pywinauto Documentation

Release 0.5.4

Lots of contributors

Apr 14, 2016

CONTENTS

CHAPTER

ONE

WHAT IS PYWINAUTO

0.6+ development is on the roadmap of the Open Source community (https://github.com/pywinauto)

Full table of contents.

1.1 What is it?

pywinauto is a set of python modules to automate the Microsoft Windows GUI. At it's simplest it allows you to send mouse and keyboard actions to windows dialogs and controls.

1.2 Installation

- Install the following Python packages
  - *Required* pyWin32 http://sourceforge.net/projects/pywin32/files/pywin32/Build%20219/
  - *Optional* Pillow (fork of PIL) https://pypi.python.org/pypi/Pillow/2.7.0
- Download latest pywinauto from https://github.com/pywinauto/pywinauto/releases
- Unzip the pywinauto zip file to a folder
- Run `python.exe setup.py install`

1.3 Installation in silent mode

(Python 2.7, 3.1, 3.2, 3.3, 3.4, 3.5)

- Just run `pip install pywinauto`

To check you have it installed correctly Run Python

```
>>> from pywinauto.application import Application
>>> app = Application.start("notepad.exe")
>>> app.UntitledNotepad.TypeKeys("%FX")
```

1.4 How does it work

A lot is done through attribute access (`__getattr__`) for each class. For example when you get the attribute of an Application or Dialog object it looks for a dialog or control (respectively).

```
myapp.Notepad # looks for a Window/Dialog of your app that has a title 'similar'
              # to "Notepad"

myapp.PageSetup.OK # looks first for a dialog with a title like "PageSetup"
                   # then it looks for a control on that dialog with a title
                   # like "OK"
```

This attribute resolution is delayed (currently a hard coded amount of time) until it succeeds. So for example if you Select a menu option and then look for the resulting dialog e.g.

```
app.UntitledNotepad.MenuSelect("File->SaveAs")
app.SaveAs.ComboBox5.Select("UTF-8")
app.SaveAs.edit1.SetText("Example-utf8.txt")
app.SaveAs.Save.Click()
```

At the 2nd line the SaveAs dialog might not be open by the time this line is executed. So what happens is that we wait until we have a control to resolve before resolving the dialog. At that point if we can't find a SaveAs dialog with a ComboBox5 control then we wait a very short period of time and try again, this is repeated up to a maximum time (currently 5 seconds!)

This avoid the user having to use time.sleep or a "Wait" function.

If your application performs long time operation, new dialog can appear or disappear later. You can wait for its new state like so

```
app.Open.Open.Click() # opening large file
app.Open.WaitNot('visible') # make sure "Open" dialog became invisible
# wait for up to 30 seconds until data.txt is loaded
app.Window(title='data.txt - Notepad').Wait('ready', timeout=30)
```

1.5 Some similar tools for comparison

- Python tools
  - PyAutoGui (https://github.com/asweigart/pyautogui) - it's a cross-platform but there is no windows/controls manipulation at all.
  - AXUI (https://github.com/xcgspring/AXUI) - this is one of the wrappers around UI Automation API.
  - winGuiAuto (http://www.brunningonline.net/simon/blog/archives/winGuiAuto.py.html) - another module using Win32 API.
- Other scripting language tools
  - Perl Win32::GuiTest (http://winguitest.sourceforge.net/)
  - Ruby Win32-Autogui (https://github.com/robertwahler/win32-autogui)
  - Ruby RAutomation (https://github.com/jarmo/RAutomation) - there are 3 adapters: Win32 API, UIA, AutoIt.
  - C# Winium.Desktop (https://github.com/2gis/Winium.Desktop)
  - others (http://www.opensourcetesting.org/functional.php)

- Other free tools
  - AutoIt (http://www.autoitscript.com/)
  - See collection at: https://github.com/atinfo/awesome-test-automation
- Commercial tools
  - WinRunner (http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/)
  - SilkTest (http://www.segue.com/products/functional-regressional-testing/silktest.asp)
  - Many Others (http://www.testingfaqs.org/t-gui.html)

1.6 Why write yet another automation tool if there are so many out there?

There are loads of reasons :-)

Takes a different approach: Most other tools are not object oriented you end up writing stuff like:

```
window = findwindow(title = "Untitled - Notepad", class = "Notepad")
SendKeys(window, "%OF")  # Format -> Font
fontdialog  = findwindow("title = "Font")
buttonClick(fontdialog, "OK")
```

I was hoping to create something more userfriendly (and pythonic). For example the translation of above would be:

```
win = app.UntitledNotepad
win.MenuSelect("Format->Font")
app.Font.OK.Click()
```

Python makes it easy: Python is a great programming language, but I didn't find any automation tools that were Pythonic (I only found one that was implemented in python) and I didn't care for it too much.

Localization as a main requirement: I work in the localization industry and GUI automation is used extensively as often all you need to do is ensure that your UI behaves and is correct with respect to the Source UI. This is actually an easier job then for testing the original source UI.

But most automation tools are based off of coordinates or text of the controls and these can change in the localized software. So my goal (though not yet implemented) is to allow scripts to run unchanged between original source language (often English) and the translated software (Japanese, German, etc).

CHAPTER

TWO

AUTOMATING AN APPLICATION

Once you have installed pywinauto - how do you get going?

2.1 Sit back and have a look at a little movie

Jeff Winkler has created a nice screencast of using pywinauto, you can see it at :

> http://showmedo.com/videos/video?name=UsingpyWinAutoToControlAWindowsApplication&fromSeriesID=7

2.2 Look at the examples

The following examples are included: **Note**: Examples are language dependent - they will only work on the language of product that they were programmed for. All examples have been programmed for English Software except where highlighted.

- `mspaint.py` Control MSPaint
- `notepad_fast.py` Use fast timing settings to control Notepad
- `notepad_slow.py` Use slow timing settings to control Notepad
- `notepad_item.py` Use item rather then attribute access to control Notepad.
- `MiscExamples.py` Show some exceptions and how to get control identifiers.
- `SaveFromInternetExplorer.py` Save a Web Page from Internet Explorer -
- `SaveFromFirefox.py` Save a Web Page from Firefox.
- `get_winrar_info.py` Example of how to do multilingual automation. This is not an ideal example (works on French, Czech and German WinRar)
- `ForteAgentSample.py` Example of dealing with a complex application that is quite dynamic and gives different dialogs often when starting.
- `windowmediaplayer.py` Just another example - deals with check boxes in a ListView.
- `test_sakura.py`, `test_sakura2.py` Two examples of automating a Japanase product.

2.3 Automate notepad at the command line

Please find below a sample run

```
      C:\>python
      Python 2.4.2 (#67, Sep 28 2005, 12:41:11) [MSC v.1310 32 bit (Intel)] on win32
      Type "help", "copyright", "credits" or "license" for more information.
(1)   >>> from pywinauto import application
(2)   >>> app = application.Application()
(3)   >>> app.start("Notepad.exe")
      <pywinauto.application.Application object at 0x00AE0990>
(4)   >>> app.Notepad.DrawOutline()
(5)   >>> app.Notepad.MenuSelect("Edit -> Replace")
(6)   >>> app.Replace.PrintControlIdentifiers()
      Control Identifiers:
      Static - 'Fi&nd what:'   (L1018, T159, R1090, B172)
              'Fi&nd what:' 'Fi&nd what:Static' 'Static' 'Static0' 'Static1'
      Edit - ''   (L1093, T155, R1264, B175)
              'Edit' 'Edit0' 'Edit1' 'Fi&nd what:Edit'
      Static - 'Re&place with:'   (L1018, T186, R1090, B199)
              'Re&place with:' 'Re&place with:Static' 'Static2'
      Edit - ''   (L1093, T183, R1264, B203)
              'Edit2' 'Re&place with:Edit'
      Button - 'Match &case'   (L1020, T245, R1109, B265)
              'CheckBox2' 'Match &case' 'Match &caseCheckBox'
      Button - '&Find Next'   (L1273, T151, R1348, B174)
              '&Find Next' '&Find NextButton' 'Button' 'Button0' 'Button1'
      Button - '&Replace'   (L1273, T178, R1348, B201)
              '&Replace' '&ReplaceButton' 'Button2'
      Button - 'Replace &All'   (L1273, T206, R1348, B229)
              'Button3' 'Replace &All' 'Replace &AllButton'
      Button - 'Cancel'   (L1273, T233, R1348, B256)
              'Button4' 'Cancel' 'CancelButton'
(7)   >>> app.Replace.Cancel.Click()
(8)   >>> app.Notepad.Edit.TypeKeys("Hi from Python interactive prompt %s" % str(dir()), with_spaces
      <pywinauto.controls.win32_controls.EditWrapper object at 0x00DDC2D0>
(9)   >>> app.Notepad.MenuSelect("File -> Exit")
(10)  >>> app.Notepad.No.Click()
      >>>
```

1. Import the pywinauto.application module (usually the only module you need to import directly)
2. Create an Application instance. All access to the application is done through this object.
3. We have created an Application instance in step 2 but we did not supply any information on the Windows application it referred to. By using the start() method we execute that application and connect it to the Application instance app.
4. Draw a green rectangle around the Notepad dialog - so that we know we have the correct window.
5. Select the Replace item from the Edit Menu on the Notepad Dialog of the application that app is connected to. This action will make the Replace dialog appear.
6. Print the identifiers for the controls on the Replace dialog, for example the 1st edit control on the Replace dialog can be referred to by any of the following identifiers:

```
app.Replace.Edit
app.Replace.Edit0
app.Replace.Edit1
app.FindwhatEdit
```

The last is the one that gives the user reading the script aftewards the best idea of what the script does.

7. Close the Replace dialog. (In a script file it is safer to use CloseClick() rather than Click() because CloseClick() waits a little longer to give windows time to close the dialog.)
8. Let's type some text into the Notepad text area. Without the `with_spaces` argument spaces would not be typed. Please see documentation for SendKeys for this method as it is a thin wrapper around SendKeys.
9. Ask to exit Notepad
10. We will be asked if we want to save - Click on the "No" button.

CHAPTER

THREE

HOW TO'S

3.1 How to specify an usable Application instance

An `Application()` instance is the point of contact for all work with the app you are automating. So the Application instance needs to be connected to a process. There are two ways of doing this:

```
start(self, cmd_line, timeout = app_start_timeout) # instance method:
```

or:

```
connect(self, **kwargs) # instance method:
```

`start()` is used when the application is not running and you need to start it. Use it in the following way:

```
app = Application.start(r"c:\path\to\your\application -a -n -y --arguments")
```

The timeout parameter is optional, it should only be necessary to use it if the application takes a long time to start up.

`connect()` is used when the application to be automated is already running. To specify an already running application you need to specify one of the following:

process the process id of the application, e.g.

```
app = Application.connect(process = 2341)
```

handle The windows handle of a window of the application, e.g.

```
app = Application.connect(handle = 0x010f0c)
```

path The path of the executable of the process (`GetModuleFileNameEx` is used to find the path of each process and compared against the value passed in) e.g.

```
app = Application.connect(path = r"c:\windows\system32\notepad.exe")
```

or any combination of the parameters that specify a window, these get passed to the `findwindows.find_windows()` function. e.g.

```
app = Application.connect(title_re = ".*Notepad", class_name = "Notepad")
```

Note: I have since added static methods Application.start() and Application.connect() these can be used the same as above - except that you no longer need to instantiate an Application object first.

Note2: The application has to be ready before you can use connect*(). There is no timeout or retries like there is when finding the application after start*(). So if you start the application outside of pywinauto you need to either sleep or program a wait loop to wait until the application has fully started.

3.2 How to specify a dialog of the application

Once the application instance knows what application it is connected to a dialog to work on needs to be specified.

There are many different ways of doing this. The most common will be using item or attribute access to select a dialog based on it's title. e.g

```
dlg = app.Notepad
```

or equivalently

```
dlg = app['Notepad']
```

The next easiest method is to ask for the `top_window_()` e.g.

```
dlg = app.top_window_()
```

This will return the window that has the highest Z-Order of the top-level windows of the application.

Note: This is currently fairly untested so I am not sure it will return the correct window. It will definitely be a top level window of the application - it just might not be the one highest in the Z-Order.

If this is not enough control they you can use the same parameters as can be passed to `findwindows.find_windows()` e.g.

```
dlg = app.window_(title_re = "Page Setup", class_name = "#32770")
```

Finally to have the most control you can use

```
dialogs = app.windows_()
```

this will return a list of all the visible, enabled, top level windows of the application. You can then use some of the methods in `handleprops` module select the dialog you want. Once you have the handle you need then use

```
Application.window_(handle = win)
```

Note: If the title of the dialog is very long - then attribute access might be very long to type, in those cases it is usually easier to use

```
app.window_(title_re = ".*Part of Title.*")
```

3.3 How to specify a control on a dialog

There are a number of ways to specify a control, the simplest are

```
app.dlg.control
app['dlg']['control']
```

The 2nd is better for non English OS's where you need to pass unicode strings e.g. app[u'your dlg title'][u'your ctrl title']

The code builds up multiple identifiers for each control from the following:

- title
- friendly class
- title + friendly class

If the control's text is empty (after removing non char characters) text is not used. Instead we look for the closest control above and to the right fo the contol. And append the friendly class. So the list becomes

- friendly class
- closest text + friendly class

Once a set of identifiers has been created for all controls in the dialog we disambiguate them.

use the *WindowSpecification.PrintControlIdentifiers()* method

e.g.

```
app.YourDialog.PrintControlIdentifiers()
```

Sample output

```
Button - Paper    (L1075, T394, R1411, B485)
        'PaperGroupBox' 'Paper' 'GroupBox'
Static - Si&ze:    (L1087, T420, R1141, B433)
        'SizeStatic' 'Static' 'Size'
ComboBox -    (L1159, T418, R1399, B439)
        'ComboBox' 'SizeComboBox'
Static - &Source:    (L1087, T454, R1141, B467)
        'Source' 'Static' 'SourceStatic'
ComboBox -    (L1159, T449, R1399, B470)
        'ComboBox' 'SourceComboBox'
Button - Orientation    (L1075, T493, R1171, B584)
        'GroupBox' 'Orientation' 'OrientationGroupBox'
Button - P&ortrait    (L1087, T514, R1165, B534)
        'Portrait' 'RadioButton' 'PortraitRadioButton'
Button - L&andscape    (L1087, T548, R1165, B568)
        'RadioButton' 'LandscapeRadioButton' 'Landscape'
Button - Margins (inches)    (L1183, T493, R1411, B584)
        'Marginsinches' 'MarginsinchesGroupBox' 'GroupBox'
Static - &Left:    (L1195, T519, R1243, B532)
        'LeftStatic' 'Static' 'Left'
Edit -    (L1243, T514, R1285, B534)
        'Edit' 'LeftEdit'
Static - &Right:    (L1309, T519, R1357, B532)
        'Right' 'Static' 'RightStatic'
Edit -    (L1357, T514, R1399, B534)
        'Edit' 'RightEdit'
Static - &Top:    (L1195, T550, R1243, B563)
        'Top' 'Static' 'TopStatic'
Edit -    (L1243, T548, R1285, B568)
        'Edit' 'TopEdit'
Static - &Bottom:    (L1309, T550, R1357, B563)
        'BottomStatic' 'Static' 'Bottom'
Edit -    (L1357, T548, R1399, B568)
        'Edit' 'BottomEdit'
Static - &Header:    (L1075, T600, R1119, B613)
        'Header' 'Static' 'HeaderStatic'
Edit -    (L1147, T599, R1408, B619)
        'Edit' 'TopEdit'
Static - &Footer:    (L1075, T631, R1119, B644)
        'FooterStatic' 'Static' 'Footer'
Edit -    (L1147, T630, R1408, B650)
        'Edit' 'FooterEdit'
Button - OK    (L1348, T664, R1423, B687)
        'Button' 'OK' 'OKButton'
```

```
    Button - Cancel    (L1429, T664, R1504, B687)
            'Cancel' 'Button' 'CancelButton'
    Button - &Printer...    (L1510, T664, R1585, B687)
            'Button' 'Printer' 'PrinterButton'
    Button - Preview    (L1423, T394, R1585, B651)
            'Preview' 'GroupBox' 'PreviewGroupBox'
    Static -     (L1458, T456, R1549, B586)
            'PreviewStatic' 'Static'
    Static -     (L1549, T464, R1557, B594)
            'PreviewStatic' 'Static'
    Static -     (L1466, T586, R1557, B594)
            'Static' 'BottomStatic'
```

This example has been taken from test_application.py

Note The identifiers printed by this method have been run through the process that makes the identifier unique. So if you have 2 edit boxes, they will both have "Edit" listed in their identifiers. In reality though the first one can be refered to as "Edit", "Edit0", "Edit1" and the 2nd should be refered to as "Edit2"

Note You do not have to be exact!. Say we take an instance from the example above

```
    Button - Margins (inches)    (L1183, T493, R1411, B584)
            'Marginsinches' 'MarginsinchesGroupBox' 'GroupBox'
```

Let's say that you don't like any of these

- `GroupBox` - too generic, it could be any group box
- `Marginsinches` and `MarginsinchesGroupBox` - these just don' look right, it would be nicer to leave out the 'inches' part

Well you CAN! The code does a best match on the identifer you use against all the available identifiers in the dialog.

