Written by Nicholas Robinson
7th May 2020

Overview

In this document, the use of the python library named "Winium" is discussed with an example of how it was used. The use case is very specific so the intent of the document is not to provide code or a tool that can be used as is, but rather to provide an example to help understand how to use the Winium Library.

The Winium library is a tool to help automate operation of Windows based programs, or to move things automatically, by "the hand of god" so to speak. In this case, it is a central processing unit doing the movement, not Diego Maradona.

Use Case

In order to get video camera still images from many different locations, it was necessary to log into each server, open the desired camera, take a snapshot, then save the photo. Because there were hundreds of locations and in order to shorten that process, I began to search for ways to perform the task automatically.

Environment

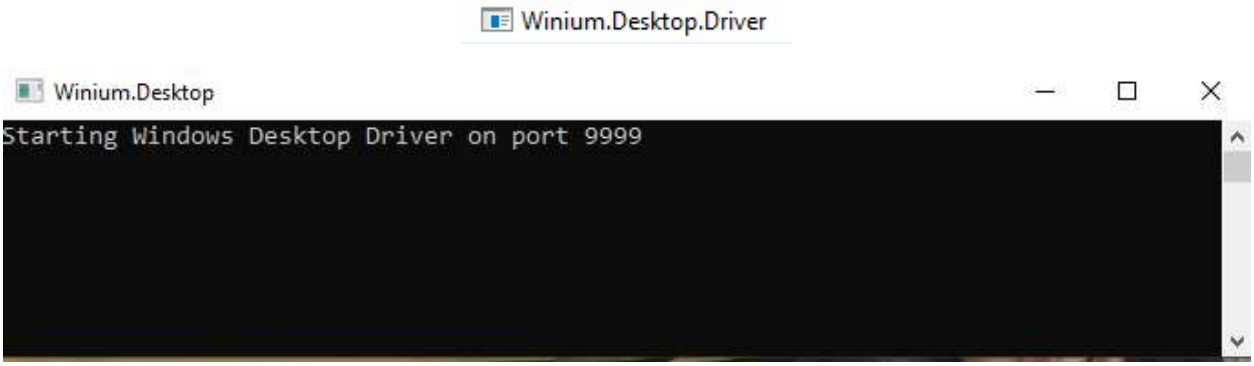
Although Winium states it can be run with a variety of different programming languages, this example is for python. Winium utilizes the selenium library in python with Winium itself running simultaneously as an independent executable. The official Winium desktop download is at the following link:

- https://github.com/2gis/Winium.Desktop

However, the above version will only work correctly if the desktop zoom is set to 100%, including primary and secondary screens. The result is that things on the screen become too small to look at. The version at the link below allows you to use Winium at whatever screen resolution you like:

- https://github.com/2gis/Winium.Desktop/issues/213

Before running your python script, make sure that you start Winium and that it is running. You can simply minimize it to keep it out of view.



The following python libraries are used in this example:

```
import os
import time
import tablib
import pyautogui
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
from selenium.common.exceptions import NoSuchElementException
```

"OS" is used to check if folders or files exist. "time" is used to measure how long certain process steps take to complete. "tablib" is used to grab data from different excel spreadsheets. "pyautogui" is used to make a mouse or keyboard action when finding method to do so using Winium/Selenium is either not possible or is more complex while still achieving the same behavior. The selenium "ActionChains" and "NoSuchElementException" classes must be imported to perform double clicks and for exception handling.

Timing

Because the program being controlled uses network resources, its display speed depends on the speed of the resource its contacting and the available speed of the network at that time. The challenge is that the python code will run faster than the windows program. When Winium attempts to find an element that is not available yet, it will create an exception, and the python script will stop. In order to compensate for this speed difference, the following two methods are used in this example:

1) While loops with try/except that runs until an element is found
2) Fixed length time pauses "time.sleep(seconds)"

