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

| | | | |
|---|---|---|---|
| | | `list = [1, 2, 3, 4, 5, 4]`<br>`print list.count(4)` | |
| list.append (x) | Will add **x** to the end of the list. | `list = [1, 2, 3, 4, 5]`<br>`list.append(6)`<br>`print list` | [1, 2, 3, 4, 5, 6] |
| list.insert(i, x) | Will insert **x** at position **i**. | `list = [1, 2, 3, 4, 5]`<br>`list.insert(1, 5)`<br>`print list` | [1, 5, 2, 3, 4, 5] |
| list.remove (x) | Will remove the first **x** from the list. | `list = [1, 2, 3, 2, 4, 5]`<br>`list.remove(2)`<br>`print list` | [1, 3, 2, 4, 5] |
| 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. | `list = [1, 2, 3, 4, 5]`<br>`list.pop(2)`<br>`print list`<br>`list.pop()`<br>`print list` | 3<br>[1, 2, 4, 5]<br>5<br>[1, 2, 4] |
| list.reverse() | Will reverse the items in the list. | `list = [1, 2, 3, 4, 5]`<br>`list.reverse()`<br>`print list` | [5, 4, 3, 2, 1] |

## 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')
```

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

Related Topics ...

- Numeric Types
- Strings
- Dictionaries
- Datasets
- Dates

# 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}
```

### On this page ...





### 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 | Output |
|---|---|---|---|
| len(dictionary) | Returns the number of items in the dictionary. | `myDictionary = {"name":"John Smith", "id": 12345}`<br><br>`print len (myDictionary)` | 2 |
| del dictionary[key] | Will remove the named key. | `myDictionary = {"name":"John Smith", "id": 12345}`<br><br>`del myDictionary["id"]`<br><br>`print myDictionary` | {'name': 'John Smith'} |
| key in dictionary | Will return True if the dictionary has that key.<br><br>Can also use "key not in dictionary" | `myDictionary = {"name":"John Smith", "id": 12345}`<br><br>`if "name" in myDictionary:`<br>`  print myDictionary["name"]` | John Smith |
| dictionary.clear() | Remove all of the items in the dictionary. | `myDictionary = {"name":"John Smith", "id": 12345}`<br><br>`myDictionary.clear()`<br><br>`print myDictionary` | {} |
| dictionary.keys() | Returns a list of the dictionary's keys. | `myDictionary = {"name":"John Smith", "id": 12345}`<br><br>`print myDictionary.keys()` | ['name', 'id'] |
| dictionary. values() | Returns a list of the dictionary's values. | `myDictionary = {"name":"John Smith", "id": 12345}`<br><br>`print myDictionary.values()` | ['John Smith', 12345] |

Related Topics ...

- Client Message Handler
- Gateway Message Handler
- Numeric Types
- Strings
- Lists and Tuples
- Datasets
- Dates

# 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)
```

**Note:** 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 | Output |
|---|---|---|---|
| data.getColumnCount() | Returns the number of columns in the dataset. | `print cities.getColumnCount()` | 4 |
| data.getColumnIndex(colName) | Returns the index of the column with the name colName. | | 2 |

| | | | |
|---|---|---|---|
| | | `print cities.getColumnIndex("Timezone")` | |
| data.getColumnName(colIndex) | Returns the name of the column at the index colIndex. | `print cities.getColumnName(1)` | Population |
| data.getColumnNames() | Returns a list with the names of all the columns. | `print cities.getColumnNames()` | [City, Population, Timezone, GMTOffset] |
| data.getColumnType(colIndex) | Returns the type of the column at the index colIndex. | `print cities.getColumnType(3)` | `<type 'java.lang.Integer'>` |
| data.getColumnTypes() | Returns a list with the types of all the columns. | `print cities.getColumnTypes()` | [class java.lang.String, class java.lang.Integer, class java.lang.String, class java.lang.Integer] |
| data.getRowCount() | Returns the number of rows in the dataset. | `print cities.getRowCount()` | 5 |
| data.getValueAt(rowIndex, colIndex) | Returns the value at the specified row and column indexes. | `print cities.getValueAt(1, 2)` | PST |
| data.getValueAt(rowIndex, colName) | Returns the value at the specified row index and column name. | `print cities.getValueAt(2, "Population")` | 2853114 |

### 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. You can use the functions above to do this.

**Python - Looping Through a Dataset**

```
# 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

**Note:** PyDatasets can 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 accessed 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
```

## PyRow

A PyRow is a row in a PyDataset. It works similarly to a Python list.

The examples and outputs are based on the results in the table below. In addition, "print" commands are used, but should be replaced by appropriate logging methods (such as system.util.getLogger) depending on the scope of the script.

| A | B |
|---|---|
| Apple | Orange |
| Banana | Orange |
| Apple | Apple |

| Method | Description | Syntax | Example | Output |
|---|---|---|---|---|
| index() | Returns the index of first occurrence of the element. Returns a ValueError if the element isn't present in the list. | index (element) | `for row in pyDataset:`<br>`  try:` | |

| | | | | |
|---|---|---|---|---|
| | | | print row.<br>index("Apple")<br>except:<br>print "No<br>apples in this row" | 0<br>No apples<br>in this row<br>0 |
| count() | Calculates total occurrence of given element in the row. | count<br>(element) | for row in pyDataset:<br>print row.count<br>("Apple") | 1<br>0<br>2 |

### Repeating Elements

You can also have repeating elements in a row:

| Example | Output |
|---|---|
| for row in PyDataset<br>print row * 2 | [u'Apple', u'Orange', u'Apple', u'Orange']<br>[u'Banana', u'Orange', u'Banana', u'Orange']<br>[u'Apple', u'Apple', u'Apple', u'Apple'] |

## Altering a Dataset

Technically, you cannot alter a dataset. Datasets are immutable, meaning they cannot change. You can, however, create new datasets. 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)
```

Related Topics ...

- Numeric Types
- Strings
- Lists and Tuples
- Dictionaries
- Dates

# Dates

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.

## On this page ...





## 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. Additional examples and functions can be found in the scripting appendix.

### 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 Vision - 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) |
| 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

While Java's Calendar class if 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.

**Ignition's System Functions vs Java's Calendar Class**
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

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.

**Ignition's System Functions vs Python's Libraries**
It is highly recommended to use Ignition's built-in system.date functions.

### Creating Dates - Python's Time Library

The time library can be use 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.

Notice 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
```

Related Topics ...

- Numeric Types
- Strings
- Lists and Tuples
- Dictionaries
- Datasets

# 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.

### 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 | Output |
|---|---|
| `x = 5`<br>`z = 15`<br>`if x < 10:`<br>`  # Since the condition "x < 10" is true,`<br>`  # the following line will execute`<br>`  print "'x' is less than 10"`<br>`if z < 10:`<br>`  # This condition "z < 10" is false,`<br>`  # so the following line will not execute`<br>`  print "this will never show"` | `'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 | Output |
|---|---|
| `x = 15`<br>`if x < 10:`<br>`  print "x is less than 10"`<br>`else:`<br>`  print "x is not less than 10"` | `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. For example, this script will print out **`three`**:

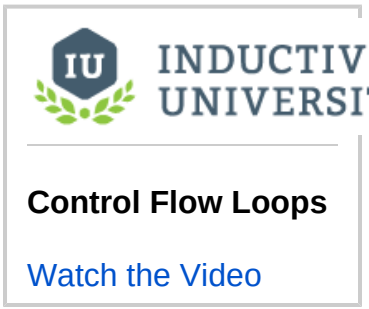
| Example | Output |
|---|---|
| `x = 3`<br>`if x == 1:`<br>`  print "one"`<br>`elif x == 2:`<br>`  print "two"` | three |

## On this page ...

```
elif x == 3:
   print "three"
else:
   print "not 1-3"
```

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

## For-Loops and While-Loops




### 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 | Output |
|---|---|
| `listOfFruit = ['Apples', 'Oranges', 'Bananas']`<br>`for fruit in listOfFruit:`<br>`  print fruit` | `Apples`<br>`Oranges`<br>`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 | Output |
|---|---|
| `# Even though this example isn't using the value of "x",`<br>`# the print statement will still be executed once for each item`<br>`# in the list returned by range().`<br>`for x in range(4):`<br>`  print "this will print 4 times"` | `this will print 4 times`<br>`this will print 4 times`<br>`this will print 4 times`<br>`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 | Output |
| --- | --- |
| `listOfFruit = ['Apples', 'Oranges', 'Bananas']`<br>`x = 0`<br>`while x < len(listOfFruit):`<br>`  print listOfFruit[x]`<br>`  x = x + 1` | `Apples`<br>`Oranges`<br>`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 " Fini shed ".

| Example | Output |
| --- | --- |
| `for x in range(10):`<br>`  if x >= 2:`<br>`  break`<br>`  print "Loop"`<br>`print "Finished"` | `Loop`<br>`Loop`<br>`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 | Output |
| --- | --- |
| `for x in range(10):`<br>`  if x == 4:`<br>`  continue`<br>`  print x` | `0`<br>`1`<br>`2`<br>`3`<br>`5`<br>`6`<br>`7`<br>`8`<br>`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.
```

Related Topics ...

- Error Handling
- Getting Started with Scripting in Ignition

# 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 it 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.

**On this page ...**



**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
```

## Displaying Error Text

Sometimes you want to get the actual text from an error in addition to protecting your script. There's an easy way to fetch that information that's built into Python. When you are inside an except section of code, sys.exc_info() gives you access to the error text as a list of values. You can use this to print out your message, display it on the screen, send it to the database, or anything else.

This example should be put on a button, and will write the error text to a Label component that is a sibling to the button. This is useful in Perspective to get error messages out of views and event actions.

```
try:
        # cause an error
        x=[1,2]
        val = x[5]
except:
        # push the error text to a sibling label
        self.getSibling("Label").props.text = sys.exc_info()
```

You can also use the Ignition loggers to push these error messages out to the Gateway console. You can find these in the Gateway Webpage under the Status section, on the **Logs** page.

```
# This code would log an error to the gateway
try:
        # cause an error
        100/0
except:
        # push the error text to the logger
        logger = system.util.getLogger("myLogger")
        # convert the sys.exc_info() to a string and log it
        logger.info(str(sys.exc_info()))
```

## 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,<br>but the index specified doesn't exist. This exception is unique to datasets in Ignition. Lists,<br>and other built-in Python objects will return the IndexError below. | `myDataset = system.dataset.toDataSet (["colName"],[[0]])`<br><br>`# This will fail because the dataset only has a single row,`<br>`# so trying to read the value at row index 5 means we're trying to`<br>`# read something that doesn't exist.`<br>`print myDataset.getValueAt(0,5)` |
| AttributeError | 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. | `myVar = 10`<br><br>`# integers do not natively have a name variable, so this will fail with an AttributeError:`<br>`# there isn't an 'attribute' on an integer by the name of 'name'`<br>`print myVar.name` |
| IndexError | Similar to ArrayIndexOutOfBoundsException above, but occurs when Python specific object, such as a list. | `myList = [1,2,3]`<br><br>`# There isn't an element in the list at index 4, so this will fail.`<br>`print myList[4]` |
| NameError | Occurs when the local or global name is not found. Typically happens when referencing a variable that hasn't been assigned a value. | `# We haven't given a value to the variable myValue, so it will fail.`<br>`print "The value of the variable is: " , myValue` |
| TypeError | A TypeError occurs when a function or similar operation is applied to another object of an incorrect type. | `myList = [1,2,3]`<br><br>`# The first argument for pop() expects an integer, so passing a string value is innappropriate.`<br>`# Passing a string to pop() will return a TypeError.`<br>`print myList.pop("0")` |
| ValueError | A ValueError is returned when the value of an argument is inappropriate. Typically the exact issue is returned in the returned exception. | `myVar = "Hello"`<br>`# Strings can be passed to the int() function, but the value needs to be something that`<br>`# can be coerced into an integer, like "0".`<br>`# Because "Hello" can't be easily converted, the line below will fail.`<br>`print int(myVar)` |

Related Topics ...

- Conditions and Loops

# 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.

**On this page ...**



### 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<br><br>When casting a String or Unicode value to a boolean, an empty string is False, any other string is True. | `# Results in False`<br>`print bool("")`<br><br>`# Results in True`<br>`print bool("Test")` | `False`<br>`True` |
| `int() and long()` | When casting a float, rounding will not occur automatically to the decimal value. Instead, the round() function should be called.<br><br>When casting a String or Unicode value, the string literal needs to be a valid integer or long: decimal values contained in the string will result in a ValueError<br><br>Integers have at least 32 bits of precision, while Longs have unlimited precision. | `# Float to Integer`<br>`print int(123.8)`<br><br>`# Float to Long`<br>`print long(321.8)`<br><br>`# String to Integer`<br>`print int("400")`<br><br>`# ValueError: the value is not base 10`<br>`print int("400.5")` | `123`<br>`321`<br>`400`<br>`ValueError` |
| `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 ("."). | `# Integer to Float`<br>`print float(123)`<br><br>`# String to Float`<br>`print float("400.5")` | `123.0`<br>`400.5` |
| `str() and unicode()` | Most objects can be cast as a string representation of some sort, including sequences. | `print "First line:" + str(80)`<br><br>`# Even sequences can be cast as a string,`<br>`# making for easy concatenation`<br>`myList = [1,2,3]`<br>`print str(myList)` | `80`<br>`[1, 2, 3]` |

## Checking an Object's Type

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

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

**Python - Type Validation: type vs isinstance**

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

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 | Description | 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.<br><br>If the optional digits argument is used, then it rounds to that many decimal places. | `var = 10.236981`<br>`print round (var)`<br>`print round (var,3)` | `10`<br>`10.237` |
| `math.floor (number)` | Returns a truncated integer from the number given. The largest integer value less than or equal to *number*.<br><br>Note that the example needs to import that math library before being able to call floor(). | `import math`<br><br>`var = 100.938`<br>`print math. floor(var)` | `100.0` |
| math.ceil(number) | Returns the ceiling integer from the number given. The smallest integer value greater than or equal to *numb er*.<br><br>Note that the example needs to import that math library before being able to call ceil(). | `import math`<br><br>`var = 100.138`<br>`print math. ceil(var)` | `101.0` |

**Python - Simple Casting**

```
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))
```

Related Topics ...

- User Defined Functions
- Libraries

# 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.

## 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 Libray**

```
# 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.
```

### On this page ...




**System Library**

Watch the Video

```
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.

> Ignition uses Python version 2.7. This means that any imported libraries must be compatible with Python 2.7.

## 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
```

Related Topics ...

- Built-in Functions

# User Defined Functions

## Functions

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.

### On this page ...

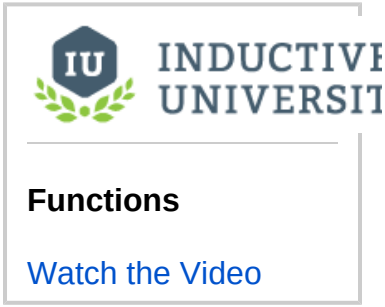


### Defining Functions

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 Arugments**

```
# 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:

> 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 as 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 scripts 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
```

Related Topics ...

- Built-in Functions
- Libraries

# 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 Vision Clients and Perspective Sessions.

### Script Scope

One important thing to keep in mind before scripting in Ignition, is to understand the concept of scope. Within Ignition, there are different scopes: the Gateway Scope, the Perspective Session 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.

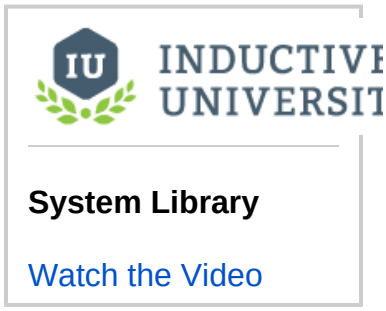
**Python - Simple Script Using a System Function**

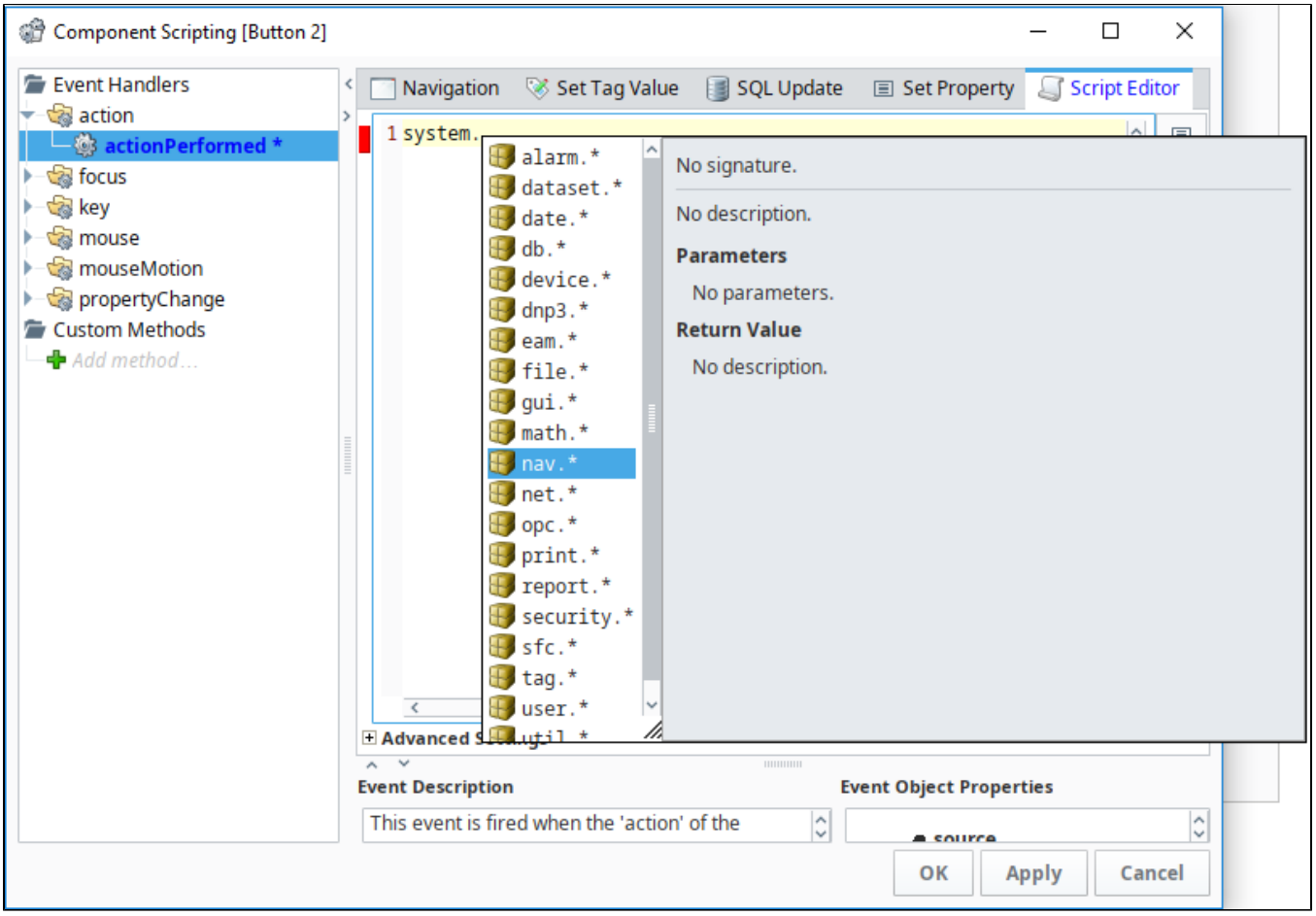
```
value = system.tag.read("tagPath").value
```

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.

## On this page ...

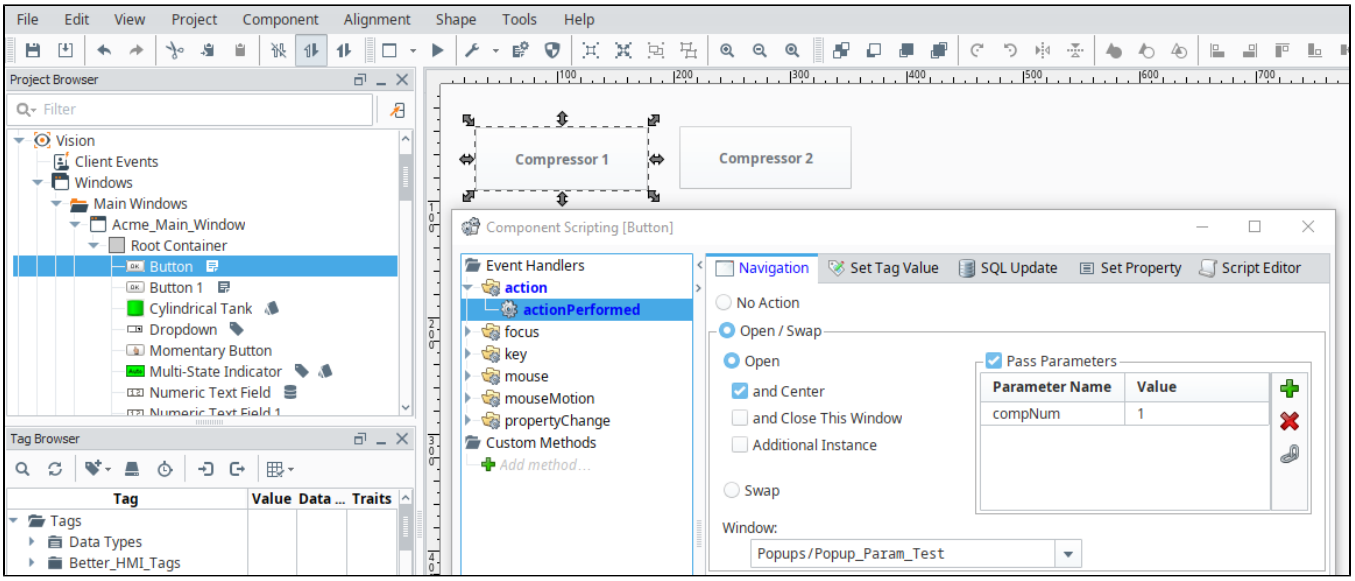





## Components

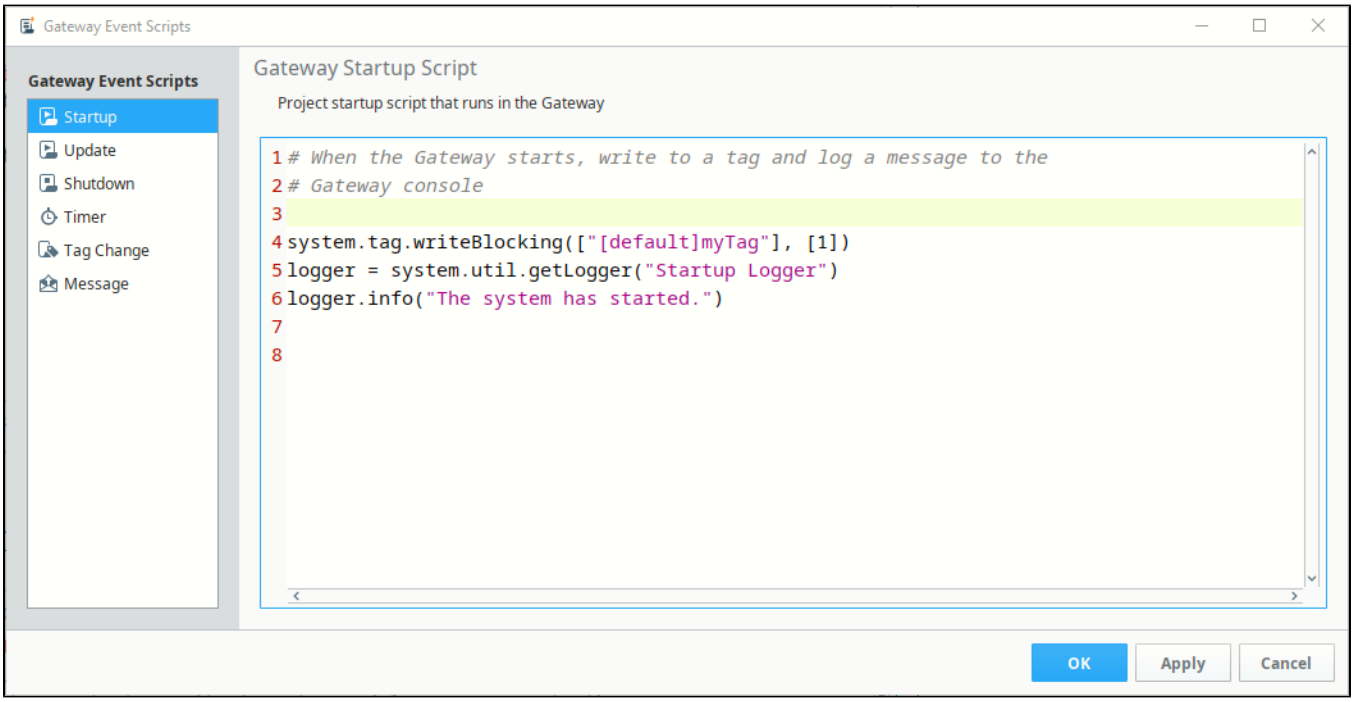
Both Perspective and Vision offer component based scripting triggers, providing a means to execute a script under a number of different situations, such as a user interacting with a component or a component property value changing. For more information on how both module handle component based scripts, take a look at the Scripting in Perspective and Scripting in Vision sections.



## Client, Gateway, and Session Event Scripts

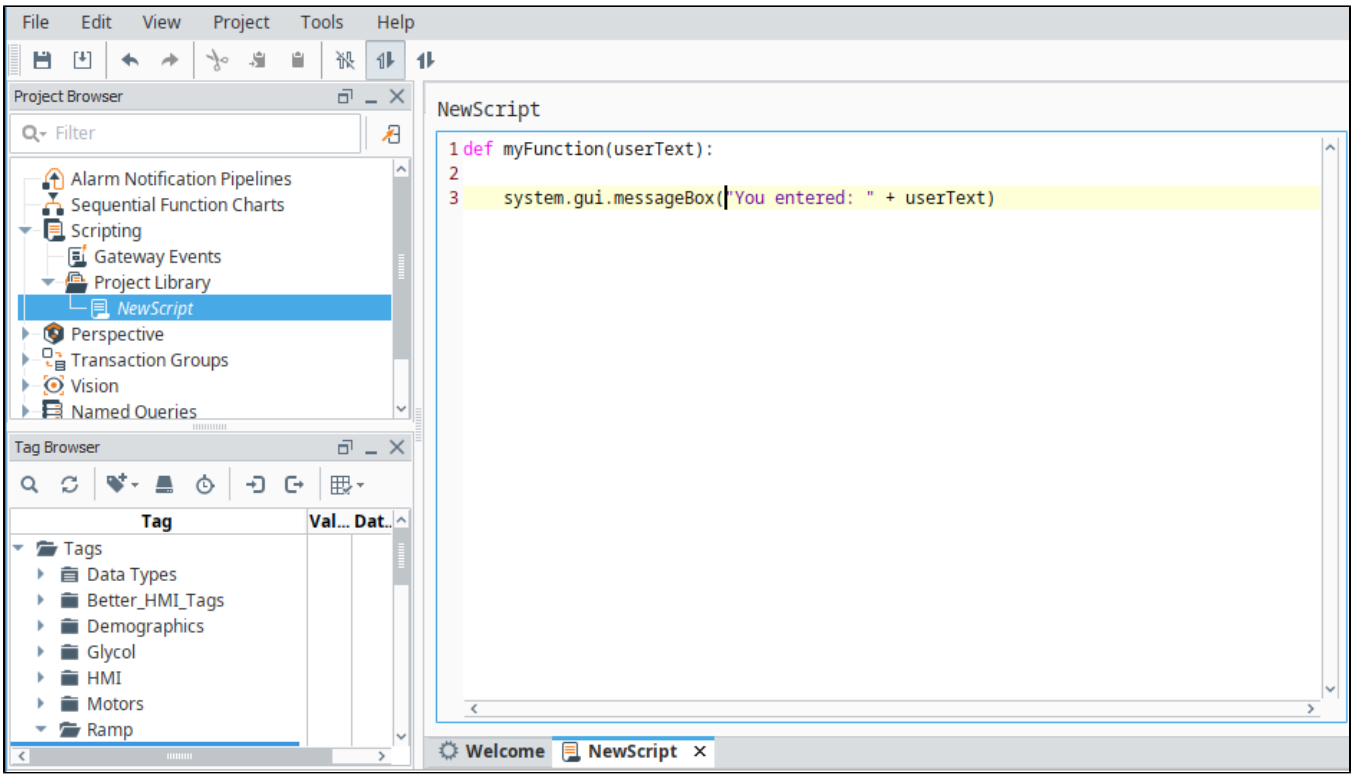
Scripts can be set to activate on specific events that occur during runtime. For example, you can triggering a script to run when a vision client starts, or on certain time intervals.

More information on these events can be found on the Client Event Scripts, Gateway Event Scripts, and Perspective Session Event Scripts pages.



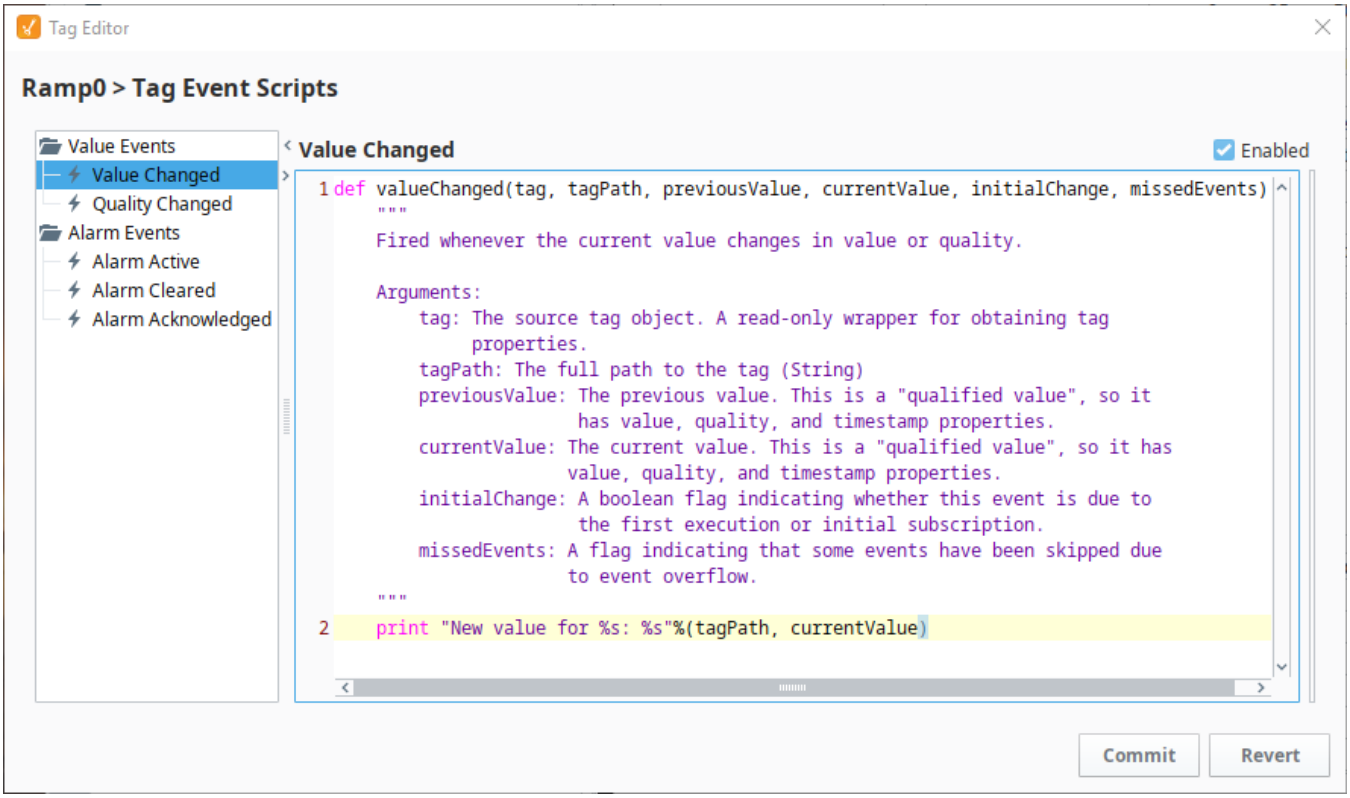

## Project Scripts

You can create your own reusable blocks of code in the Project Library. Once configured, these functions can be called from anywhere in a project, just like our system.* functions.



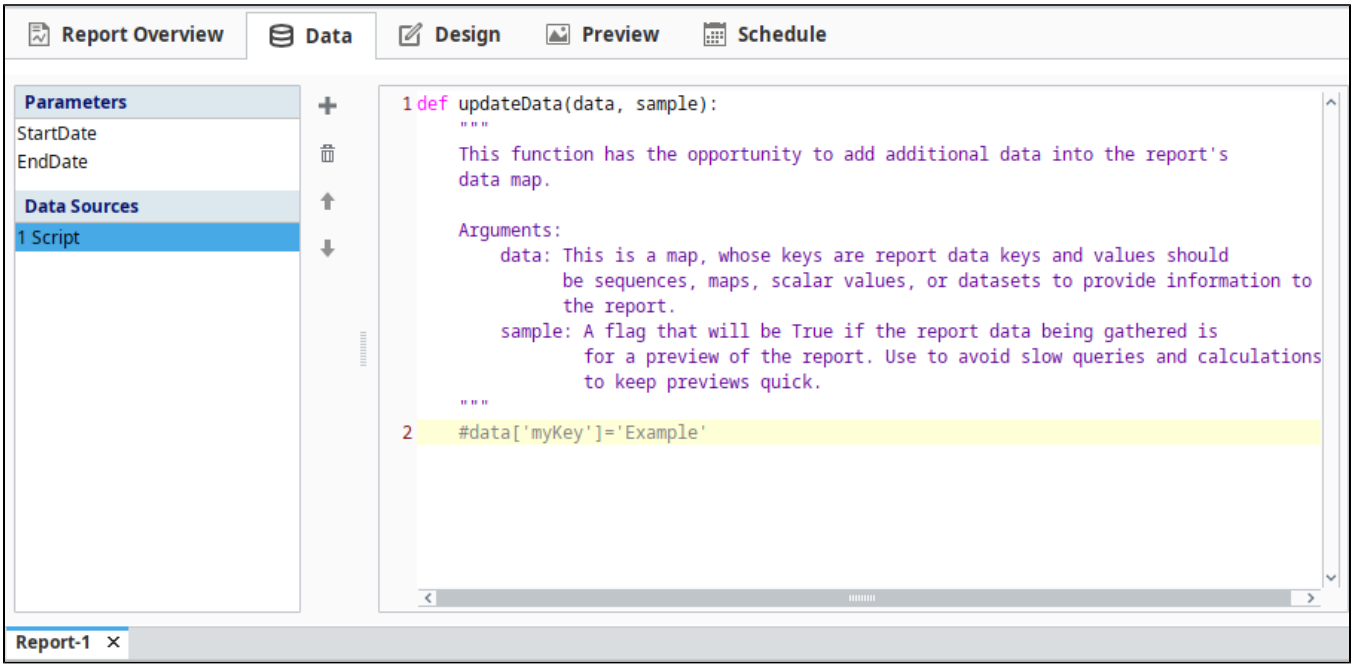

## Tag Scripts

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.



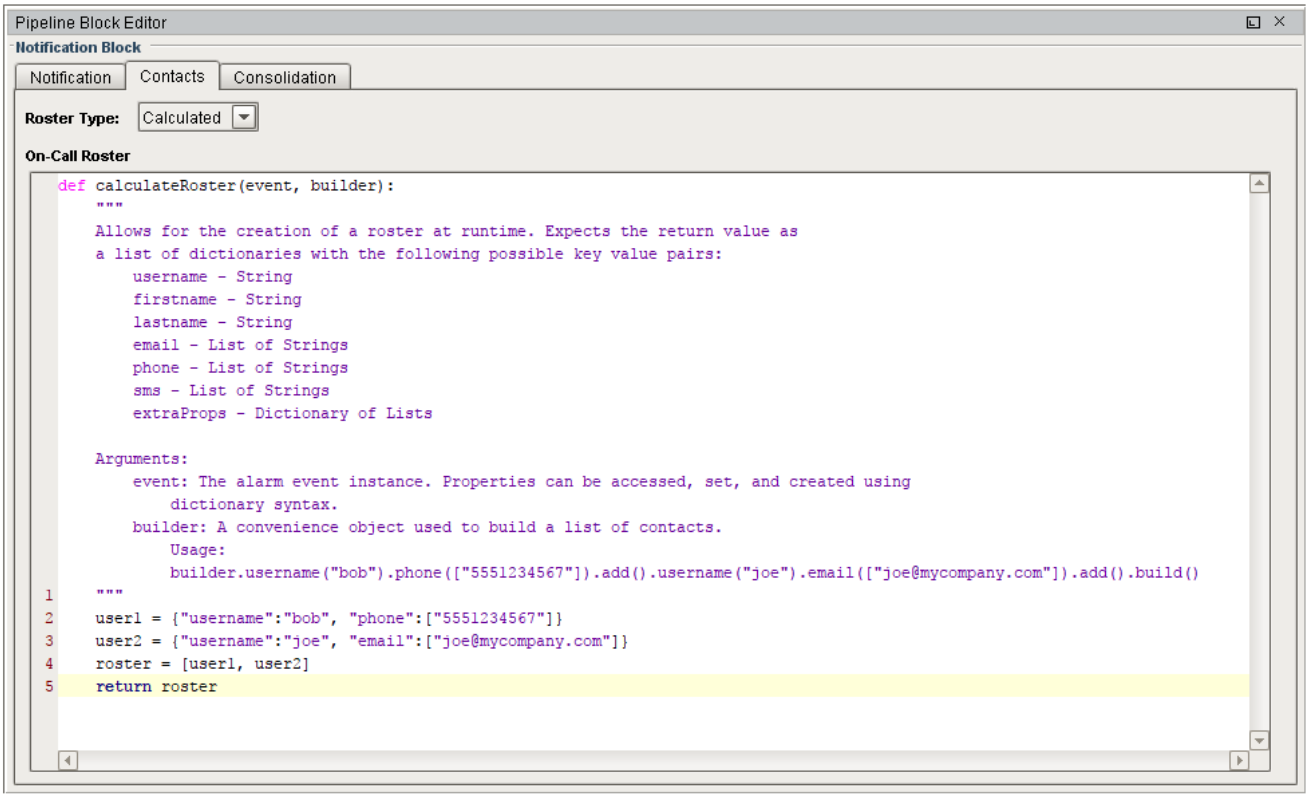
## Reporting

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.



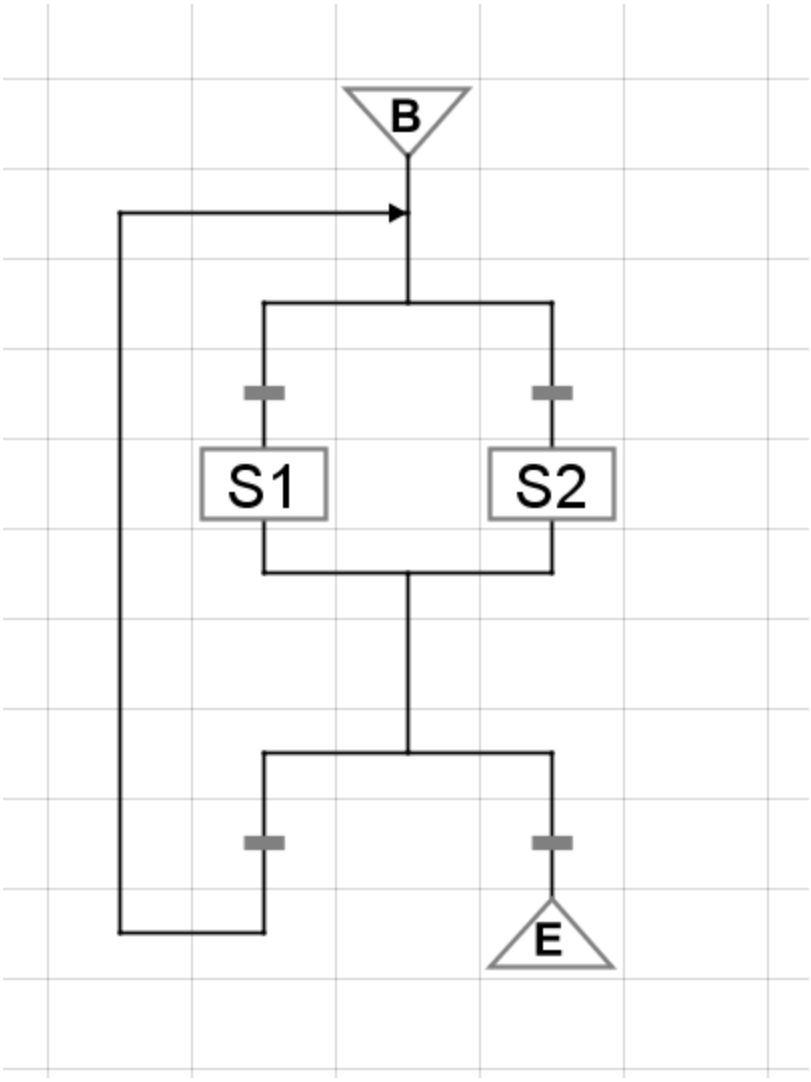

## Alarming

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.




Related Topics ...

- Scripting Data Source
- Sequential Function Charts
- Notification Block
- Tag Event Scripts
- Client Event Scripts
- Gateway Event Scripts
- Perspective Component Methods
- Perspective Session Event Scripts
- Project Library

In This Section ...

# Getting Started with Scripting in Ignition

## 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 flexibile 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.

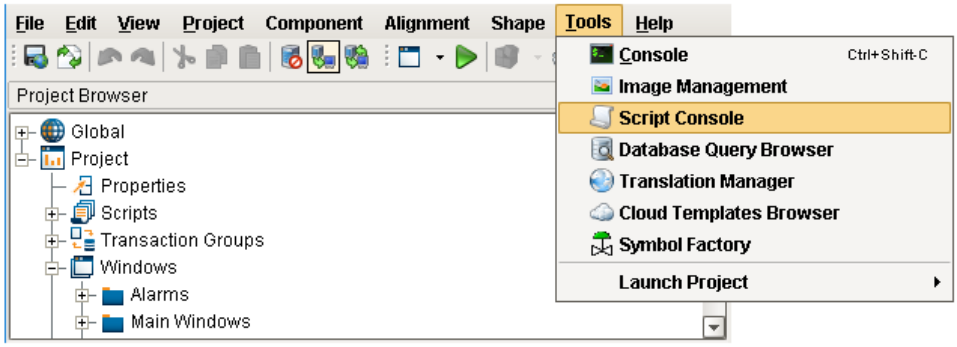
### On this page ...



## 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.**

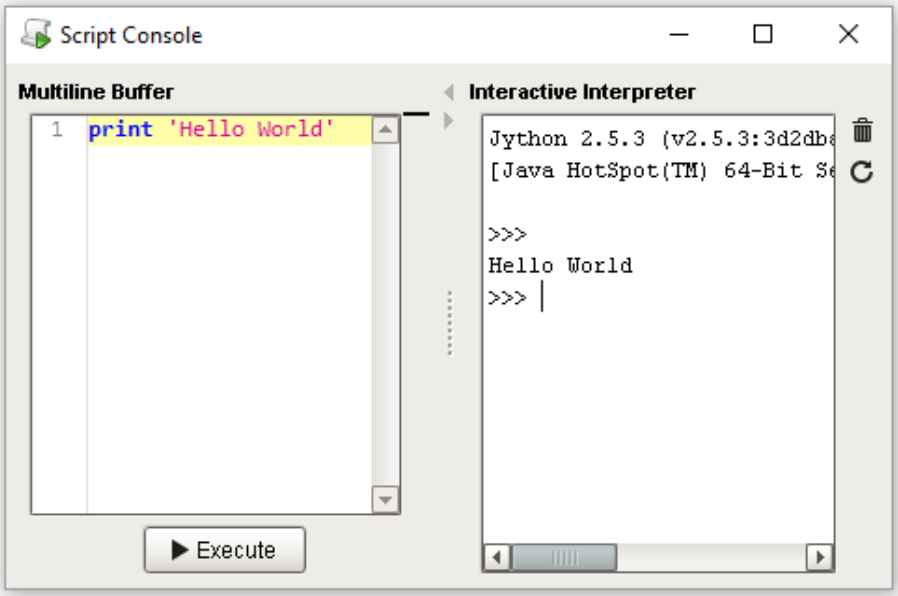



2. The Script Console will appear. Type the following code, or simply copy and paste it into the text area in the Multiline Buffer on the left side of the Script Console:

**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.

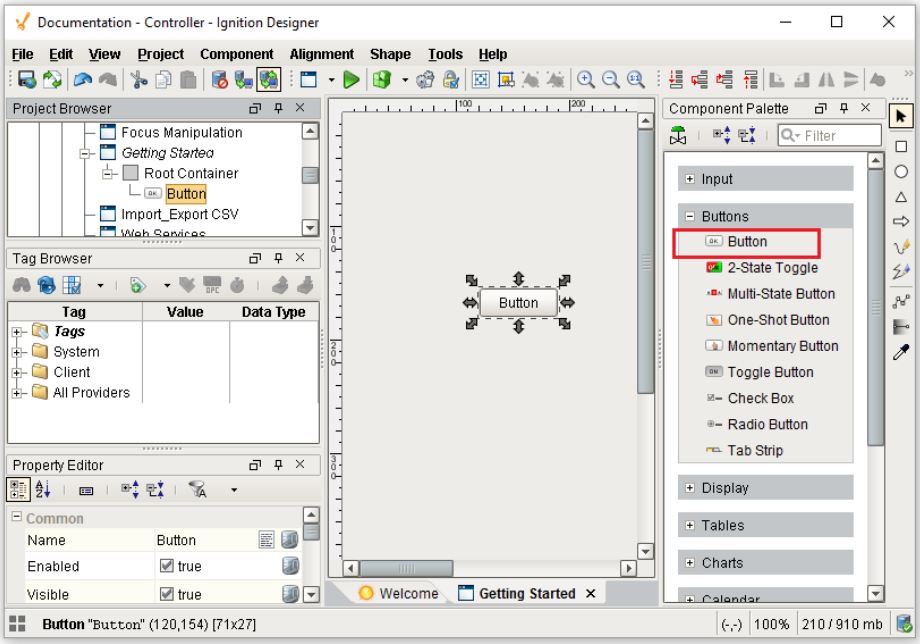



## Example - "Hello World" on a Button Component

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.

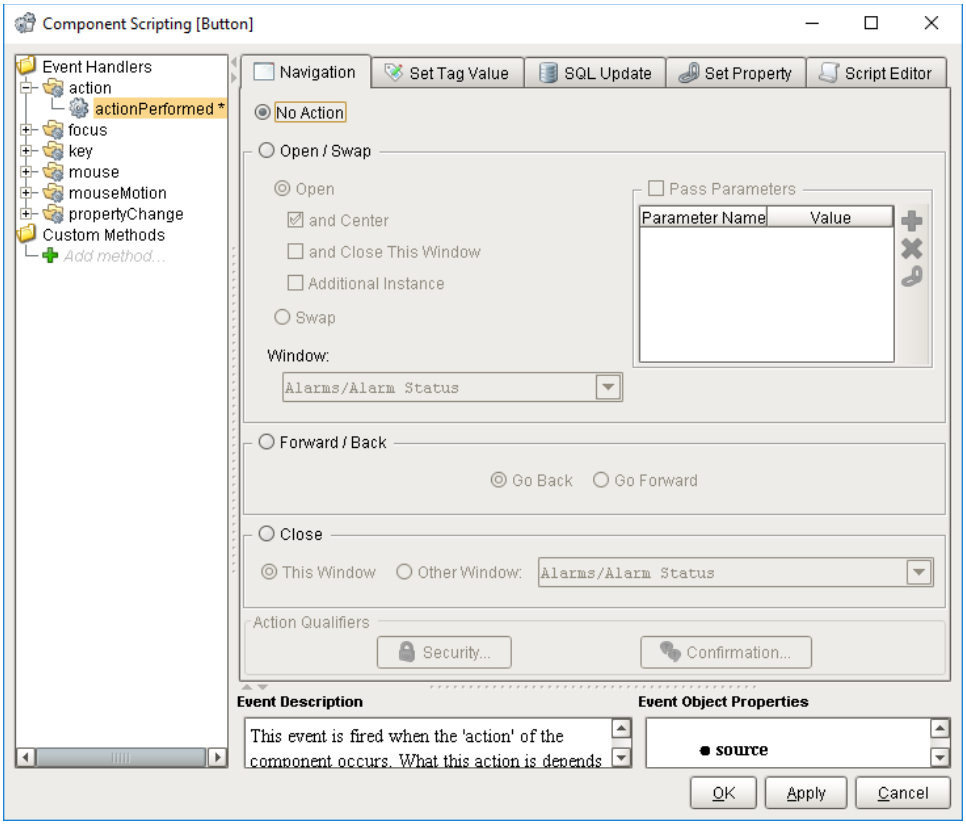
1. In the **Designer**, create a new Main Window. Give it a meaningful name if you like. (We won't need to reference the name of the window in any of these examples).

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.

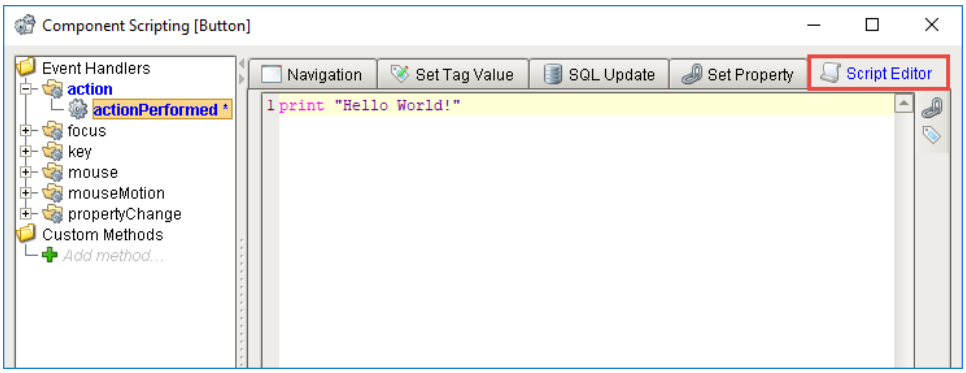
4. Under Event Handlers, select the **action > actionPerformed** option, and click the **Script Editor** tab. Make sure the **actionPerformed** event is highlighted. If an event is not selected, all of the remaining features in this window will be disabled, so we will not be able to write a script. Additionally, 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!"
```



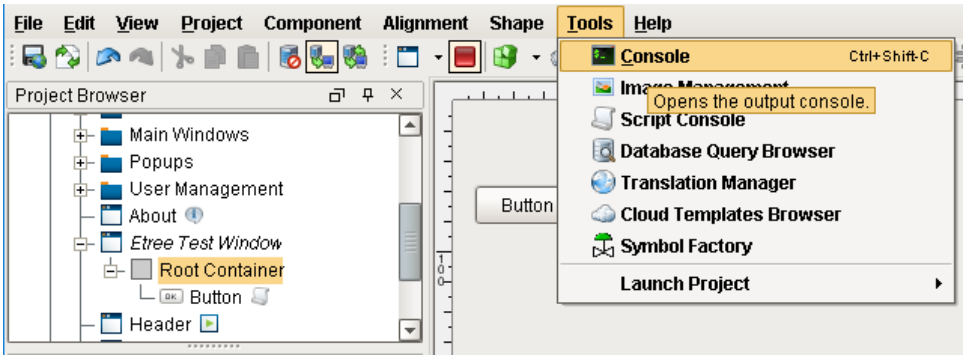

Notice the **actionPerformed** event is blue and bold. This means there is a script on this event. This is useful to know in situations where a component has scripts on multiple events.

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.

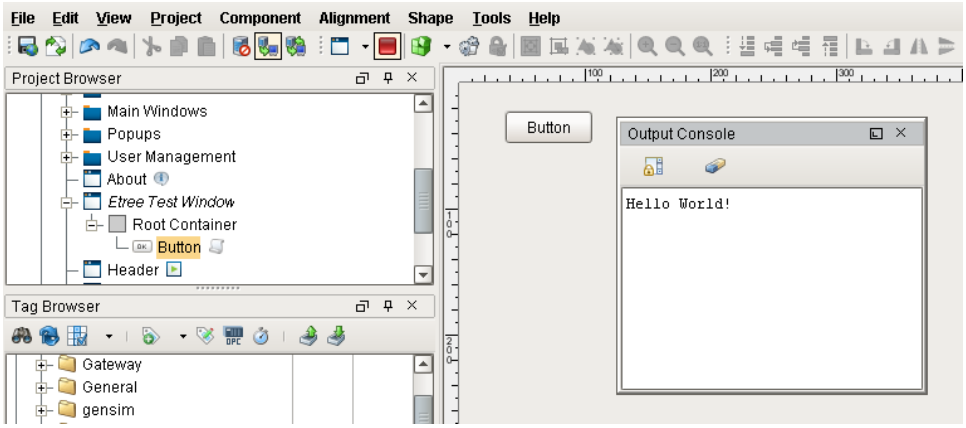
The **Script Editor** tab also has a blue color, denoting where the script is. An event will only ever have a script located on a single tab at any time. If a new tab is selected and configured, it will wipe out the work on the prior tab, for example, writing a script on **Script Editor** and then

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

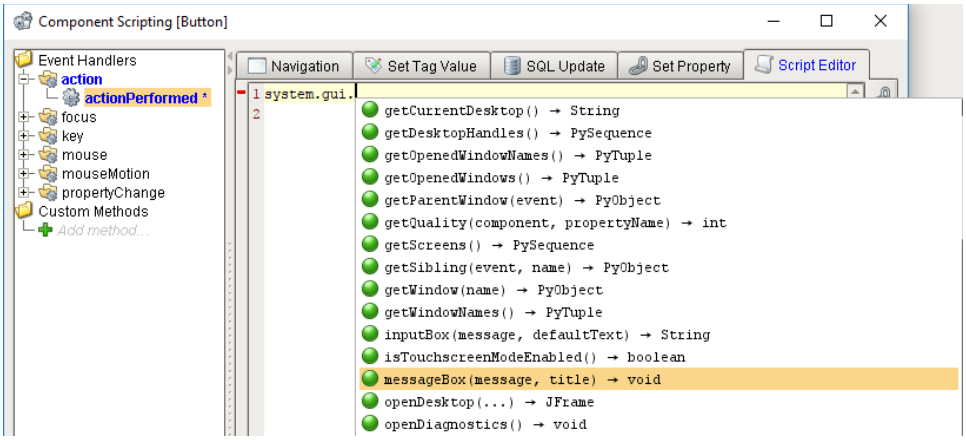
Now that we've seen how to print to the Output Console, lets make our message appear when we call it. This time, we will modify the script on the Button component to bring up a window that the user will see.