For example if you break into the debugger you can see how different identifiers can be used

```
    (Pdb) print app.PageSetup.Margins.Text()
    Margins (inches)
    (Pdb) print app.PageSetup.MarginsGroupBox.Text()
    Margins (inches)
```

And this will also cater for typos. Though you still have to be careful as if there are 2 similar identifiers in the dialog the typo you have used might be more similar to another control then the one you were thinking of.

3.4 How to use pywinauto with application languages other than English

Because Python does not support unicode identifiers in code you cannot use attribute access to reference a control so you would either have to use item access or make an explicit calls to `window_()`.

So instead of writing

```
    app.dialog_ident.control_ident.Click()
```

You would have to write

```
    app['dialog_ident']['control_ident'].Click()
```

Or use `window_()` explictly

```
app.window_(title_re = "NonAsciiCharacters").window_(title = "MoreNonAsciiCharacters").Click()
```

To see an example of this see `examples\MiscExamples.py.GetInfo()`

3.5 How to deal with controls that do not respond as expected (e.g. OwnerDraw Controls)

Some controls (especially Ownerdrawn controls) do not respond to events as expected. For example if you look at any HLP file and go to the Index Tab (click 'Search' button) you will see a listbox. Running Spy or Winspector on this will show you that it is indeed a list box - but it is ownerdrawn. This means that the developer has told Windows that they will override how items are displayed and do it themselves. And in this case they have made it so that strings cannot be retrieved :-(.

So what problems does this cause?

```
app.HelpTopics.ListBox.Texts()                # 1
app.HelpTopics.ListBox.Select("ItemInList")   # 2
```

1. Will return a list of empty strings, all this means is that pywinauto has not been able to get the strings in the listbox
2. This will fail with an IndexError because the Select(string) method of a ListBox looks for the item in the Texts to know the index of the item that it should select.

The following workaround will work on this control

```
app.HelpTopics.ListBox.Select(1)
```

This will select the 2nd item in the listbox, because it is not a string lookup it works correctly.

Unfortunately not even this will always work. The developer can make it so that the control does not respond to standard events like Select. In this case the only way you can select items in the listbox is by using the keyboard simulation of TypeKeys().

This allows you to send any keystrokes to a control. So to select the 3rd item you would use

```
app.Helptopics.ListBox1.TypeKeys("{HOME}{DOWN 2}{ENTER}")
```

- `{HOME}` will make sure that the first item is highlighted.
- `{DOWN 2}` will then move the highlight down 2 items
- `{ENTER}` will select the highlighted item

If your application made extensive use of a similar control type then you could make using it easier by deriving a new class from ListBox, that could use extra knowledge about your particular application. For example in the WinHelp example evertime an item is highlighted in the list view, it's text is inserted into the Edit control above the list, and you CAN get the text of the item from there e.g.

```
# print the text of the item currently selected in the list box
# (as long as you are not typing into the Edit control!)
print app.HelpTopics.Edit.Texts()[1]
```

3.6 How to Access the System Tray (aka SysTray, aka 'Notification Area')

Near the clock are icons representing running applications, this area is normally referred to as the "System Tray". There are actually many different windows/controls in this area. The control that contains the icons is actually a toolbar. It is in a Pager control, in within a window with a class TrayNotifyWnd, which is in yet another window with a class Shell_TrayWnd and all these windows are part of the running Explorer instance. Thankfully you don't need to remeber all that :-).

The things that are important to remember is that you are looking for a window in the "Explorer.exe" application with the class "Shell_TrayWnd" that has a Toolbar control with a title "Notification Area".

One way to get this is to do the following

```
imprt pywinauto.application
app = pywinauto.application.Application().connect_(path = "explorer")
systray_icons = app.ShellTrayWnd.NotificationAreaToolbar
```

The taskbar module provides very preliminary access to the System Tray.

It defines the following variables:

explorer_app defines an Application() object connected to the running explorer. You probably don't need to use this your self very much.

TaskBar The handle to the task bar (the bar containing Start Button, the QuickLaunch icons, running tasks, etc

StartButton "Start me up" :-) I think you might know what this is!

QuickLaunch The Toolbar with the quick launch icons

SystemTray The window that contains the Clock and System Tray Icons

Clock The clock

SystemTrayIcons The toolbar representing the system tray icons

RunningApplications The toolbar representing the running applications

I have also provided 2 functions in the module that can be used to click on system tray icons:

`ClickSystemTrayIcon(button)` You can use this to left click a visible icon in the system tray. I had to specifically say visible icon as there may be many invisible icons that obviously cannot be clicked. Button can be any integer. If you specify 3 then it will find and click the 3rd visible button. (very little error checking is performed and this method will more then likely be moved/renamed in the futures.

`RightClickSystemTrayIcon(button)` Similar to `ClickSytemTrayIcon` but performs a right click.

Often when you click/right click on an icon - you get a popup menu. The thing to remember at this point is that the popup menu is part of the application being automated not part of explorer.

e.g.

```
# connect to outlook
outlook = Application.connect(path = 'outlook.exe')

# click on Outlook's icon
taskbar.ClickSystemTrayIcon("Microsoft Outlook")
```

```
    # Select an item in the popup menu
    outlook.PopupMenu.Menu().GetMenuPath("Cancel Server Request")[0].Click()
```

CHAPTER

FOUR

WAITING FOR LONG OPERATIONS

A GUI application behaviour is often unstable and your script needs waiting until a new window appears or an existing window is closed/hidden. pywinauto can wait for a dialog initialization implicitly (with the default timeout). There are few methods/functions that could help you to make your code easier and more reliable.

4.1 Application methods

- WaitCPUUsageLower (new in pywinauto 0.5.2)

This method is useful for multi-threaded interfaces that allow a lazy initialization in another thread while GUI is responsive and all controls already exist and ready to use. So waiting for a specific window existence/state is useless. In such case the CPU usage for the whole process indicates that a task calculation is not finished yet.

Example:

```
app.WaitCPUUsageLower(threshold=5) # wait until CPU usage is lower than 5%
```

4.2 WindowSpecification methods

These methods are available to all controls.

- Wait
- WaitNot

There is an example containing long waits: install script for 7zip 9.20 x64 (https://gist.github.com/vasily-v-ryabov/7a04717af4584cbb840f).

A `WindowSpecification` object isn't necessarily related to an existing window/control. It's just a description namely a couple of criteria to search the window. The `Wait` method (if no any exception is raised) can guarantee that the target control exists or even visible, enabled and/or active.

4.3 Functions in `timings` module

There are also low-level methods useful for any Python code.

- WaitUntil
- WaitUntilPasses

4.4 Identify controls

The methods to help you to find a needed control.

- PrintControlIdentifiers
- DrawOutline

4.4.1 How To‘s

- *How To’s*

CHAPTER

FIVE

METHODS AVAILABLE TO EACH DIFFERENT CONTROL TYPE

Windows have many controls, buttons, lists, etc

5.1 All Controls

These functions are aviailable to all controls.

- CaptureAsImage
- Click
- ClickInput
- Close
- CloseClick
- DebugMessage
- DoubleClick
- DoubleClickInput
- DragMouse
- DrawOutline
- GetFocus
- GetShowState
- Maximize
- MenuSelect
- Minimize
- MoveMouse
- MoveWindow
- NotifyMenuSelect
- NotifyParent
- PressMouse
- PressMouseInput
- ReleaseMouse
- ReleaseMouseInput

- Restore
- RightClick
- RightClickInput
- RightClickInput
- SendMessage
- SendMessageTimeout
- SetFocus
- SetWindowText
- TypeKeys
- Children
- Class
- ClientRect
- ClientRects
- ContextHelpID
- ControlID
- ExStyle
- Font
- Fonts
- FriendlyClassName
- GetProperties
- HasExStyle
- HasStyle
- IsChild
- IsDialog
- IsEnabled
- IsUnicode
- IsVisible
- Menu
- MenuItem
- MenuItems
- Owner
- Parent
- PopupWindow
- ProcessID
- Rectangle
- Style

- Texts
- TopLevelParent
- UserData
- VerifyActionable
- VerifyEnabled
- VerifyVisible
- WindowText

5.2 Button, CheckBox, RadioButton, GroupBox

- ButtonWrapper.Check
- ButtonWrapper.GetCheckState
- ButtonWrapper.SetCheckIndeterminate
- ButtonWrapper.UnCheck

5.3 ComboBox

- ComboBoxWrapper.DroppedRect
- ComboBoxWrapper.ItemCount
- ComboBoxWrapper.ItemData
- ComboBoxWrapper.ItemTexts
- ComboBoxWrapper.Select
- ComboBoxWrapper.SelectedIndex

5.4 Dialog

- DialogWrapper.ClientAreaRect
- DialogWrapper.RunTests
- DialogWrapper.WriteToXML

5.5 Edit

- EditWrapper.GetLine
- EditWrapper.LineCount
- EditWrapper.LineLength
- EditWrapper.Select
- EditWrapper.SelectionIndices

- EditWrapper.SetEditText
- EditWrapper.SetWindowText
- EditWrapper.TextBlock

5.6 Header

- HeaderWrapper.GetColumnRectangle
- HeaderWrapper.GetColumnText
- HeaderWrapper.ItemCount

5.7 ListBox

- ListBoxWrapper.GetItemFocus
- ListBoxWrapper.ItemCount
- ListBoxWrapper.ItemData
- ListBoxWrapper.ItemTexts
- ListBoxWrapper.Select
- ListBoxWrapper.SelectedIndices
- ListBoxWrapper.SetItemFocus

5.8 ListView

- ListViewWrapper.Check
- ListViewWrapper.ColumnCount
- ListViewWrapper.Columns
- ListViewWrapper.ColumnWidths
- ListViewWrapper.GetColumn
- ListViewWrapper.GetHeaderControl
- ListViewWrapper.GetItem
- ListViewWrapper.GetSelectedCount
- ListViewWrapper.IsChecked
- ListViewWrapper.IsFocused
- ListViewWrapper.IsSelected
- ListViewWrapper.ItemCount
- ListViewWrapper.Items
- ListViewWrapper.Select
- ListViewWrapper.Deselect

- ListViewWrapper.UnCheck

5.9 PopupMenu

(no extra visible methods)

5.10 ReBar

- ReBarWrapper.BandCount
- ReBarWrapper.GetBand
- ReBarWrapper.GetToolTipsControl

5.11 Static

(no extra visible methods)

5.12 StatusBar

- StatusBarWrapper.BorderWidths
- StatusBarWrapper.GetPartRect
- StatusBarWrapper.GetPartText
- StatusBarWrapper.PartCount
- StatusBarWrapper.PartRightEdges

5.13 TabControl

- TabControlWrapper.GetSelectedTab
- TabControlWrapper.GetTabRect
- TabControlWrapper.GetTabState
- TabControlWrapper.GetTabText
- TabControlWrapper.RowCount
- TabControlWrapper.Select
- TabControlWrapper.TabCount
- TabControlWrapper.TabStates

5.14 Toolbar

- ToolbarWrapper.Button
- ToolbarWrapper.ButtonCount
- ToolbarWrapper.GetButton
- ToolbarWrapper.GetButtonRect
- ToolbarWrapper.GetToolTipsControl
- ToolbarWrapper.PressButton

  ToolbarButton (returned by `Button()`)

  - ToolbarButton.Rectangle
  - ToolbarButton.Style
  - ToolbarButton.ClickInput
  - ToolbarButton.Click
  - ToolbarButton.IsCheckable
  - ToolbarButton.IsChecked
  - ToolbarButton.IsEnabled
  - ToolbarButton.IsPressable
  - ToolbarButton.IsPressed
  - ToolbarButton.State

5.15 ToolTips

- ToolTipsWrapper.GetTip
- ToolTipsWrapper.GetTipText
- ToolTipsWrapper.ToolCount

5.16 TreeView

- TreeViewWrapper.EnsureVisible
- TreeViewWrapper.GetItem
- TreeViewWrapper.GetProperties
- TreeViewWrapper.IsSelected
- TreeViewWrapper.ItemCount
- TreeViewWrapper.Root
- TreeViewWrapper.Select

  TreeViewElement (returned by `GetItem()` and `Root()`)

  - TreeViewElement.Children

- TreeViewElement.Item
- TreeViewElement.Next
- TreeViewElement.Rectangle
- TreeViewElement.State
- TreeViewElement.SubElements
- TreeViewElement.Text

5.17 UpDown

- UpDownWrapper.GetBase
- UpDownWrapper.GetBuddyControl
- UpDownWrapper.GetRange
- UpDownWrapper.GetValue
- UpDownWrapper.SetValue
- UpDownWrapper.Increment
- UpDownWrapper.Decrement

CHAPTER SIX

CREDITS

Stefaan Himpe - Lots of speed and stability improvements early on<br/>

Jeff Winkler - Early encouragement, creation of screencasts<br/>

Dalius Dobravolskas - Help on the forums and prompted major improvements on the wait* functionality<br/>

Daisuke Yamashita - Bugs/suggestions for 2.5 that MenuWrapper.GetProperties() returns a list rather then a dict<br/>

Raghav - idea with using metaclass for finding wrapper<br/>

Michael Herrmann - bug fixes, project maintenance<br/>

Vasily Ryabov (Intel Corporation) - Port to 64-bit Python and Python 3.x, moving repo to GitHub, project maintenance, other contributions<br/>

airelil - continuous integration with AppVeyor, Python 3.x bug reports, moving unit tests to VC2010 MFC samples, other contributions<br/>

CHAPTER

SEVEN

DEV NOTES

7.1 FILE LAYOUT

# used by just about everything (and considered a block!) win32defines.py win32functions.py win32structures.py

# Find windows and their attributes findwindows.py handleprops.py

# wrap windows, get extra info for particular controls # set the friendly class name controlscommon_controls.py controlscontrolactions.py controlsHwndWrapper.py controlswin32_controls.py

# currently depends on the Friendly class name # probably needs to be refactored to make it independent of controls! # maybe move that stuff to _application_? findbestmatch.py # currently depends on controls!

controlactions.py

testsallcontrols.py testsasianhotkey.py testscomboboxdroppedheight.py testscomparetoreffont.py testsleadtrailspaces.py testsmiscvalues.py testsmissalignment.py testsmissingextrastring.py testsoverlapping.py testsrepeatedhotkey.py teststranslation.py teststruncation.py

controlproperties.py

XMLHelpers.py

    FindDialog.py PyDlgCheckerWrapper.py

application.py test_application.py

7.2 Best matching

difflib provides this support For menu's it is simple we match against the text of the menu item. For controls the story is more complicated because we want to match against the following:

- Control text if it exists
- Friendly Class name
- Control text + Friendly class name (if control text exists)
- (Possibly) closest static + FriendlyClassName

e.g. FindWhatCombo, ComboBox1,

or Text, TextRiadio, RadioButton2

1. the control itself knows what it should be referred to
2. Need to disambiguate across all controls in the dialog

3. then we need to match

7.3 ATTRIBUTE RESOLUTION

Thinking again... app.dlg.control

TWO LEVELS

- **application.member (Python resolves)** an attribute of application object
- **application.dialog** a dialog reference

THREE LEVELS

- **application.member.attr (Python resolves)** another attribute of the previous member
- **application.dialog.member** a member of the dialog object
- **application.dialog.control** a control on the dialog

FOUR LEVELS (leaving out Python resolved)

- application.dialog.member.member
- application.dialog.control.member

DELAYED RESOLUTION FOR SUCCESS Taking the example

```
app.dlg.control.action()
```

If we leave out syntax and programming errors there are still a number of reasons why it could fail.

dlg might not be found control might not be found either dlg or control may be disabled

dialog and control may be found but on the wrong dialog (e.g. in Notepad you can bring up 2 “Page Setup” dialogs both with an OK button)

One solution would just be to add a “sleep” before trying to find each new dialog (to ensure that it is there and ready) - but this will mean lots of unnecessary waiting.