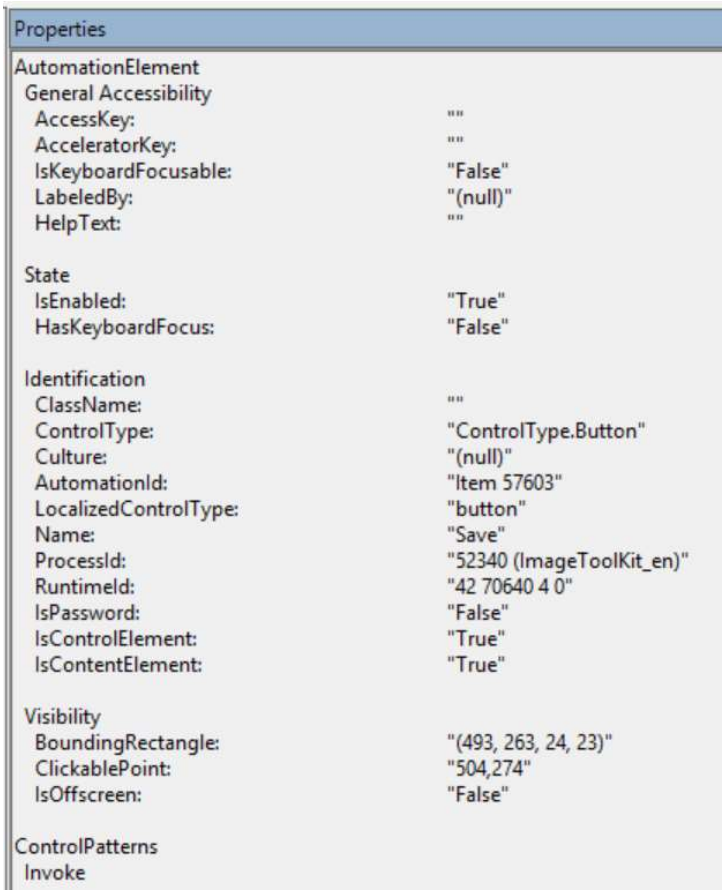
The latter is probably not the best or most sustainable way to write code, however for this example it proved effective in the fact that it made the code run more stable and with a higher probability to complete the assigned mission without stopping.

Finding Elements

In order to discover the names or automation ID of components on the target program, the tool “UI Spy” can be used.

- https://github.com/blackrosezy/gui-inspect-tool

Note that in some cases the name or ID of an element may be difficult to find or is named in a way that is not logical to its function so a certain amount of experimentation is needed to find the element that provides the use case with the needed functionality and stable running behavior. The following example shows the photo save button. Note the automation ID has no relation to what the button does. However, because there may be other buttons available to find on the screen with the name “Save”, it is better to use “Item 57603” with an ID search method because it is a unique identifier.



Code Description

The first part of this section will describe the class “Verint” which is the custom library made for this use case. Each defined function is shown with descriptions and key points described. At the end, an example is given for how it is initialized and called in a main program.

Initialization

This is a place to store variables for the class instance which can be accessed by all its functions. The part to point out is that in this case the provided store number could be 1, 2, 3, or 4 digits long but the store number in the URL always needs to be a fixed length so the necessary number of leading zeros are added. The actual text in the full URL definition is not shown for privacy reasons.

```
import os
import time
import pyautogui
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
from selenium.common.exceptions import NoSuchElementException

class Verint:

    def __init__(self, storeNum, password, saveDir):
        """automate getting data from verint review"""
        self.storeNum = str(storeNum)
        self.fullStoreNum = ''

        if (len(self.storeNum)) == 1:
            self.fullStoreNum = '000' + self.storeNum
        if (len(self.storeNum)) == 2:
            self.fullStoreNum = '00' + self.storeNum
        if (len(self.storeNum)) == 3:
            self.fullStoreNum = '0' + self.storeNum
        else:
            self.fullStoreNum = self.storeNum

        self.fullUrl = 'beginning' + self.fullStoreNum + 'end'

        self.password = password
        self.saveDir = saveDir
        self.driver = ''
        self.actions = ''
        self.searchWindow = ''
        self.previewErrorCount = 0
        self.exitStatus = ''
```

Start Driver

The path to the program executable is set here and this portion of the code is responsible to open the program. An instance of "ActionChains" is declared so that double clicks can be done.

```
def startDriver(self):
    """Initialize driver and actions"""
    self.driver = webdriver.Remote(
        command_executor='http://localhost:9999',
        desired_capabilities={
            "debugConnectToRunningApp": 'false',
            "app": r"C:\Program Files (x86)\Verint\Review\Review.exe",
            'launchDelay': '2'
        })
    self.actions = ActionChains(self.driver)
```

Login

This section handles the login screen. As a general statement, while Winium is running, if you try to do anything else it will upset it. For example, if while the login window is still opening, a popup window appears, the program would stop in error. By adding the try/except action, if something happens like a pop up, all that needs to be done is close the pop up then wait and the program will continue normally. For the password field, if a login to that server had been done before, the password automatically appears, and the field is inactive. In that case, attempting to enter a password will result in an error that prevents forward progress. The simple workaround is to check if the field is enabled beforehand.

```
def login(self):
    """Handles the login screen"""
    windowFound = False
    while(windowFound == False):
        try:
            time.sleep(3)
            loginWindow = self.driver.find_element_by_id('LoginView')
            url = loginWindow.find_element_by_class_name('Edit')
            url.clear()
            url.send_keys(self.fullUrl)

            password = loginWindow.find_element_by_id('m_Password')

            if password.is_enabled():
                password.clear()
                password.send_keys(self.password)

            loginButton = loginWindow.find_element_by_id('m_LoginButton')
            loginButton.click()
            windowFound = True
        except NoSuchElementException:
            continue
```

