Chuco Gives a FAQ 2024: Python Sensor Authoring Guide

Version: 1.0

Last Update: Oct 22nd, 2024

Table of Contents

When to use Python for sensor code

Python sensors are useful anytime you need a cross-platform solution or anytime you are doing something that Python is natively good at. For example, if I need to parse a JSON file or query some parameter across Windows, Linux and Mac. If you need to support AIX or Solaris, then you will need to write separate code for those OS's.

Python sensors are inherently slower than other options. While we might be talking fractions of a second for a simple script, that does add up over thousands of clients. So if you're writing a sensor that can't particularly benefit from using Python, Powershell or a shell script might be your best choice.

Core Python

Tanium Core Python is the underlying layer that supports execution of python code from the Tanium client. It includes a Python module called `tanium` which is used to access some Tanium client functions and return sensor results.

Writing Python Sensors

Note: *When writing a python sensor, you can only use modules built into the Python Standard Library.*

Below are some examples for writing Python sensors including how to return results, handle parameters, and access some Tanium client functions.

Basic Structure

This is the basic structure of a Python sensor. Note that `print()` will not return results to the console, instead, you need to add results via `tanium.results.add()`.

We are also cleanly handling exceptions and returning them to the console as a TSE-Error.

Python

```python
import traceback
import tanium

def format_error_message(str_in):
    # Formats the error message string with the "TSE-Error:" prefix.
    return f'TSE-Error: {str_in}'

def sensor_main():
    # This is where your code goes.
    result = "Do Something"
    return result

if __name__ == "__main__":
    try:
        r = sensor_main()
    except Exception:
        r = format_error_message(traceback.format_exc().replace("\n", "->"))
    finally:
        tanium.results.add(r)
```

Multiple Results

The results class can accept strings, objects and iterables. So if you have a list of results, you can add the list directly:

```python
def sensor_main():
    # You can return a list.
    result = ["Do Something", "Do Another Thing"]
    return result
```

You can also add results multiple times from within `sensor_main()` rather than return results from the function:

```python
def sensor_main():
    # Adding results from within sensor_main().
    colors = ["blue", "green", "red"]
    for color in colors:
        tanium.results.add(color)
```

To split results into multiple columns, just like any other language, use a delimiter:

```python
def sensor_main():
    # Using | as a delimiter.
    column_data = ["data_a", "data_b", "data_c"]
    result = "|".join(column_data)
    return result
```

Parameterized Sensors

Sensor parameters must be url decoded. For more details, see Tanium's Documentation. The `tanium` module has a convenient way of handling parameters using the `sensor_input.SensorInputs()` class.

Let's say we have four parameters with the types of: text, number, checkbox, and date time. These parameters will be substituted into the scripts as URL encoded strings, those are the values below enclosed in double pipes. (e.g. `||text||`) Oftentimes we will want to convert those parameters into different types better suited for Python. In most cases we can just do a simple type conversion. (e.g. `int(inputs.get_param('number'))`) But in the case of date time, we need a function to help us convert the 13 digit (Python expects 10) epoch to a datetime object:

Python

```python
from datetime import datetime
from tanium.sensor_io import sensor_input

inputs = sensor_input.SensorInputs()
inputs.add_param('text', '||text||')
inputs.add_param('check', '||check||')
inputs.add_param('number', '||number||')
inputs.add_param('date_time', '||date_time||')

def epoch_to_datetime(epoch, str_format = "%Y-%m-%d %H:%M:%S"):
    # Convert epoch to datetime object, specifically handles the 13 digit epoch
provided by a tanium parameter.
    return datetime.fromtimestamp(int(epoch[0:9])).strftime(str_format)

def sensor_main():
    # Convert user inputs to Python types.
    text_input = inputs.get_param('text')
    check_box_input = bool(inputs.get_param('check'))
    number_input = int(inputs.get_param('number'))
    date_time_input = epoch_to_datetime(inputs.get_param('date_time'))
```

Getting the Tanium Directory Path

You can get the path to the Tanium Client directory from the `tanium` module using `tanium.client.common.get_client_dir()`.

get_client_dir(*subdir_path=None*)

You can specify a sub directory path when calling `get_client_dir`. This will return the full path along with the sub directory you specify. If the sub directory does not exist an OSError will be raised.
Note: *This is case sensitive on Linux clients.*

```python
import tanium
def sensor_main():
    # Get the Tanium Client root directory
    tanium_dir = tanium.client.common.get_client_dir()
    # Get a sub directory within the Tanium Client directory
    deploy_dir = tanium.client.common.get_client_dir('Tools/Deploy')
```

Using os.path

You can use `os.path` as you normally would in a python script. You can review the documentation for more info.
Note: *You can also use `pathlib`, but some `pathlib` functionality may be included in a version of Python beyond what is included in core content.*

```python
import Tanium
from os import path, makedirs

def sensor_main():
    # Get the Tanium Client root directory
    tanium_dir = tanium.client.common.get_client_dir()

    # Get a sub directory within the Tanium Client directory
    deploy_dir = path.join(tanium_dir, 'Tools', 'Deploy')
```

```
    # Get a file from a sub directory, basicly the same as above
    subprocess_log = path.join(tanium_dir, 'Tools', 'SoftwareManagement',
'logs', 'subprocess.log')
```

Query Sensor Results (Sensorception)

Sometimes you need to modify or further process the results of an existing sensor. Normally, you might just copy the sensor and modify the code. This method of copying sensors maintained by Tanium has some drawbacks, most notable is that they can break if the sensor relies on Tanium specific functionality. A good example of this is Custom Tags. The location of custom tags has changed in the past, breaking custom content. When appropriate, you can query the results of a Tanium sensor within a Python sensor using `tanium.evaluate_sensor_result_by_name()`.

Multi-row

Multi-row data can be split by the new line character. This allows you to iterate through each row using a loop, or the more pythonic list comprehension.

Python
```python
import tanium

def sensor_main():
    # This is where your code goes.
    custom_tags = tanium.evaluate_sensor_result_by_name("Custom Tags")
    custom_tags = custom_tags.split("\n")
    for tag in custom_tags:
        # Do Something
```

Multi-row and Multi-column

Let's say you want to get the first column of data from a sensor. (You can change the index at `application.split("|")[0]` to get other columns)

```python
import tanium

def sensor_main():
    # This is where your code goes.
    installed_applications = tanium.evaluate_sensor_result_by_name("Installed Applications")
    installed_applications = [application.split("|")[0] for application in installed_applications.split("\n")]
```

Parameterized

Parameterized sensors can be called by passing the sensor parameters as string key/value pairs.

```python
import tanium

def sensor_main():
    # This is where your code goes.
    result = tanium.evaluate_sensor_result_by_name("Custom Tag Exists", {"tag": "test", "exactMatch": "0"})
    return result
```