So the solution I have tried is:

- perform the complete attribute access resolution at the latest possible time
- if it fails then wait and try again
- after a specified timeout fail raising the original exception.

This means that in the normal case you don’t have unnecessary waits - and in the failure case - you still get an exception with the error.

Also waiting to do resolution as late as possible stops errors where an earlier part of the path succeedes - but finds the wrong item.

So for example if finds the page setup dialog in Notepad # open the Printer setup dialog (which has “Page Setup” as title) app.PageSetup.Printer.Click()

# if this runs too quickly it actually finds the current page setup dialog # before the next dialog opens, but that dialog does not have a Properties # button - so an error is raised. # because we re-run the resolution from the start we find the new pagesetup dialog. app.PageSetup.Properties.Click()

7.4 WRITING TO DIALOGS

We need a way of making sure that the dialog is active without having to access a control on it. e.g.

```
app.MainWin.MenuSelect("Something That->Loads a Dialog")
app.Dlg._write("dlg.xml")
```

or a harder problem:

```
app.PageSetup.Printer.Click()
app.PageSetup._write("pagesetup.xml")
```

In this second example it is very hard to be sure that the correct Page Setup dialog is shown.

The only way to be realy sure is to check for the existance of certain control(s) (ID, Class, text, whatever) - but it would be nice to not have to deal with those :-(

Another less declarative (more magic?) is to scan the list of available windows/controls and if they haven't changed then accept that the correct one is shown.

When testing and having XML files then we should use those to make sure that we have the correct dialog up (by using Class/ID)

CHAPTER

EIGHT

PYWINAUTO TODO'S

- Make sure to add documentation strings for all undocumented methods/functions
- Check coverage of the tests and work to increase it.
- Add tests for SendInput click methods
- Implement findbestmatch using FuzzyDict.
- Find a way of doing application data in a better way. Currently if someone even adds a call to print_control_identifiers() it will break the matching algorithm!
- Need to move the checking if a control is a Ownerdrawn/bitmap control out of __init__ methods and into it's own method something like IsNormallyRendered() (Why?)
- Give example how to work with Tray Window
- Fix ToolbarWrapper.PressButton() which doesn't seem to work (found wile working on IE example script)
- Maybe supply an option so that scripts can be run by using:

```
pywinauto.exe yourscript.py
```

  This would work by creating a Py2exe wrapper that would import the script (and optionally call a particular function?)

  This way pywinauto could be made available to people without python installed (whether this is a big requirement or not I don't know because the automation language is python anyway!.

- Message traps - how to handle unwanted message boxes popping up?
  1. Wait for an Exception then handle it there
  2. set a trap waiting for a specific dialog
  3. on calls to window specification, if we fail to find our window then we can run quickly through the available specified traps to see if any of them apply - then if they do we can run the associated actions - then try our original dialog again
- Handle adding reference controls (in that they should be the controls used for finding windows)
- Find the reference name of a variable e.g so that in Dialog._write() we can know the variable name that called the _write on (this we don't have to repeat the XML file name!)
- If we remove the delay after a button click in controlactions then trying to close two dialogs in a row might fail because the first dialog hasn't closed yet and the 2nd may have similar title and same closing button e.g PageSetup.OK.Click(), PageSetup2.OK.Click(). A possible solution to this might be to keep a cache of windows in the application and no two different dialog identifiers (PageSetup and PageSetup2 in this case) can have the same handle - so returning the handle of PageSetup when we call PageSetup2 would fail (and we would do our usual waiting until it succeeds or times out).

- Investigate using any of the following
  - BringWindowToTop: probably necessary before image capture
  - GetTopWindow: maybe to re-set top window after capture?
  - EnumThreadWindows
  - GetGUIThreadInfo
- Make it easy to work with context(right click) menu's
- Further support .NET controls and download/create a test .NET application
- Look at supporting the Sytem Tray (e.g. right click on an icon)
- supply SystemTray class (singleton probably)
- Look at clicking and text input - maybe use SendInput
- Support Up-Down controls and other common controls
- Find out whether control.item.action() or control.action(item) is better
- Create a Recorder to visually create tests

LOW PRIORITY

- Create a class that makes it easy to deal with a single window (e.g. no application)
- Allow apps to be started in a different thread so we don't lock up
  - this is being done already - the problem is that some messages cannot be sent across processes if they have pointers (so we need to send a synchronous message which waits for the other process to respond before returning)
  - But I guess it would be possible to create a thread for sending those messages?
- Liberate the code from HwndWrapper - there is very little this add's beyond what is available in handleprops. The main reason this is required is for the FriendlyClassName. So I need to look to see if this can be moved elsewhere.

  Doing this might flatten the heirarchy quite a bit and reduce the dependencies on the various packages
- Need to make Menu items into classes so instead of Dlg.MenuSelect we should be doing

```
dlg.Menu("blah->blah").Select()
```

  or even

```
dlg.Menu.Blah.Blah.Select()
```

  To do this we need to change how menu's are retrieved - rather than get all menuitems at the start - then we just get the requested level.

  This would also enable things like

```
dlg.Menu.Blah.Blah.IsChecked()  IsEnabled(), etc
```

8.1 CLOSED (in some way or the other)

- Allow delay after click to be removed. The main reason that this is needed at the moment is because if you close a dialog and then try an action on the parent immediately it may not yet be active - so the delay is needed to allow it to become active. To fix this we may need to add more magic around calling actions on dialogs e.g. on an attribute access for an ActionDialog do the following:

- Check if it is an Action
- If it is not enabled then wait a little bit
- If it is then wait a little bit and try again
- repeat that until success or timeout

The main thing that needs to be resolved is that you don't want two of these waits happening at once (so a wait in a function at 1 level, and another wait in a function called by the other one - because this would mean there would be a VERY long delay while the timeout of the nested function was reached the number of times the calling func tried to succeed!)

- Add referencing by closest static (or surrounding group box?)
- Need to modularize the methods of the common_controls because at the moment they are much too monolithic.
- Finish example of saving a page from IE
- Document that I have not been able to figure out how to reliably check if a menu item is enabled or not before selecting it. (Probably FIXED NOW!)

  For Example in Media Player if you try and click the View->Choose Columns menu item when it is not enabled it crashes Media Player. Theoretically MF_DISABLED and MF_GRAYED should be used - but I found that these are not updated (at least for Media Player) until they are dropped down.
- Implement an opional timing/config module so that all timing can be customized

CHAPTER

NINE

CHANGE LOG

9.1 0.5.4 Bug fixes and partial MFC Menu Bar support

30-October-2015

- Fix bugs and inconsistencies:
  - Add *where="check"* possible value to the ListViewWrapper.Click/ClickInput‘ methods.
  - Add *CheckByClickInput* and *UncheckByClickInput* methods for a plain check box.
  - Fix crash while waiting for the window start.
- Add partial MFC Menu Bar support. The menu bar can be interpreted as a toolbar. Items are clickable by index through experimental *MenuBarClickInput* method of the *ToolbarWrapper*.
- Python 3.5 is supported.

9.2 0.5.3 Better Unicode support for SetEditText/TypeKeys and menu items

25-September-2015

- Better backward compatibility with pywinauto 0.4.2:
  - support Unicode symbols in the `TypeKeys` method again;
  - allow `SetEditText/TypeKeys` methods to take non-string arguments;
  - fix taking Unicode parameters in `SetEditText/TypeKeys`.
- Fix bug in `Wait("active")`, raise a SyntaxError when waiting for an incorrect state.
- Re-consider some timings, update docs for the default values etc.
- Fix several issues with an owner-drawn menu.
- `MenuItem` method `Click` is renamed to `ClickInput` while `Click = Select` now.
- New `SetTransparency` method can make a window transparent in a specified degree.

9.3 0.5.2 Improve ListView, new methods for CPU usage, DPI awareness

07-September-2015

- New Application methods: `CPUUsage` returns CPU usage as a percent (float number), `WaitCPUUsageLower` waits until the connected process' CPU usage is lower than a specified value (2.5% by default).
- A new class `_listview_item`. It is very similar to `_treeview_element`.
- Add DPI awareness API support (Win8+). It allows correct work when all fonts are scaled at 125%, 150% etc (globally or per monitor).
- "Tools overview" section in docs.
- Fix number of bugs:
  - `TreeViewWrapper.Select` doesn't work when the control is not in focus.
  - `TabControlWrapper.Select` doesn't work in case of TCS_BUTTONS style set.
  - `ListViewWrapper` methods `Check/UnCheck` are fixed.
  - Toolbar button: incorrect access by a tooltip text.
  - Warning "Cannot retrieve text length for handle" uses print() instead of actionlogger.
  - `ClientToScreen` method doesn't return a value (modifying mutable argument is not good practice).

9.4 0.5.1 Several fixes, more tests

13-July-2015

- Resolve pip issues
- Warn user about mismatched Python/application bitness (64-bit Python should be used for 64-bit application and 32-bit Python is for 32-bit app)
- Add "TCheckBox" class name to ButtonWrapper detection list
- Fix `DebugMessage` method
- Disable logging (actionlogger.py) by default, provide shortcuts: `actionlogger.enable()` and `actionlogger.disable()`. For those who are familiar with standard `logging` module there's method `actionlogger.set_level(level)`

9.5 0.5.0 64-bit Py2/Py3 compatibility

30-June-2015

- 64-bit Python and 64-bit apps support (but 32-bit Python is recommended for 32-bit apps)
- Python 2.x/3.x compatibility
- Added pyWin32 dependency (silent install by pip for 2.7 and 3.1+)
- Improvements for Toolbar, TreeView, UpDown and DateTimePicker wrappers
- Improved `best_match` algorithm allows names like `ToolbarFile`

- Clicks can be performed with pressed Ctrl or Shift
- Drag-n-drop and scrolling methods (DragMouse, DragMouseInput, MouseWheelInput)
- Improved menu support: handling OWNERDRAW menu items; access by command_id (like `$23453`)
- Resolved issues with py2exe and cx_freeze
- `RemoteMemoryBlock` can now detect memory corruption by checking guard signature
- Upgraded `taskbar` module
- `sysinfo` module for checking 32-bit or 64-bit OS and Python
- `set_foreground` flag in `TypeKeys` method for typing into in-place controls
- flags `create_new_console` and `wait_for_idle` in `Application.start` method

9.6 0.4.0 Various cleanup and bug fixes

03-April-2010

- Gracefully Handle dir() calls on Application or WindowSpecification objects (which used hang for a while as these classes would search for windows matching __members__, __methods__ and __bases__). The code now checks for any attribute that starts with '__' and ends with '__' and raises AttributeError immediately. Thanks to Sebastian Haase for raising this.
- Removed the reference to an Application object in WindowSpecification. It was not used in the class and made the class harder to use. WindowSpecification is now more useful as a utility class.
- Add imports of application.WindowSpecification and application.Application to pywinauto.__init__.py so that these classes can be used more easily (without having to directly import pywinauto.application). Thanks again to Sebastian Haase.
- Added a function to empty the clipboard (thanks to Tocer on Sourceforge)
- Use 'SendMessageTimeout' to get the text of a window. (SendMessage will hang if the application is not processing messages)
- Fixed references to PIL.ImageGrab. PIL add's it's module directly to the module path, so it should just be referenced by ImageGrab and not PIL.ImageGrab.
- Use AttachThreadInput + PostMessage rather than SendMessageTimeout to send mouse clicks.
- Fix how timeout retry times are calculated in timings.WaitUntil() and timings.Wait
- Fixed some issues with application.Kill_() method, highlighted due to the changes in the HwndWrapper.Close() method.
- Fix writing images to XML. It was broken with updates to PIL that I had not followed. Changed the method of knowing if it is an image by checking for various attributes.
- Renamed WindowSpecification.(Ww)indow() to ChildWindow() and added deprecation messages for the other functions.
- Improved the tests (fixed test failures which were not pywinauto issues)

9.7 0.3.9 Experimental! New Sendkeys, and various fixes

27-November-2009

- Major change this release is that Sendkeys is no longer a requirement! A replacement that supports Unicode is included with pywinauto. (hopefully soon to be released as a standalone module). Please note - this is still quite untested so this release should be treated with some care..
- Made sure that default for WindowSpecification.Window_() was to look for non top level windows. The defaults in find_windows() had been changed previously and it now needed to be explicitly overridden.
- Fixed a missing reference to 'win32defines' when referencing WAIT_TIMEOUT another typo of false (changed to False)
- Removed the restriction to only get the active windows for the process, now it will be possible to get the active windows, even if a process is not specified. From http://msdn.microsoft.com/en-us/library/ms633506%28VS.85%29.aspx it gets the active window for the foreground thread.
- Hopefully improved Delphi TreeView and ListView handling (added window class names as supported window classes to the appropriate classes).
- Added support for running UI tests with reference controls. (requried for some localization tests)
- Various PyLint and PEP8 fixes made.

9.8 0.3.8 Collecting improvements from last 2 years

10-March-2009

- Fixed toolbar button pressing - This required for HwndWrapper.NotifyParent() to be updated (to accept a new ID parameter)
- Fixed a bug wherea listview without a column control would make pywinauto fail to capture the dialog.
- Converted documenation from Pudge generated to Sphinx Generated
- Added some baic support for Pager and Progress controls (no tests yet)
- Added some more VB 'edit' window classes
- Added some more VB 'listbox' window classes
- Added some more VB 'button' window classes
- Ensured that return value from ComboBoxWrapper.SelectedIndices is always a tuple (there was a bug where it would sometimes be a ctypes array)
- Changed default for finding windows to find disabled windows as well as enabled ones (previous was to find enabled windows only) (note this may impact scripts that relied on the previous setting i.e. in cases where two dialogs have the same title!)
- Much better handling of InvalidWindowHandle during automation runs. This could be raised when a closing window is still available when the automation was called, but is gone half way through whatever function was called.
- Made clicking more robust by adding a tiny wait between each SendMessageTimeout in _perform_click().
- Added attributes `can_be_label` and `has_title` to `HwndWrapper` and subclasses to specify whether a control can act as a label for other controls, and whether the title should be used for identifying the control. If you have created your own HwndWrapper subclasses you may need to override the defaults.
- Added a `control_id` parameter to find_windows which allows finding windows based off of their control id's
- Added a FriendlyClassName method to MenuItem
- Split up the functions for button truncation data

- Commented out code to get a new font if the font could not be recovered
- Moved code to get the control font from Truncation test to handleprops
- Added a function to get the string representation of the bug. (need to refactor PrintBugs at some point).
- Fixed a variable name (from fname -> font_attrib as fname was not a defined variable!)
- Forced some return values from MissingExtraString test to be Unicode
- Fixed the MiscValues test (converted to Unicode and removed some extraneous characters)
- Updated the path for all unittests
- Made two unit tests sligthly more robust and less dependent on computer/app settings
- Updated timing settings for unit tests
- Updated the examples to work in dev environment.