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:

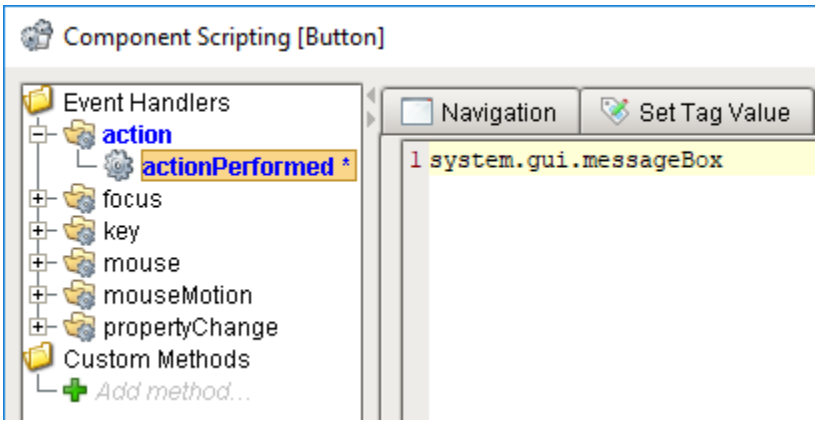
**Pseudocode - Built-in Functions Begin with `system.`**

```
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.

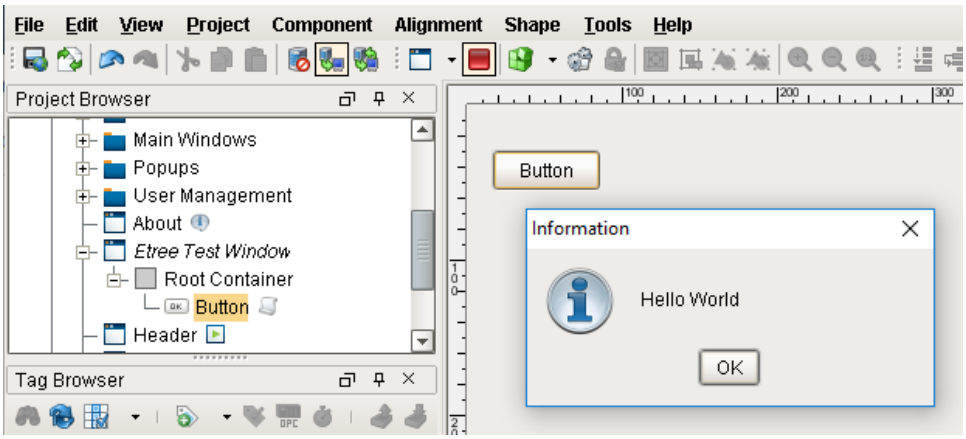



5. Complete the code by placing some parentheses and a message as a string. Alternatively, you can copy the example below.

**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.




Related Topics ...

- Python Scripting
- Scripting Functions
- Component Events

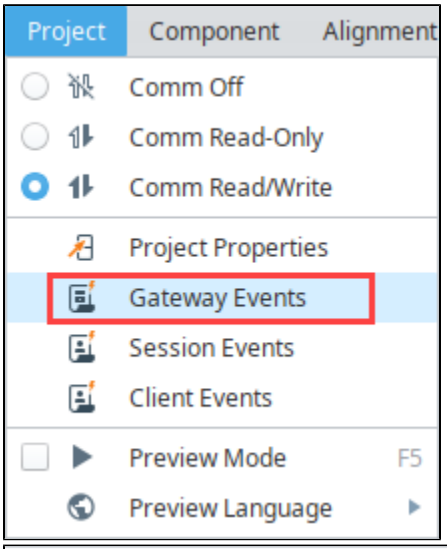
# Gateway Event Scripts

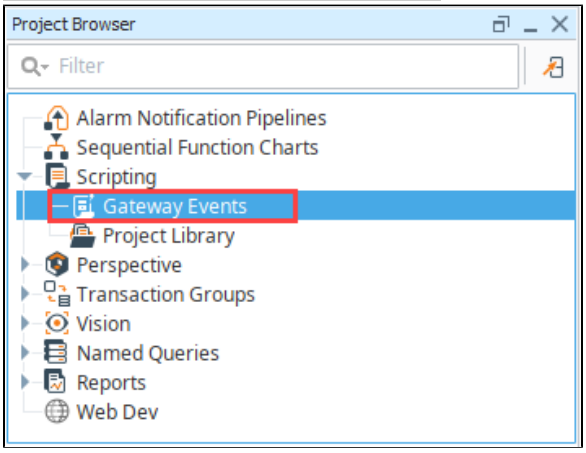
## Gateway Event Scripts Overview

Gateway Event Scripts are scripts that run directly on the Gateway. They are useful because they always run, regardless if any sessions or clients are open. They also provide a guaranteed way to make sure only a single execution of a particular script occurs at a time, as opposed to placing a script in a window, as there could be multiple instances of the window open at a given point of time.

Note that even though Gateway Event Scripts run on the Gateway, they're still considered a project resource. Project backups will include any Gateway Event Scripts.

The Gateway Event scripting workspace is located in the Scripting menu of the Designer or in the Project Browser under **Scripting > Gateway Events**.




### Other Event Scripts

The content on this page will focus primarily on Gateway Event Scripts. However, there is some overlap with Client Event Scripts, as they have similar events. More information can be found on the Client Event Scripts and Perspective Session Event Scripts pages.

**Note:** System functions are available for both Client Event Scripts 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're 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.

## On this page ...

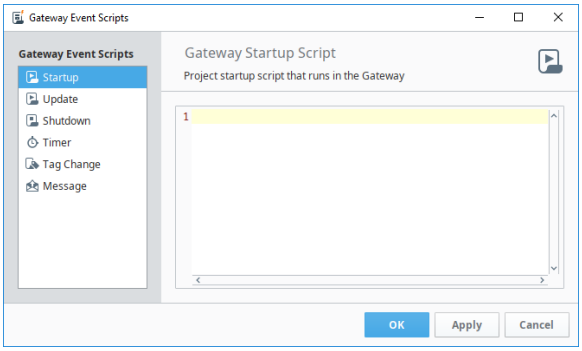



**Gateway vs Client Event Scripts**

Watch the Video

## Startup Script

The Startup Script event runs at the startup of the Gateway. Additionally, if the project is restarted in someway, such as by making a change to a Gateway Event Script and saving, then the Startup Script will be called. This means that while editing scripts frequently in the Designer, the startup and shutdown events may happen frequently.




INDUCTIVE UNIVERSITY

**Startup Scripts**

Watch the Video

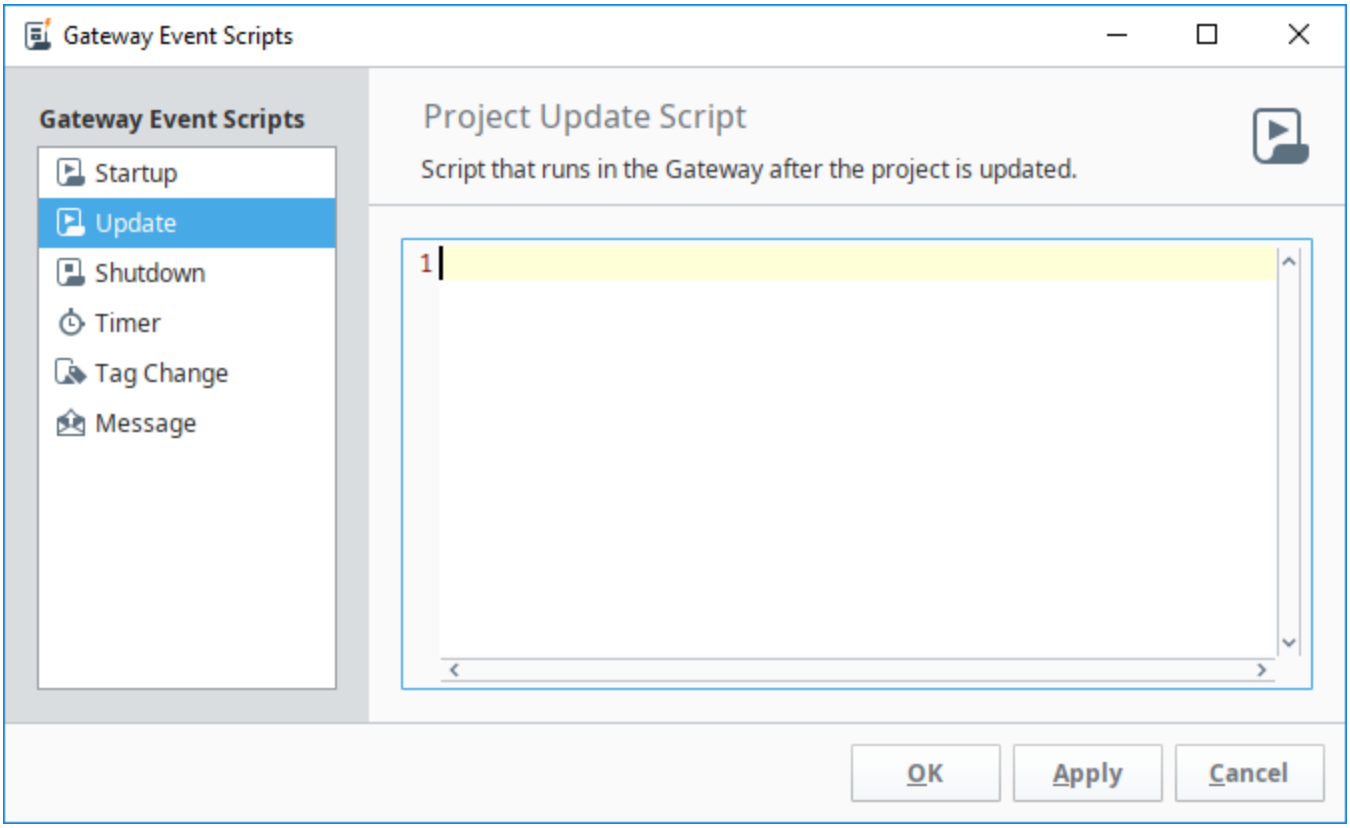
### Gateway Startup Behavior

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.

## Update Script

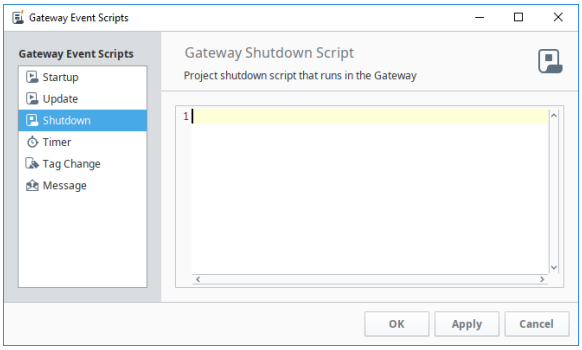
The Update Script event runs after a project is saved or updated on the Gateway. This enables you to insert a script that will run every time a project is saved.




## Shutdown Script

The Shutdown Script event runs at the shutdown of the project, which means it can be used as a way to trigger a script when the Gateway has to be restarted. It allows you to run a piece of code as the shutdown is occurring. After the script completes, the shutdown will finish.

Note that the Shutdown Script event only gets called if the Gateway is requested to shut down: if the computer power is lost abruptly (power outage, hard restart, etc.) this shutdown script will not run.




INDUCTIVE UNIVERSITY

**Shutdown Scripts**

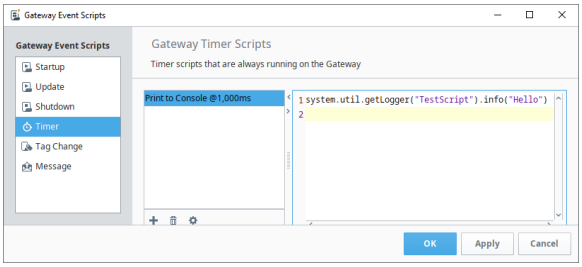
Watch the Video

### Gateway Shutdown Behavior

Similar to how a Startup Script behaves, "Shutdown" in the context of this event means "project shutdown", so shutting down the Gateway would trigger this event, as well as disabling the project containing a script on this event. Additionally, making a change to a Gateway Event Script in the Designer, and then saving the project will cause the project to restart, which means this event can get called by simply making changes in the Designer and saving.

## Timer Scripts

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 on the Gateway. This is the ideal event to use if you need the Gateway to periodically perform some scripting task.




INDUCTIVE UNIVERSITY

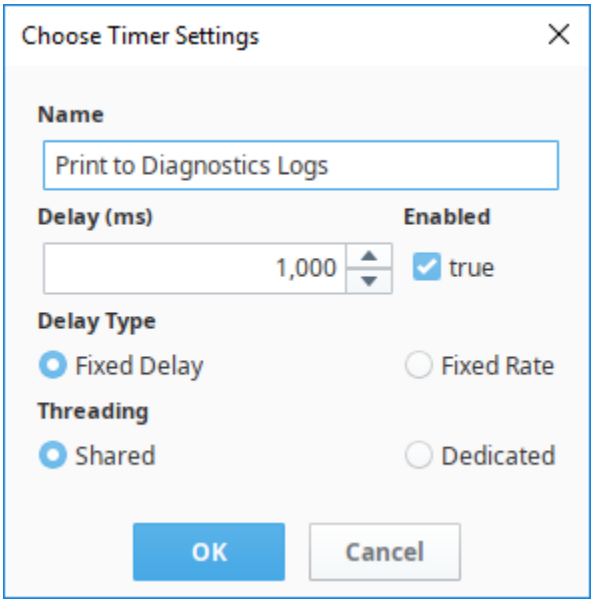
**Timer Scripts**

Watch the Video

Since multiple Timer Scripts can be added, there are separate buttons that allow you to manage each Timer Script.

- **Add Timer Script** - Adds a new Timer Script.
- **Remove Timer Script** - Will delete the selected Timer Script.
- **Modify Timer Script** - Will modify the settings for the selected Timer Script.

## Timer Script Settings




Below is an overview of the settings for a Timer Script.

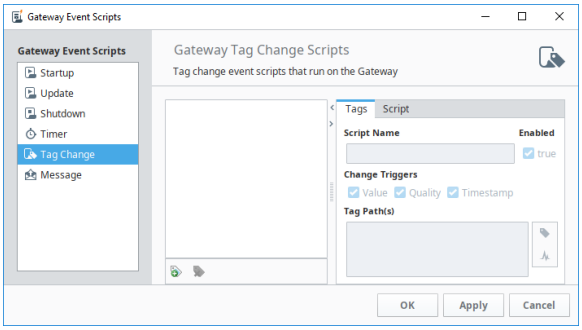
- **Name**: The name of the Timer script. Names must be unique per project, so two timer scripts in the same project cannot have the same name.
- **Delay**: The delay period in milliseconds. The meaning of this setting is dependent on the **Delay Type** setting.
- **Enabled**: Allows you disable the Timer Script when set to false.
- **Delay Type**: Determines how the **Delay** setting is utilized.
  - 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.
- **Threading**: Determines which thread this script should run in. In other words, this setting allows you to specify if you want this timer script to share execution resources or not. 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.
  - The **Shared** setting means that all timer scripts will share a thread. This is usually desirable, as it prevents creating lots of unnecessary threads: threads have some overhead, so a small amount of resources are used per thread. However, if your script takes a long time to run, it will block other timer tasks on the shared thread.
  - The **Dedicated** setting will create a separate thread specifically for the timer script to use. This setting is desirable when your scripts executions must be as consistent as possible, as other timer scripts can't slowdown or otherwise impact the execution of a script in a separate thread.

# Tag Change Scripts

The Tag Change Script event allows you to specify any number of Tags, and trigger a script when one of them change. Since these execute based on a Tag changing value, Tag Change Scripts are ideal when you need a script to run based on some some signal from a PLC.

Having the Tag Change Scripts run in the Gateway Scope means that the scripts are active as long as the Gateway is running. Thus, you do not need a client or session to be open for a Gateway Tag Change Script to execute. When executing, each Tag Change Script runs in a separate thread. This prevents long running scripts from blocking the execution of other Tag Change Scripts.

**Note:** Due to their nature, Client Tags will not trigger Gateway Tag Change Scripts. However, there are other similar events, such as a Client Event Script, that enable you to trigger a script based on a Client Tag changing value.




## Tag Change Script Interface

### Tag List

Lists all of the available Tag Change scripts in the project. Two icons are available below the list:

- **Add Script** - Adds a new Tag Change Script to the list.
- **Remove Script** - Removes the currently selected script from the list.

### Tags Tab

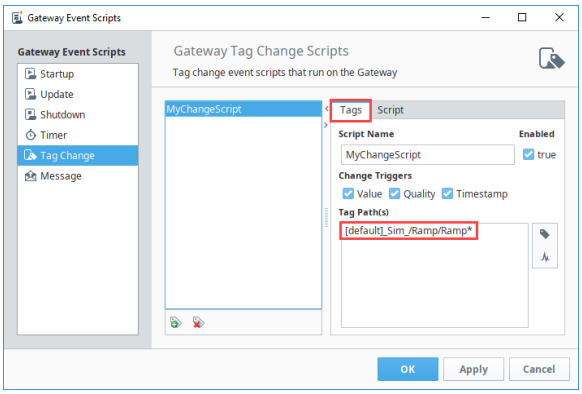
The **Tags** tab contains settings for the script. See the **Script Settings** description below

- **Script Name** - The name of the script. Script names must be unique per project.
- **Enabled** - Determines if the script is active or not. Set to false to disable the script.
- **Change Triggers** - When the Tag changes, the script can trigger based on the **Value**, **Quality**, and/or **Timestamp**. Note that regardless of how many triggers changed, the script only executes once per tag, so leaving all triggers enabled will not trigger three executions each time the Tag changes.
- **Tag Paths** - A list of Tag paths to monitor. When any of the Tags listed in this area change, the script will trigger. Note that the list is not comma separated: new paths are specified each line.
    - **Tag Browser** - Opens a Tag Browser window, allowing you to quickly lookup and add Tag paths to the Tag Change Script.
    - **Path Diagnostic** - Click this icon to verify the paths specified under the **Tag Paths** list. This is useful for checking for typos in the list.
    - Wildcards can be used at the end of configured tag paths. For example, [provider]folder/*.

INDUCTIVE UNIVERSITY

**Tag Change Scripts**

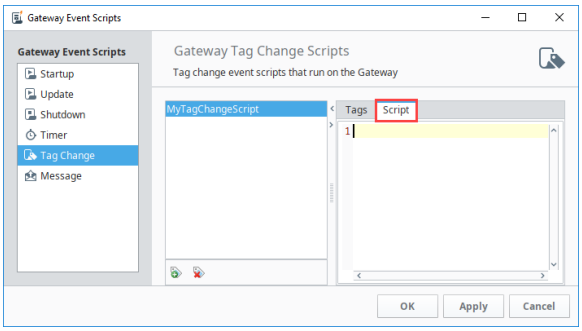
Watch the Video




## Script Tab

The **Script** tab is where the Python script associated with this event will be placed. Two icons are available below the list:

- **Add Script** - Adds a new Tag Change Script to the list.
- **Remove Script** - Removes the currently selected script from the list.




## Tag Change Objects

Tag Change Scripts contain several built-in objects that are useful for inspecting the event, such as seeing what value the Tag changed to. These objects are listed below:

The initialChange Value

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 inital subscription, preventing a script from running when the values haven't actually changed.

```
if not initialChange:

    # Do something useful here
```

The newValue Object

A QualifiedValue object that contains the following methods

| Method | Description | Usage Example |
|---|---|---|
| `getValue()` | Returns the new value on the tag | `newValue.getValue()` |
| `getQuality()` | Returns the new quality on the tag | `newValue.getQuality()` |
| `getTimestamp()` | Returns the new timestamp value | `newValue.getTimestamp()` |

The event Object

This object offers some additional utility, such as accessing the previous values on the tag.

| Method | Description | Usage Example |
|---|---|---|
| `getCurrentValue()` | Returns a QualifiedValue object (similar to the newValue object), representing the new values on the tag. | `event.getCurrentValue().getValue()` |
| `getPreviousValue()` | Returns a QualifiedValue object, representing the values on the tag before the change. | `event.getPreviousValue().getValue()` |
| `getTagPath()` | Returns a TagPath object, that can be further examened for details about the path on the tag that changed. See the TagPath Object table below. Additionally, the TagPath object can easily be turned into a string, providing quick access to the path of the tag that changed value. | `event.getTagPath()` |
| `getValue()` | Returns the new value on the tag, similar to getCurrentValue(). This method is a convenient way to retrieve just the new value, without needing to interact with the QualifiedValue object returned by getCurrentValue(). | `event.getValue()` |

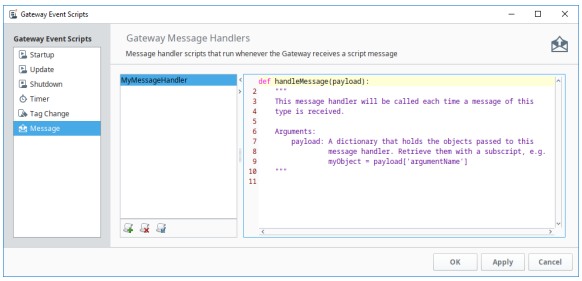
The TagPath Object

These methods are available on the TagPath object returned by `event.getTagPath()` calls

| Method | Description | Usage Example |
|---|---|---|
| `getItemName()` | Returns the name of the item at the end of the path, which can be used to get the name of the tag that changed. | `event.getTagPath().getItemName()` |
| `getParentPath()` | Returns the path to the Tag's parent folder. | `event.getTagPath().getParentPath()` |

## Message Scripts

Message Handlers allow you to write a script that will run in the scope 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.



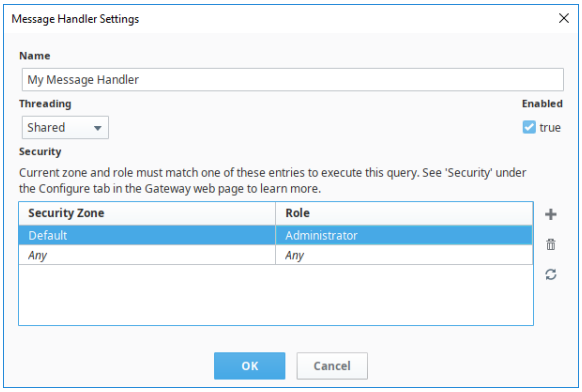



Under the list of handlers, three small buttons allow you to add, remove and manage your handlers.

- **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.

## Gateway Message Handler Settings

When adding or modifying a message handler, a Message Handler settings window will popup.




The following settings are available:

- **Name** - The name of the message handler. Each message handler must have a unique name per project.
- **Threading** - Determines the threading for the message handler. Contains the following options:
  - **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. Threads have a bit of overhead, so this option uses more of the Gateway's resources, but is desirable if you want the message handler to not be impeded by the execution of other message handlers.
- **Security** - Allows you to specify security zone and role combinations that are allow to request this message handler.

## The Payload

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. In essence, the payload is the mechanism that allows you to pass the message handler values.

The payload is simply a python dictionary, so extracting values involves specifying the key:

**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.
```

# Calling 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 Scripts

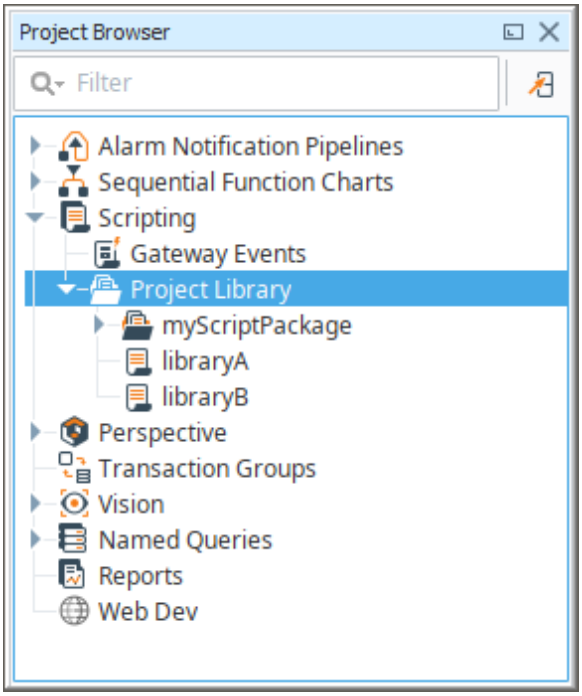
While they are technically project resources, remember that Gateway Event Scripts technically run on the Gateway. Thus the Status section of the Gateway is useful for diagnosing issues with Gateway Event Scripts.

Related Topics ...

- Client Event Scripts
- Perspective Session Event Scripts

# Project Library

Scripts under the Project Library are a project-based resource that allows user created Python scripts to be configured. Objects and functions created in a Project Library script can be called from anywhere in the project. Project Library scripts are accessible from the Project Browser, under the **Scripting** item.

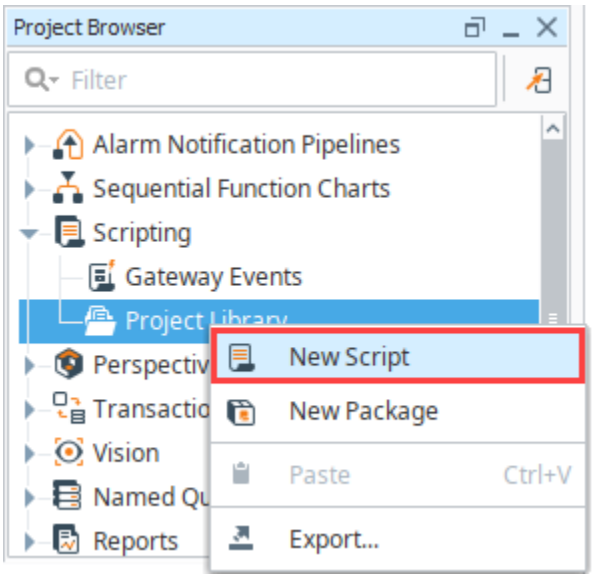



Additionally, a single project can be designated as the **Gateway Scripting Project**, meaning that scripts defined in the stated project can be called from the Gateway scope.



## Add a Script

To add a a Project Library script, right click the **Project Library** item and click the **New Script** option.




### Scripts and Packages

There are two main types of resources under the Project Library.

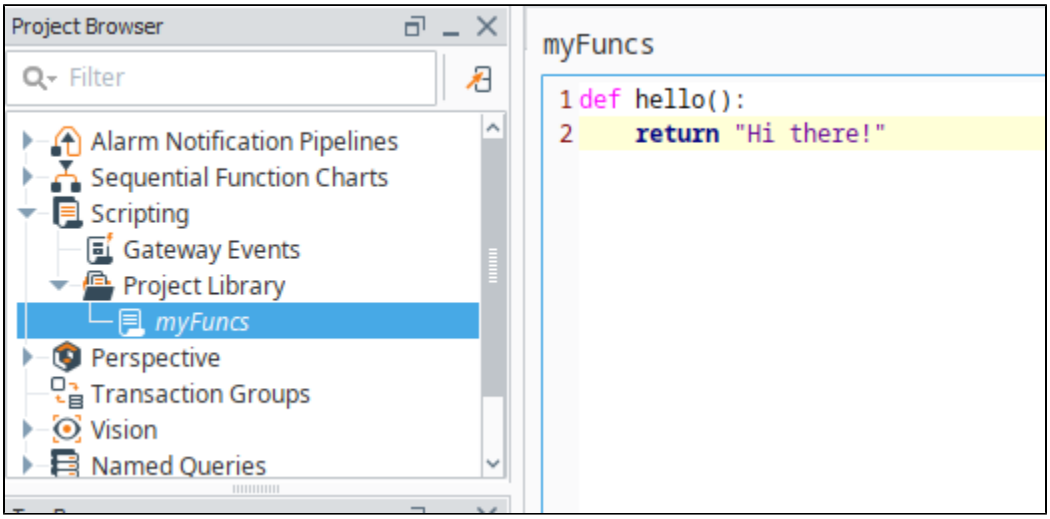
- **Scripts** - Each script resource can contain many functions and objects.
- **Packages** - Each package effectively acts as a folder, allowing you to better organize each script resource.

## Usage Example

For example, let's suppose you added the following script module named **myFuncs**, whose body is shown below.

**Python**

```
def hello():
        return "Hi there!"
```



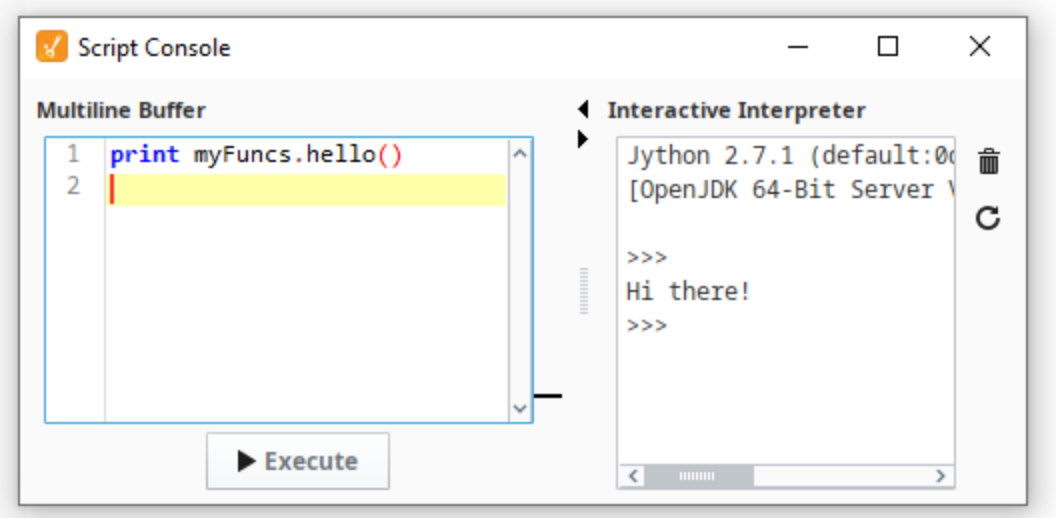

Once we **save** our project, we can now call this function from anywhere within the project using the following syntax

**Python - Calling the Project Script**

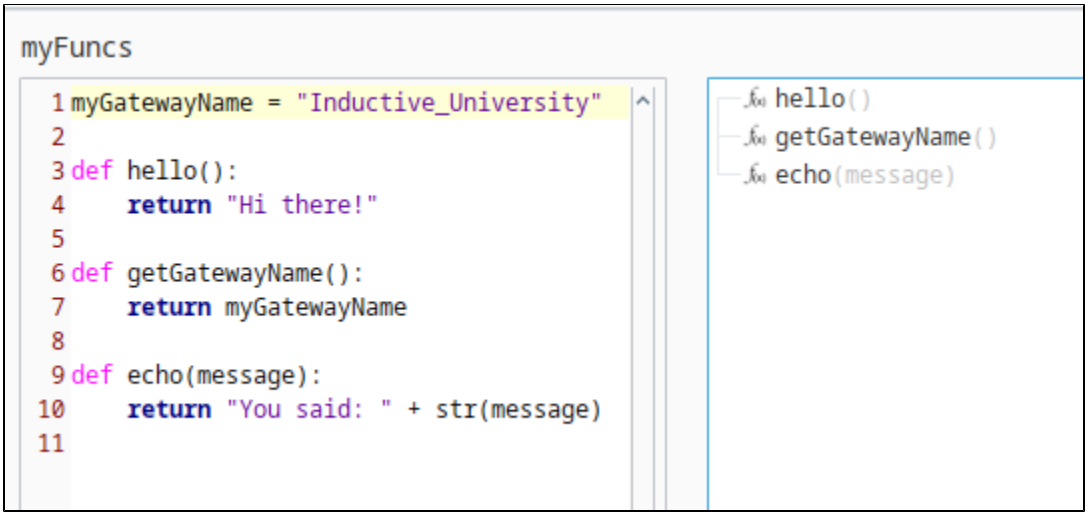
```
myFuncs.hello()
```

**Note:** Project Library scripts are not accessible to the other resources until the project is saved.

For example, we could open the Script Console (Tools menu > Script Console), write the following, and execute the script.