Open Search

After logging in, the program must contact network resources so the amount of time it takes to open varies greatly depending on location and time of day. To compensate for this, "Cameras" is searched for until found inside of a while(try/except) loop. The element "Cameras" is used because unless it is findable, the steps after it cannot be done. The "toggleButton" element switches between two methods of searching for videos. By clicking on the right side of the button, the desired search method is displayed. However, by default the mouse moves to the left side and in the discoverable elements, the two sides are not differentiated. The work around for this is to move the mouse 10 pixels to the right before performing the click action.

```
def openSearch(self):
    """Open the camera search pane"""
    windowFound = False
    while(windowFound == False):
        try:
            self.driver.find_element_by_name('Cameras')
            windowFound = True
        except NoSuchElementException:
            continue

    toggleButton = self.driver.find_element_by_name('ngToggleButton1')
    toggleButton.click()

    self.actions.move_by_offset(10, 0).perform()
    self.actions.click().perform()
    time.sleep(3)

    self.searchWindow = self.driver.find_element_by_id('m_TopWorkspace')

    expander = self.searchWindow.find_element_by_name('m_CameraNameExpander')
    expander.click()
```

Camera Search

This function types in the camera name, clicks search, and then double clicks the camera to open it. It is written as a separate function from open search so that in case a list of multiple cameras is given, it can cycle through and open them all. The double click is done using pyautogui because it was not clear which element to do it with in Winium. This works in case the screen size is always the same. If running on a different sized computer, then coordinates would need to be adjusted. The quickest way to get the coordinates is with the "Inspect.exe" tool.

```
def cameraSearch(self, cameraName):
    """Filter to and open a single camera view"""
    textBox = self.searchWindow.find_element_by_id('m_CameraNameTextBox')
    textBox.clear()
    textBox.send_keys(cameraName)

    searchButton = self.searchWindow.find_element_by_name('m_SearchBtn')
    searchButton.click()
    time.sleep(1)

    pyautogui.doubleClick(x = 50, y = 535)
```

Open Image

After double clicking to open a camera the image could appear quickly, take a long time to load, or not appear at all due to other errors. The first while(try/except) loop waits for the container which holds the camera view to open. However, this did not prove 100% effective which is why the 15 seconds of sleep time was added to give the image more time to open. Overall, this increased the time to get a single image. The goal was to reach the end without stopping for errors rather than super-quick operation so stable operation was given priority.

The button to open the snapshot function did not have a clear automation ID or name but luckily it did have a hot key (ctrl + I). After clicking (ctrl+ I), if the window is open but the image is not yet shown an error appears saying that a snapshot cannot be made on an inactive view, or that some other error is present. Rather than dealing with all of them one by one the simplest solution was a while(try/except) loop which searched for the snapshot save button "Item 57603" because if it could be found it meant that the snapshot image had successfully opened. This is tried 5 times and if not successful by then, an error flag is raised, and the loop quits trying. Pyautogui presses "enter" to close the error window and (ctrl + I) to try to open the snapshot window again.

```
def openImage(self, cameraName):
    """Opens the snapshot tool"""
    windowFound = False
    while(windowFound == False):
        try:
            preview = self.driver.find_element_by_id('NextivaVideoControlPlayBarOverlay')
            time.sleep(2)
            windowFound = True
        except NoSuchElementException:
            continue

    time.sleep(15)
    pyautogui.hotkey('ctrl', 'i')
    time.sleep(2)

    windowFound = False
    while(windowFound == False):
        try:
            save = self.driver.find_element_by_id('Item 57603')
            save.click()
            time.sleep(2)
            windowFound = True

        except NoSuchElementException:
            self.previewErrorCount += 1
            print(f'{self.storeNum} Error opening preview: {self.previewErrorCount}')

            pyautogui.press('enter')
            time.sleep(2)
            pyautogui.hotkey('ctrl', 'i')
            time.sleep(2)

            if self.previewErrorCount > 4:
                self.exitStatus = 'ERROR'
                return
```

Save Image

Once the preview is open this function clicks save, changes the save directory to the user specified one, enters a unique file name, and clicks save via pyautogui pressing “enter”. The file name is given a time stamp to make it unique which eliminates the need to perform file duplicate message handling. The file name is returned to the main control loop so that it can verify that the file was successfully written.