9.9 0.3.7 Merge of Wait changes and various bug fixes/improvements

10-April-2007

- Added Timings.WaitUntil() and Timings.WaitUntilPasses() which handle the various wait until something in the code. Also refactored existing waits to use these two methods.
- Fixed a major Handle leak in RemoteMemorBlock class (which is used extensively for 'Common' controls. I was using OpenHandle to open the process handle, but was not calling CloseHandle() for each corresponding OpenHandle().
- Added an active_() method to Application class to return the active window of the application.
- Added an 'active' option to WindowSpecification.Wait() and WaitNot().
- Some cleanup of the clipboard module. GetFormatName() was improved and GetData() made a little more robust.
- Added an option to findwindows.find_windows() to find only active windows (e.g. active_only = True). Default is False.
- Fixed a bug in the timings.Timings class - timing values are Now accessed through the class (Timings) and not through the intance (self).
- Updated ElementTree import in XMLHelpers so that it would work on Python 2.5 (where elementtree is a standard module) as well as other versions where ElementTree is a separate module.
- Enhanced Item selection for ListViews, TreeViews - it is now possible to pass strings and they will be searched for. More documentation is required though.
- Greatly enhanced Toolbar button clicking, selection, etc. Though more documentation is required.
- Added option to ClickInput() to allow mouse wheel movements to be made.
- menuwrapper.Menu.GetProperties() now returns a dict like all other GetProperties() methods. This dict for now only has one key 'MenuItems' which contains the list of menuitems (which had been the previous return value).

9.10 0.3.6b Changes not documented in 0.3.6 history

31-July-2006

- Fixed a bug in how findbestmatch.FindBestMatches was working. It would match against text when it should not!
- Updated how timings.Timings.Slow() worked, if any time setting was less then .2 after 'slowing' then set it to .2

9.11 0.3.6 Scrolling and Treview Item Clicking added

28-July-2006

- Added parameter to `_treeview_item.Rectangle()` to have an option to get the Text rectangle of the item. And defaulted to this.
- Added `_treeview_item.Click()` method to make it easy to click on tree view items.
- Fixed a bug in `TreeView.GetItem()` that was expanding items when it shouldn't.
- Added `HwndWrapper.Scroll()` method to allow scrolling. This is a very minimal implementation - and if the scrollbars are implemented as separate controls (rather then a property of a control - this will probably not work for you!). It works for Notepad and Paint - that is all I have tried so far.
- Added a call to `HwndWrapper.SetFocus()` in `_perform_click_input()` so that calls to `HwndWrapper.ClickInput()` will make sure to click on the correct window.

9.12 0.3.5 Moved to Metaclass control wrapping

24-May-2006

- Moved to a metaclass implementation of control finding. This removes some cyclic importing that had to be worked around and other then metaclass magic makes the code a bit simpler.
- Some of the sample files would not run - so I updated them so they would (Thanks to Stefaan Himpe for pointing this out)
- Disabled saving application data (it was still being saved in Application.RecordMatch() even if the rest of the application data code is disabled. This was causing what appeared to be a memory leak where pywinauto would keep grabbing more and more memory (especially for controls that contain a lot of information). Thanks to Frank Martinez for leading me to this).
- Added ListViewWrapper.GetItemRect() to enable retrieving the rectangle for a particular item in the listview.
- Removed references to _ctrl() method within pywinauto as it was raising a DeprecationWarning internally even if the user was not using it.

9.13 0.3.4 Fixed issue with latest ctypes, speed gains, other changes

25-Apr-2006

- The latest version of ctypes (0.9.9.6) removed the code generator I was using some generated code in win32functions.py (stdcall). I was not using those functions so I just commented them out.
- Started the process of renaming methods of the `Application` and `WindowSpecification` classes. I will be converting names to `UppercaseNames_()`. The trailing _ is to disambiguate the method names from potential Window titles.

- Updated how print_control_identifiers works so that it now always prints the disambiguated control name. (even for single controls)
- Added __hash__ to HwndWrapper so that controls could be dictionary keys.
- Caching various information at various points. For example I cache how well two pieces of text match. For short scripts this has little impact - but for larger script it could well have a major impact. Also caching information for controls that cannot change e.g. TopLeveParent, Parent, etc

9.14 0.3.3 Added some methods, and fixed some small bugs

19-Apr-2006

- Added a wait for the control to be active and configurable sleeps after 'modifying' actions (e.g. Select, Deselect, etc)
- Fixed Timings.Slow() and Timings.Fast() - they could in certain circumstances do the opposite! If you had already set a timing slower or faster then they would set it then they would blindly ignore that and set their own times. I added functionality that they will take either the slowest or fastest of the new/current setting rather then blindly setting to the new value.
- Fixed some hidden bugs with HwndWrapper.CloseClick()
- Fixed a bug in setup.py that would raise an error when no argument was specified
- Added an argument to HwndWrapper.SendMessageTimeout so that the wait options could be passed in.
- Added HwndWrapper.Close(), Maximize(), Minimize(), Restore() and GetShowState().
- Commented out all deprecated methods (will be removed completely in some future release).
- Added Application.kill_() method - which closes all windows and kills the application. If the application is asking if you want to save your changes - you will not be able to click yes or no and the application will be killed anyway!.

9.15 0.3.2 Fixed setup.py and some typos

31-Mar-2006

- Fixed the spelling of Stefaan Himpe's name
- Fixed setup.py which was working for creating a distribution but not for installing it (again thanks to Stefaan for pointing it out!)

9.16 0.3.1 Performance tune-ups

30-Mar-2006

- Change calculation of distance in findbestmatch.GetNonTextControlName() so that it does not need to square or get the square root to find the real distance - as we only need to compare values - not have the actual distance. (Thanks to Stefaan Himpe)
- Compiled regular expression patterns before doing the match to avoid compiling the regular expression for window that is being tested (Thanks to Stefaan Himpe)

- Made it easier to add your own control tests by adding a file extra_tests.py which needs to export a ModifyRegisteredTests() method. Also cleaned up the code a little.
- Updated notepad_fast.py to make it easier to profile (adde a method)
- Changed WrapHandle to use a cache for classes it has matched - this is to avoid having to match against all classes constantly.
- Changed default timeout in SendMessageTimeout to .001 seconds from .4 seconds this results in a significant speedup. Will need to make this value modifiable via the timing module/routine.
- WaitNot was raising an error if the control was not found - it should have returned (i.e. success - control is not in any particular state because it does not exist!).
- Added ListViewWrapper.Deselect() per Chistophe Keller's suggestion. While I was at it I added a check on the item value passed in and added a call to WaitGuiIdle(self) so that the control has a chance to process the message.
- Changed doc templates and moved dependencies into pywinauto subversion to ensure that all files were availabe at www.openqa.org and that they are not broken when viewed there.
- Moved all timing information into the timings.Timings class. There are some simple methods for changing the timings.

9.17 0.3.0 Added Application data - now useful for localization testing

20-Mar-2006

- Added automatic Application data collection which can be used when running the same test on a different spoken language version. Support is still preliminary and is expected to change. Please treat as early Alpha.

  If you have a different language version of Windows then you can try this out by running the notepad_fast.py example with the langauge argument e.g.

  ```
  examples\notepad_fast.py language
  ```

  This will load the application data from the supplied file notepad_fast.pkl and use it for finding the right menu items and controls to select.

- Test implementation to make it easier to start using an application. Previously you needed to write code like

  ```
  app = Application().connect_(title = 'Find')
  app.Find.Close.Click()
  app.NotePad.MenuSelect("File->Exit")
  ```

  1st change was to implement static methods `start()` and `connect()`. These methods return a new Application instance so the above code becomes:

  ```
  app = Application.connect(title = 'Find')
  app.Find.Close.Click()
  app.NotePad.MenuSelect("File->Exit")
  ```

  I also wanted to make it easier to start working with a simple application - that may or may not have only one dialog. To make this situation easier I made `window_()` not throw if the application has not been `start()ed` or `connect()ed` first. This leads to simpler code like:

  ```
  app = Application()
  app.Find.Close.Click()
  app.NotePad.MenuSelect("File->Exit")
  ```

What happens here is that when you execute any of Application.window_(), Application.__getattr__() or Application.__getitem__() when the application hasn't been connected or started. It looks for the window that best matches your specification and connects the application to that process.

This is extra functionality - existing connect_() and start_() methods still exist

- Fixed HwndWrapper.SetFocus() so that it would work even if the window was not in the foreground. (it now makes the window foreground as well as giving it focus). This overcomes a restriction in Windows where you can only change the foreground window if you own the foreground window.
- Changed some 2.4'isms that an anonymous commenter left on my blog :-) with these changes pywinauto should run on Python 2.3 (though I haven't done extensive testing).
- Commented out controls.common_controls.TabControlWrapper.GetTabState() and TabStates() as these did not seem to be returning valid values anyway.
- Fixed documentation issues were parts of the documentation were not getting generated to the HTML files.
- Fixed issue where MenuSelect would sometimes not work as expected. Some Menu actions require that the window that owns the menu be active. Added a call to SetFocus() before selecting a menu item to ensure that the window was active.
- Fixed Bug 1452832 where clipboard was not closed in clipboard.GetData()
- Added more unit tests now up to 248 from 207

9.18 0.2.5 More refactoring, more tests

07-Mar-2006

- Added wrapper classes for Menus and MenuItems this enabled cleaner interaction with Menu's. It also gives more functionality - you can now programmatically Click() on menus, and query if a menu item is checked or not.
- Added application.WindowSpecification.Wait() and WaitNot() methods. These methods allow you to wait for a control to exist, be visible, be enabled, be ready (both enabled and visible!) or to wait for the control to not be in any of these states. WaitReady(), WaitNotEnabled(), WaitNotVisible() now use these methods. I was able to also add the missing methods WaitNotReady(), WaitEnabled(), WaitVisible(), WaitExists(), WaitnotExists(). Please use Wait() and WaitNot() as I have Deprecated these Wait* methods.
- Slightly modified timeout waits for control resolution so that a timed function more accurately follows the timeout value specified.
- Added application.Application.start() and connect() static methods. These methods are factory methods in that they will return an initialized Application instance. They work exactly the same as start_() and connect() as they are implemented in terms of those.

  from pywinauto.application import Application notepad = Application.start("notepad") same_notepad = Application.connect(path = "notepad")
- Updated the examples to follow changes to the code - and to make them a little more robust.
- Added a new Controls Overview document page which lists all the actions on all controls.
- Added more unit tests now up to 207 from 134 (added 68 tests)

9.19 0.2.1 Small Release number - big changes

17-Feb-2006

- Quick release to get many changes out there - but this release has been less tested then I would like for a .3 release.
- Allow access to non text controls using the closest Text control. This closest text control will normally be the static/label associated with the control. For example in Notepad, Format->Font dialog, the 1st combobox can be refered to as “FontComboBox” rather than “ComboBox1”
- Added a new control wrapper - `PopupMenuWrapper` for context menu’s You can now work easily with context menu’s e.g.

```
app.Notepad.Edit.RightClick()
# need to use MenuClick rather then MenuSelect
app.PopupMenu.MenuClick("Select All")
app.Notepad.Edit.RightClick()
app.PopupMenu.MenuClick("Copy")
```

  I could think of merging the `RightClick()` and `MenuSelect()` into one method `ContextMenuSelect()` if that makes sense to most people.

- Added Support for Up-Down controls
- Not all top level windows now have a FriendlyClassName of “Dialog”. I changed this because it made it hard to get windows of a particular class. For example the main Notepad window has a class name of “Notepad”.

  This was primarily implemented due to work I did getting the System Tray.

- Renamed `StatusBarWrapper.PartWidths()` to `PartRightEdges()` as this is more correct for what it returns.
- Changed HwndWrapper.Text() and SetText() to WindowText() and SetWindowText() respectively to try and make it clearer that it is the text returned by GetWindowText and not the text that is visible on the control. This change also suggested that EditWrapper.SetText() be changed to SetEditText() (though this is not a hard requirement EditWrapper.SetText() still exists - but may be deprecated.
- Added ClickInput, DoubleClickInput, RightClickInput, PressMouseInput ReleaseMouseInput to HwndWrapper - these use SendInput rather then WM_LBUTTONDOWN, WM_RBUTTONUP, etc used by Click, DoubleClick etc.

  I also added a MenuClick method that allows you to click on menu items. This means you can now ‘physically’ drop menus down.

- Some further working with tooltips that need to be cleaned up.
- Fixed a bug where coordinates passed to any of the Click operations had the X and Y coordinates swapped.
- Added new MenuItem and Menu classes that are to the most part hidden but you can get a menu item by doing

```
app.Notepad.MenuItem("View")
app.Notepad.MenuItem("View->Status Bar")
```

  MenuItems have various actions so for example you can use `MenuItem.IsChecked()` to check if the menu item is checked. Among other methods there are `Click()` and `Enabled()`.

- Modified the ‘best match’ algorithm for finding controls. It now searches a couple of times, and tries to find the best fit for the text passed to it. The idea here is to make it more “Select what I want - not that other thing that looks a bit like what I want!”. It is possible this change could mean you need to use new identifiers in scripts - but in general very little modification should be necessary.

  There was also a change to the algorithm that looked for the closest text control. It missed some obvious controls in the previous implementation. It also had a bug for controls above the control rather than to the left.

- Added a new example scripts SaveFromInternetExplorer.py and SaveFromFirefox.py which show automating downloading of a page from either of these browsers.

- Added yet more unit tests, there are now a total of 134 tests.

9.20 0.2.0 Significant refactoring

06-Feb-2006

- Changed how windows are searched for (from application) This chage should not be a significant change for users
- Started adding unit tests (and the have already uncovered bugs that been fixed). They also point to areas of missing functionality that will be addded with future updates
- Changed from property access to Control attributes to function access If your code was accessing properties of controls then this might be a significant change! The main reasons for doing this were due to the inheritability of properties (or lack there-of!) and the additional scafolding that was required to define them all.
- Updated the `DialogWrapper.MenuSelect()` method to notify the parent that it needs to initialize the menu's before it retrieves the items
- Added functionality to associate 'non-text' controls with the 'text' control closest to them. This allows controls to be referenced by:

```
app.dlg.<Nearby_text><Window_class>
```

  e.g. to reference the "Footer" edit control in the Page Setup dialog you could use:

```
app.PageSetup.FooterEdit
```

- Added a MoveWindow method to HwndWrapper
- Did some more cleanup (fixing pylint warnings) but still not finished
- Added some better support for .NET controls (not to be considered final)

9.21 0.1.3 Many changes, few visible

15-Jan-2006

- Wrote doc strings for all modules, classes and functions
- Ran pychecker and pylint and fixed some errors/warning
- changed

```
_connect, _start, _window, _control, _write
```

  respectively to

```
connect_, start_, window_, connect_, write_
```

  If you forget to change `_window`, `_connect` and `_start` then you will probably get the following error.

```
TypeError: '_DynamicAttributes' object is not callable
```

- pywinauto is now a package name - you need to import it or its modules
- Changes to the code to deal with pywinauto package name
- Fixed searching for windows if a Parent is passed in

- Added Index to retrieved MenuItem dictionary
- Added a check to ensure that a windows Handle is a valid window
- Refactored some of the methods in common_controls
- Refactored how FriendlyClassName is discovered (and still not really happy!

9.22 0.1.2 Add Readme and rollup various changes

15-Jan-2006

- Updated Readme (original readme was incorrect)
- Added clipboard module
- Fixed DrawOutline part of tests.__init__.print_bugs
- Added a NotifyParent to HwndWrapper
- Make sure that HwndWrapper.ref is initialized to None
- Refactored some methods of ComboBox and ListBox
- Updated Combo/ListBox selection methods
- Removed hardcoded paths from test_application.py
- Added section to save the document as UTF-8 in MinimalNotepadTest
- Fixed EscapeSpecials and UnEscapeSpecials in XMLHelpers
- Made sure that overly large bitmaps do not break XML writing

9.23 0.1.1 Minor bug fix release

12-Jan-2006

- Fixed some minor bugs discovered after release

9.24 0.1.0 Initial Release

6-Jan-2006

CHAPTER

TEN

SOURCE CODE REFERENCE

10.1 Main user modules

10.1.1 pywinauto.application module

The application module is the main one that users will use first.

When starting to automate an application you must initialize an instance of the Application class. Then you must `Application.Start()` that application or `Application.Connect()` to a running instance of that application.