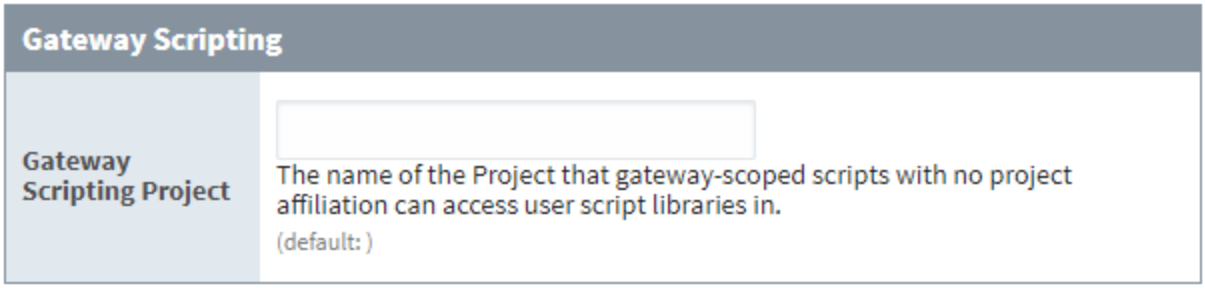

Each script resource can contain multiple functions and objects. As you add new function definitions, the list on the right will populate giving you a quick way to navigate through long scripts.



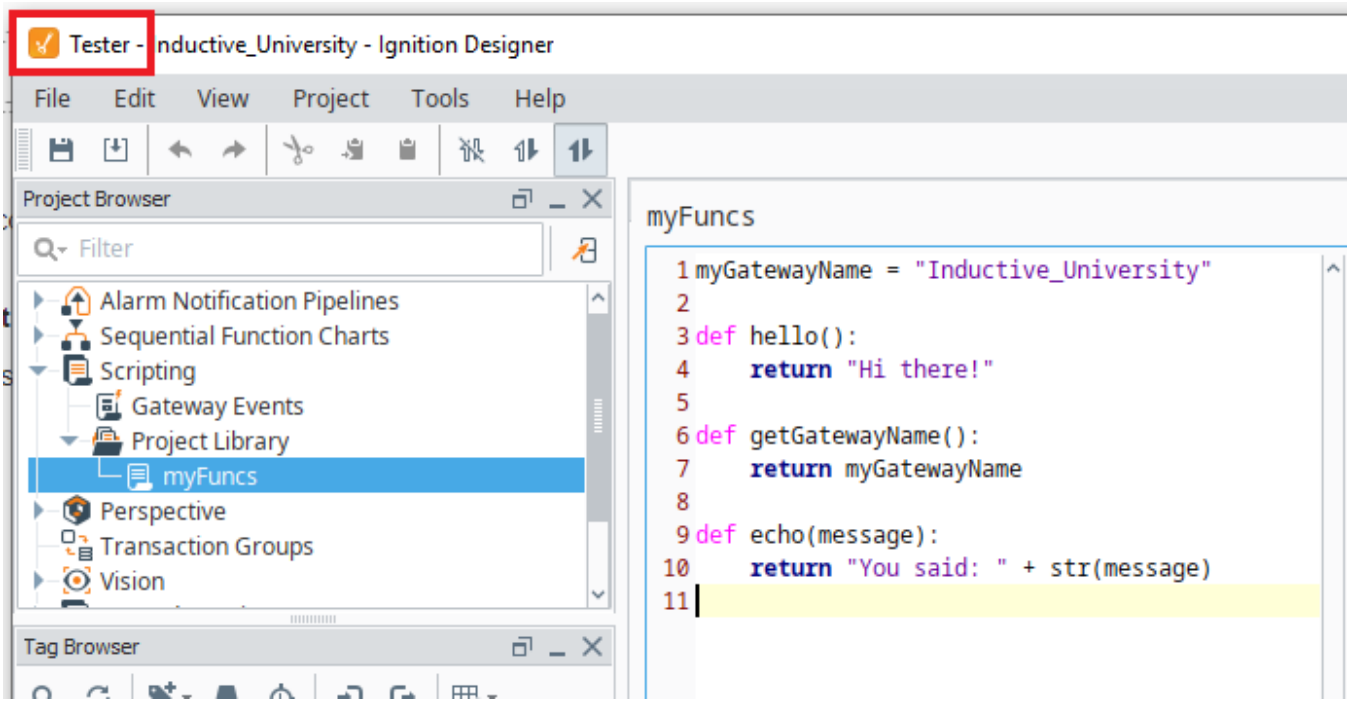

## Gateway Scripting Project

Project Library scripts are normally only accessible from the project they were defined in. Thus objects that exist in other scopes, such as Tags that exist in the Gateway scope, are unable to call Project Library scripts. Attempting to do so will result in Gateway log errors stating that "global name 'yourScript' is not defined'.

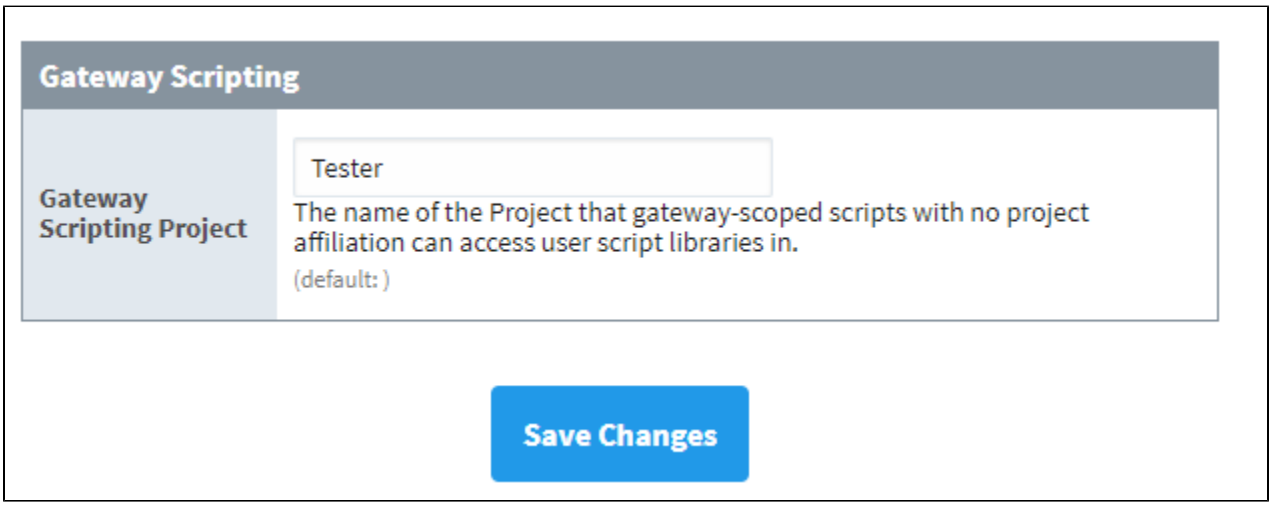
The exception to this rule is the Gateway Scripting Project. This project is specified by the **Gateway Scripting Project** property, which is set in the Con fig section of the Gateway Webpage under Gateway Settings. Entering in the name of a project under this property allows the Gateway access to Project Library scripts configured in the specified project.




Thus, if the **myFuncs** library in the prior section was configured in a project named "Tester"




We could use that name in the Gateway Scripting Project.




After we save, Tags and other Gateway-scoped resources can then start calling any of the scripts from our Project Library in the "Tester" project.

Related Topics ...

- User Defined Functions
- Scripting in Ignition

# 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.

### On this page ...



### 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.


Related Topics ...

- WebDev Module

In This Section ...

# 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%20woeid%20from%
20geo.places(1)%20where%20text%3D%22maui%2C%20hi%22)&format=json&env=store%
3A%2F%2Fdatatables.org%2Falltableswithkeys
```

## On this page ...



## 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%20text%3D"
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 ouf 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%20text%3D"
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 Dictionary.
decodedDict = system.util.jsonDecode(results)

# Now we can treat the results like a nested dictionary, thus we can specify the "query" 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 bay 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**

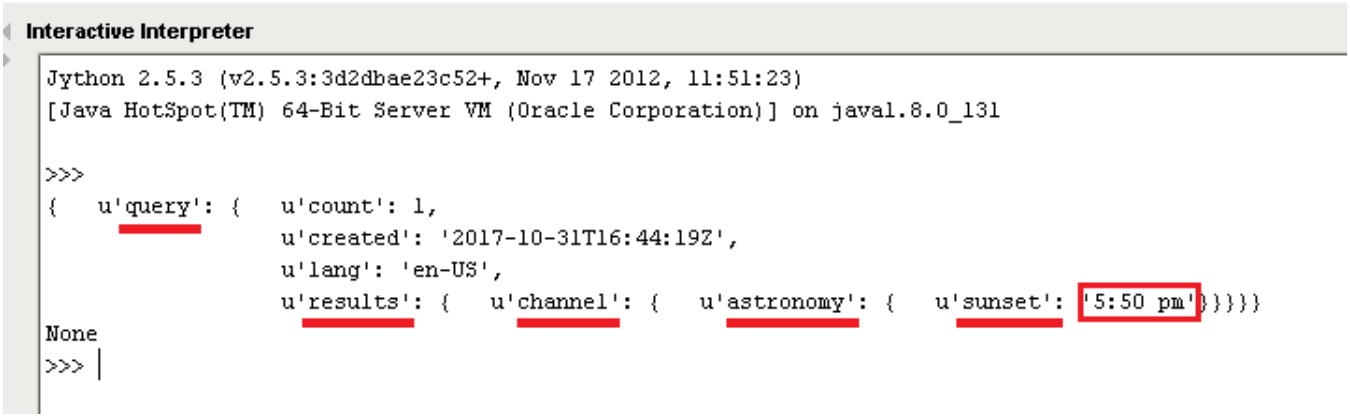
```
# Import the pprint library
import pprint

# 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%20text%3D"
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 Dictionary.
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'}}}}}
None
```

From here we can see all of the keys that lead to our final value:




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 a GET call with a bad
/incomplete address.
# myEndpoint += "forecast%20where%20woeid%20in%20(select%20woeid%20from%20geo.places(1)%20where%20text%3D"
# 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 | Description |
|---|---|

| Code | |
|---|---|
| 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). |

Related Topics ...

- system.net

# 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.

**On this page ...**



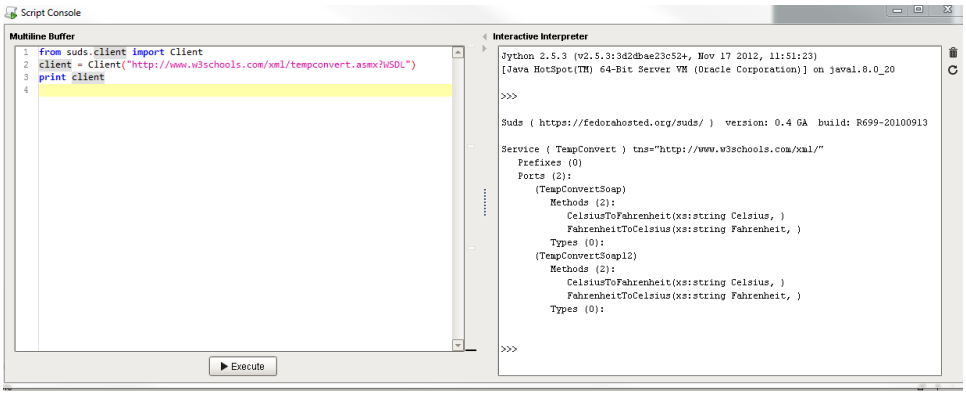
## Simple Example

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.

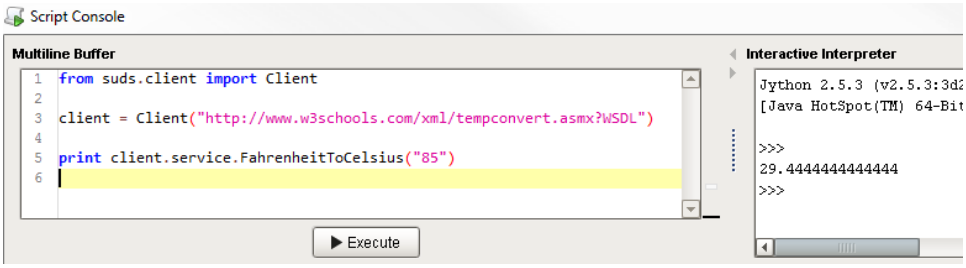
**Python - W3Schools WSDL**

```
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

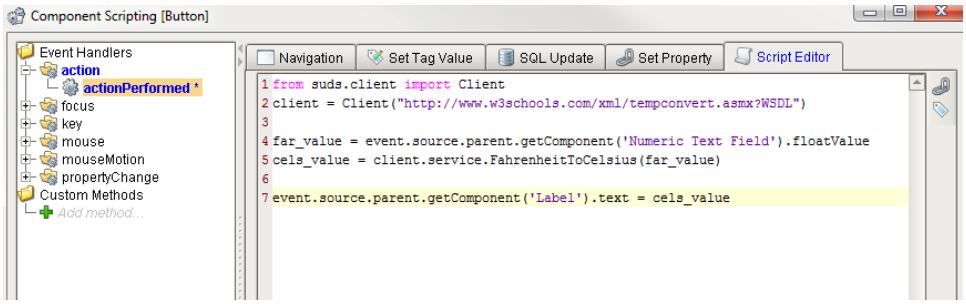
scripting, you have to make use of the "service" member of the client object. You can see printing the returned results shows the conversion.




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
```

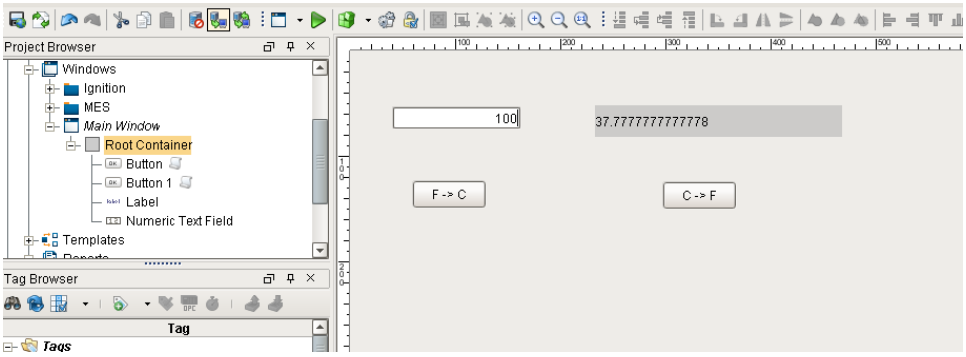



3. Then you can make a second button to do the opposite: calculate Celsius to Fahrenheit.

**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_wsdl")

#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.

**Note:** This example is completely hypothetical. It is intended to give insight into complex types. It does not make use of an 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
```

Related Topics ...

- Web Services, SUDS, and REST

# JSON Format

## About JSON

JavaScript Object Notation (JSON) is a language-independent data format. It's a lightweight format for storing and transporting data (i.e., when data is sent from a server to a web page). JSON is easy for users to read and write and for machines to parse and generate.

## JSON Rules

JSON has a few simple rules:

- Data is in name/value pairs.
- Data is separated by commas.
- Curly braces hold objects.
- Square brackets hold arrays.

### On this page ...



## How JSON Works

### JSON Data - Name and a Value

JSON data is written as name/value pairs. A name/value pair consists of a field name in double quotes, followed by a colon, followed by a value.

```
{
        "companyName":"Inductive Automation"
}
```

### JSON Objects

JSON objects are written inside curly braces.

**Note:** The properties of a JSON object have no defined order. If you need a defined order, use a dataset or an array instead.

```
{
        "firstName":"Sally",
        "lastName":"Smith"
}
```

### JSON Arrays

JSON Arrays are written inside square brackets. An array can contain objects. In the following example, the object "companies" is an array and contains three objects.

```
{
   "companies":[
      {
         "companyName":"Inductive Automation",
         "cityName":"Folsom",
         "stateName":"CA"
      },
      {
         "companyName":"Hewlett Packard",
         "cityName":"Palo Alto",
```

```
         "stateName":"CA"
      },
      {
         "companyName":"Apple",
         "cityName":"Cupertino",
         "stateName":"CA"
      }
   ]
}
```

## Where JSON Is Used in Ignition

Ignition uses the JSON format to store much of its data internally, including Tags and Perspective component properties.

### Perspective Component Properties

Components have properties (props), which are simply named values. These properties are arranged in a tree structure following the structure and data model of the common JSON document format. Component properties are defined as a JSON structure, and are variable according to the type of component that the config object represents. All components registered in the module have a set of default properties included. These defaults are provided to the instantiated component at runtime, and so default props are not saved. Instead, only those which have a value that differs from the default are stored, serialized and sent to the client during loading.

**Example - Sample Data from Table Component**

```
[ { "city": "Helsinki", "country": "Finland", "population": 635591 }, { "city": "Jakarta", "country":
"Indonesia", "population": 10187595 }, { "city": "Madrid", "country": "Spain", "population": 3233527 },
{ "city": "Prague", "country": "Czech Republic", "population": 1241664 }, { "city": "San Diego",
"country": "United States", "population": 1406630 }, { "city": "Tunis", "country": "Tunisia",
"population": 1056247 } ]
```

### Tags

Ignition exports and imports Tag configurations to and from JSON. Tags are defined as JSON objects, which consist of properties, arrays, and sub-objects. The system.tag.configure function can take either a String document definition, or a JSON object that defines one or more Tags. Overrides for UDTs are created by simple redefinition of properties, and complex structures like Event Scripts and Alarm configurations will be merged with inherited definitions.

You can copy the JSON or one or or more Tags in the Tag Browser. This copies them into the system clipboard. In addition, pasting the JSON into a different provider/designer will create or overwrite tags.

**Example 1 - Tag Export**

```
{
  "name": "Tank Instance",
  "typeId": "Tank UDT",
  "tagType": "UdtInstance",
  "tags": [
    {
      "value": "80",
      "name": "Tank Level",
      "tagType": "AtomicTag"
    },
    {
      "value": 80,
      "name": "sliderValue",
      "tagType": "AtomicTag"
    }
  ]
}
```

**Example 2 - Tag Export**


```
{
  "tags": [
    {
      "valueSource": "memory",
      "dataType": "Boolean",
      "alarms": [
        {
          "setpointA": 1,
          "name": "Above Normal"
        }
      ],
      "name": "Boolean Tag",
      "value": false,
      "tagType": "AtomicTag"
    },
    {
      "valueSource": "memory",
      "dataType": "Boolean",
      "name": "One Shot Trigger",
      "tagGroup": "Driven One Shot",
      "value": true,
      "tagType": "AtomicTag",
      "enabled": true
    },
    {
      "valueSource": "opc",
      "opcItemPath": "ns\u003d1;s\u003d[Generic]_Meta:Random/RandomDouble1",
      "dataType": "Float8",
      "name": "Pressure3",
      "tagGroup": "Driven One Shot",
      "tagType": "AtomicTag",
      "enabled": true,
      "opcServer": "Ignition OPC UA Server"
    },
    {
      "valueSource": "opc",
      "opcItemPath": "ns\u003d1;s\u003d[Generic]_Meta:Random/RandomDouble2",
      "dataType": "Float8",
      "name": "Thickness3",
      "tagGroup": "Driven One Shot",
      "tagType": "AtomicTag",
      "opcServer": "Ignition OPC UA Server"
    }
  ]
}
```


## JSON in Tag UDTs

You can also set JSON strings as properties in an Ignition Tag. Any properties of a Tag can be set to a string that represents a JSON object.

**Note:** In a UDT, the { } braces are used to denote a property reference. Make sure you don't have any properties with names that look like JSON objects so there is no overlap. Strings will appear as black text, parameter references will appear as grey and italicized.

## Looping through JSON Objects with Scripting

Traversing a JSON object in scripting is simple as long as the structure is known. If there are objects within objects, you can use multiple loops to get through it all.

Let's use the JSON Array object above for a simple example. You can loop through the list to get the repeating items by name.

```
# loop through the JSON data
# fetch the data, this will change depending on where the script is in relation to the table
json = self.items
# access the companies object (which is a list)
```

```
companies = json["companies"]

# loop through the companies list
for company in companies :
        # get each item out of the row object
        name = company["companyName"]
        city = company["cityName"]
        state= company["stateName"]
        # now do something with the data
```

Related Topics ...

- JSON Functions

# 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.

## On this page ...



## 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:

| Scope | Print Command | Print Output Location |
|---|---|---|
| Gateway<br><br>*Note: anything that isn't in the Client or Session scopes listed below uses this scope* | system.util.getLogger | 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:<br><br>• **Windows:** Program Files/Inductive Automation/Ignition/logs<br>• **Lunix:** /var/log/ignition<br>• **Mac OS X:** /Users/UserName/Documents/Ignition-osx-x.x.x/logs |
| Vision | Python's `print` command, or<br><br>system.util.getLogger | 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). |
| Perspective | system.perspective.print | Web browsers generally offer a way to inspect a page, which usually contains a console of some sort. |
| Designer | Depends. See the output location. | 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.<br><br>While in the Designer, and working on Perspective resources, calls to system.perspective.print will appear in the Designer's Console as well. |

#### 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.

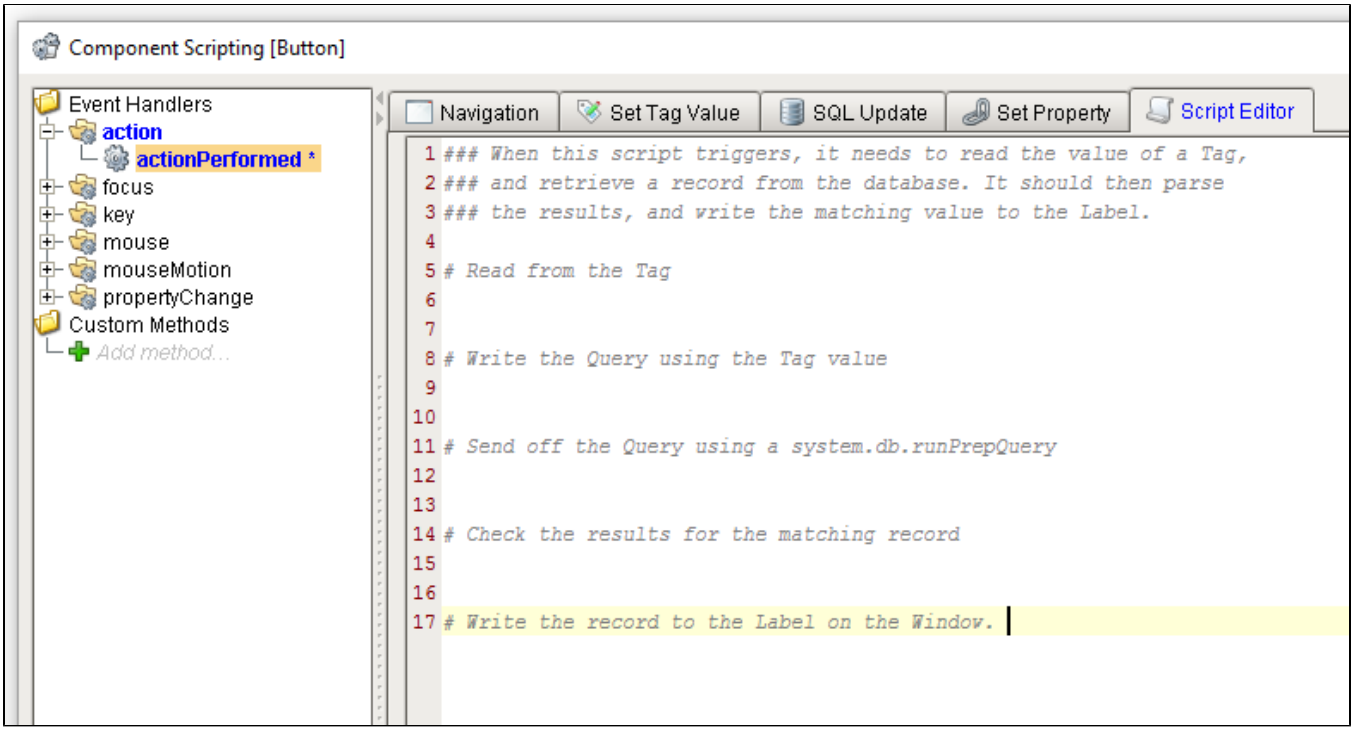
Once your code works as expected, remember to either remove or comment out the print statements, so they don't flood the console during normal use as this makes troubleshooting other issue more difficult. Just be mindful of the scope of the script, as that determines how you generate print messages, as well as where the console is.

**Python - Using print to Troubleshoot**

```
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:

**Python - Hard Coded Message Box**

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

Try to get into the habit of doing this:

**Python - Message Box Variable**

```
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 colleges and agree upon some naming conventions before you write any code.

Ignition's **system.* functions** use the 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 camelCase 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 you're 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.

Related Topics ...

- Python Scripting

In This Section ...

# Reading Error Messages

When an error occurs in the execution of a script, an error message box will pop up. The popup box appears in front of any open Designer windows, and will remain in view until you close it or click on something behind it.

There are two modes for the Error box selectable by the **Send to Front** and **Send to Back** icons.

- **Send to Front** means additional error messages will cause the popup to reappear on top.
- **Send to Back** will cause the errors to remain hidden below the Designer.

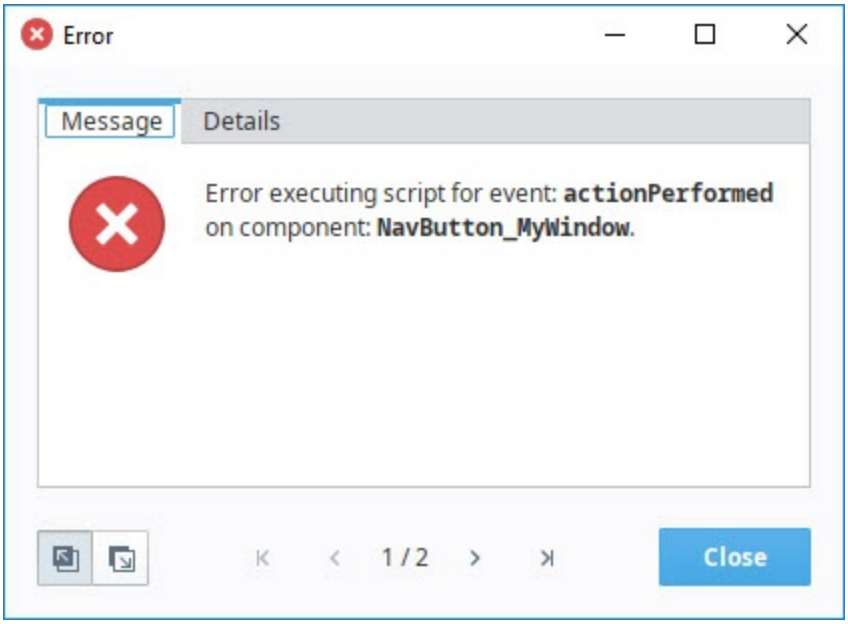
## On this page ...



## 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.

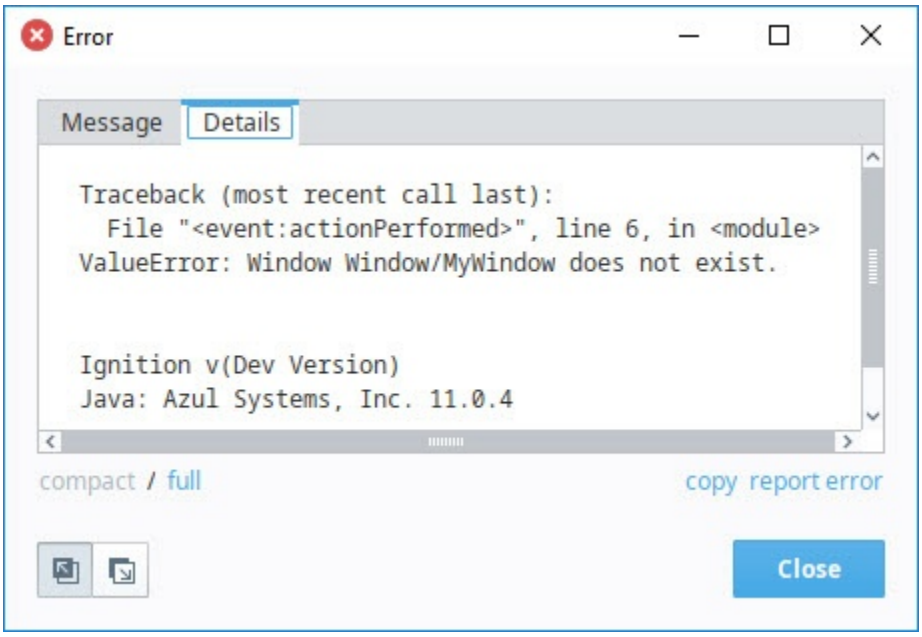
When testing a script, you may come across an Error Message Box like the following.  Your first inclination may be to close the error without closer examination: resist it!




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

In the following script, when a Button is pressed a Message Box is supposed to open on a window and display "Hello World".

