems > Gateway Scripts**) where you can quickly check to make sure your Gateway scripts are running properly. If any of your scripts have an error, you can find the details of the error to help you troubleshoot what went wrong. You can also find a list of logged errors for all Gateway Event Scripts under Log Activity. To learn more about troubleshooting Gateway scripts, refer to [Gateway Scripts](#).

Client Scripts

The Console is very a important tool in Ignition for troubleshooting Client scripts. You can check to see if your script is working directly from the Client window, or the Designer while in Preview Mode. Any client scripting errors along with printouts go to the Console. The Console will identify the script name, error message, what line the script error is in, and a description of the problem.

To access the Console from a Client, go to the menubar and select **Help > Diagnostics > Console**. To access the Console from Preview Mode in the Designer, go to the menubar **Tools > Console**.