Once you have an Application instance you can access dialogs in that application either by using one of the methods below.

```
dlg = app.YourDialogTitle
dlg = app.ChildWindow(title = "your title", classname = "your class", ...)
dlg = app['Your Dialog Title']
```

Similarly once you have a dialog you can get a control from that dialog in almost exactly the same ways.

```
ctrl = dlg.YourControlTitle
ctrl = dlg.ChildWindow(title = "Your control", classname = "Button", ...)
ctrl = dlg["Your control"]
```

Note: For attribute access of controls and dialogs you do not have to have the title of the control exactly, it does a best match of the available dialogs or controls.

See also:

`pywinauto.findwindows.find_windows()` for the keyword arguments that can be passed to both: `Application.Window_()` and `WindowSpecification.Window()`

exception `pywinauto.application.`**`AppNotConnected`**

Bases: `exceptions.Exception`

Application has been connected to a process yet

exception `pywinauto.application.`**`AppStartError`**

Bases: `exceptions.Exception`

There was a problem starting the Application

class `pywinauto.application.`**`Application`**(*backend='native'*, *datafilename=None*)

Bases: `object`

Represents an application

`__getattribute__`(*attr_name*)
Find the specified dialog of the application

`__getitem__`(*key*)
Find the specified dialog of the application

`CPUUsage`(*interval=None*)
Return CPU usage percentage during specified number of seconds

`Connect`(***kwargs*)
Connects to an already running process

`Connect_`(***kwargs*)
Deprecated method. Performs PendingDeprecationWarning before calling the .connect(). Should be also removed in 0.6.X.

`GetMatchHistoryItem`(*index*)
Should not be used - part of application data implementation

`Kill_`()
Try and kill the application

Dialogs may pop up asking to save data - but the application will be killed anyway - you will not be able to click the buttons. this should only be used when it is OK to kill the process like you would in task manager.

`Start`(*cmd_line*, *timeout=None*, *retry_interval=None*, *create_new_console=False*, *wait_for_idle=True*)
Starts the application giving in cmd_line

`Start_`(**args*, ***kwargs*)
Deprecated method. Performs PendingDeprecationWarning before calling the .start(). Should be also removed in 0.6.X.

`WaitCPUUsageLower`(*threshold=2.5*, *timeout=None*, *usage_interval=None*)
Wait until process CPU usage percentage is less than specified threshold

`Window_`(***kwargs*)
Return a window of the application

You can specify the same parameters as findwindows.find_windows. It will add the process parameter to ensure that the window is from the current process.

`Windows_`(***kwargs*)
Return list of wrapped windows of the top level windows of the application

`WriteAppData`(*filename*)
Should not be used - part of application data implementation

`active_`()
Return the active window of the application

`connect`(***kwargs*)
Connects to an already running process

`connect_`(***kwargs*)
Deprecated method. Performs PendingDeprecationWarning before calling the .connect(). Should be also removed in 0.6.X.

`is64bit`()
Return True if running process is 64-bit

kill_()
: Try and kill the application

  Dialogs may pop up asking to save data - but the application will be killed anyway - you will not be able to click the buttons. this should only be used when it is OK to kill the process like you would in task manager.

start(*cmd_line*, *timeout=None*, *retry_interval=None*, *create_new_console=False*, *wait_for_idle=True*)
: Starts the application giving in cmd_line

start_(**args*, ***kwargs*)
: Deprecated method. Performs PendingDeprecationWarning before calling the .start(). Should be also removed in 0.6.X.

top_window_()
: Return the current top window of the application

window_(***kwargs*)
: Return a window of the application

  You can specify the same parameters as findwindows.find_windows. It will add the process parameter to ensure that the window is from the current process.

windows_(***kwargs*)
: Return list of wrapped windows of the top level windows of the application

`pywinauto.application.`**AssertValidProcess**(*process_id*)
: Raise ProcessNotFound error if process_id is not a valid process id

exception `pywinauto.application.`**ProcessNotFoundError**
: Bases: `exceptions.Exception`

  Could not find that process

class `pywinauto.application.`**WindowSpecification**(*search_criteria*)
: Bases: `object`

  A specification for finding a window or control

  Windows are resolved when used. You can also wait for existance or non existance of a window

  __getattribute__(*attr_name*)
  : Attribute access for this class

    If we already have criteria for both dialog and control then resolve the control and return the requested attribute.

    If we have only criteria for the dialog but the attribute requested is an attribute of DialogWrapper then resolve the dialog and return the requested attribute.

    Otherwise delegate functionality to `__getitem__()` - which sets the appropriate criteria for the control.

  __getitem__(*key*)
  : Allow access to dialogs/controls through item access

    This allows:

```
app['DialogTitle']['ControlTextClass']
```

    to be used to access dialogs and controls.

    Both this and `__getattribute__()` use the rules outlined in the HowTo document.

  ChildWindow(***criteria*)
  : Add criteria for a control

When this window specification is resolved then this will be used to match against a control.

Exists (*timeout=None*, *retry_interval=None*)
Check if the window exists, return True if the control exists

Parameters

- **timeout** – the maximum amount of time to wait for the control to exists. Defaults to `Timings.exists_timeout`
- **retry_interval** – The control is checked for existance this number of seconds. `Defaults to Timings.exists_retry`

PrintControlIdentifiers ()
Prints the 'identifiers'

If you pass in a control then it just prints the identifiers for that control

If you pass in a dialog then it prints the identifiers for all controls in the dialog.

Note: The identifiers printed by this method have not been made unique. So if you have 2 edit boxes, they will both have "Edit" listed in their identifiers. In reality though the first one can be refered to as "Edit", "Edit0", "Edit1" and the 2nd should be refered to as "Edit2".

WAIT_CRITERIA_MAP = {'ready': ('is_visible', 'is_enabled'), 'visible': ('is_visible',), 'enabled': ('is_enabled',), 'active':

Wait (*wait_for*, *timeout=None*, *retry_interval=None*)
Wait for the window to be in a particular state/states.

Parameters

- **wait_for** – The state to wait for the window to be in. It can be any of the following states, also you may combine the states by space key.
  - 'exists' means that the window is a valid handle
  - 'visible' means that the window is not hidden
  - 'enabled' means that the window is not disabled
  - 'ready' means that the window is visible and enabled
  - 'active' means that the window is active
- **timeout** – Raise an *`pywinauto.timings.TimeoutError()`* if the window is not in the appropriate state after this number of seconds.
- **retry_interval** – How long to sleep between each retry.

Default: `pywinauto.timings.Timings.window_find_retry`.

An example to wait until the dialog exists, is ready, enabled and visible:

```
self.Dlg.Wait("exists enabled visible ready")
```

See also:

`WindowSpecification.WaitNot()`

`pywinauto.timings.TimeoutError()`

WaitNot (*wait_for_not*, *timeout=None*, *retry_interval=None*)
Wait for the window to not be in a particular state/states.

Parameters

- **wait_for_not** – The state to wait for the window to not be in. It can be any of the following states, also you may combine the states by space key.
  - 'exists' means that the window is a valid handle
  - 'visible' means that the window is not hidden
  - 'enabled' means that the window is not disabled
  - 'ready' means that the window is visible and enabled
  - 'active' means that the window is active
- **timeout** – Raise an *pywinauto.timings.TimeoutError()* if the window is sill in the state after this number of seconds.
- **retry_interval** – How long to sleep between each retry.

Default: `pywinauto.timings.Timings.window_find_retry`.

An example to wait until the dialog is not ready, enabled or visible:

```
self.Dlg.WaitNot("enabled visible ready")
```

See also:

WindowSpecification.Wait()

pywinauto.timings.TimeoutError()

Window(***criteria*)

Window_(***criteria*)

WrapperObject()
Allow the calling code to get the HwndWrapper object

print_control_identifiers()
Prints the 'identifiers'

If you pass in a control then it just prints the identifiers for that control

If you pass in a dialog then it prints the identifiers for all controls in the dialog.

Note: The identifiers printed by this method have not been made unique. So if you have 2 edit boxes, they will both have "Edit" listed in their identifiers. In reality though the first one can be refered to as "Edit", "Edit0", "Edit1" and the 2nd should be refered to as "Edit2".

window_(***criteria*)

pywinauto.application.**assert_valid_process**(*process_id*)
Raise ProcessNotFound error if process_id is not a valid process id

pywinauto.application.**process_from_module**(*module*)
Return the running process with path module

pywinauto.application.**process_get_modules**()

pywinauto.application.**process_module**(*process_id*)
Return the string module name of this process

10.1.2 pywinauto.findbestmatch

Module to find the closest match of a string in a list

exception `pywinauto.findbestmatch.`**`MatchError`**(*items=None*, *tofind=u''*)
A suitable match could not be found

class `pywinauto.findbestmatch.`**`UniqueDict`**
A dictionary subclass that handles making it's keys unique

`FindBestMatches`(*search_text*, *clean=False*, *ignore_case=False*)
Return the best matches for search_text in the items

- **search_text** the text to look for
- **clean** whether to clean non text characters out of the strings
- **ignore_case** compare strings case insensitively

`pywinauto.findbestmatch.`**`build_unique_dict`**(*controls*)
Build the disambiguated list of controls

Separated out to a different function so that we can get the control identifiers for printing.

`pywinauto.findbestmatch.`**`find_best_control_matches`**(*search_text*, *controls*)
Returns the control that is the the best match to search_text

This is slightly differnt from find_best_match in that it builds up the list of text items to search through using information from each control. So for example for there is an OK, Button then the following are all added to the search list: "OK", "Button", "OKButton"

But if there is a ListView (which do not have visible 'text') then it will just add "ListView".

`pywinauto.findbestmatch.`**`find_best_match`**(*search_text*, *item_texts*, *items*, *limit_ratio=0.5*)
Return the item that best matches the search_text

- **search_text** The text to search for
- **item_texts** The list of texts to search through
- **items** The list of items corresponding (1 to 1) to the list of texts to search through.
- **limit_ratio** How well the text has to match the best match. If the best match matches lower then this then it is not considered a match and a MatchError is raised, (default = .5)

`pywinauto.findbestmatch.`**`get_control_names`**(*control*, *allcontrols*, *textcontrols*)
Returns a list of names for this control

`pywinauto.findbestmatch.`**`get_non_text_control_name`**(*ctrl*, *controls*, *text_ctrls*)
return the name for this control by finding the closest text control above and to its left

`pywinauto.findbestmatch.`**`is_above_or_to_left`**(*ref_control*, *other_ctrl*)
Return true if the other_ctrl is above or to the left of ref_control

10.1.3 pywinauto.findwindows

Provides functions for iterating and finding windows/elements

exception `pywinauto.findwindows.`**`ElementAmbiguousError`**
There was more then one element that matched

exception `pywinauto.findwindows.`**`ElementNotFoundError`**
No element could be found

exception `pywinauto.findwindows.`**`WindowAmbiguousError`**

There was more then one window that matched

`pywinauto.findwindows.`**`enum_windows`**`()`

Return a list of handles of all the top level windows

`pywinauto.findwindows.`**`find_element`**`(`***kwargs*`)`

Call find_elements and ensure that only one element is returned

Calls find_elements with exactly the same arguments as it is called with so please see find_elements for a description of them.

`pywinauto.findwindows.`**`find_elements`**`(`*class_name=None*, *class_name_re=None*, *parent=None*, *process=None*, *title=None*, *title_re=None*, *top_level_only=True*, *visible_only=True*, *enabled_only=False*, *best_match=None*, *handle=None*, *ctrl_index=None*, *found_index=None*, *predicate_func=None*, *active_only=False*, *control_id=None*, *auto_id=None*, *framework_id=None*, *backend=None*`)`

Find elements based on criteria passed in

Possible values are:

- **class_name** Elements with this window class
- **class_name_re** Elements whose class match this regular expression
- **parent** Elements that are children of this
- **process** Elements running in this process
- **title** Elements with this text
- **title_re** Elements whose text match this regular expression
- **top_level_only** Top level elements only (default=True)
- **visible_only** Visible elements only (default=True)
- **enabled_only** Enabled elements only (default=False)
- **best_match** Elements with a title similar to this
- **handle** The handle of the element to return
- **ctrl_index** The index of the child element to return
- **found_index** The index of the filtered out child lement to return
- **active_only** Active elements only (default=False)
- **control_id** Elements with this control id
- **auto_id** Elements with this automation id (for UIAutomation elements)
- **framework_id** Elements with this framework id (for UIAutomation elements)
- **backend** Back-end name to use while searching (default=None means current active backend)

10.1.4 pywinauto.timings

Timing settings for all of pywinauto

This module has one object that should be used for all timing adjustments timings.Timings

There are a couple of predefined settings

timings.Timings.Fast() timings.Timings.Defaults() timings.Timings.Slow()

The Following are the individual timing settings that can be adjusted:

- window_find_timeout (default 5)
- window_find_retry (default .09)
- app_start_timeout (default 10)
- app_start_retry (default .90)
- cpu_usage_interval (default .5)
- cpu_usage_wait_timeout (default 20)
- exists_timeout (default .5)
- exists_retry (default .3)
- after_click_wait (default .09)
- after_clickinput_wait (default .05)
- after_menu_wait (default .1)
- after_sendkeys_key_wait (default .01)
- after_button_click_wait (default 0)
- before_closeclick_wait (default .1)
- closeclick_retry (default .05)
- closeclick_dialog_close_wait (default 2)
- after_closeclick_wait (default .2)
- after_windowclose_timeout (default 2)
- after_windowclose_retry (default .5)
- after_setfocus_wait (default .06)
- setfocus_timeout (default 2)
- setfocus_retry (default .1)
- after_setcursorpos_wait (default .01)
- sendmessagetimeout_timeout (default .01)
- after_tabselect_wait (default .05)
- after_listviewselect_wait (default .01)
- after_listviewcheck_wait default(.001)
- after_treeviewselect_wait default(.1)
- after_toobarpressbutton_wait default(.01)
- after_updownchange_wait default(.1)
- after_movewindow_wait default(0)
- after_buttoncheck_wait default(0)

- after_comboboxselect_wait default(.001)
- after_listboxselect_wait default(0)
- after_listboxfocuschange_wait default(0)
- after_editsetedittext_wait default(0)
- after_editselect_wait default(.02)
- drag_n_drop_move_mouse_wait default(.1)
- before_drag_wait default(.2)
- before_drop_wait default(.1)
- after_drag_n_drop_wait default(.1)
- scroll_step_wait default(.1)

class `pywinauto.timings.`**`TimeConfig`**

Central storage and manipulation of timing values

`Defaults``()`

Set all timings to the default time

`Fast``()`

Set fast timing values

Currently this changes the timing in the following ways: timeouts = 1 second waits = 0 seconds retries = .001 seconds (minimum!)