**Python - Broken: Incorrect Attribute**

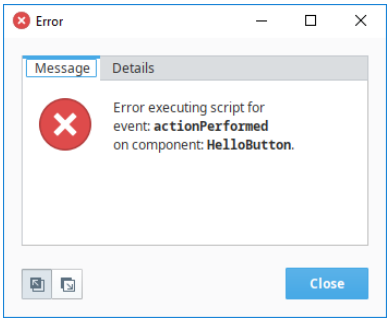
```
system.gui.messagebox("Hello World")
```

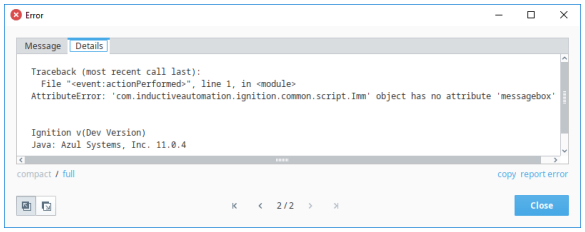
From the error message box, you know two important things right away:

- The script is using the actionPerformed event on the Hello Button.
- Line 1 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 autocompletion 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., **messageBox**).




In this case, "messagebox" must 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:

**Python - Corrected: Incorrect Attribute**

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

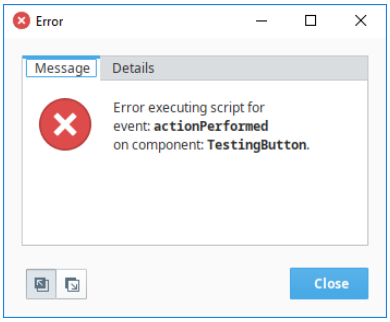
## Broken Example - Undefined

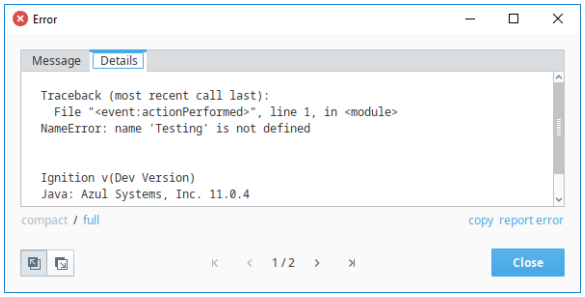
Here is a similar script. When a Button is pressed, it's supposed to open a Message Box on a window and display "Testing".

**Python - Broken: Undefined**

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

The Error Message Box displays an error on the actionPerformed script on the button component in line 1 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:

**Python - Corrected: Undefined**

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

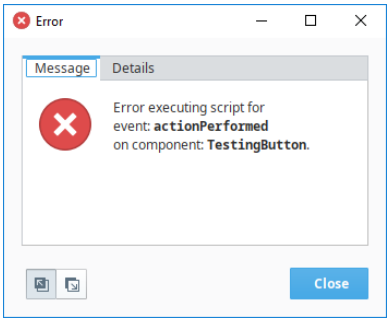
## Broken Example - Type Error

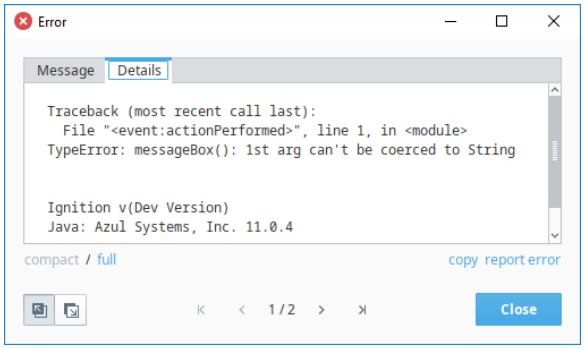
Similar to the other scripts, this script opens a Message Box and is supposed to display "100" when a Button is pressed.

**Python - Broken: TypeError**

```
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 1 with a **TypeError**. This error tells us that the first 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.

**Python - Corrected: TypeError**

```
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.

Related Topics ...

- Basic Python Troubleshooting
- Scripting in Ignition
- Getting Started with Scripting in Ignition

# Troubleshooting - Nothing Happened

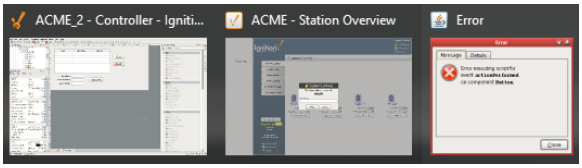
## Did it Work?

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 **T ools > Console**.

## On this page ...



## Common Scenarios when Nothing Happens

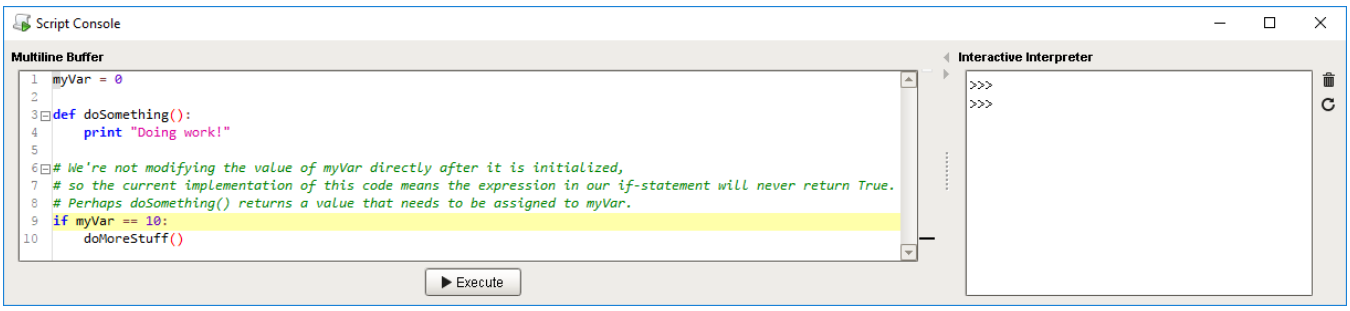
### An Important Line Is Missing

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.

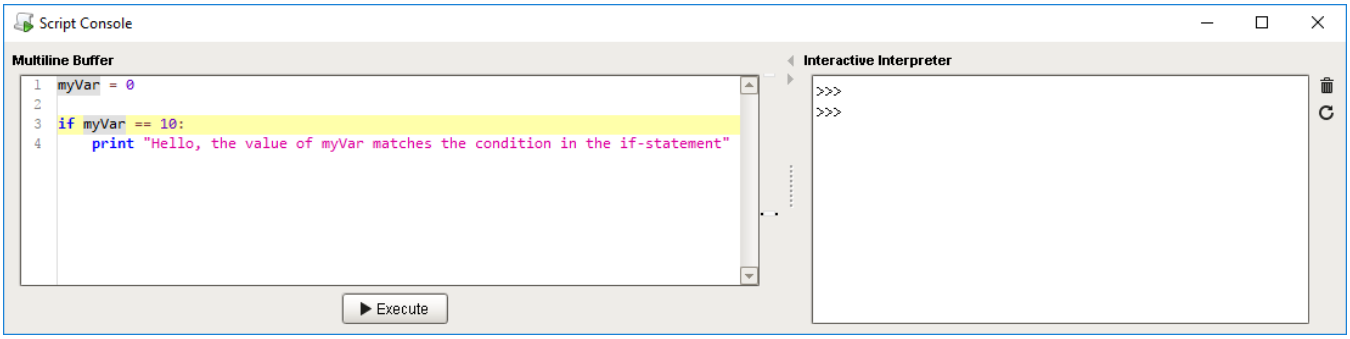
### Important Lines Are Being Skipped

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.

Related Topics ...

- Script Console
- Basic Python Troubleshooting
- Reading Error Messages

# Troubleshooting Workflow

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 indepth 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:

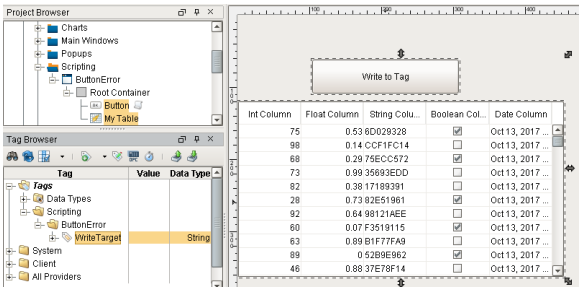
- Reading Error Messages
- Troubleshooting - Nothing Happened

## On this page ...



## 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.



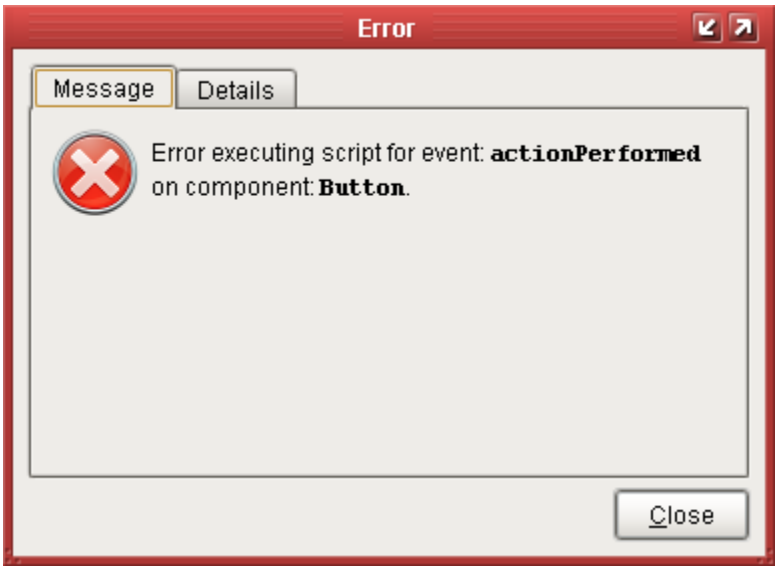

The code on the button is listed below.

**Python - Sample Code Block**

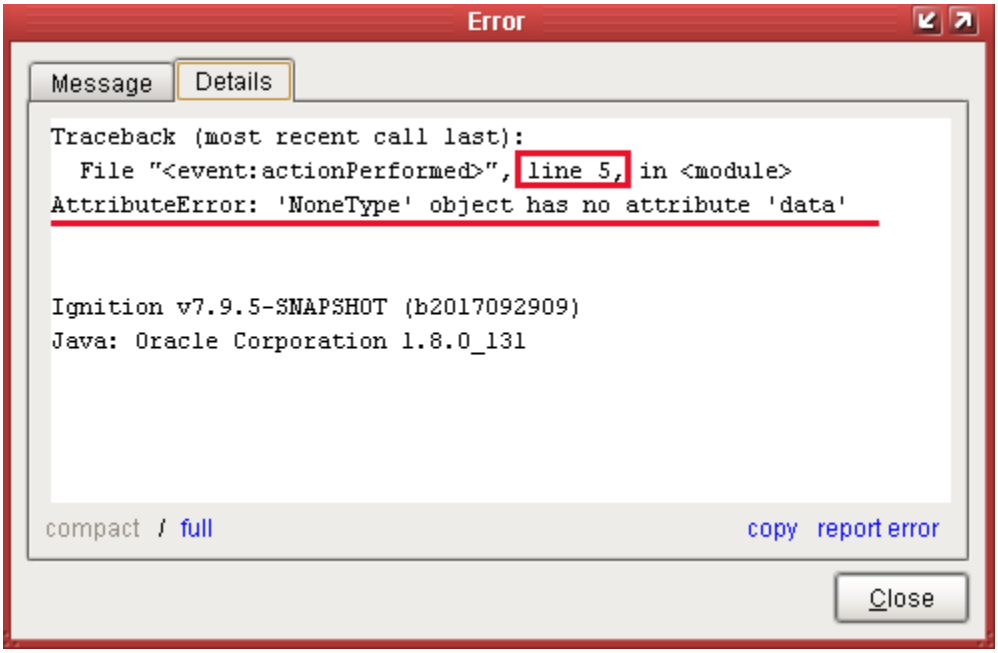
```
# Create a variable that references the Power Table
table = event.source.parent.getComponent('Power Table')

# 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)
```

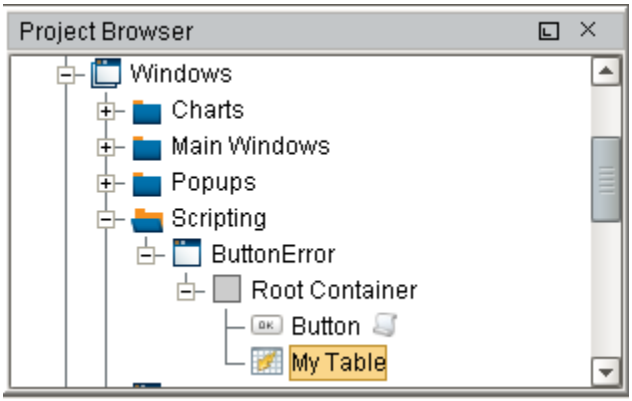
Based on this information, we should take a closer look at the table variable since our code is specifying **'data'** as an attribute of the table. The error states that it could not find the data attribute on a NoneType object, but our code only references data in regards to the table object, so maybe 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

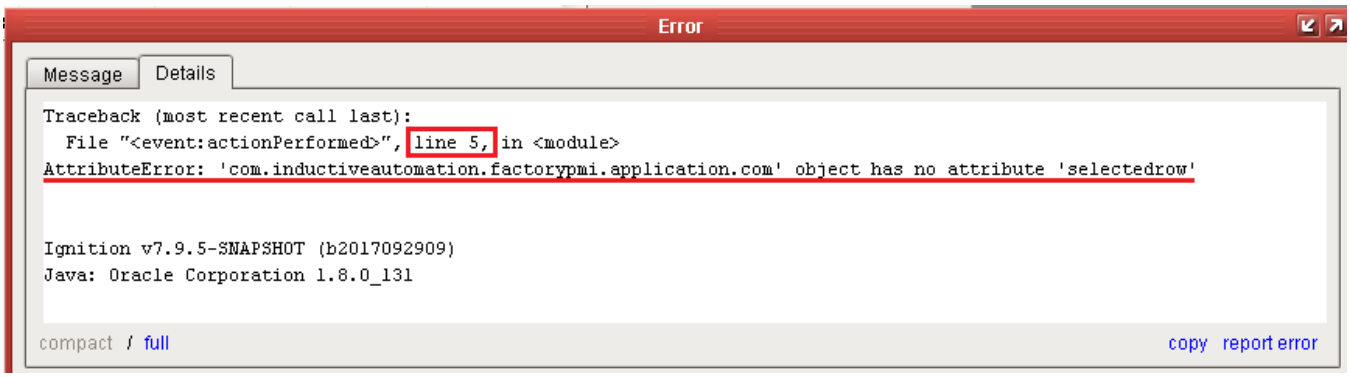
To fix this issue, we can simply update our code to use the new name of the Power Table, which is '**My Table**'. We can type this in manually, but since it is case sensitive, it must match exactly. It may be easier to find the **Name** property on the component, copy the name to the system clipboard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

**Python - Updated Code Block**

```
# 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.

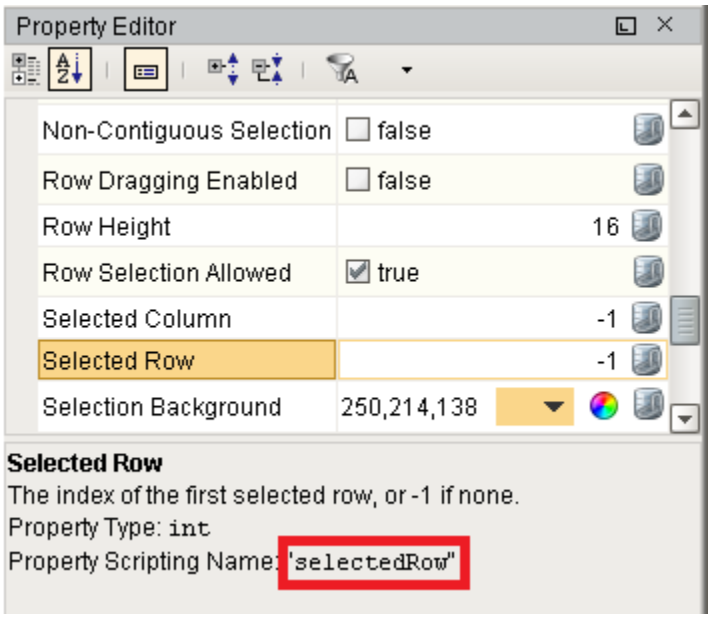
## What Should We Look For?

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 component 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

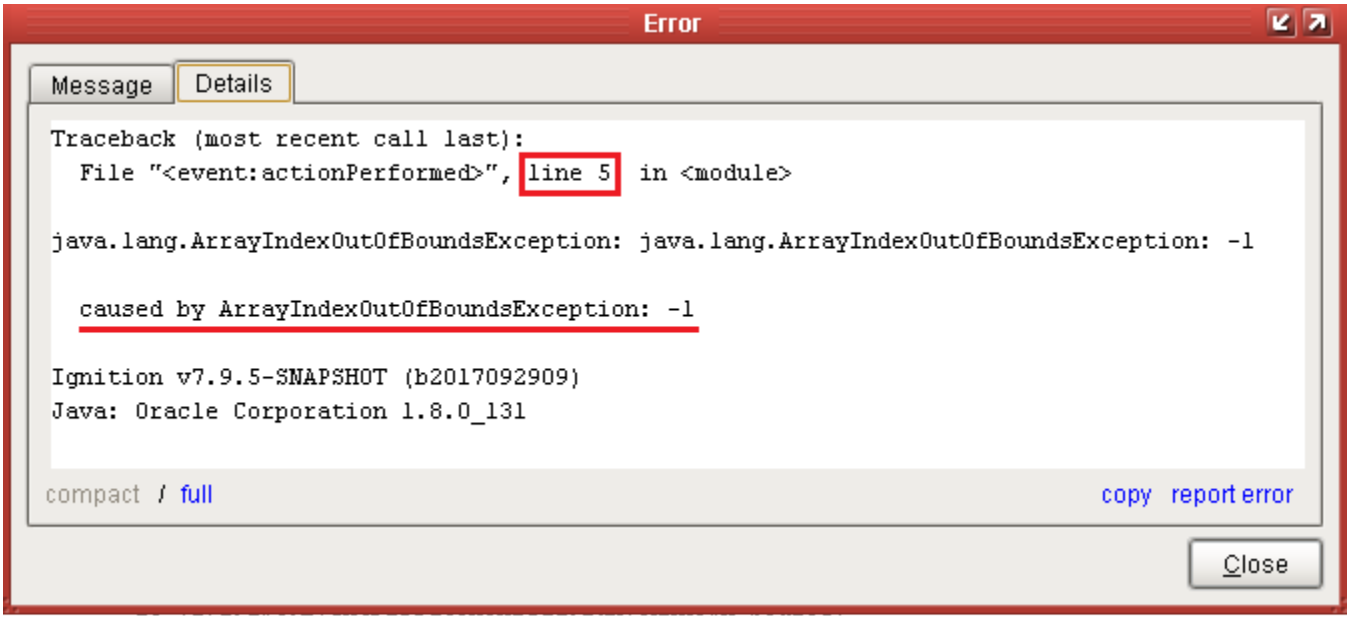
We should fix the typo in the attribute name.

**Python - Line 5 Updated**

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

## Third Error - ArrayIndexOutOfBounds

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 **Selec ted 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:

**Python - Code Block Update #3**

```
# 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!")
```

Related Topics ...

- Reading Error Messages
- Troubleshooting - Nothing Happened

# Scripting Vs. SQL Vs. Expressions

## 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

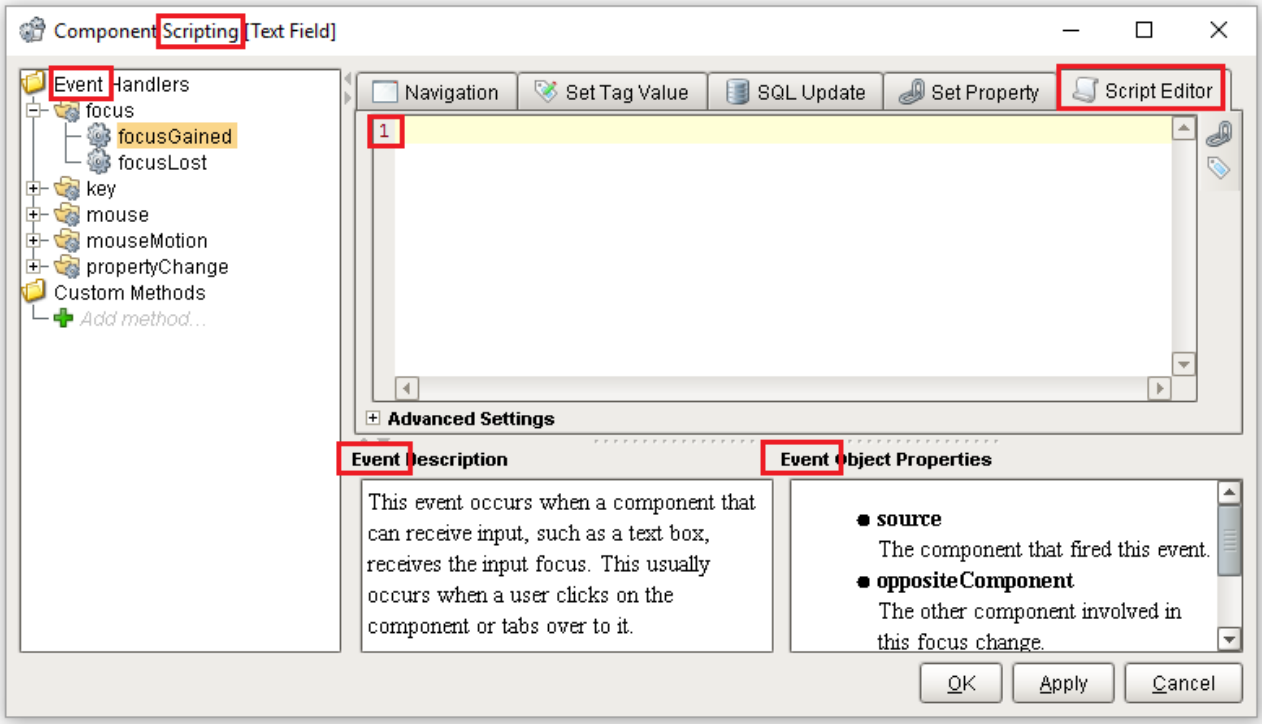
### Overview

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.

## On this page ...




### 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:

- Scripting in Vision
- Extension Functions
- Tag Event Scripts
- Client Event Scripts
- Gateway Event Scripts

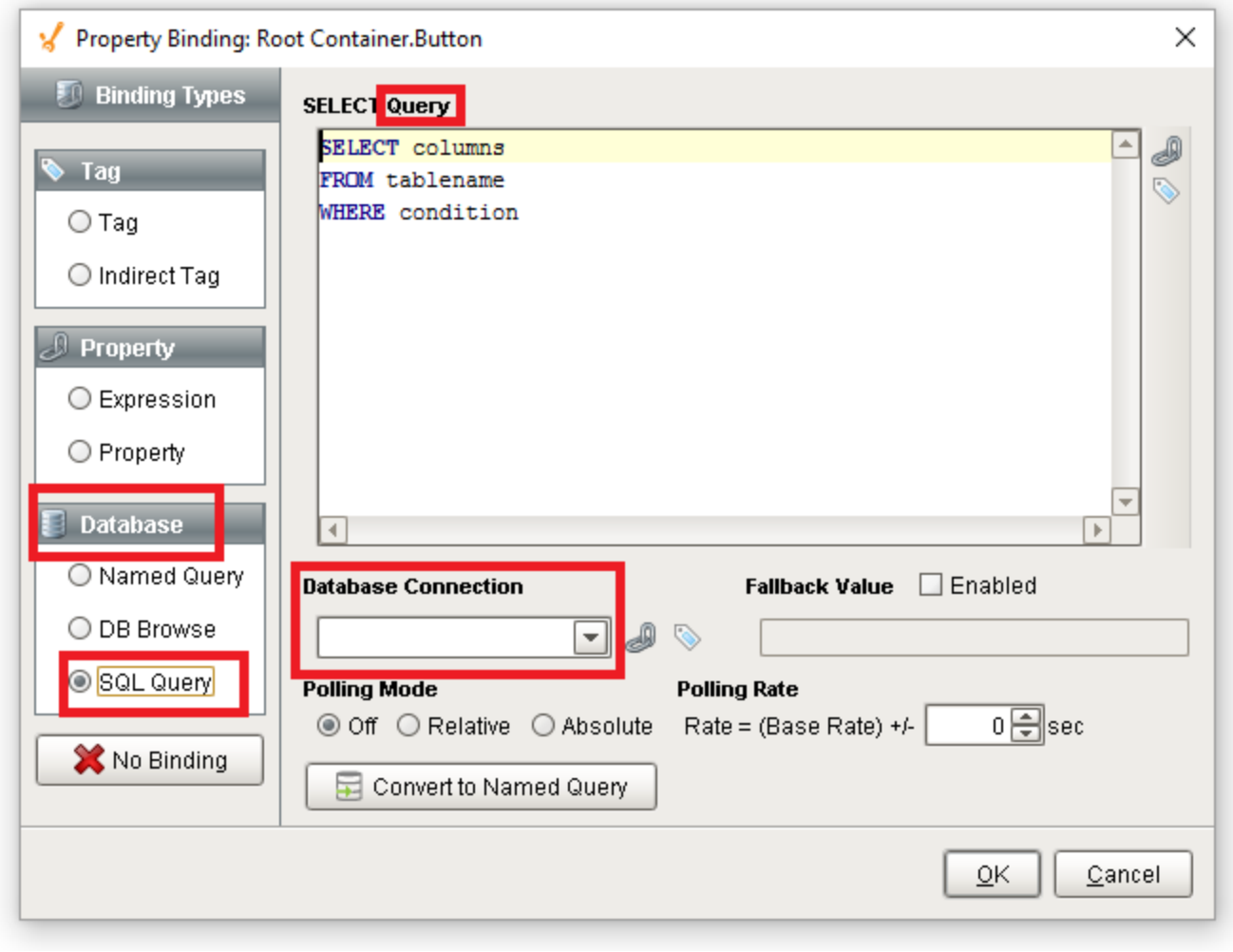
## SQL Queries

### 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 and Basic SQL Query
- Query Tags
- Python Scripts - See the SQL in Python header below for more details

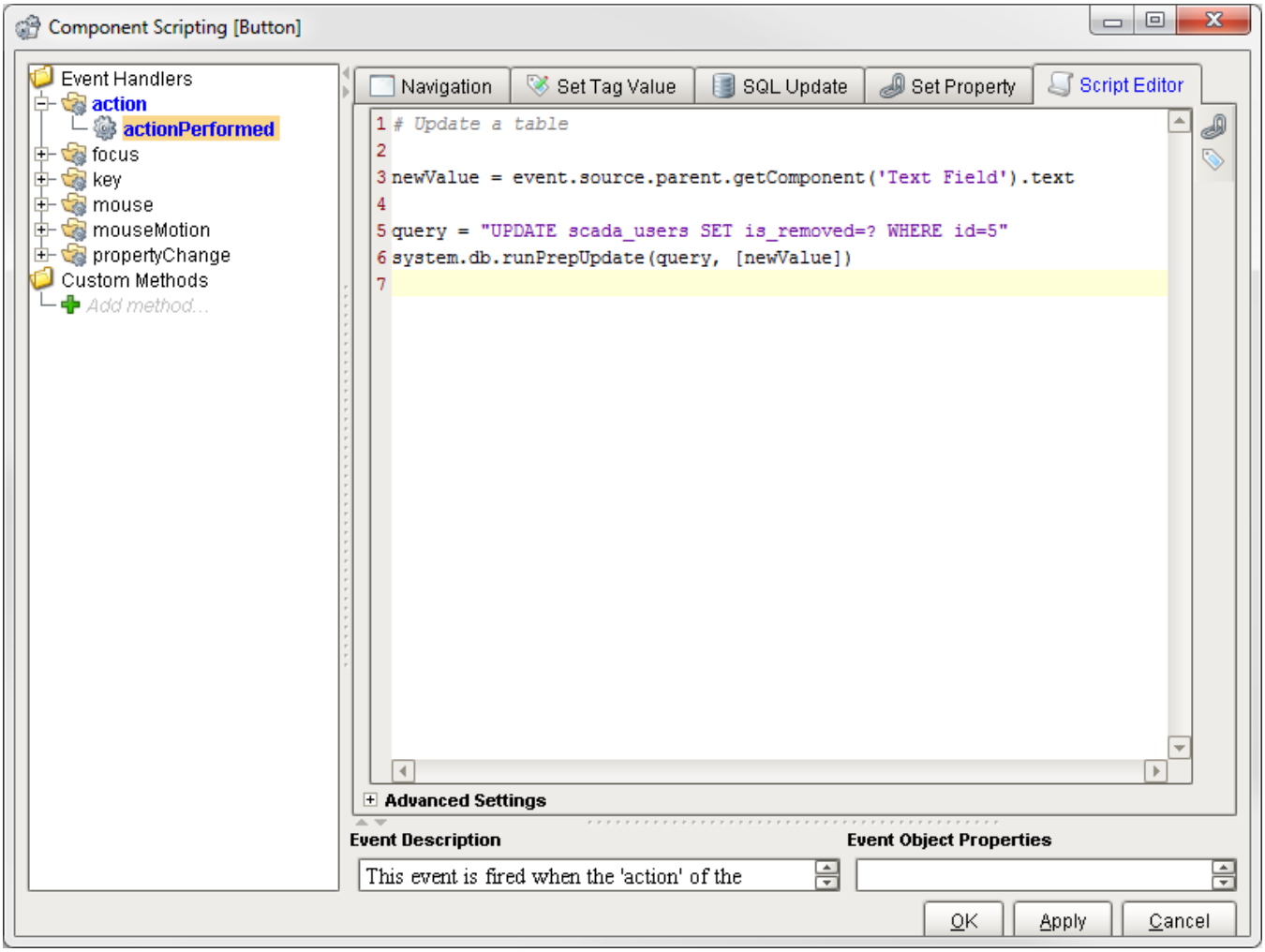
## SQL in Python

**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

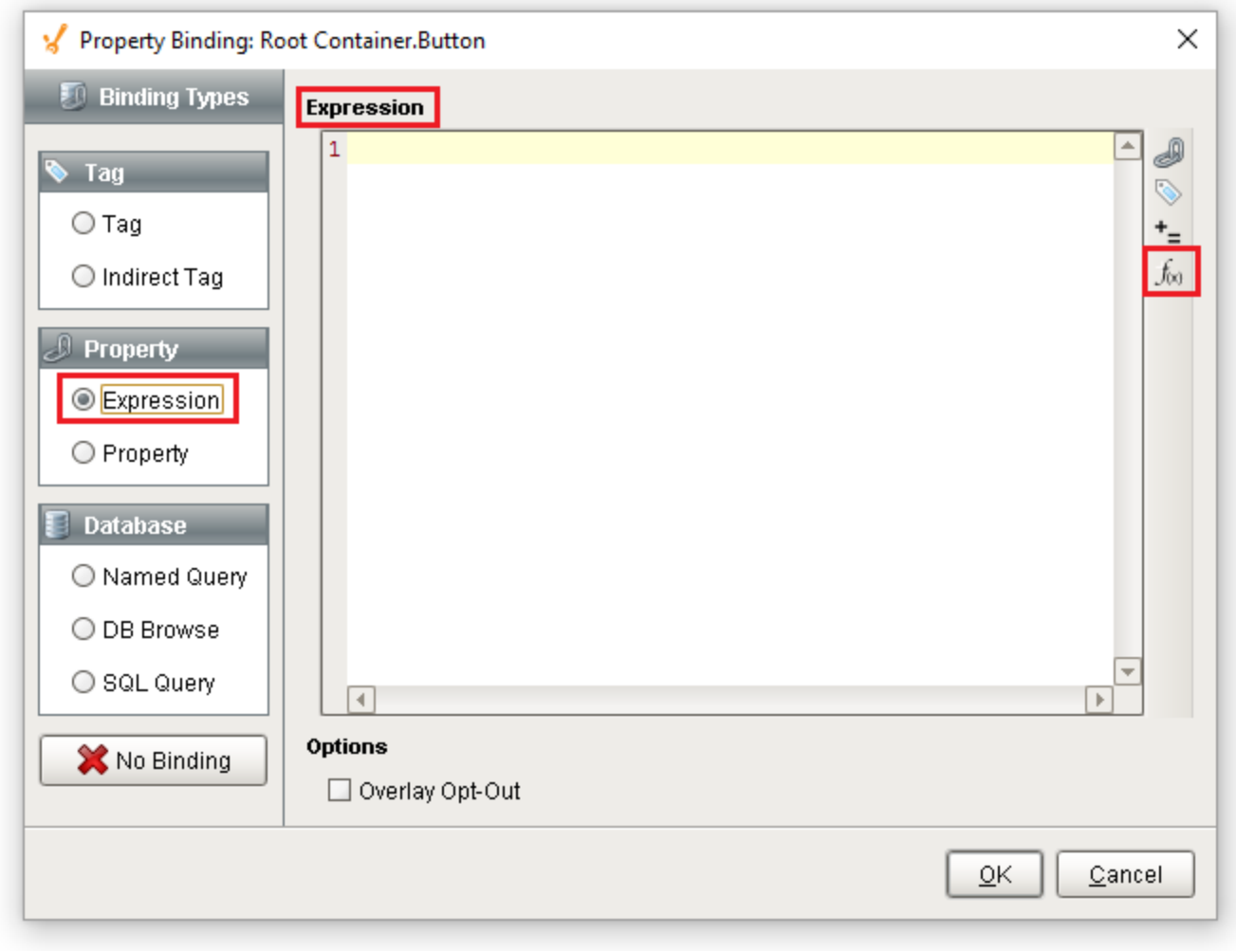
## Overview

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 calculated 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 $f(x)$ 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

Related Topics ...

- Expression Language and Syntax
- SQL in Ignition
- Scripting
- Python Scripting

# Scripting Examples

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 Ig nition. 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.

## Adding a Delay to a Script

In some cases, being able to halt execution of a script can be helpful while waiting for some other event to occur. Adding a Delay to a Script page details some common approaches, as well as approaches to avoid.

## Export Tag Historian to CSV

The Tag Historian Data can be great, but it's sometimes difficult to view it outside of Ignition. The Export Tag Historian to CSV page details how to pull out a subset of Tag history data and export it to a CSV file.

## Parsing XML with the Etree Library

XML files are a great way to share information between two systems, but parsing them can seem like a daunting task. Parsing XML with the Etree Library shows how to use Python's Etree Library to read through an XML document.

### On this page ...



Related Topics ...

- Basic Python Troubleshooting

In This Section ...

# Location Based Vision Startup Scripts

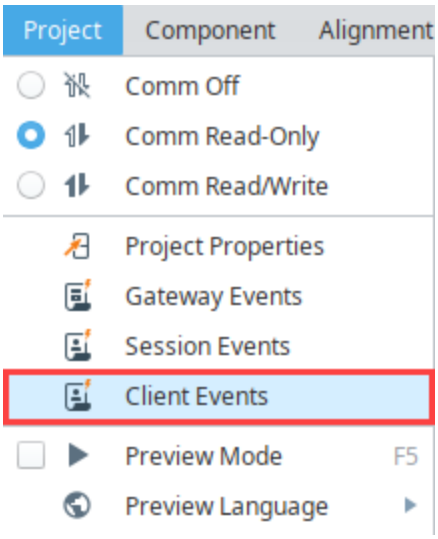
## Client Startup Scripts

Security can be created on a Vision Client using a Startup Script that checks certain user information and then customizes the level of access within the project based on that information. We can pull user information from places like System Tags for Vision Clients or Session Properties for Perspective Sessions. Using that information, the script can then customize which windows or views the user sees, provide some additional information to the user, or even prevent the user from accessing the project completely. An additional example is located here.

## Location Based Vision Client Restrictions

Each Client has access to distinct hostname and IP addresses. These can be used in Client startup scripts that evaluate the Client's information and compares it to a list of acceptable host names or IP addresses. This information can come from a database or a set of Tags. In this example, we will prevent access to the client if the user is logging in from an incorrect location.

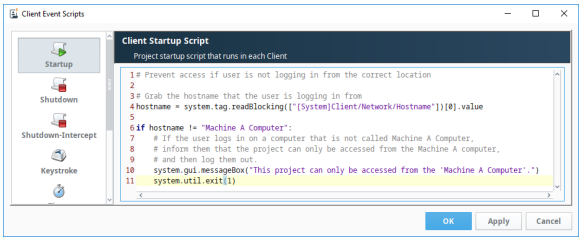
1. On the Project tab, choose Client Events.



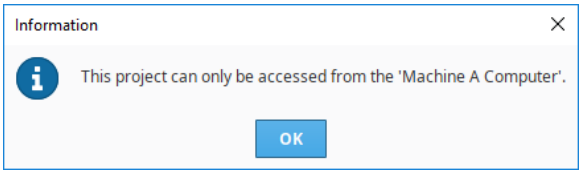

2. Select the Startup  icon.
3. Add the following script:

```
# Prevent access if user is not logging in from the correct location

# Grab the hostname that the user is logging in from
hostname = system.tag.readBlocking(["[System]Client/Network
/Hostname"])[0].value

if hostname != "Machine A Computer":
        # If the user logs in on a computer that is not called
Machine A Computer,
        # inform them that the project can only be accessed from the
Machine A computer,
        # and then log them out.
        system.gui.messageBox("This project can only be accessed
from the 'Machine A Computer'.")
        system.util.exit(1)
```

### On this page ...



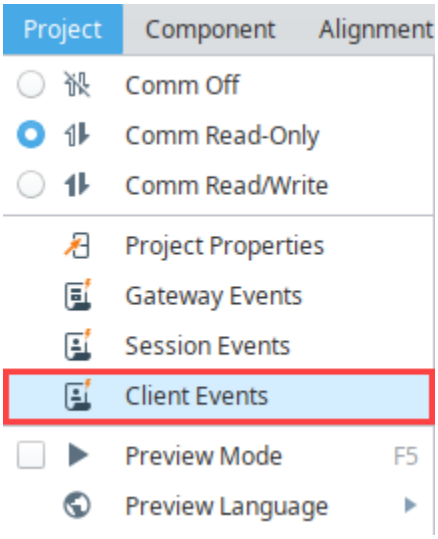

4. Click **OK** to save the script.
5. Save your project and launch the client. You will get a popup that informs you that the project must be launched using the Machine A Computer.




## Location Based Startup Display

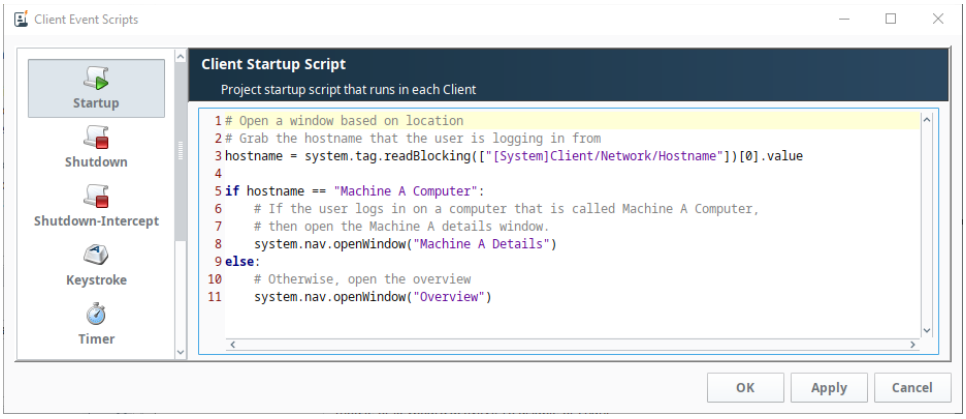
Another common use for startup scripts on Vision Clients is to open a specific window depending on the location of the log in. In this example, if the user logs in on Machine A Computer, then the Machine A Details window will be displayed. If they log in on a different computer, the Overview window is displayed.

1. On the Project tab, choose Client Events.




2. Select the Startup  icon.
3. Add the following script:

```
# Open a window based on location

# Grab the hostname that the user is logging in from
hostname = system.tag.readBlocking(["[System]Client/Network/Hostname"])[0].value

if hostname == "Machine A Computer":
	# If the user logs in on a computer that is called Machine A Computer,
	# then open the Machine A details window.
	system.nav.openWindow("Machine A Details")
else:
	# Otherwise, open the overview
	system.nav.openWindow("Overview")
```




4. Click **OK** to save the script.

   **Note:** Any windows that are set to Open on Startup will still open in addition to the window specified in the startup script. You should disable any main windows from opening on startup if using this method.

5. Save your project.

Related Topics ...

- Client Tags for Indirection

# Reading and Writing to Tags

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.

## On this page ...



## 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.readBlocking(["My/Tag/Path"])

# Reading from the same Tag, but specifying the Tag Provider. This format
may be used safely in either scope.
system.tag.readBlocking(["[default]My/Tag/Path"])
```


## Manual Tag Reads

### Reading from a Single Tag

Reading a Tag from a script is accomplished with the `system.tag.readBlocking()` function, which requires the Tag path you wish to read from. 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.

The function normally expects a list of Tag paths, but can be used with a single Tag: simply provide a list of only a single Tag path.

When handling the results of the read, the results will be returned in a list of qualified values, so you'll need to specify which qualified value you're interested in via Python slicing, even if there is only a single qualified value in the list.

**Pseudocode - Reading Tag Attributes**

```
# get the Tag value
value = system.tag.readBlocking(["tagPath"])[0].value

# get all three attributes
tag = system.tag.readBlocking(["tagPath"])[0]
value = tag.value
quality = tag.quality
timestamp= tag.timestamp
```

### Reading from Multiple Tags

The `system.tag.readBlocking()` function can easily be used to read multiple Tags in a single call:

**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.readBlocking(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.readBlocking(["[.]otherMember"]).value
```

## Manual Tag Writes

### Writing to a Single Tag

Much like reading, there is a function you can use to write to Tags: `system.tag.writeBlocking`. It requires a list of Tag paths, as well as a list of values to write to those Tags.

**Pseudocode - Tag Write**

```
system.tag.writeBlocking(["tagPath"],["Hello World"])
```

### Writing to Multiple Tags

The `system.tag.writeBlocking` function can also write to multiple Tags, again by providing multiple paths and values.

**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.writeBlocking(paths, values)
```

Related Topics ...

- Tags

# 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 flexibe 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 CSV**

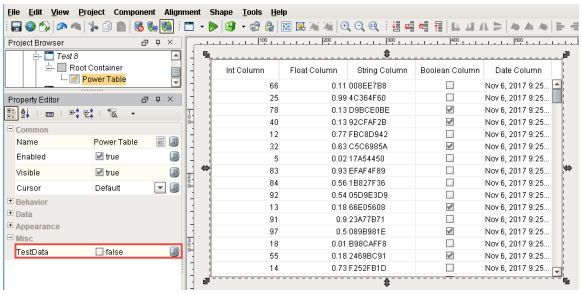
Watch the Video

The examples on this page make use of Vision components, and run in the Vision Client scope.

## 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.



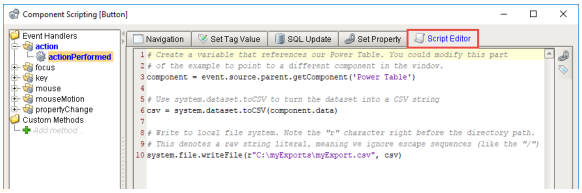

2. Drag a **Button** component on the window, and double click on the Button to open the **Component Scripting** window.

3. Next, let's add our script on the Button component's actionPerformed event. Select the **actionPerformed** event, and click on the **Script Editor** tab.

4. 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)
```

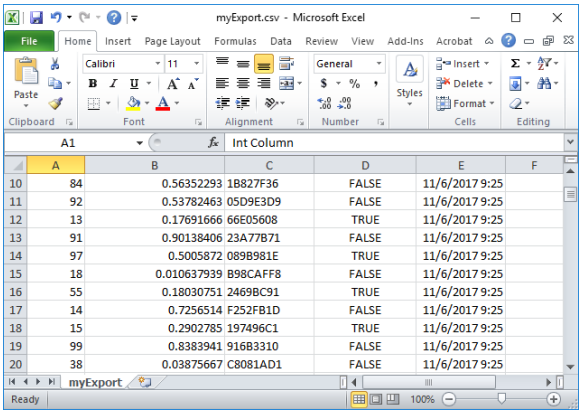



5. Instead of hardcoding the path as we did in the above example, we could ask the user to select a directory on the local system with system.file.saveFile:

**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.




## 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** componen t's **Data** property. We have a couple of approaches listed below.

### Calling the system.dataset.fromCSV Function

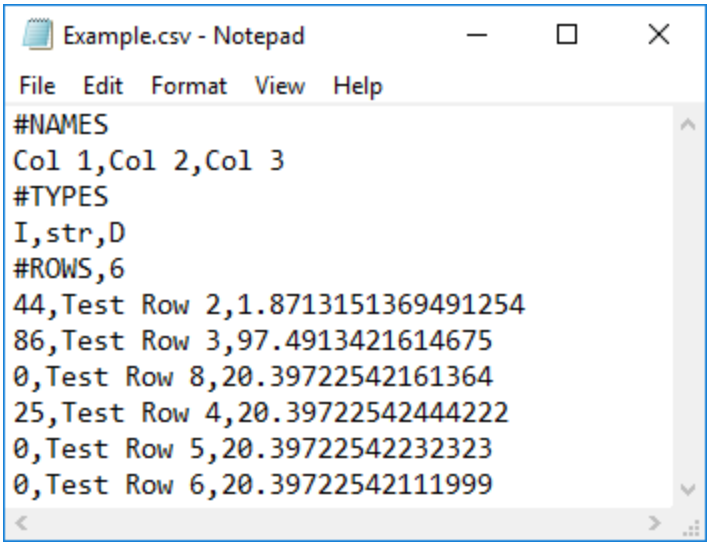
The system.dataset.fromCSV function can take a string and convert it to a dataset. Note, that the function expects a very specific format:

**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

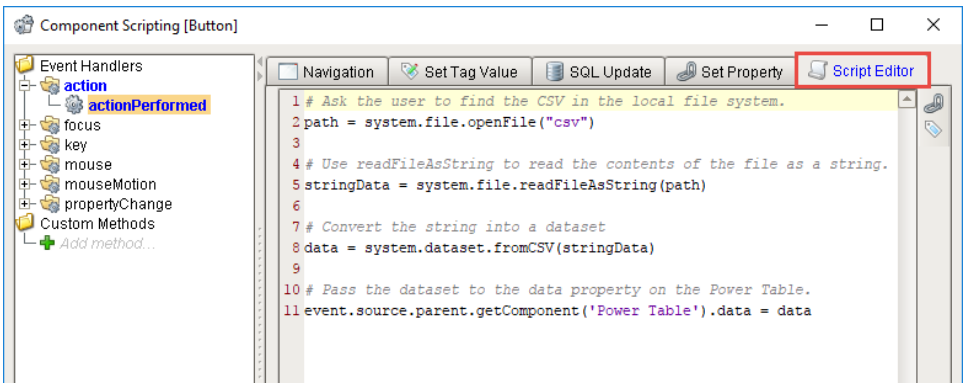
1. Create a Text file on your local system named "**example.csv**."
2. Open the Text file, copy the contents of the "**CSV file Content**" box above, and paste the contents to the file. Save the changes to the **examp le.csv** file.



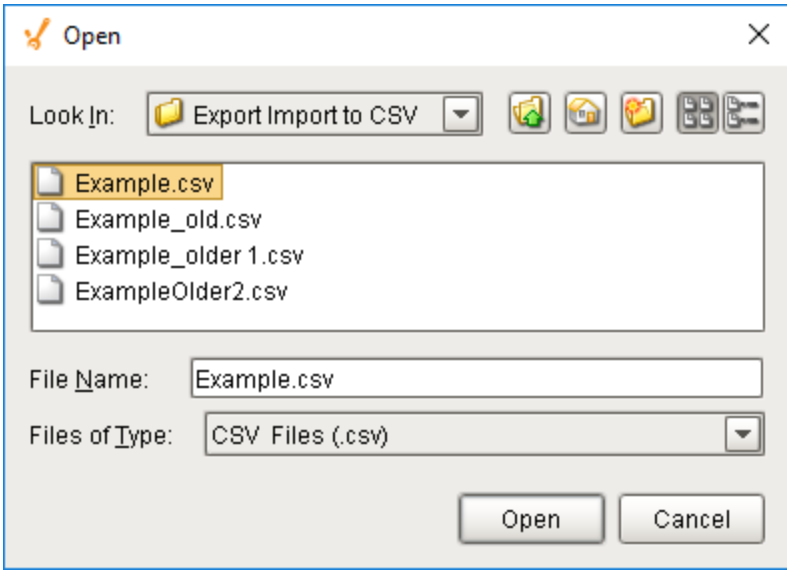

3. Let's move the contents of a CSV file to a Power Table. Add a **Power Table** and a **Button** component to your window. Double click on the **Button** component, and paste the following code into the **Script Editor** of the **actionPerformed** event:

**Python - Using system.dataset.fromCSV()**

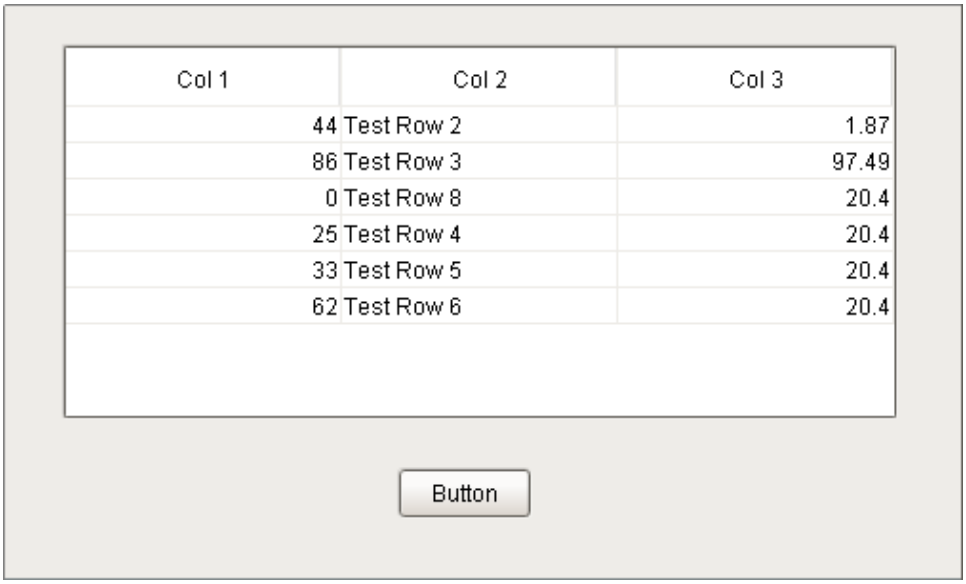
```
# Ask the user to find the CSV in the local file system.
path = system.file.openFile("csv")

# 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.




## 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 at 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
```

Related Topics ...

- system.dataset.exportCSV

# Adding a Delay to a Script

## Overview

In some cases, having a script execute after a delay is preferable. A common use case is waiting for some event elsewhere in the system to finish: a Tag change script executes that needs to wait for a new value from a separate Tag. One approach to this is to trigger our script, and then hold or wait until the other event occurs. On this page, we'll take a look at a couple of different approaches to this problem.

It is important to note that pausing a script can cause your client to lock. It is often preferred to look for another event to trigger the script you need. For the example above where we are waiting on a Tag to change, you might be able to use the Tag change to fire your script instead of waiting in the original script. It is up to you to determine the best trigger based on what exactly your script does.

### On this page ...



## Using the system.util.invokeLater Function

The system.util.invokeLater function is a great way to add a delay mid-way through the script. Simply create a function that represents all of the work that should occur after the delay, and pass the function to invokeLater, along with a delay period.

The example below calls two Message Boxes: once initially when pressed, and the other after a three second delay.

**Python - Two Message Boxes**

```
message = "All Done!"

# Create a function that will be called by invokeLater.
def runThisLater():
        system.gui.messageBox(message)

# Call invokeLater with a 3000ms delay. Note that our function will not
run immediately
# because invokeLater always executes once the rest of this script is
complete.
system.util.invokeLater(runThisLater, 3000)

# Bring up another Message Box. This will appear before the "All Done!"
message, because of invokeLater.
system.gui.messageBox("Waiting...")
```

One of the main limitations with invokeLater is that you can not pass parameters to the function that will be called. Parameters need to be initialized and determined elsewhere in your script, usually in the function definition.

## Using a Timer from Python's Threading Library

The Threading Library has a Timer function that works in a very similar fashion to invokeLater. The main difference is that you can pass parameters to the function parameter when calling the Timer. Take a look at Python's official documentation for more details on the Timer object.

The example below will again call two Message Boxes with a three second delay between them. However, the text that is defined in the second Message Box is specified when starting the Timer.

**Python - Python Threading Timer**

```
from threading import Timer

# Create a function that will be called by the Timer.
def runThisLater(param):
        system.gui.messageBox(str(param))

# Constructs a Timer object that runs a function after a specified interval of time has passed.
# Note the start() at the end: this is required to start the Timer. Don't forget this part!
Timer(3.0, runThisLater, ["Stop Waiting, I'm done"]).start()
```

```
# Bring up another Message Box. This will appear before the "All Done!" message, because of the Timer
system.gui.messageBox("Waiting")
```

Another benefit to using the Timer, is you can cancel the execution of the Timer's action using the cancel() function, but it only works if the Timer is still in its waiting stage.

## Calling time.sleep from Python's Time Library

The simplest approach to pausing a script is to use the sleep() function in Python's Time Library:

**Python - Sleeping the Code**

```
from time import sleep

# This will pause execution of the script for 3 seconds. After that time, the script will continue.
sleep(3)

print "I'm awake!"
```

## Using a While Loop

The While Loop is another simple approach to adding a delay: simply keep looping until the other event occurs. As always, you will want to take steps to ensure that an infinite loop never occurs: easiest by initializing a counter variable before iterating in the loop, and breaking out if the counter reaches a certain value.

**Python - While Loop Safeguard**