At the end, pyautogui presses (alt + space + c) to close the snapshot window. If only a single image is being saved this is not important. However for multiple images it is necessary because if on the next image when Winium tries to find “Item 57603” it will falsely find it on the previously opened window and then generate an exception when trying to find “Save As” which will stop the python script.

```
def saveImage(self, cameraName):
    """saves the image file"""
    saveAs = self.driver.find_element_by_name('Save As')
    filePath = saveAs.find_element_by_name('Address: Documents')
    filePath.click()
    filePath.send_keys(self.saveDir)
    filePath.submit()
    time.sleep(1)

    filename = saveAs.find_element_by_id('1001')
    fileTimeStamp = time.strftime('%m.%d.%Y %H.%M', time.gmtime(time.time()))
    file = self.storeNum + ' ' + cameraName + ' ' + fileTimeStamp
    filename.send_keys(file)
    time.sleep(1)

    pyautogui.hotkey('enter')
    time.sleep(3)
    pyautogui.hotkey('alt', 'space', 'c')
    return(f'{file}.bmp')
```

Snapshot

Sequence control is performed here. The “for” loop allows it to process multiple cameras if more than one camera name is passed to it. If less than 5 errors occurred trying to open a camera view, it will verify that the image was correctly saved and retry if not. If errors occurred but the image was saved successfully, the error count is reset to zero so that if there is still another image to save, it can start with zero errors. In case an exit error has occurred, a message is printed to the console with details for the user so they can go back and investigate that case manually. Total time processing time is tracked and displayed as well.

```
def snapshot(self, cameraNames):
    """Saves snapshot images for a list of camera"""
    startTime = time.time()
    self.startDriver()
    self.login()
    self.openSearch()

    for cameraName in cameraNames:
        self.cameraSearch(cameraName)
        self.openImage(cameraName)

        if self.previewErrorCount <= 4:
            saveSuccess = False
            while(saveSuccess == False):
                file = self.saveImage(cameraName)
                os.chdir(self.saveDir)
                saveSuccess = os.path.isfile(file)

                if saveSuccess == False:
                    print(f'{self.storeNum}: {file} was not saved')
                    self.openImage(cameraName)
                else:
                    print(f'{self.storeNum}: {file} saved ok!')
                    self.previewErrorCount = 0
                    self.exitStatus = 'OK'

    if self.exitStatus != 'ERROR':
        totalTime = time.time() - startTime
        print(f'{self.storeNum}: {totalTime:.2f} seconds for {len(cameraNames)} cameras')

    self.driver.close()
    print(f'{self.storeNum} exit status: {self.exitStatus}')
```

Usage Example

In the example shown on the next page, there are two excel sheets from which tablib grabs data and builds a dictionary whose keys are the store numbers and the values are lists of camera names. If no cameras containing the text “PARKING” or “EXTERIOR” are present, the value “none” will be placed in the list.

Marked with yellow are the three places where the Verint class is used. At the beginning, the class is imported. In order to login to each store, a new instance is made by passing the store number, password, and save directory. Then finally, the snapshot function is called by passing a list of cameras associated with the store.

```
from Verint import *
import tablib

#Verint password and snapshot save directory
state = 'ND'
password = 'password'
saveDir = 'C:\\Temp\\' + state

if not os.path.isdir(saveDir):
    os.mkdir(saveDir)

#Location of source excel files
storesFile = 'C:/Temp/summary.xlsx'
camerasFile = 'C:/Temp/cameras.xlsx'

#Create tablib instances and get header indexes
stores = tablib.Dataset()
stores.xls = open(storesFile, 'rb').read()
cameras = tablib.Dataset()
cameras.xls = open(camerasFile, 'rb').read()

storeIndex = stores.headers.index('Store')
cityIndex = stores.headers.index('City')
stateIndex = stores.headers.index('State')
serverIndex = stores.headers.index('Dell Server')
cameraIndex = stores.headers.index('IP Camera')

siteIndex = cameras.headers.index('Site')
cameraNameIndex = cameras.headers.index('Camera Name\xa0')

#Build store # and camera name dictionary
storeList = {}

for row in range(0, stores.height):
    if stores[row][stateIndex] == state:
        if stores[row][serverIndex] == 'Yes':
            if stores[row][cameraIndex] == 'Yes':
                storeList[int(stores[row][storeIndex])] = ''

for storeNum in storeList.keys():
    cameraList = []
    for row in range(0, cameras.height):
        if cameras[row][siteIndex] == storeNum:
            if ('PARKING' in cameras[row][cameraNameIndex].upper()
            or 'EXTERIOR' in cameras[row][cameraNameIndex].upper()):
                cameraList.append(cameras[row][cameraNameIndex])
    if cameraList == []:
        cameraList.append('none')

    storeList[storeNum] = cameraList

print(f'State: {state}')

for store in storeList.keys():
    print(f'{store}: {storeList[store]}')

#Get camera snapshots
startTime = time.time()

for storeNum in storeList.keys():
    verint = Verint(storeNum, password, saveDir)
    if storeList[storeNum] != ['none']:
        verint.snapshot(storeList[storeNum])

totalMin = (time.time() - startTime) / 60
print(f'Completed in {totalMin:.2f} minutes')
```