(if existing times are faster then keep existing times)

`Slow``()`

Set slow timing values

Currently this changes the timing in the following ways: timeouts = default timeouts * 10 waits = default waits * 3 retries = default retries * 3

(if existing times are slower then keep existing times)

exception `pywinauto.timings.`**`TimeoutError`**

`pywinauto.timings.`**`WaitUntil`**`(`*timeout*, *retry_interval*, *func*, *value=True*, *op=<built-in function eq>*, **args*`)`

Wait until `op(function(*args), value)` is True or until timeout expires

- **timeout** how long the function will try the function
- **retry_interval** how long to wait between retries
- **func** the function that will be executed
- **value** the value to be compared against (defaults to True)
- **op** the comparison function (defaults to equality) * **args** optional arguments to be passed to func when called

Returns the return value of the function If the operation times out then the return value of the the function is in the 'function_value' attribute of the raised exception.

e.g.

```
        try:
            # wait a maximum of 10.5 seconds for the
            # the objects item_count() method to return 10
            # in increments of .5 of a second
            WaitUntil(10.5, .5, self.item_count, 10)
        except TimeoutError as e:
            print("timed out")
```

`pywinauto.timings.`**`WaitUntilPasses`**(*timeout*, *retry_interval*, *func*, *exceptions=<type 'exceptions.Exception'>*, **args*)

Wait until `func(*args)` does not raise one of the exceptions in exceptions

- **timeout** how long the function will try the function
- **retry_interval** how long to wait between retries
- **func** the function that will be executed
- **exceptions** list of exceptions to test against (default: Exception)
- **args** optional arguments to be passed to func when called

Returns the return value of the function If the operation times out then the original exception raised is in the 'original_exception' attribute of the raised exception.

e.g.

```
        try:
            # wait a maximum of 10.5 seconds for the
            # window to be found in increments of .5 of a second.
            # P.int a message and re-raise the original exception if never found.
            WaitUntilPasses(10.5, .5, self.Exists, (ElementNotFoundError))
        except TimeoutError as e:
            print("timed out")
            raise e.
```

10.2 Specific functionality

10.2.1 pywinauto.clipboard

Some clipboard wrapping functions - more to be added later

`pywinauto.clipboard.`**`EmptyClipboard`**()

`pywinauto.clipboard.`**`GetClipboardFormats`**()
Get a list of the formats currently in the clipboard

`pywinauto.clipboard.`**`GetData`**(*format_id=<MagicMock name='mock.CF_UNICODETEXT' id='47454946085840'>*)
Return the data from the clipboard in the requested format

`pywinauto.clipboard.`**`GetFormatName`**(*format_id*)
Get the string name for a format value

10.2.2 pywinauto.taskbar

Module showing how to work with the task bar

This module will likely change significantly in the future!

pywinauto.taskbar.**ClickHiddenSystemTrayIcon**(*button*, *exact=False*, *by_tooltip=False*, *double=False*)
Click on a hidden tray icon given by button

pywinauto.taskbar.**ClickSystemTrayIcon**(*button*, *exact=False*, *by_tooltip=False*, *double=False*)
Click on a visible tray icon given by button

pywinauto.taskbar.**RightClickHiddenSystemTrayIcon**(*button*, *exact=False*, *by_tooltip=False*)
Right click on a hidden tray icon given by button

pywinauto.taskbar.**RightClickSystemTrayIcon**(*button*, *exact=False*, *by_tooltip=False*)
Right click on a visible tray icon given by button

pywinauto.taskbar.**TaskBarHandle**()
Return the first window that has a class name 'Shell_TrayWnd'

10.3 Controls Reference

10.3.1 pywinauto.controls.common_controls

Classes that wrap the Windows Common controls

class pywinauto.controls.common_controls.**_toolbar_button**(*index_*, *tb_handle*)
Bases: object

Wrapper around Toolbar button (TBBUTTONINFO) items

Click(*button='left'*, *pressed='`*)
Click on the Toolbar button

ClickInput(*button='left'*, *double=False*, *wheel_dist=0*, *pressed='`*)
Click on the Toolbar button

HasStyle(*style*)
Return True if the button has the specified style

IsCheckable()
Return if the button can be checked

IsChecked()
Return if the button is in the checked state

IsEnabled()
Return if the button is in the pressed state

IsPressable()
Return if the button can be pressed

IsPressed()
Return if the button is in the pressed state

Rectangle()
Get the rectangle of a button on the toolbar

State()
Return the state of the button

Style()
Return the style of the button

Text()
Return the text of the button

click(*button='left'*, *pressed=''*)
Click on the Toolbar button

click_input(*button='left'*, *double=False*, *wheel_dist=0*, *pressed=''*)
Click on the Toolbar button

has_style(*style*)
Return True if the button has the specified style

is_checkable()
Return if the button can be checked

is_checked()
Return if the button is in the checked state

is_enabled()
Return if the button is in the pressed state

is_pressable()
Return if the button can be pressed

is_pressed()
Return if the button is in the pressed state

rectangle()
Get the rectangle of a button on the toolbar

state()
Return the state of the button

style()
Return the style of the button

text()
Return the text of the button

class pywinauto.controls.common_controls.**_treeview_element**(*elem*, *tv_handle*)
Bases: object

Wrapper around TreeView items

Children()
Return the direct children of this control

Click(*button='left'*, *double=False*, *where='text'*, *pressed=''*)
Click on the treeview item

where can be any one of "text", "icon", "button", "check" defaults to "text"

ClickInput(*button='left'*, *double=False*, *wheel_dist=0*, *where='text'*, *pressed=''*)
Click on the treeview item

where can be any one of “text”, “icon”, “button”, “check” defaults to “text”

Collapse()
Collapse the children of this tree view item

Drop(*button='left'*, *pressed=''*)
Drop at the item

EnsureVisible()
Make sure that the TreeView item is visible

Expand()
Expand the children of this tree view item

GetChild(*child_spec*, *exact=False*)
Return the child item of this item

Accepts either a string or an index. If a string is passed then it returns the child item with the best match for the string.

IsChecked()
Return whether the TreeView item is checked or not

IsExpanded()
Indicate that the TreeView item is selected or not

IsSelected()
Indicate that the TreeView item is selected or not

Item()
Return the item itself

Next()
Return the next item

Rectangle(*text_area_rect=True*)
Return the rectangle of the item

If text_area_rect is set to False then it will return the rectangle for the whole item (usually left is equal to 0). Defaults to True - which returns just the rectangle of the text of the item

Select()
Select the TreeView item

StartDragging(*button='left'*, *pressed=''*)
Start dragging the item

State()
Return the state of the item

SubElements()
Return the list of children of this control

Text()
Return the text of the item

children()
Return the direct children of this control

click(*button='left'*, *double=False*, *where='text'*, *pressed=''*)
Click on the treeview item

where can be any one of “text”, “icon”, “button”, “check” defaults to “text”

click_input (*button='left'*, *double=False*, *wheel_dist=0*, *where='text'*, *pressed='‘*)
Click on the treeview item

where can be any one of "text", "icon", "button", "check" defaults to "text"

collapse ()
Collapse the children of this tree view item

drop (*button='left'*, *pressed='‘*)
Drop at the item

ensure_visible ()
Make sure that the TreeView item is visible

expand ()
Expand the children of this tree view item

get_child (*child_spec*, *exact=False*)
Return the child item of this item

Accepts either a string or an index. If a string is passed then it returns the child item with the best match for the string.

is_checked ()
Return whether the TreeView item is checked or not

is_expanded ()
Indicate that the TreeView item is selected or not

is_selected ()
Indicate that the TreeView item is selected or not

item ()
Return the item itself

next_item ()
Return the next item

rectangle (*text_area_rect=True*)
Return the rectangle of the item

If text_area_rect is set to False then it will return the rectangle for the whole item (usually left is equal to 0). Defaults to True - which returns just the rectangle of the text of the item

select ()
Select the TreeView item

start_dragging (*button='left'*, *pressed='‘*)
Start dragging the item

state ()
Return the state of the item

sub_elements ()
Return the list of children of this control

text ()
Return the text of the item

class pywinauto.controls.common_controls.**_listview_item** (*lv_ctrl*, *item_index*, *subitem_index=0*)
Bases: object

Wrapper around ListView items

Check()
Check the ListView item

Click(*button='left'*, *double=False*, *where='text'*, *pressed=''*)
Click on the list view item

where can be any one of "all", "icon", "text", "select", "check" defaults to "text"

ClickInput(*button='left'*, *double=False*, *wheel_dist=0*, *where='text'*, *pressed=''*)
Click on the list view item

where can be any one of "all", "icon", "text", "select", "check" defaults to "text"

Deselect()
Mark the item as not selected

The ListView control must be enabled and visible before an Item can be selected otherwise an exception is raised

EnsureVisible()
Make sure that the ListView item is visible

Image()
Return the image index of the item

Indent()
Return the indent of the item

IsChecked()
Return whether the ListView item is checked or not

IsFocused()
Return True if the item has the focus

IsSelected()
Return True if the item is selected

Item()
Return the item itself (LVITEM instance)

ItemData()
Return the item data (dictionary)

Rectangle(*area='all'*)
Return the rectangle of the item.

Possible `area` values:

- "`all`" Returns the bounding rectangle of the entire item, including the icon and label.
- "`icon`" Returns the bounding rectangle of the icon or small icon.
- "`text`" Returns the bounding rectangle of the item text.
- "`select`" Returns the union of the "icon" and "text" rectangles, but excludes columns in report view.

Select()
Mark the item as selected

The ListView control must be enabled and visible before an Item can be selected otherwise an exception is raised

State()
Return the state of the item

Text()
Return the text of the item

UnCheck()
Uncheck the ListView item

check()
Check the ListView item

click(*button='left'*, *double=False*, *where='text'*, *pressed=''*)
Click on the list view item

where can be any one of "all", "icon", "text", "select", "check" defaults to "text"

click_input(*button='left'*, *double=False*, *wheel_dist=0*, *where='text'*, *pressed=''*)
Click on the list view item

where can be any one of "all", "icon", "text", "select", "check" defaults to "text"

deselect()
Mark the item as not selected

The ListView control must be enabled and visible before an Item can be selected otherwise an exception is raised

ensure_visible()
Make sure that the ListView item is visible

image()
Return the image index of the item

indent()
Return the indent of the item

is_checked()
Return whether the ListView item is checked or not

is_focused()
Return True if the item has the focus

is_selected()
Return True if the item is selected

item()
Return the item itself (LVITEM instance)

item_data()
Return the item data (dictionary)

rectangle(*area='all'*)
Return the rectangle of the item.

Possible `area` values:

- `"all"` Returns the bounding rectangle of the entire item, including the icon and label.
- `"icon"` Returns the bounding rectangle of the icon or small icon.
- `"text"` Returns the bounding rectangle of the item text.
- `"select"` Returns the union of the "icon" and "text" rectangles, but excludes columns in report view.

select()
Mark the item as selected

The ListView control must be enabled and visible before an Item can be selected otherwise an exception is raised

state()
Return the state of the item

text()
Return the text of the item

uncheck()
Uncheck the ListView item

class pywinauto.controls.common_controls.**AnimationWrapper**(*element_info*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows Animation common control

friendlyclassname = 'Animation'

windowclasses = ['SysAnimate32']

class pywinauto.controls.common_controls.**CalendarWrapper**(*element_info*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows Calendar common control

friendlyclassname = 'Calendar'

has_title = False

windowclasses = ['SysMonthCal32']

class pywinauto.controls.common_controls.**ComboBoxExWrapper**(*element_info*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows ComboBoxEx common control

friendlyclassname = 'ComboBoxEx'

has_title = False

windowclasses = ['ComboBoxEx32']

class pywinauto.controls.common_controls.**DateTimePickerWrapper**(*element_info*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows DateTimePicker common control

GetTime()
Get the currently selected time

SetTime(*year*, *month*, *day_of_week*, *day*, *hour*, *minute*, *second*, *milliseconds*)
Get the currently selected time

friendlyclassname = 'DateTimePicker'

get_time()
Get the currently selected time

has_title = False

set_time(*year*, *month*, *day_of_week*, *day*, *hour*, *minute*, *second*, *milliseconds*)
Get the currently selected time

windowclasses = ['SysDateTimePick32']

class pywinauto.controls.common_controls.**HeaderWrapper**(*hwnd*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows ListView Header common control

GetColumnRectangle(*column_index*)
Return the rectangle for the column specified by column_index

GetColumnText (*column_index*)
Return the text for the column specified by column_index

ItemCount ()
Return the number of columns in this header

client_rects ()
Return all the client rectangles for the header control

friendlyclassname = 'Header'

get_column_rectangle (*column_index*)
Return the rectangle for the column specified by column_index

get_column_text (*column_index*)
Return the text for the column specified by column_index

item_count ()
Return the number of columns in this header

texts ()
Return the texts of the Header control

windowclasses = ['SysHeader32', 'msvb_lib_header']

class pywinauto.controls.common_controls.**HotkeyWrapper** (*element_info*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows Hotkey common control

friendlyclassname = 'Hotkey'

has_title = False

windowclasses = ['msctls_hotkey32']

class pywinauto.controls.common_controls.**IPAddressWrapper** (*element_info*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows IPAddress common control

friendlyclassname = 'IPAddress'

has_title = False

windowclasses = ['SysIPAddress32']

class pywinauto.controls.common_controls.**ListViewWrapper** (*hwnd*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows ListView common control

This class derives from HwndWrapper - so has all the methods o that class also

see HwndWrapper.HwndWrapper

Check (*item*)
Check the ListView item

ColumnCount ()
Return the number of columns

ColumnWidths ()
Return a list of all the column widths

Columns ()
Get the information on the columns of the ListView

Deselect(*item*)
Mark the item as not selected

The ListView control must be enabled and visible before an Item can be selected otherwise an exception is raised

GetColumn(*col_index*)
Get the information for a column of the ListView

GetHeaderControl()
Returns the Header control associated with the ListView

GetItem(*item_index*, *subitem_index=0*)
Return the item of the list view"

- **item_index** Can be either an index of the item or a string with the text of the item you want returned.
- **subitem_index** A zero based index of the item you want returned. Defaults to 0.

GetItemRect(*item_index*)
Return the bounding rectangle of the list view item

GetSelectedCount()
Return the number of selected items

IsChecked(*item*)
Return whether the ListView item is checked or not

IsFocused(*item*)
Return True if the item has the focus

IsSelected(*item*)
Return True if the item is selected

Item(*item_index*, *subitem_index=0*)
Return the item of the list view"

- **item_index** Can be either an index of the item or a string with the text of the item you want returned.
- **subitem_index** A zero based index of the item you want returned. Defaults to 0.

ItemCount()
The number of items in the ListView

Items()
Get all the items in the list view

Select(*item*)
Mark the item as selected

The ListView control must be enabled and visible before an Item can be selected otherwise an exception is raised

UnCheck(*item*)
Uncheck the ListView item

check(*item*)
Check the ListView item

column_count()
Return the number of columns

column_widths()
Return a list of all the column widths

columns()
Get the information on the columns of the ListView

deselect(*item*)
Mark the item as not selected

The ListView control must be enabled and visible before an Item can be selected otherwise an exception is raised

friendlyclassname = 'ListView'

get_column(*col_index*)
Get the information for a column of the ListView

get_header_control()
Returns the Header control associated with the ListView

get_item(*item_index*, *subitem_index=0*)
Return the item of the list view"

- **item_index** Can be either an index of the item or a string with the text of the item you want returned.
- **subitem_index** A zero based index of the item you want returned. Defaults to 0.

get_item_rect(*item_index*)
Return the bounding rectangle of the list view item

get_selected_count()
Return the number of selected items

is_checked(*item*)
Return whether the ListView item is checked or not

is_focused(*item*)
Return True if the item has the focus

is_selected(*item*)
Return True if the item is selected

item(*item_index*, *subitem_index=0*)
Return the item of the list view"

- **item_index** Can be either an index of the item or a string with the text of the item you want returned.
- **subitem_index** A zero based index of the item you want returned. Defaults to 0.

item_count()
The number of items in the ListView

items()
Get all the items in the list view

select(*item*)
Mark the item as selected

The ListView control must be enabled and visible before an Item can be selected otherwise an exception is raised

texts()
Get the texts for the ListView control

uncheck(*item*)
Uncheck the ListView item

windowclasses = ['SysListView32', 'WindowsForms\\d*\\.SysListView32\\..*', 'TSysListView', 'ListView20WndC

writable_props
Extend default properties list.

class pywinauto.controls.common_controls.**PagerWrapper**(*element_info*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows Pager common control

GetPosition()
Return the current position of the pager

SetPosition(*pos*)
Set the current position of the pager

friendlyclassname = 'Pager'

get_position()
Return the current position of the pager

set_position(*pos*)
Set the current position of the pager

windowclasses = ['SysPager']

class pywinauto.controls.common_controls.**ProgressWrapper**(*element_info*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows Progress common control

GetPosition()
Return the current position of the progress bar

GetState()
Get the state of the progress bar
State will be one of the following constants:
- PBST_NORMAL
- PBST_ERROR
- PBST_PAUSED

GetStep()
Get the step size of the progress bar

SetPosition(*pos*)
Set the current position of the progress bar

StepIt()
Move the progress bar one step size forward

friendlyclassname = 'Progress'

get_position()
Return the current position of the progress bar

get_step()
Get the step size of the progress bar

has_title = False

set_position(*pos*)
Set the current position of the progress bar

set_state()
Get the state of the progress bar
State will be one of the following constants:

- PBST_NORMAL
- PBST_ERROR
- PBST_PAUSED

step_it()
Move the progress bar one step size forward

windowclasses = ['msctls_progress', 'msctls_progress32']

class pywinauto.controls.common_controls.**ReBarWrapper**(*hwnd*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows ReBar common control

BandCount()
Return the number of bands in the control

GetBand(*band_index*)
Get a band of the ReBar control

GetToolTipsControl()
Return the tooltip control associated with this control

band_count()
Return the number of bands in the control

friendlyclassname = 'ReBar'

get_band(*band_index*)
Get a band of the ReBar control

get_tool_tips_control()
Return the tooltip control associated with this control

texts()
Return the texts of the Rebar

windowclasses = ['ReBarWindow32']

writable_props
Extend default properties list.

class pywinauto.controls.common_controls.**StatusBarWrapper**(*hwnd*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows Status Bar common control

BorderWidths()
Return the border widths of the StatusBar

A dictionary of the 3 available widths is returned: Horizontal - the horizontal width Vertical - The width above and below the status bar parts Inter - The width between parts of the status bar

GetPartRect(*part_index*)
Return the rectangle of the part specified by part_index

GetPartText(*part_index*)
Return the text of the part specified by part_index

PartCount()
Return the number of parts

PartRightEdges()
Return the widths of the parts

border_widths()
Return the border widths of the StatusBar

A dictionary of the 3 available widths is returned: Horizontal - the horizontal width Vertical - The width above and below the status bar parts Inter - The width between parts of the status bar

client_rects()
Return the client rectangles for the control

friendlyclassname = 'StatusBar'

get_part_rect(*part_index*)
Return the rectangle of the part specified by part_index

get_part_text(*part_index*)
Return the text of the part specified by part_index

part_count()
Return the number of parts

part_right_edges()
Return the widths of the parts

texts()
Return the texts for the control

windowclasses = ['msctls_statusbar32', '.*StatusBar', 'WindowsForms\\d*\\.msctls_statusbar32\\..*']

writable_props
Extend default properties list.

class pywinauto.controls.common_controls.**TabControlWrapper**(*hwnd*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows Tab common control

GetSelectedTab()
Return the index of the selected tab

GetTabRect(*tab_index*)
Return the rectangle to the tab specified by tab_index

GetTabText(*tab_index*)
Return the text of the tab

RowCount()
Return the number of rows of tabs

Select(*tab*)
Select the specified tab on the tab control

TabCount()
Return the number of tabs

client_rects()
Return the client rectangles for the Tab Control

friendlyclassname = 'TabControl'

get_properties()
Return the properties of the TabControl as a Dictionary

get_selected_tab()
Return the index of the selected tab

get_tab_rect (*tab_index*)
Return the rectangle to the tab specified by tab_index

get_tab_text (*tab_index*)
Return the text of the tab

row_count ()
Return the number of rows of tabs

select (*tab*)
Select the specified tab on the tab control

tab_count ()
Return the number of tabs

texts ()
Return the texts of the Tab Control

windowclasses = ['SysTabControl32', 'WindowsForms\\d*\\.SysTabControl32\\..*']

class pywinauto.controls.common_controls.**ToolTip** (*ctrl*, *tip_index*)
Bases: `object`

Class that Wraps a single tip from a ToolTip control

class pywinauto.controls.common_controls.**ToolTipsWrapper** (*hwnd*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows ToolTips common control (not fully implemented)

GetTip (*tip_index*)
Return the particular tooltip

GetTipText (*tip_index*)
Return the text of the tooltip

ToolCount ()
Return the number of tooltips

friendlyclassname = 'ToolTips'

get_tip (*tip_index*)
Return the particular tooltip

get_tip_text (*tip_index*)
Return the text of the tooltip

texts ()
Return the text of all the tooltips

tool_count ()
Return the number of tooltips

windowclasses = ['tooltips_class32', '.*ToolTip', '#32774', 'MS_WINNOTE', 'VBBubble']

class pywinauto.controls.common_controls.**ToolbarWrapper** (*hwnd*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows Toolbar common control

Button (*button_identifier*, *exact=True*, *by_tooltip=False*)
Return the button at index button_index

ButtonCount ()
Return the number of buttons on the ToolBar

CheckButton (*button_identifier*, *make_checked*, *exact=True*)
Find where the button is and click it if it's unchecked and vice versa

GetButton (*button_index*)
Return information on the Toolbar button

GetButtonRect (*button_index*)
Get the rectangle of a button on the toolbar

GetButtonStruct (*button_index*)
Return TBBUTTON structure on the Toolbar button

GetToolTipsControl ()
Return the tooltip control associated with this control

MenuBarClickInput (*path*, *app*)
Select menu bar items by path (experimental!)

The path is specified by a list of items separated by '->' each Item can be the zero based index of the item to return prefaced by # e.g. #1.
Example: "#1 -> #0", "#1->#0->#0"

PressButton (*button_identifier*, *exact=True*)
Find where the button is and click it

TipTexts ()
Return the tip texts of the Toolbar (without window text)

button (*button_identifier*, *exact=True*, *by_tooltip=False*)
Return the button at index button_index

button_count ()
Return the number of buttons on the ToolBar

check_button (*button_identifier*, *make_checked*, *exact=True*)
Find where the button is and click it if it's unchecked and vice versa

friendlyclassname = 'Toolbar'

get_button (*button_index*)
Return information on the Toolbar button

get_button_rect (*button_index*)
Get the rectangle of a button on the toolbar

get_button_struct (*button_index*)
Return TBBUTTON structure on the Toolbar button

get_tool_tips_control ()
Return the tooltip control associated with this control

menu_bar_click_input (*path*, *app*)
Select menu bar items by path (experimental!)

The path is specified by a list of items separated by '->' each Item can be the zero based index of the item to return prefaced by # e.g. #1.
Example: "#1 -> #0", "#1->#0->#0"

press_button (*button_identifier*, *exact=True*)
Find where the button is and click it

texts ()
Return the texts of the Toolbar

tip_texts()
Return the tip texts of the Toolbar (without window text)

windowclasses = ['ToolbarWindow32', 'WindowsForms\\d*\\.ToolbarWindow32\\..*', 'Afx:ToolBar:.*']

writable_props
Extend default properties list.

class pywinauto.controls.common_controls.**TrackbarWrapper**(*element_info*)
Bases: pywinauto.controls.HwndWrapper.HwndWrapper

Class that wraps Windows Trackbar common control

friendlyclassname = 'Trackbar'

windowclasses = ['msctls_trackbar']

class pywinauto.controls.common_controls.**TreeViewWrapper**(*hwnd*)
Bases: pywinauto.controls.HwndWrapper.HwndWrapper

Class that wraps Windows TreeView common control

EnsureVisible(*path*)
Make sure that the TreeView item is visible

GetItem(*path*, *exact=False*)
Read the TreeView item

- **path** the path to the item to return. This can be one of the following:
  - A string separated by characters. The first character must be . This string is split on the characters and each of these is used to find the specific child at each level. The represents the root item - so you don't need to specify the root itself.
  - A list/tuple of strings - The first item should be the root element.
  - A list/tuple of integers - The first item the index which root to select.

IsSelected(*path*)
Return True if the item is selected

Item(*path*, *exact=False*)
Read the TreeView item

- **path** the path to the item to return. This can be one of the following:
  - A string separated by characters. The first character must be . This string is split on the characters and each of these is used to find the specific child at each level. The represents the root item - so you don't need to specify the root itself.
  - A list/tuple of strings - The first item should be the root element.
  - A list/tuple of integers - The first item the index which root to select.

ItemCount()
Return the count of the items in the treeview

PrintItems()
Print all items with line indents

Root()
Return the root element of the tree view

Roots()

Select(*path*)
Select the treeview item

ensure_visible(*path*)
Make sure that the TreeView item is visible

friendlyclassname = 'TreeView'

get_item(*path*, *exact=False*)

Read the TreeView item

- **path** the path to the item to return. This can be one of the following:
  - A string separated by characters. The first character must be . This string is split on the characters and each of these is used to find the specific child at each level. The represents the root item - so you don't need to specify the root itself.
  - A list/tuple of strings - The first item should be the root element.
  - A list/tuple of integers - The first item the index which root to select.

get_properties()

Get the properties for the control as a dictionary

is_selected(*path*)

Return True if the item is selected

item(*path*, *exact=False*)

Read the TreeView item

- **path** the path to the item to return. This can be one of the following:
  - A string separated by characters. The first character must be . This string is split on the characters and each of these is used to find the specific child at each level. The represents the root item - so you don't need to specify the root itself.
  - A list/tuple of strings - The first item should be the root element.
  - A list/tuple of integers - The first item the index which root to select.

item_count()

Return the count of the items in the treeview

print_items()

Print all items with line indents

roots()

select(*path*)

Select the treeview item

texts()

Return all the text for the tree view

tree_root()

Return the root element of the tree view

windowclasses = ['SysTreeView32', 'WindowsForms\\d*\\.SysTreeView32\\..*', 'TTreeView', 'TreeList.TreeListC

class pywinauto.controls.common_controls.**UpDownWrapper**(*hwnd*)

Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Class that wraps Windows UpDown common control

Decrement()

Decrement the number in the UpDown control by one

GetBase()

Get the base the UpDown control (either 10 or 16)

GetBuddyControl()

Get the buddy control of the updown control

GetRange()

Return the lower, upper range of the up down control

GetValue()
Get the current value of the UpDown control

Increment()
Increment the number in the UpDown control by one

SetBase(*base_value*)
Get the base the UpDown control (either 10 or 16)

SetValue(*new_pos*)
Set the value of the of the UpDown control to some integer value

decrement()
Decrement the number in the UpDown control by one

friendlyclassname = 'UpDown'

get_base()
Get the base the UpDown control (either 10 or 16)

get_buddy_control()
Get the buddy control of the updown control

get_range()
Return the lower, upper range of the up down control

get_value()
Get the current value of the UpDown control

increment()
Increment the number in the UpDown control by one

set_base(*base_value*)
Get the base the UpDown control (either 10 or 16)

set_value(*new_pos*)
Set the value of the of the UpDown control to some integer value

windowclasses = ['msctls_updown32', 'msctls_updown']

10.3.2 pywinauto.controls.HwndWrapper

Basic wrapping of Windows controls

exception `pywinauto.controls.HwndWrapper.`**ControlNotEnabled**
Bases: `exceptions.RuntimeError`

Raised when a control is not enabled

exception `pywinauto.controls.HwndWrapper.`**ControlNotVisible**
Bases: `exceptions.RuntimeError`

Raised when a control is not visible

`pywinauto.controls.HwndWrapper.`**GetDialogPropsFromHandle**(*hwnd*)
Get the properties of all the controls as a list of dictionaries

class `pywinauto.controls.HwndWrapper.`**HwndMeta**(*name*, *bases*, *attrs*)
Bases: `pywinauto.base_wrapper.BaseMeta`

Metaclass for HwndWrapper objects

static find_wrapper(*element*)
Find the correct wrapper for this native element

re_wrappers = {<_sre.SRE_Pattern object at 0x2b28f6253030>: <class 'pywinauto.controls.common_controls.Too

str_wrappers = {'.*ToolTip': <class 'pywinauto.controls.common_controls.ToolTipsWrapper'>, 'WindowsForms

class `pywinauto.controls.HwndWrapper.`**`HwndWrapper`**(*element_info*)
Bases: `pywinauto.base_wrapper.BaseWrapper`

Default wrapper for controls.

All other wrappers are derived from this.

This class wraps a lot of functionality of underlying windows API features for working with windows.

Most of the methods apply to every single window type. For example you can click() on any window.

Most of the methods of this class are simple wrappers around API calls and as such they try do the simplest thing possible.

An HwndWrapper object can be passed directly to a ctypes wrapped C function - and it will get converted to a Long with the value of it's handle (see ctypes, _as_parameter_).

`Click`(*button=u'left'*, *pressed=u''*, *coords=(0, 0)*, *double=False*, *absolute=False*)
Simulates a mouse click on the control

This method sends WM_* messages to the control, to do a more 'realistic' mouse click use click_input() which uses mouse_event() API to perform the click.

This method does not require that the control be visible on the screen (i.e. it can be hidden beneath another window and it will still work).

`ClientRect`()
Returns the client rectangle of window

The client rectangle is the window rectangle minus any borders that are not available to the control for drawing.

Both top and left are always 0 for this method.

This method returns a RECT structure, Which has attributes - top, left, right, bottom. and has methods width() and height(). See win32structures.RECT for more information.

`ClientRects`()
Return the client rect for each item in this control

It is a list of rectangles for the control. It is frequently over-ridden to extract all rectangles from a control with multiple items.

It is always a list with one or more rectangles:
- First elemtent is the client rectangle of the control
- Subsequent elements contain the client rectangle of any items of the control (e.g. items in a listbox/combobox, tabs in a tabcontrol)

`Close`(*wait_time=0*)
Close the window

Code modified from http://msdn.microsoft.com/msdnmag/issues/02/08/CQA/

`CloseAltF4`()
Close the window by pressing Alt+F4 keys.

`CloseClick`(*button=u'left'*, *pressed=u''*, *coords=(0, 0)*, *double=False*)
Perform a click action that should make the window go away

The only difference from click is that there are extra delays before and after the click action.

ContextHelpID()
Return the Context Help ID of the window

DebugMessage(*text*)
Write some debug text over the window

DoubleClick(*button=u'left'*, *pressed=u''*, *coords=(0, 0)*)
Perform a double click action

DragMouse(*button=u'left'*, *press_coords=(0, 0)*, *release_coords=(0, 0)*, *pressed=u''*)
Drag the mouse

ExStyle()
Returns the Extended style of window

Return value is a long.

Combination of WS_* and specific control specific styles. See HwndWrapper.has_style() to easily check if the window has a particular style.

Font()
Return the font of the window

The font of the window is used to draw the text of that window. It is a structure which has attributes for font name, height, width etc.

See win32structures.LOGFONTW for more information.

Fonts()
Return the font for each item in this control

It is a list of fonts for the control. It is frequently over-ridden to extract all fonts from a control with multiple items.

It is always a list with one or more fonts:
- First elemtent is the control font
- Subsequent elements contain the font of any items of the control (e.g. items in a listbox/combobox, tabs in a tabcontrol)

GetActive()
Return a handle to the active window within the process

GetFocus()
Return the control in the process of this window that has the Focus

GetShowState()
Get the show state and Maximized/minimzed/restored state

Returns a value that is a union of the following
- SW_HIDE the window is hidden.
- SW_MAXIMIZE the window is maximized
- SW_MINIMIZE the window is minimized
- SW_RESTORE the window is in the 'restored' state (neither minimized or maximized)
- SW_SHOW The window is not hidden

GetToolbar()
Get the first child toolbar if it exists

HasExStyle(*exstyle*)
Return True if the control has the specified extended style

HasStyle(*style*)
Return True if the control has the specified style

IsActive()
Whether the window is active or not

IsUnicode()
Whether the window is unicode or not

A window is Unicode if it was registered by the Wide char version of RegisterClass(Ex).

Maximize()
Maximize the window

Menu()
Return the menu of the control

MenuItem(*path*, *exact=False*)
Return the menu item specified by path

Path can be a string in the form "MenuItem->MenuItem->MenuItem..." where each MenuItem is the text of an item at that level of the menu. E.g.

```
File->Export->ExportAsPNG
```

spaces are not important so you could also have written...

```
File -> Export -> Export As PNG
```

MenuItems()
Return the menu items for the dialog

If there are no menu items then return an empty list

MenuSelect(*path*, *exact=False*)
Select the menuitem specified in path

Minimize()
Minimize the window

MoveMouse(*coords=(0, 0)*, *pressed=u''*, *absolute=False*)
Move the mouse by WM_MOUSEMOVE

MoveWindow(*x=None*, *y=None*, *width=None*, *height=None*, *repaint=True*)
Move the window to the new coordinates

- **x** Specifies the new left position of the window. Defaults to the current left position of the window.
- **y** Specifies the new top position of the window. Defaults to the current top position of the window.
- **width** Specifies the new width of the window. Defaults to the current width of the window.
- **height** Specifies the new height of the window. Default to the current height of the window.
- **repaint** Whether the window should be repainted or not. Defaults to True

NotifyParent(*message*, *controlID=None*)
Send the notification message to parent of this control

Owner()
Return the owner window for the window if it exists

Returns None if there is no owner

PopupWindow()
Return owned enabled Popup window wrapper if shown.