```
counter = 0

while not otherEvent:

        checkOtherEvent()
        counter += 1

        # Use this to break out if the event takes too long. You'll need the rest of your script to account
for this possibility.
        if  counter >= 10000:
                print "Took too long. Leaving the While Loop"
                break
```

## Approaches to Avoid - Locking the Client

While the sleep() and while loop functions are simple to use, they can cause problems by locking up the client and because of this, we generally recommend avoiding them if possible. Typically, system.util.invokeLater on a delay can accomplish the same task.

Note, that there are use cases for both approaches outlined below especially so in regard to the While Loop. However, neither function should be used to force a delay in a Client.

> Using either of these methods to pause or delay a script can lock up the entire client, which may be very dangerous.

### Reasons to Avoid time.sleep and While Loops

If used on a component, these approaches could lock up the Client or Designer. Scripts called from a component run in the same thread that the Client and Designer use to refresh the screen. Whenever a script on a component triggers, the screen is unable to refresh until the script finishes. In most cases, this is so fast that no one notices. If your script is intentionally calling sleep() or using a long running While Loop, then the Client will lock up for the duration.

Additionally, these approaches run an extended period of time, and block other component based scripts from executing. They do not yield to other scripts while waiting.

Because of these two reasons, the sleep() and While Loop functions can cause your clients to appear unresponsive, when really they are just running a script that prevents the screen from refreshing.

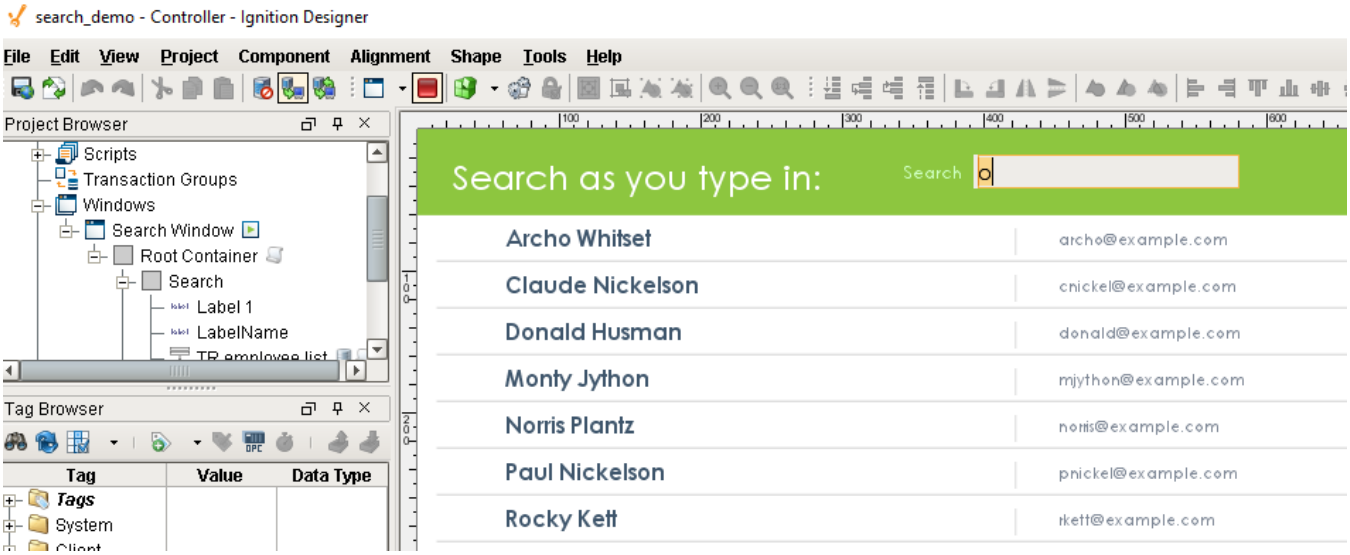
### Recommended Alternative

As mentioned earlier, the system.util.invokeLater function (also mentioned on this page) can provide the same functionality. If a sleep() or While Loop function must be used, then they should be called from system.util.invokeAsynchronous to prevent blocking any other scripts.

## Demonstration - Executing a Delay Between User Keyboard Input

In many cases, you may need to show your users a large number of entries on a Table or Template Repeater. As more entries are added to the system, adding a search field that can filter results becomes more appealing. Furthermore, being able to filter as the user types (as opposed to forcing them to hit enter every time) adds some polish to the window. However, if the entries are backed by the database, you will not want to run a query every time a user presses a key. Instead, it would be preferable to wait for a delay in input, and run one query to limit strain on the database.

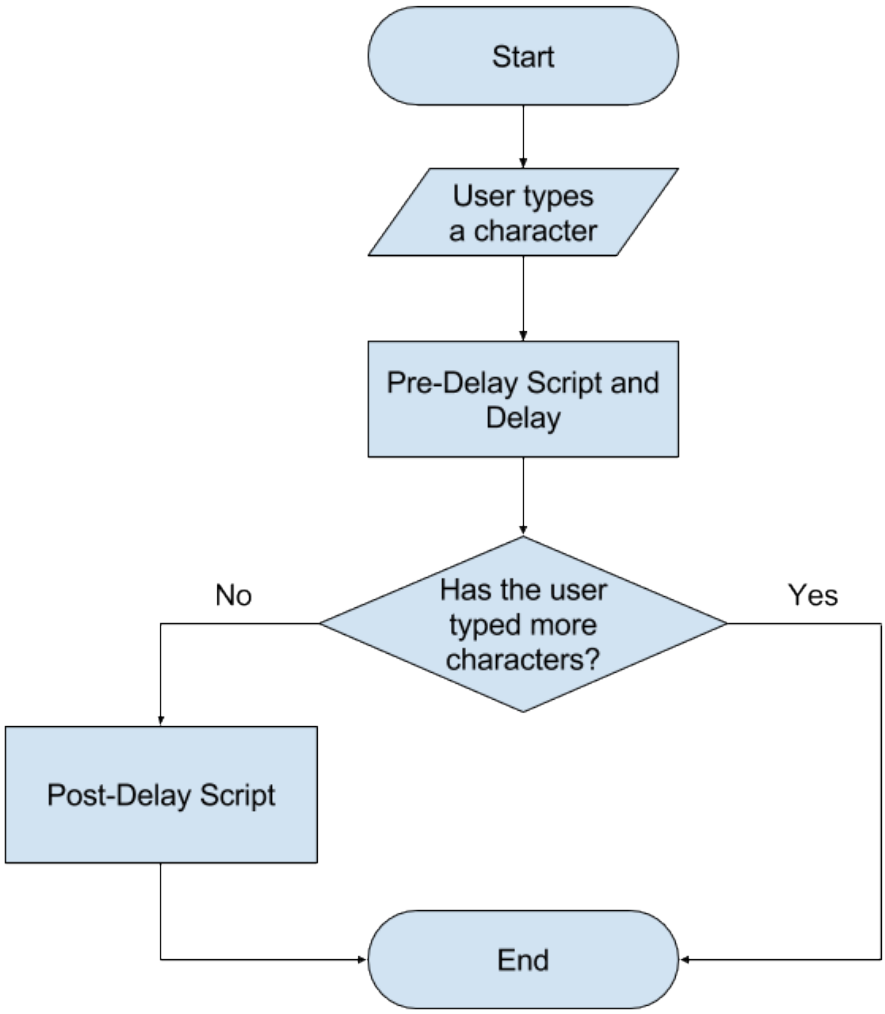
The following is not a traditional example that you would find here in the User Manual, and assumes you are comfortable with many concepts in Ignition.




### Workflow for a Delay Using Two Scripts

One approach to adding a delay involves two scripts:

- **Pre-Delay Script**: This script fires before the delay. In this case, we will use a script on the Text Field's keyReleased event. This will call the script everytime a new key is entered into the Text Field, and calls the Post-Delay Script on a delay. It also passes the text that is currently in the Text Field, and increments an edit count (could be configured as a custom property on the same component).
- **Post-Delay Script**: This script runs after the delay, so it needs to check if there were any new keystrokes that occurred during the delay. If there were no new keystrokes, then it can run the query. In this demonstration, we will use a Custom Method not the Text Field component.




Additionally, we will need some criteria that our script can use to determine if it is safe to run our query. You can have multiple criteria here, but for the sake of simplicity we will use the **Text Property on the Text Field**. This way we will know what the user typed before the delay, and then can compare it back to the Text property after the delay. If the Text on the component after the delay is different, then we don't want to run the query as the user may still be typing.

The keyReleased script is shown below. It is using a .05 second delay, but could of course can be modified. We're creating a new Timer object every time a key is released, so there will be many of these scripts firing as the user types. However, they are fairly lightweight, and don't directly interact with the database, so having many executions of this script isn't taxing on the system.

**Python - Pre-Delay Script**

```
# We're using the Timer object for this because we want to pass parameters to the function that will run
after the delay.
from threading import Timer

# Ignoring arrow keys.
if (event.keyCode < 37 or event.keyCode > 40):

	# Grab the text in the component currently.
	currentText = event.source.text

	# Calls the function after 0.5 seconds, and pass the currentText.
	Timer (0.5, event.source.sendingQuery,[currentText]).start()
```

Below is the sendingQuery script. As mentioned, this script will determine if the user ceased keyboard activity before running a query against the database. This example is using a Custom Method on the Text Field, so it uses the `self` argument to reference the source component.

**Python - Post-Delay Script**

```
def sendingQuery (self, oldText):

	# Read the value of the text property, since it may have changed during the delay.
	currentSearchText = self.text

	# If the text before the delay is the same as the after the delay, there has been a pause in
keyboard activity, so we should run the query.
	if (currentSearchText == oldText):

		# If the text field isn't empty, then we need to filter the results with a WHERE clause and
our criteria.
		if(currentSearchText != ""):
			newQuery = "SELECT * FROM employees WHERE CONCAT(firstname, ' ', lastname) like '%"
+ currentSearchText + "%'"
		# If the Text Field Is blank, then we should run the query again, but this time without a
filter, so all results appear.
		else:
			newQuery = "SELECT * FROM employees"
```

We can now run our query in between user keyboard activity.

Related Topics ...

- system.util.invokeLater
- Timer Scripts

# 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.

### On this page ...



## 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.

**Python - Export History to CSV**

```
# 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.

| Part of Script | Description |
|---|---|

| | |
|---|---|
| `path='histprov:myDB:/drv:myIgnitionGateway:myTagProvider:/tag:myFolderOfTags'` | 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.<br><br>• histprov:myDB:/ - This is the name of your Historical Tag Provider that you are searching in. Replace **myDB** with the name of your historical Tag Provider.<br>• drv:myIgnitionGateway:myTagProvider:/ - The first thing is the name of your Ignition Gateway. Replace **myIgnition Gateway** with the name of your Ignition Gateway, which can be found by going to the Gateway Configure Webpage and navigating to **System > Gateway Settings**. It is the **System Name**. The second thing is the name of the Realtime Tag Provider where the Tags reside. Replace **myTagProvider** with the name of your Tag Provider.<br>• tag:myFolderOfTags - This is the name of the folder you want to search for historical Tags. Replace **myFolderOfTags** with the name of a folder in your Tag Browser to search through.<br><br>**Note:** Paths to Internal Historian providers (such as those used by Edge Gateways) do not need to include the Gateway name. |
| `system.tag.queryTagHistory(...)` | 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. |
| `r"C:\myExports\myExport.csv"` | 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. |

**Caution:**

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)
```

Related Topics ...

- Exporting and Importing a CSV

# Parsing XML with the Etree Library

## What is the xml.etree Library?

The etree (ElementTree) library is a part of the python standard library, and contains many tools that make it simple to parse through and pull information out of an XML document. There are other libraries that can parse through XML documents, but etree is commonly used and very easy to get started with. The etree library will break up the XML into easily accessible elements, each representing a single node in the entire XML tree. For more information on using the etree library beyond the scope of this page, see the python documentation.

### On this page ...



### Using the xml.etree Library

There are a couple of different ways to import the XML data from etree, depending on how it is being stored. It can pull the data in from an XML file using the filepath, or it can read a string. Notice how regardless of how we import the XML, we end up with a root object.

**Python - Reading a File**

```
# The library must first be imported no matter how we pull in the data.
import xml.etree.ElementTree as ET

# Here we can grab the filepath using Ignition's built in openFile function, parse that into a tree, then
grab the root element.
filepath = system.file.openFile()
tree = ET.parse(filepath)
root = tree.getroot()
```

**Python - Reading from a String**

```
# The library must first be imported no matter how we pull in the data.
import xml.etree.ElementTree as ET

# Alternately, we can start with a string of the xml data.
xmlString = """
<employee id="1234">
        <name>John Smith</name>
        <start_date>2010-11-26</start_date>
        <department>IT</department>
        <title>Tech Support</title>
</employee>
"""

# Then parse through the string using a different function that takes us straight to the root element.
root = ET.fromstring(xmlString)
```

Each Tag is considered an element object. In the example above, the root element would be the employee Tag. Elements can also have attributes, which are within the Tag itself. In the example above, the employee element has an attribute id with a value 1234. Finally, each element can also have additional data, typically in the form of a string. This additional data is usually placed in between the element's start and end Tags. In the example above, the employee element has no additional data, but its children do. The name element would have an additional data value of John Smith. All of this data can be accessed using the Element object's built-in functionality. The major functions are listed below, and each example uses the reading from a string root XML example from above.

| Function | Description | Example | Output |
|---|---|---|---|
| Element.tag | Returns the name of the Element's Tag. | `print root.tag` | employee |
| Element.attrib | Returns a dictionary of the Element's attributes. | | {'id':'1234'} |

| | | | |
|---|---|---|---|
| | | `print root.attrib` | |
| Element.text | Returns the additional data of the Element. The example here will return nothing because the root does not have any text. The next example uses children which do have text. | `print root.text` | |
| for child in Element | Will iterate through the Element's children. Each child is then its own element, complete with Tag, attrib, and text properties. | `for child in root:`<br><br>`print child.tag, child.text` | name John Smith start_date 2010-11-26 department IT title Tech Support |
| Element [index] | Allows direct reference to an Element's children by index. Since Tags can be nested many times, further nested children can be accessed by adding an additional index in square brackets as many times as necessary: Element[1][4][0] From the original element, we would go to the child located in the first position, that child's fourth position child, and that child's zero position child.<br><br>When direct referencing child elements in this way, they still have access to the Tag, attrib, and text properties. | `root[2].tag`<br>`root[3].text` | department<br><br>Tech Support |

## A Simple Book Example

Using the functions above, we can now easily parse through an XML file and use the results for something. Lets keep it simple, and parse through a document and then place the values into a table. First we need to start with an XML document. We have one below for you to test with in a string form, which would need to be pasted at the top of the script.

**XML String**

```
document = """
<catalog>
        <book id="bk101">
                <author>Gambardella, Matthew</author>
                <title>XML Developer's Guide</title>
                <genre>Computer</genre>
                <price>44.95</price>
                <publish_date>2000-10-01</publish_date>
                <description>An in-depth look at creating applications
                with XML.</description>
        </book>
        <book id="bk102">
                <author>Ralls, Kim</author>
                <title>Midnight Rain</title>
                <genre>Fantasy</genre>
                <price>5.95</price>
                <publish_date>2000-12-16</publish_date>
                <description>A former architect battles corporate zombies,
                an evil sorceress, and her own childhood to become queen
                of the world.</description>
        </book>
        <book id="bk103">
                <author>Corets, Eva</author>
                <title>Maeve Ascendant</title>
                <genre>Fantasy</genre>
                <price>5.95</price>
                <publish_date>2000-11-17</publish_date>
```

```
        <description>After the collapse of a nanotechnology
        society in England, the young survivors lay the
        foundation for a new society.</description>
</book>
<book id="bk104">
        <author>Corets, Eva</author>
        <title>Oberon's Legacy</title>
        <genre>Fantasy</genre>
        <price>5.95</price>
        <publish_date>2001-03-10</publish_date>
        <description>In post-apocalypse England, the mysterious
        agent known only as Oberon helps to create a new life
        for the inhabitants of London. Sequel to Maeve
        Ascendant.</description>
</book>
<book id="bk105">
        <author>Corets, Eva</author>
        <title>The Sundered Grail</title>
        <genre>Fantasy</genre>
        <price>5.95</price>
        <publish_date>2001-09-10</publish_date>
        <description>The two daughters of Maeve, half-sisters,
        battle one another for control of England. Sequel to
        Oberon's Legacy.</description>
</book>
<book id="bk106">
        <author>Randall, Cynthia</author>
        <title>Lover Birds</title>
        <genre>Romance</genre>
        <price>4.95</price>
        <publish_date>2000-09-02</publish_date>
        <description>When Carla meets Paul at an ornithology
        conference, tempers fly as feathers get ruffled.</description>
</book>
<book id="bk107">
        <author>Thurman, Paula</author>
        <title>Splish Splash</title>
        <genre>Romance</genre>
        <price>4.95</price>
        <publish_date>2000-11-02</publish_date>
        <description>A deep sea diver finds true love twenty
        thousand leagues beneath the sea.</description>
</book>
<book id="bk108">
        <author>Knorr, Stefan</author>
        <title>Creepy Crawlies</title>
        <genre>Horror</genre>
        <price>4.95</price>
        <publish_date>2000-12-06</publish_date>
        <description>An anthology of horror stories about roaches,
        centipedes, scorpions  and other insects.</description>
</book>
<book id="bk109">
        <author>Kress, Peter</author>
        <title>Paradox Lost</title>
        <genre>Science Fiction</genre>
        <price>6.95</price>
        <publish_date>2000-11-02</publish_date>
        <description>After an inadvertant trip through a Heisenberg
        Uncertainty Device, James Salway discovers the problems
        of being quantum.</description>
</book>
<book id="bk110">
        <author>O'Brien, Tim</author>
        <title>Microsoft .NET: The Programming Bible</title>
        <genre>Computer</genre>
        <price>36.95</price>
        <publish_date>2000-12-09</publish_date>
        <description>Microsoft's .NET initiative is explored in
        detail in this deep programmer's reference.</description>
</book>
```

```
        <book id="bk111">
                <author>O'Brien, Tim</author>
                <title>MSXML3: A Comprehensive Guide</title>
                <genre>Computer</genre>
                <price>36.95</price>
                <publish_date>2000-12-01</publish_date>
                <description>The Microsoft MSXML3 parser is covered in
                detail, with attention to XML DOM interfaces, XSLT processing,
                SAX and more.</description>
        </book>
        <book id="bk112">
                <author>Galos, Mike</author>
                <title>Visual Studio 7: A Comprehensive Guide</title>
                <genre>Computer</genre>
                <price>49.95</price>
                <publish_date>2001-04-16</publish_date>
                <description>Microsoft Visual Studio 7 is explored in depth,
                looking at how Visual Basic, Visual C++, C#, and ASP+ are
                integrated into a comprehensive development
                environment.</description>
        </book>
</catalog>
"""
```

We can then place a Table component and a Button component on the window, and place this script on the Button's actionPerformed event.

**Python - Complete XML Parsing**

```
# Start by importing the library
import xml.etree.ElementTree as ET

######
# Here is where you would paste in the document string.
# Simply remove this comment, and paste in the document string.
######

# We can then parse the string into useable elements.
root = ET.fromstring(document)

# This creates empty header and row lists that we will add to later.
# These are used to create the dataset that will go into the Table.
# We could fill in the names of the headers beforehand, since we know what each will be.
# However, this allows us to add or remove children keys, and the script will automatically adjust.
headers = []
rows = []

# Now we can loop through each child of the root.
# Since the root is catalog, each child element is an individual book.
# We also create a single row empty list. We can add all of the data for a single book to this list.
for child in root:
        oneRow = []

        # Check if the book has any attributes.
        if child.attrib != {}:

                # If it does contain attributes, we want to loop through all of them.
                for key in child.attrib:

                        # Since we only want to add the attributes to our header list once, first check if
it is there.
                        # If it isn't add it.
                        if key not in headers:
                                headers.append(key)

                        # Add the attribute value to the oneRow list.
                        oneRow.append(child.attrib[key])

        # Loop through the children of the book.
        for child2 in child:
```

```
                # Similar to above, we check if the tag is present in the header list before adding it.
                if child2.tag not in headers:
                        headers.append(child2.tag)

                # We can then add the text of the Element to the oneRow list.
                oneRow.append(child2.text)

        # Finally, we want to add the oneRow list to our list of rows.
        rows.append(oneRow)

# Once the loop is complete, this will print out the headers and rows list so we can manually check them in
the console.
print headers
print rows

# Convert to a dataset, and insert into the Table.
data = system.dataset.toDataSet(headers, rows)
event.source.parent.getComponent('Table').data = data
```

Related Topics ...

- system.file.openFile
- Libraries

# Audit Log and Profiles

Ignition's built-in auditing system automatically records certain actions that occur in the system, such as a Tag writes or User Source authentication, into a SQL database table. Utilizing the system involves creating an Audit Profile, followed by enabling auditing in a project. Once both prerequisites have been met, the Gateway will automatically create a database table named AUDIT_EVENTS, and use the table to start tracking user actions.

The Remote Audit Log configuration option allows audit events to be automatically sent to a remote Gateway's audit log. The remote Gateway you plan to connect to must have a Audit Profile created. To learn more about sending audit events to a remote Gateway, refer to section Creating a Remote Gateway Audit Profile on this page.

### Auditing Actions

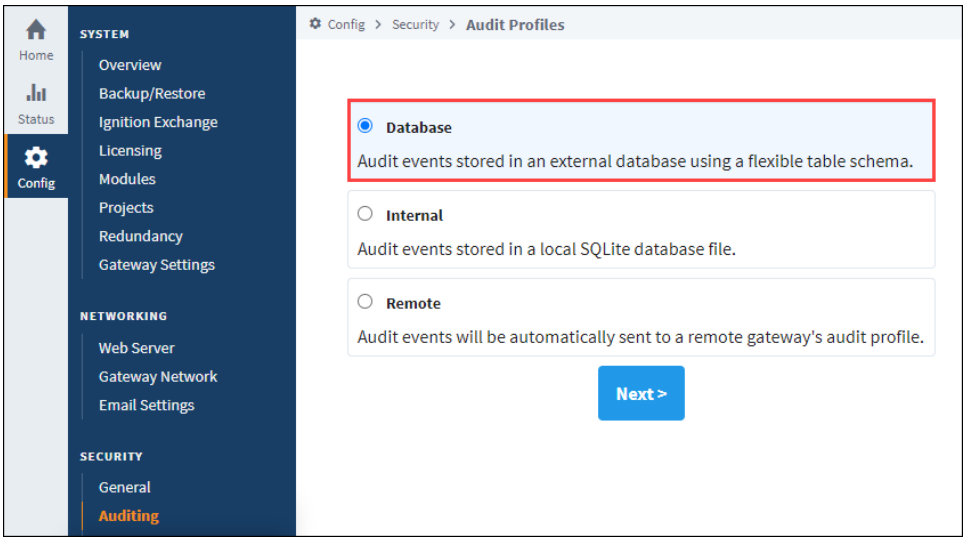
For a list of actions that are recorded by an audit profile, see the Auditing Actions Reference page.

## On this page ...

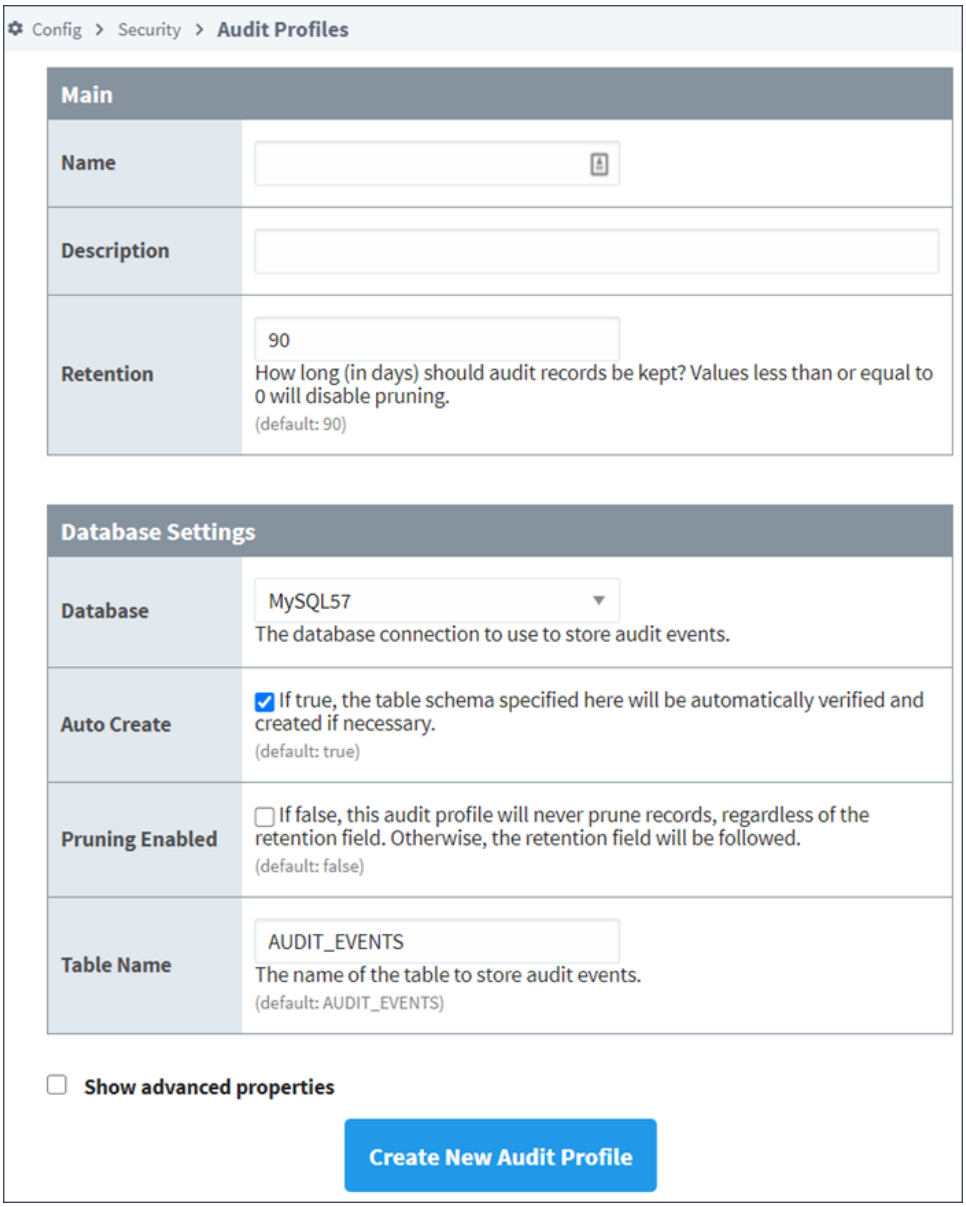


## Create a Database Audit Profile

1. Go to the **Config** section of the Gateway Webpage.
2. Scroll down to the **Security > Auditing** from the menu on the left.  The Audit Profiles page is displayed.
3. Click the **Create a new Audit Profile** link.
4. You have the option of storing audit logs into an external database or sending them to a remote Gateway. For this example, select **Database**. (Configuring audit events to be sent to a remote Gateway's audit log is addressed in Creating a Remote Gateway Audit Profile section on this page).




5. Enter the **Name** of the audit log and **Description** (optional).
6. In the Retention field, set a value in days for how long you want audit records kept. (The default is 90 days.)
7. Under the **Database Settings**, select the **Database** where the table will be stored, select the **Auto Create** check box, and enter the desired **Table Name**.
8. Click **Create New Audit Profile**.




Once some changes have been made to a Tag or a Database table, Ignition will begin recording.

| AUDIT_EVENTS_ID | EVENT_TIMESTAMP | ACTOR | ACTOR_HOST | ACTION | ACTION_TARGET | ACTION_VALUE |
|---|---|---|---|---|---|---|
| 1 | 2016-07-25 17:50:09 | admin | IU-WorkStation | tag write | B Tags/B3:1 | 1.0 |
| 2 | 2016-07-25 17:50:51 | admin | IU-WorkStation | tag write | B Tags/B3:1 | 100.0 |
| 3 | 2016-07-25 17:50:53 | admin | IU-WorkStation | tag write | B Tags/B3:1 | 2.0 |
| 4 | 2016-07-25 17:50:56 | admin | IU-WorkStation | tag write | B Tags/B3:1 | 8.0 |
| 5 | 2016-07-25 17:51:20 | admin | IU-WorkStation | query | update audit_events set acto... | 4 |
| 6 | 2016-07-25 17:51:51 | admin | IU-WorkStation | query | UPDATE audit_events SET `A... | 1 |

### Database Audit Profile Properties Table

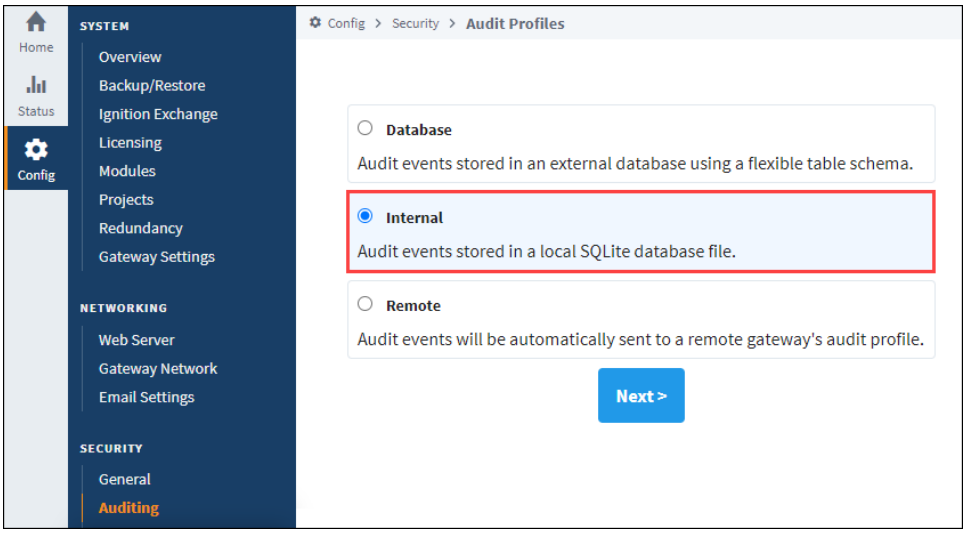
| **Main** | |
|---|---|
| Name | The default name, is the name of the Audit Profile. |
| Description | Description of the audit profile. Optional. |

| | |
|---|---|
| Retention | This feature is new in Ignition version **8.1.1**<br>Click here to check out the other new features<br><br>How long (in days) should audit records be kept? Values less than or equal to 0 will disable pruning. **Default is 90 days.** |
| **Database Settings** | |
| Database | The database connection to use to store audit events. |
| Auto Create | If true (selected), the table schema specified here will be automatically verified and created if necessary. Default is true. |
| Pruning Enabled | This feature is new in Ignition version **8.1.3**<br>Click here to check out the other new features<br><br>If false, this audit profile will never prune records, regardless of the retention field. Otherwise, the retention field will be followed. De fault is false. |
| Table Name | The name of the table to store audit events. Default is AUDIT_EVENTS. |

## Create an Internal Audit Profile

The Internal Audit Profile option allows an Ignition Gateway to store audit records without an external SQL database. The only way to interact with the Internal Audit Profile is via the Status page of the Gateway webpage.

1. Go to the **Config** section of the Gateway Webpage.
2. Scroll down to the **Security > Auditing** from the menu on the left.  The Audit Profiles page is displayed.
3. Click the **Create a new Audit Profile** link.
4. Select **Internal**.




5. Enter a name for the audit log and a description (optional).
6. In the Retention field, set a value in days for how long you want audit records kept. (The default is 90 days.)
7. Click **Create New Audit Profile**.

### Internal Audit Profile Properties Table

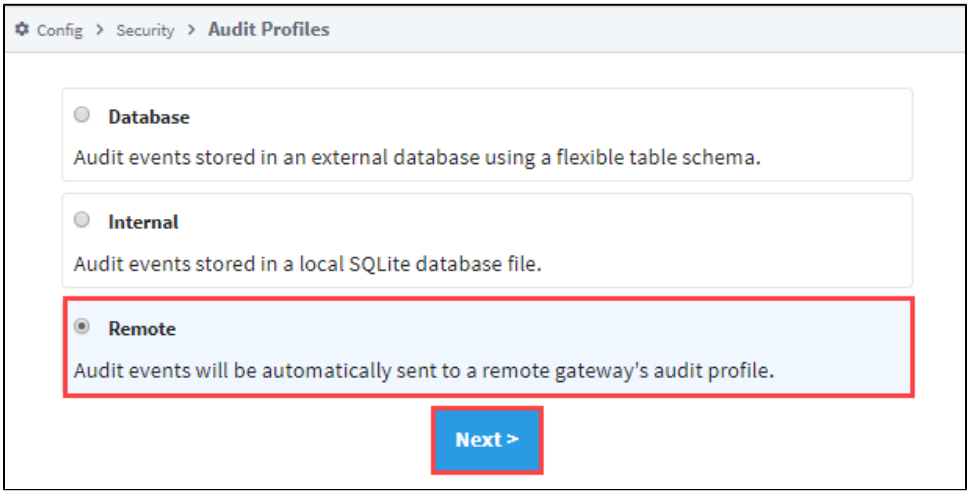
| | |
|---|---|
| **Main** | |
| Name | The default name, is the name of the Audit Profile. |
| Description | Description of the audit profile. Optional. |
| Retention | This feature is new in Ignition version **8.1.1**<br>Click here to check out the other new features |

| | |
|---|---|
| | Value in days for how long you want audit records kept. (The default is 90 days.) |

## Creating a Remote Gateway Audit Profile

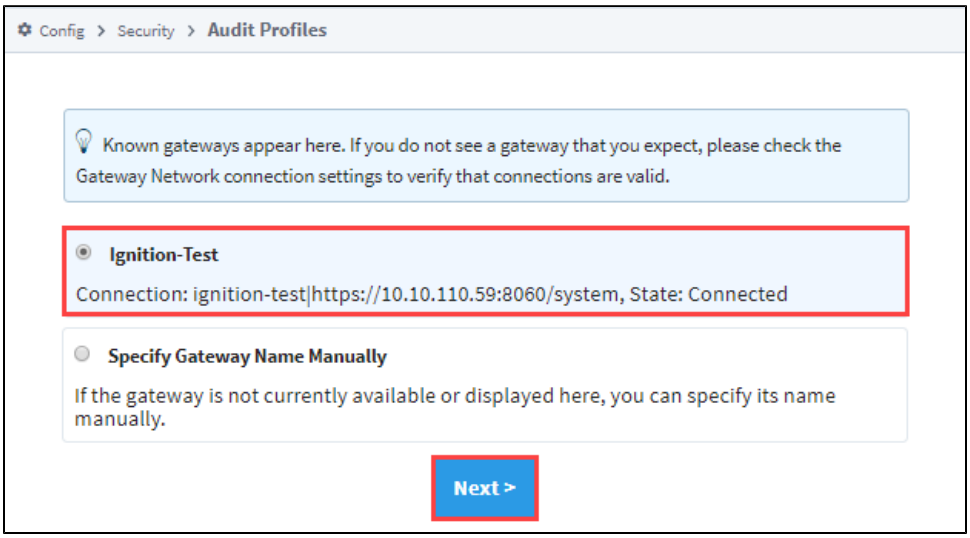
Just like configuring audit events to be logged into an external database, it is done from the Gateway Webpage, **Config > Security > Auditing.**

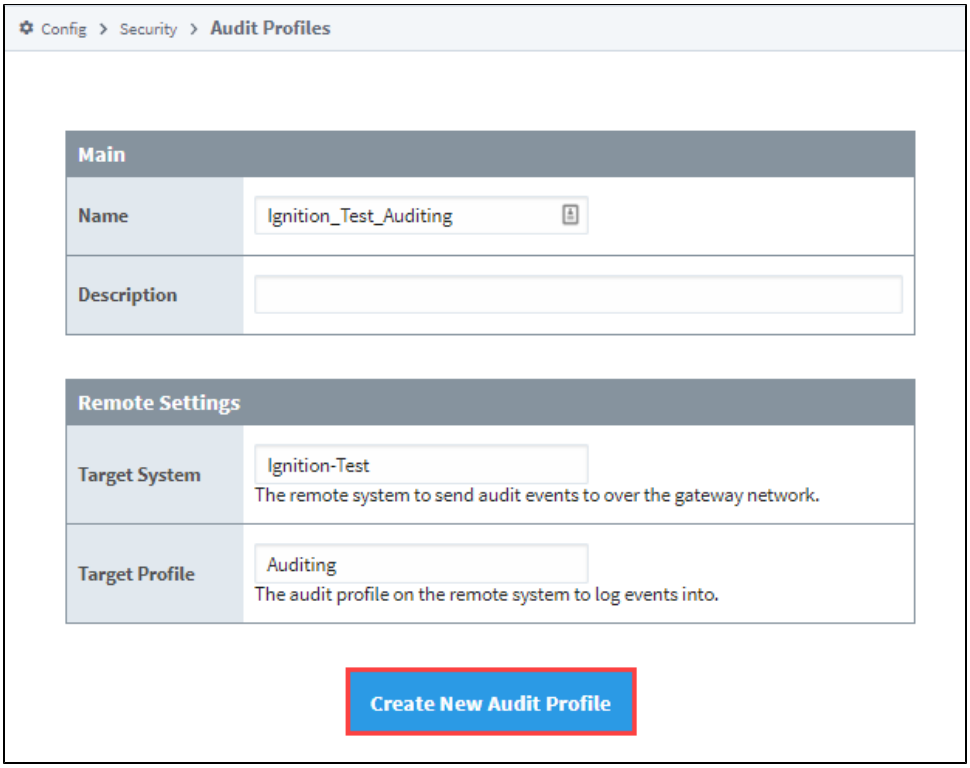
1. To have your audit events automatically sent to a remote Gateway's audit profile, select **Remote**, and click **Next**.



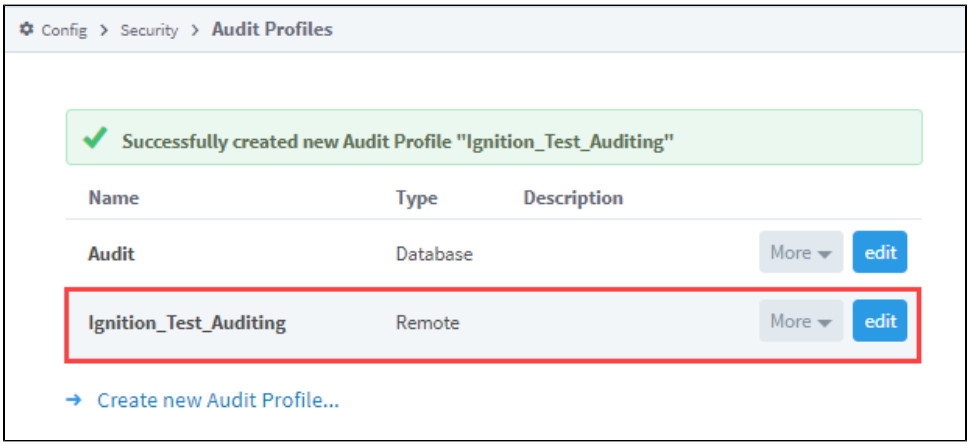

2. A list of known Gateways will be displayed. If you don't see a Gateway that you expected to see, check your Gateway Network settings to verify that the connections are valid. You also have the option to specify a Gateway manually. This example selects a valid Gateway. Click **Ne xt**.

3. If an Audit profile exists, the fields will auto-populate. The name of the Gateway will appear in the **Name** field prefaced with the audit profile name (i.e., Ignition_Test_Auditing), as shown in the following example. Click **Create New Audit Profile**.






4. You will receive a successful message stating your new Audit Profile was created.



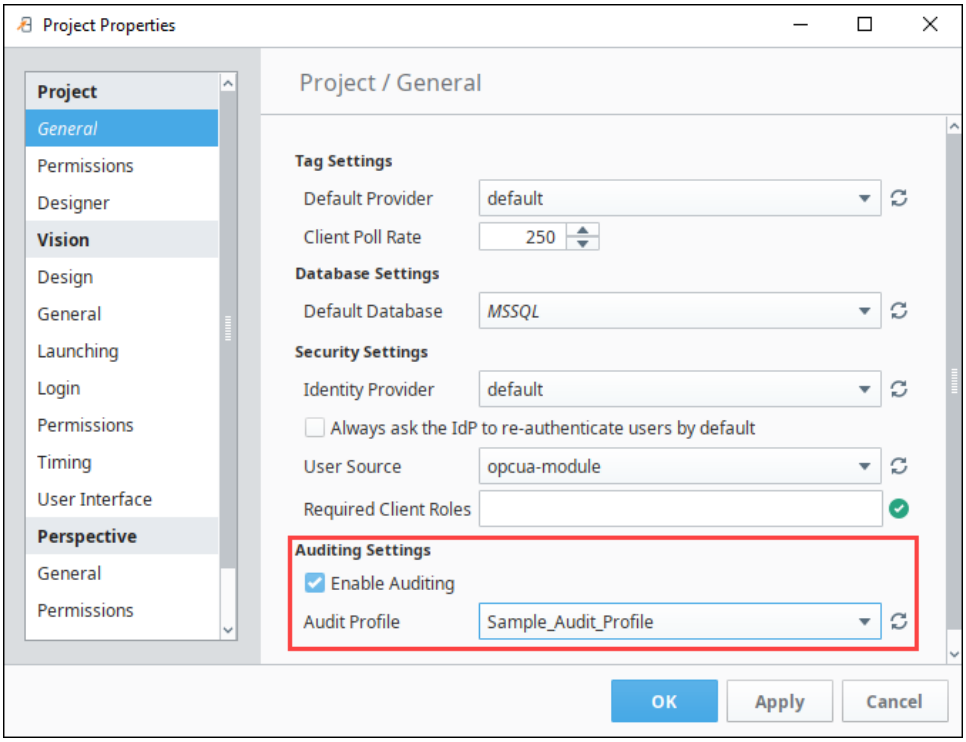

### Remote Gateway Audit Profile Properties Table

| **Main** | |
|---|---|
| Name | The default name, is the name of the Remote Gateway and Audit Profile. |
| Description | Description of the audit profile. Optional. |
| Enabled | By default. the journal profile is enabled. |
| **Remote Settings** | |
| Target System | The remote system to send audit events to over the Gateway network. |
| Target Profile | The audit profile on the remote system to log events into. |

## Enabling Auditing in a Project

1. Go to the Designer, open the project that you want to enable auditing on, then go to **Project > Properties**.

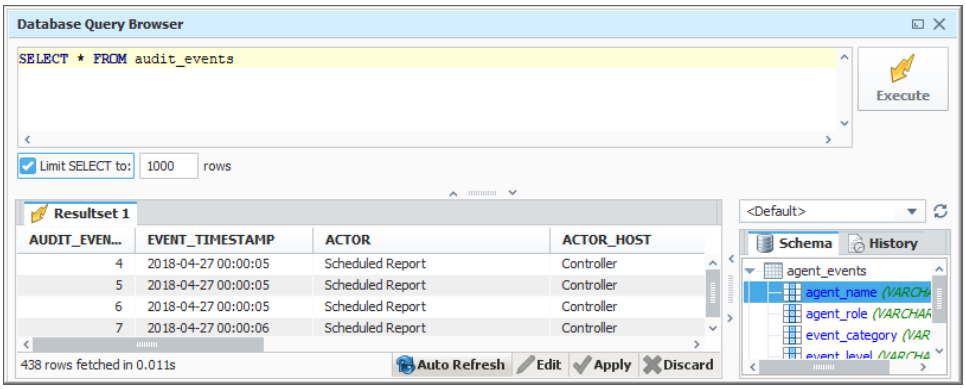
2. Go to the General section, select the **Enable Auditing** check box, and select your **Audit Profile** from the drop-down menu. The audit profile is used to record audit actions for your project. If the new audit profile does not show up, click **Refresh**.
3. Click **OK**.
4. **Save** your Project.




## Viewing Information in an Audit Log

There are a few ways to view audit information: using a Table component, interface on the Gateway, or the Database Query Browser. Here is one example of viewing an Audit Log using the Database Query Browser.

1. In the Designer, go to **Tools > Database Query Browser.**
2. Under the **Schema** area, double click on a **table**, and it will expand the query in the Database Query Browser area.
3. Click **Execute**. All the audit log data will be displayed in the Resultset1 area.




Related Topics ...

- Database Query Browser
- Audit Log Display

In This Section ...

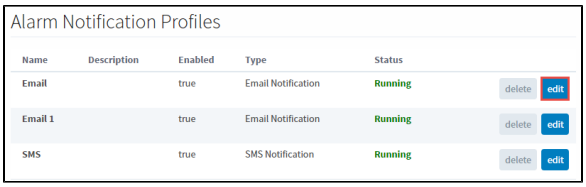
# Alarm Notification Auditing

Alarm Notification profiles can be set to store information in an Audit Profile.

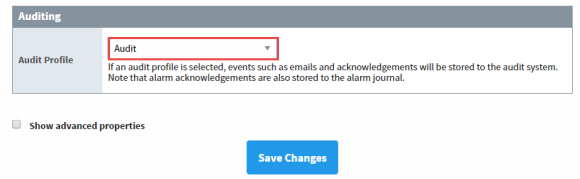
## How to Store Alarm Notifications in an Audit Profile

Once you have an audit profile created in Ignition, you can configure your Alarm Notification Profile to start using it.

1. Go to the **Config** section of the Gateway.
2. Choose **Alarming > Notification** from the menu on the left.
   The Alarm Notification Profiles page is displayed.
3. Edit the appropriate notification profile by clicking the **edit** link.




4. The Edit Alarm Notification Profile page is displayed.
   Scroll down to the **Auditing** section, and select an **Audit Profile** from the drop-down menu.
   Click **Save Changes**, and Ignition will automatically begin storing information.




Now that the Alarm Notification Profile is storing data into the Audit system, you have a complete log of all alarm emails and acknowledgements that you can review.
See Audit Log Display for more info on how to retrieve information from the Audit Log.

Related Topics ...

- Alarm Notification
- Database Connections
- Audit Log Display

**On this page ...**



# Audit Log Display

This page documents several ways to view results from the Audit Log System. For more information on how the Audit Log works, see the Audit Log and Profiles page.

## On this page ...




**View Audit Information**

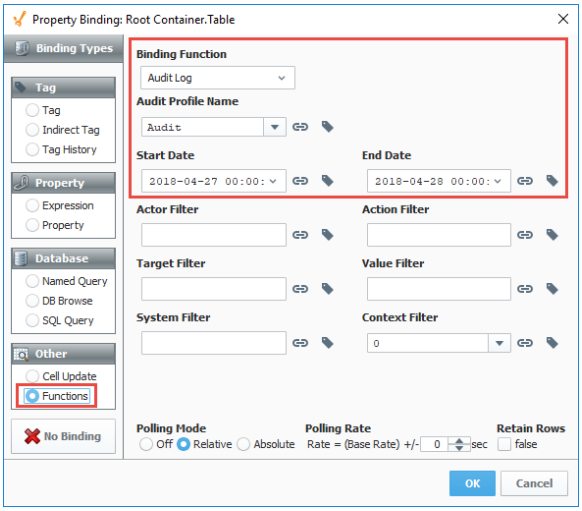
Watch the Video

## Access the Audit Log with Table Functions

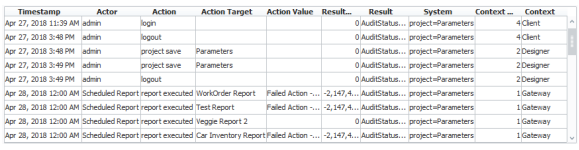
Since Ignition makes accessing data from databases seamless, it is possible to bind a data property of a table directly to database table. Alternatively, it is possible to access the contents of the audit log with table functions.

Once you have an Audit Log setup and attached to a project, you can go back to that project and see what information is in that audit log. This example uses the Table Functions to extract data from the Audit Log.

1. In **Designer**, drag a **Table** component on to a window.
2. In the **Property Editor**, click on the binding icon from the table's **Data** property.
   The Property Binding window is displayed.
3. Select the **Functions** Binding Type.
4. Select **Functions** from the various binding options.
   a. **Binding Function**, select the **Audit Log** from the drop-down menu.
   b. **Audit Profile Name**, select **Audit** or the name of your Audit Profile.
   c. **Start Date**, enter the appropriate start date.
   d. **End Date**, enter the appropriate end date.
   e. **Polling Mode**, select **Relative**.
   f. Select any other appropriate function options from the menu, and click **OK**.




5. The table will populate with all the information stored in the audit log based on the Table Functions you selected. You can use the Table Customizer to configure how you want the table to look by reorganizing and hiding columns, making columns sortable, assigning meaningful headers, and much more.

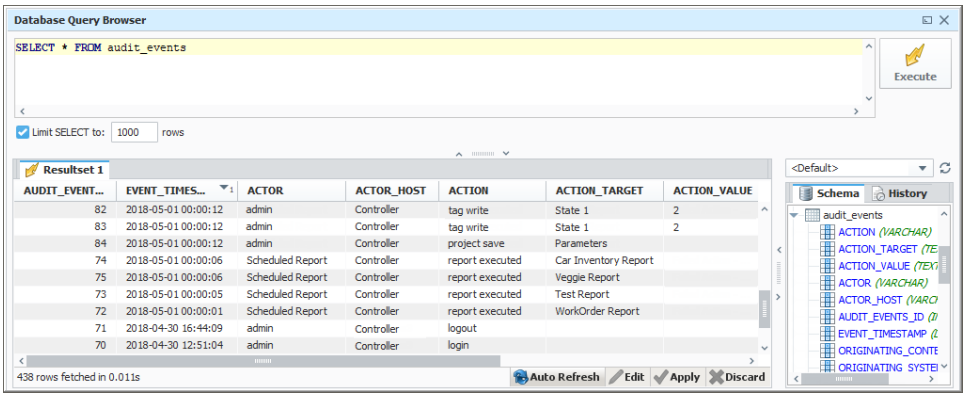



## Access the Audit Log Using the Database Query Browser

The Database Query Browser makes it easy to search your database tables to view audit information.

1. In the **Designer** under the **Tools** menubar, select **Database Query Browser**. The Database Query Browser opens.

2. On the right side of Browser window under **Schema**, will be a list of tables from the currently selected database. Double click on the **audit_events** table, and click **Execute.**
   The data from the audit_events table will appear in the Resultset 1 tab.



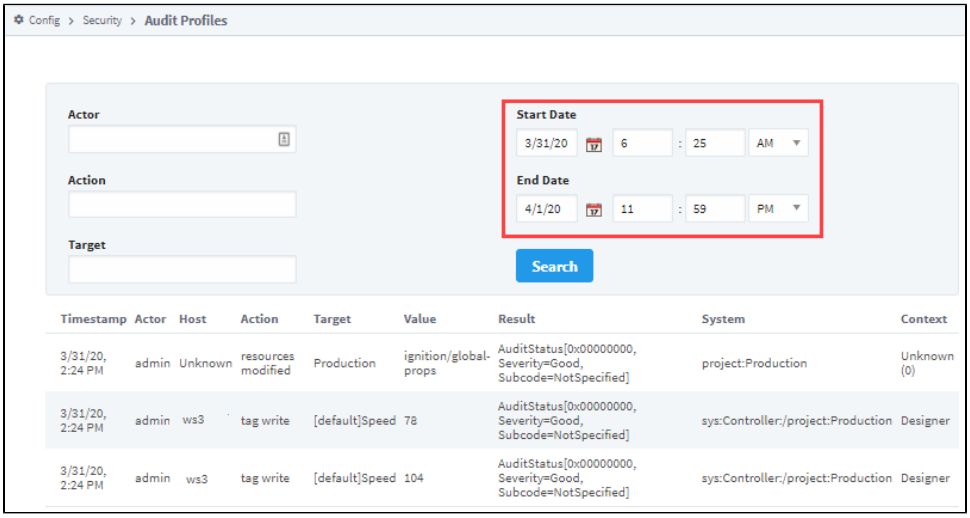

Refer to the Database Query Browser to learn more about how it works.

## Access the Audit Log on the Gateway

Ignition provides a simple interface to view Audit Logs on the Gateway.

1. On the Gateway webpage in the **Config** section, scroll down to **Security > Auditing**.
2. The Audit Profile page will be displayed. Select the Audit Profile where your information is stored, and click **More > view log**.


3. Choose the parameter settings if you're looking for something specific, otherwise, enter a **Start Date** and **End Date**, and click **Search**.




Related Topics ...

- Scripting
- Database Query Browser

- Table Customizer
- Security

# Auditing Actions Reference

The auditing system in Ignition records actions originating from the Gateway, Perspective and Vision projects. This page lists which actions are logged by the auditing system.

## On this page ...



## Gateway Audit Actions

The following actions are recorded in an audit log when the Gateway has a Gateway Audit Profile is configured.

### Project System

The following project-based actions are tracked by the auditing system.

- Project Property changes made from the Designer.
- Project setting changes made from the gateway's web interface.
- Creating or deleting a project.
- Saving a project (action recorded as "project update").

In addition, project files on the Gateway's file system are closely monitored. If a user or third-party system modifies any of the project files, an entry will be recorded in the auditing system.

The following Gateway-level actions are recorded in an audit log when the Gateway has a Gateway Audit Profile is configured.

### Modules

- Installing modules on the Gateway.
- Restarting a module.
- Deleting a module.

### Gateway - General

- Gateway startup
- Gateway shutdown (assuming the gateway was requested to shutdown: unintended shutdowns from power failures and such will not be recorded).

### Gateway - Restore

- Restoring the Gateway from a Gateway backup. Specifically, the Gateway will log that it was asked to before a restore, then perform the restoration.

### Licensing Changes

- Activating a license.
- Unactivating a license.
- Updating a license.

### Redundancy

- Saving after making any changes to the Redundancy Settings page.

### Web Server Page

- Installing or removing a security certificate.
- Making changes to the Web Server Settings page.

### Identity Providers - User Authorization

- Login requests and responses.
- Logout requests and responses.

### Gateway Network

- Saving changes to Gateway Network General Settings .
- Creating, editing, or deleting outgoing connections.

- Approving incoming connections.

### Email Settings

- Creating, editing, or deleting an SMTP Profile.

### Audit Profile

- Creating, editing, or deleting an Audit Profile.

### User Sources

- Creating, editing, or deleting a User Source.
- Creating, editing, or deleting a user.
- Creating, editing, or deleting a role.

### Service Security

- Editing and saving a policy.

### Identity Providers

- Creating, editing, or deleting an Identity Provider configuration.
- Making changes to a User Attribute Mapping.
- Creating, editing, or deleting a User Grant.
- Saving changes on a Security Level Rule.

### Security Levels

- Creating, editing, and deleting security zones

### Security Zones

- Creating, editing, and deleting security zones

### Database - Connections

- Creating, editing or deleting a database connection.

### Database - Drivers

- Creating, editing or deleting a JDBC driver
- Creating, editing or deleting a Translator

### Store and Forward

- Creating, editing or deleting a Store and Forward engine.

### Alarming - General

- Saving changes on the Alarming General settings page.

### Alarming - Alarm Journal

- Creating, editing or deleting an Alarm Journal Profile

### Alarming - Notification

- Creating, editing, or deleting an Alarm Notification Profile.

### Schedules

- Creating, editing or deleting a schedule.
- Creating, editing or deleting a holiday.

### Tags - Realtime

- Creating, editing or deleting a Realtime Tag Provider

### Tags - Historical

- Creating, editing or deleting a Historical Tag Provider.

### OPC Client Connections

- Creating, editing or deleting an OPC connection.

### OPC UA - Device Connections

- Creating, editing, or deleting a device connection (editing/saving a device connection configuration without making any changes will be recorded as an edit).
- Editing a Modbus address mapping via gateway interface on Modbus device connections.
- Editing DNP3 Aliased Points via gateway interface on DNP3 device connections.
- Editing tags via gateway interface on Omron NJ device connections.
- Adding FINS tags via gateway interface on Omron FIN device connections.

### OPC UA - Server Settings

- Editing the OPC UA Settings page.

### Enterprise Administration

- Configuring a gateway to be either an Agent or Controller.

### Enterprise Administration - Event Thresholds

- Changes made to Event thresholds.

### Enterprise Administration - Controller Settings

- Making changes to the Controller Settings page, including uninstalling the controller.

### Enterprise Administration - Agent Settings

- Making changes to the Agent Settings page, including uninstalling the agent.

### Enterprise Administration - Agent Management

- Creating, editing, deleting an Agent Group.

### Enterprise Administration - License Management

- Adding or removing a license from the License Management page.

### Enterprise Administration - Agent Tasks

- Creating, editing, or deleting an agent task
- Separate records are taken each time a task executes.

### Sequential Function Charts

- Changes made to the SFC Settings page.

### Add a Record Manually

- You can also add a record into the audit profile using the function system.util.audit.

## Perspective Auditing Actions

Perspective Sessions generate entries in an assigned audit profile. The following actions are recorded in the Audit Profile:

- Tag changes from a component binding.
- Session login.
- Session logout.
- Authentication level changes (a user's security level changes).

# Vision Auditing Actions

The Vision project needs an audit profile configured and auditing enabled. Vision Clients will then log records to an assigned audit profile. Here is a list of audit actions that will be tracked in the Ignition auditing system:

## Tags

The following Tag related actions generate entries in the audit log. Note that the functions below must originate from the Tag Browser, the Designer's Scripting Console, or Vision component-based scripts.

- Tag Creation - Including tags created with the Tag Editor and the system.tag.configure function.
- Tag Deletion - Including those deleted from the Tag Browser's UI and the system.tag.deleteTags function.
- Tag Edits - Including edits made to tags from the Tag Editor and the system.tag.configure function.
- Moving Tags - Including moves made by drag-and-drop in the Tag Browser or by calling the system.tag.move function.
- Tag Renames - Renaming a tag generates an entry.

## Vision Tag Writes

Write requests sent from a tag either through a standard Tag Binding, Indirect Tag Binding, or manual entry from the Tag Browser.

## Vision Component Database Writes

The system explicitly captures modifications made to database tables through the following methods:

- SQL Query Bindings - modifications from the UPDATE Query will be recorded.
- DB Browse Binding - modifications made with the Enable Database Writeback area will be recorded.

## Vision User Login/logout

- Logging into a Vision Client will generate an entry in the auditing system, as will logging out of a client.
- Closing the client while logged in is treated as a logout, and will generate a "logout" entry. Note that a logout entry is only recorded if the client is aware that it is closing: entries aren't generated if the designer closed unexpectedly ("crashes").

## Database Query Browser

- If the project opened in the Designer has an assigned Audit Profile, then changes made to database tables using the database query browser are automatically recorded to the audit log. "Changes" in this case refer to UPDATE, DELETE, or INSERT statements manually typed and executed from the database query browser.
- Enabling edit mode and applying changes, including typing in new values, adding rows, removing rows, and clearing out fields, are recorded as queries called from the project.

## Vision Scripting

The following functions generate entries in the audit log if called from Vision component-based scripts, or from the Designer's Scripting Console.

- system.db.execSProcCall
- system.db.runPrepUpdate
- system.db.runUpdateQuery
- system.tag.writeBlocking
- system.tag.writeAsync
- system.report.executeAndDistribute
- system.report.executeReport

# Designer

## Designer Login and Closing

- Opening a project in the Designer that has auditing enabled will also generate a login entry in the auditing system. Note that this occurs when the user opens the project, not when they log in using the Designer's login screen: auditing is project-based, so the user has to select a project that is being edited first.
- Closing the Designer effectively counts as logging off, and will generate a "logout" entry. Similar to vision, should the designer close unexpectedly, then an entry will not be recorded.

## Database Query Browser

If the project opened in the Designer has an assigned Audit Profile, then changes made to database tables using the database query browser are automatically recorded to the audit log. "Changes" in this case refer to UPDATE, DELETE, or INSERT statements manually typed and executed from the database query browser.

Enabling edit mode and applying changes, including typing in new values, adding rows, removing rows, and clearing out fields, are recorded as queries called from the project.

# Alarm Notification

## Alarm Notification Attempts

Attempts to send out alarm notifications are recorded in the auditing system. Specifically, the Gateway will record when it attempted to send out a notification, as well as if the attempt failed (such as the SMTP server refusing the request). It is important to note that the auditing system can not report failures that occur outside of the Gateway. Thus, if a voice notification fails to send due to some error in the VOIP system, it's possible that the Gateway won't report the VOIP error, but the audit log will have an entry stating that the Gateway attempted to send the notification.

# Reporting Module

## Report Execution

Reporting Module Reports generate an entry in the auditing system when a report is executed. Thus:

- Reports running on a schedule will generate an entry.
- Report schedules executed on demand will generate an entry.
- Navigating to a Vision window (in either the Designer or a Vision Client) will trigger a report execution, generating an entry in the auditing system.

# Audit Table Definition

The following table describes the audit table as it exist in the database:

| Column Name | Description |
|---|---|
| AUDIT_EVENTS_ID | The id of the row. |
| ACTION | Brief description of the action. |
| ACTION_TARGET | The target of the action. |
| ACTION_VALUE | The value acted upon the action target. |
| ACTOR | The logged in user when the action occurred or a description of the system that generated the action. |
| ACTOR_HOST | The host computer where the action occurred. |
| EVENT_TIMESTAMP | The time when the action occurred. |
| ORIGINATING_CONTEXT | A numerical description of the origin of the originating system. gateway = 1, designer = 2, client = 4 |
| ORIGINATING_SYSTEM | The name of the project or system where this action occurred. |
| STATUS_CODE | The quality code where (where applicable). 192 signifies success. 0 signifies bad or failure. |

Related Topics ...

- Audit Log and Profiles
- Audit Log Display