If there is no enabled popups at that time, it returns **self**. See MSDN reference: https://msdn.microsoft.com/en-us/library/windows/desktop/ms633515.aspx

Please do not use in production code yet - not tested fully

`PostCommand`(*commandID*)

`PostMessage`(*message*, *wparam=0*, *lparam=0*)
Post a message to the control message queue and return

`PressMouse`(*button=u'left'*, *coords=(0, 0)*, *pressed=u''*)
Press the mouse button

`ReleaseMouse`(*button=u'left'*, *coords=(0, 0)*, *pressed=u''*)
Release the mouse button

`Restore`()
Restore the window

`RightClick`(*pressed=u''*, *coords=(0, 0)*)
Perform a right click action

`Scroll`(*direction*, *amount*, *count=1*, *retry_interval=None*)
Ask the control to scroll itself

direction can be any of "up", "down", "left", "right" **amount** can be one of "line", "page", "end" **count** (optional) the number of times to scroll

`SendCommand`(*commandID*)

`SendMessage`(*message*, *wparam=0*, *lparam=0*)
Send a message to the control and wait for it to return

`SendMessageTimeout`(*message*, *wparam=0*, *lparam=0*, *timeout=None*, *timeoutflags=<MagicMock name='mock.SMTO_NORMAL' id='47454927586000'>*)
Send a message to the control and wait for it to return or to timeout

If no timeout is given then a default timeout of .01 of a second will be used.

`SetApplicationData`(*appdata*)
Application data is data from a previous run of the software

It is essential for running scripts written for one spoke language on a different spoken language

`SetTransparency`(*alpha=120*)
Set the window transparency from 0 to 255 by alpha attribute

`SetWindowText`(*text*, *append=False*)
Set the text of the window

`Style`()
Returns the style of window

Return value is a long.

Combination of WS_* and specific control specific styles. See HwndWrapper.has_style() to easily check if the window has a particular style.

`UserData`()
Extra data associted with the window

This value is a long value that has been associated with the window and rarely has useful data (or at least data that you know the use of).

click (*button=u'left'*, *pressed=u''*, *coords=(0, 0)*, *double=False*, *absolute=False*)
Simulates a mouse click on the control

This method sends WM_* messages to the control, to do a more 'realistic' mouse click use click_input() which uses mouse_event() API to perform the click.

This method does not require that the control be visible on the screen (i.e. it can be hidden beneath another window and it will still work).

client_rect ()
Returns the client rectangle of window

The client rectangle is the window rectangle minus any borders that are not available to the control for drawing.

Both top and left are always 0 for this method.

This method returns a RECT structure, Which has attributes - top, left, right, bottom. and has methods width() and height(). See win32structures.RECT for more information.

client_rects ()
Return the client rect for each item in this control

It is a list of rectangles for the control. It is frequently over-ridden to extract all rectangles from a control with multiple items.

It is always a list with one or more rectangles:

- First elemtent is the client rectangle of the control
- Subsequent elements contain the client rectangle of any items of the control (e.g. items in a listbox/combobox, tabs in a tabcontrol)

close (*wait_time=0*)
Close the window

Code modified from http://msdn.microsoft.com/msdnmag/issues/02/08/CQA/

close_alt_f4 ()
Close the window by pressing Alt+F4 keys.

close_click (*button=u'left'*, *pressed=u''*, *coords=(0, 0)*, *double=False*)
Perform a click action that should make the window go away

The only difference from click is that there are extra delays before and after the click action.

context_help_id ()
Return the Context Help ID of the window

debug_message (*text*)
Write some debug text over the window

double_click (*button=u'left'*, *pressed=u''*, *coords=(0, 0)*)
Perform a double click action

drag_mouse (*button=u'left'*, *press_coords=(0, 0)*, *release_coords=(0, 0)*, *pressed=u''*)
Drag the mouse

exstyle ()
Returns the Extended style of window

Return value is a long.

Combination of WS_* and specific control specific styles. See HwndWrapper.has_style() to easily check if the window has a particular style.

font()
Return the font of the window

The font of the window is used to draw the text of that window. It is a structure which has attributes for font name, height, width etc.

See win32structures.LOGFONTW for more information.

fonts()
Return the font for each item in this control

It is a list of fonts for the control. It is frequently over-ridden to extract all fonts from a control with multiple items.

It is always a list with one or more fonts:
- First elemtent is the control font
- Subsequent elements contain the font of any items of the control (e.g. items in a listbox/combobox, tabs in a tabcontrol)

get_active()
Return a handle to the active window within the process

get_focus()
Return the control in the process of this window that has the Focus

get_show_state()
Get the show state and Maximized/minimzed/restored state

Returns a value that is a union of the following
- SW_HIDE the window is hidden.
- SW_MAXIMIZE the window is maximized
- SW_MINIMIZE the window is minimized
- SW_RESTORE the window is in the 'restored' state (neither minimized or maximized)
- SW_SHOW The window is not hidden

get_toolbar()
Get the first child toolbar if it exists

handle = None

has_exstyle(*exstyle*)
Return True if the control has the specified extended style

has_style(*style*)
Return True if the control has the specified style

is_active()
Whether the window is active or not

is_dialog()
Return true if the control is a top level window

is_unicode()
Whether the window is unicode or not

A window is Unicode if it was registered by the Wide char version of RegisterClass(Ex).

maximize()
Maximize the window

menu()
Return the menu of the control

menu_item(*path*, *exact=False*)
Return the menu item specified by path

Path can be a string in the form "MenuItem->MenuItem->MenuItem..." where each MenuItem is the text of an item at that level of the menu. E.g.

```
File->Export->ExportAsPNG
```

spaces are not important so you could also have written...

```
File -> Export -> Export As PNG
```

menu_items()
Return the menu items for the dialog

If there are no menu items then return an empty list

menu_select(*path*, *exact=False*)
Select the menuitem specified in path

minimize()
Minimize the window

move_mouse(*coords=(0, 0)*, *pressed=u''*, *absolute=False*)
Move the mouse by WM_MOUSEMOVE

move_window(*x=None*, *y=None*, *width=None*, *height=None*, *repaint=True*)
Move the window to the new coordinates

- **x** Specifies the new left position of the window. Defaults to the current left position of the window.
- **y** Specifies the new top position of the window. Defaults to the current top position of the window.
- **width** Specifies the new width of the window. Defaults to the current width of the window.
- **height** Specifies the new height of the window. Default to the current height of the window.
- **repaint** Whether the window should be repainted or not. Defaults to True

notify_parent(*message*, *controlID=None*)
Send the notification message to parent of this control

owner()
Return the owner window for the window if it exists

Returns None if there is no owner

popup_window()
Return owned enabled Popup window wrapper if shown.

If there is no enabled popups at that time, it returns **self**. See MSDN reference: https://msdn.microsoft.com/en-us/library/windows/desktop/ms633515.aspx

Please do not use in production code yet - not tested fully

post_command(*commandID*)

post_message(*message*, *wparam=0*, *lparam=0*)
Post a message to the control message queue and return

press_mouse(*button=u'left'*, *coords=(0, 0)*, *pressed=u''*)
Press the mouse button

release_mouse(*button=u'left'*, *coords=(0, 0)*, *pressed=u''*)
Release the mouse button

restore()
Restore the window

right_click(*pressed=u'‘*, *coords=(0, 0)*)
Perform a right click action

scroll(*direction*, *amount*, *count=1*, *retry_interval=None*)
Ask the control to scroll itself

direction can be any of “up”, “down”, “left”, “right” **amount** can be one of “line”, “page”, “end” **count** (optional) the number of times to scroll

send_command(*commandID*)

send_message(*message*, *wparam=0*, *lparam=0*)
Send a message to the control and wait for it to return

send_message_timeout(*message*, *wparam=0*, *lparam=0*, *timeout=None*, *timeoutflags=<MagicMock name=’mock.SMTO_NORMAL’ id=‘47454927586000’>*)
Send a message to the control and wait for it to return or to timeout

If no timeout is given then a default timeout of .01 of a second will be used.

set_application_data(*appdata*)
Application data is data from a previous run of the software

It is essential for running scripts written for one spoke language on a different spoken language

set_focus()
Set the focus to this control.

Bring the window to the foreground first if necessary.

set_transparency(*alpha=120*)
Set the window transparency from 0 to 255 by alpha attribute

set_window_text(*text*, *append=False*)
Set the text of the window

style()
Returns the style of window

Return value is a long.

Combination of WS_* and specific control specific styles. See HwndWrapper.has_style() to easily check if the window has a particular style.

user_data()
Extra data associted with the window

This value is a long value that has been associated with the window and rarely has useful data (or at least data that you know the use of).

writable_props
Extend default properties list.

exception `pywinauto.controls.HwndWrapper.`**InvalidWindowHandle**(*hwnd*)
Bases: `exceptions.RuntimeError`

Raised when an invalid handle is passed to HwndWrapper

`pywinauto.controls.HwndWrapper.`**get_dialog_props_from_handle**(*hwnd*)
Get the properties of all the controls as a list of dictionaries

10.3.3 pywinauto.controls.menuwrapper

Wrapper around Menu's and Menu items

These wrappers allow you to work easily with menu items. You can select or click on items and check if they are checked or unchecked.

class `pywinauto.controls.menuwrapper.`**`Menu`**(*owner_ctrl*, *menuhandle*, *is_main_menu=True*, *owner_item=None*)

Bases: `object`

A simple wrapper around a menu handle

A menu supports methods for querying the menu and getting it's menu items.

`GetMenuPath`(*instance*, **args*, ***kwargs*)
Check if the instance is accessible

`GetProperties`(*instance*, **args*, ***kwargs*)
Check if the instance is accessible

`Item`(*instance*, **args*, ***kwargs*)
Check if the instance is accessible

`ItemCount`(*instance*, **args*, ***kwargs*)
Check if the instance is accessible

`Items`(*instance*, **args*, ***kwargs*)
Check if the instance is accessible

`get_menu_path`(*instance*, **args*, ***kwargs*)
Check if the instance is accessible

`get_properties`(*instance*, **args*, ***kwargs*)
Check if the instance is accessible

`item`(*instance*, **args*, ***kwargs*)
Check if the instance is accessible

`item_count`(*instance*, **args*, ***kwargs*)
Check if the instance is accessible

`items`(*instance*, **args*, ***kwargs*)
Check if the instance is accessible

exception `pywinauto.controls.menuwrapper.`**`MenuInaccessible`**
Bases: `exceptions.RuntimeError`

Raised when a menu has handle but inaccessible.

class `pywinauto.controls.menuwrapper.`**`MenuInfo`**
Bases: `object`

class `pywinauto.controls.menuwrapper.`**`MenuItem`**(*ctrl*, *menu*, *index*, *on_main_menu=False*)

Bases: `object`

Wrap a menu item

`Click`()
Select the menu item

This will send a message to the parent window that the item was picked.

ClickInput()
Click on the menu item in a more realistic way

If the menu is open it will click with the mouse event on the item. If the menu is not open each of it's parent's will be opened until the item is visible.

FriendlyClassName()
Return friendly class name

GetProperties()
Return the properties for the item as a dict

If this item opens a sub menu then call Menu.get_properties() to return the list of items in the sub menu. This is avialable under the 'menu_items' key.

ID()
Return the ID of this menu item

Index()
Return the index of this menu item

IsChecked()
Return True if the item is checked.

IsEnabled()
Return True if the item is enabled.

Rectangle()
Get the rectangle of the menu item

Select()
Select the menu item

This will send a message to the parent window that the item was picked.

State()
Return the state of this menu item

SubMenu()
Return the SubMenu or None if no submenu

Text()
Return the text of this menu item

Type()
Return the Type of this menu item

Main types are MF_STRING, MF_BITMAP, MF_SEPARATOR.

See https://msdn.microsoft.com/en-us/library/windows/desktop/ms647980.aspx for further information.

click()
Select the menu item

This will send a message to the parent window that the item was picked.

click_input()
Click on the menu item in a more realistic way

If the menu is open it will click with the mouse event on the item. If the menu is not open each of it's parent's will be opened until the item is visible.

friendly_class_name()
Return friendly class name

get_properties()
Return the properties for the item as a dict

If this item opens a sub menu then call Menu.get_properties() to return the list of items in the sub menu. This is avialable under the 'menu_items' key.

index()
Return the index of this menu item

is_checked()
Return True if the item is checked.

is_enabled()
Return True if the item is enabled.

item_id()
Return the ID of this menu item

item_type()
Return the Type of this menu item

Main types are MF_STRING, MF_BITMAP, MF_SEPARATOR.

See https://msdn.microsoft.com/en-us/library/windows/desktop/ms647980.aspx for further information.

rectangle()
Get the rectangle of the menu item

select()
Select the menu item

This will send a message to the parent window that the item was picked.

state()
Return the state of this menu item

sub_menu()
Return the SubMenu or None if no submenu

text()
Return the text of this menu item

class pywinauto.controls.menuwrapper.**MenuItemInfo**
Bases: object

exception pywinauto.controls.menuwrapper.**MenuItemNotEnabled**
Bases: exceptions.RuntimeError

Raised when a menu item is not enabled

pywinauto.controls.menuwrapper.**ensure_accessible**(*method*)
Decorator for Menu instance methods

10.3.4 pywinauto.controls.win32_controls

Wraps various standard windows controls

class pywinauto.controls.win32_controls.**ButtonWrapper**(*hwnd*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Wrap a windows Button control

Check()
Check a checkbox

CheckByClick()
Check the CheckBox control by click() method

CheckByClickInput()
Check the CheckBox control by click_input() method

GetCheckState()
Return the check state of the checkbox

The check state is represented by an integer 0 - unchecked 1 - checked 2 - indeterminate

The following constants are defined in the win32defines module BST_UNCHECKED = 0 BST_CHECKED = 1 BST_INDETERMINATE = 2

SetCheckIndeterminate()
Set the checkbox to indeterminate

UnCheck()
Uncheck a checkbox

UncheckByClick()
Uncheck the CheckBox control by click() method

UncheckByClickInput()
Uncheck the CheckBox control by click_input() method

can_be_label = True

check()
Check a checkbox

check_by_click()
Check the CheckBox control by click() method

check_by_click_input()
Check the CheckBox control by click_input() method

click(**args*, ***kwargs*)
Click the Button control

friendly_class_name()
Return the friendly class name of the button

Windows controls with the class "Button" can look like different controls based on their style. They can look like the following controls:

- Buttons, this method returns "Button"
- CheckBoxes, this method returns "CheckBox"
- RadioButtons, this method returns "RadioButton"
- GroupBoxes, this method returns "GroupBox"

friendlyclassname = u'Button'

get_check_state()
Return the check state of the checkbox

The check state is represented by an integer 0 - unchecked 1 - checked 2 - indeterminate

The following constants are defined in the win32defines module BST_UNCHECKED = 0 BST_CHECKED = 1 BST_INDETERMINATE = 2

is_dialog()
Buttons are never dialogs so return False

set_check_indeterminate()
Set the checkbox to indeterminate

uncheck()
Uncheck a checkbox

uncheck_by_click()
Uncheck the CheckBox control by click() method

uncheck_by_click_input()
Uncheck the CheckBox control by click_input() method

windowclasses = [u'Button', u'.*Button', u'WindowsForms\\d*\\.BUTTON\\..*', u'.*CheckBox']

class pywinauto.controls.win32_controls.**ComboBoxWrapper**(*hwnd*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Wrap a windows ComboBox control

DroppedRect()
Get the dropped rectangle of the combobox

ItemCount()
Return the number of items in the combobox

ItemData(*item*)
Returns the item data associated with the item if any

ItemTexts()
Return the text of the items of the combobox

Select(*item*)
Select the ComboBox item

item can be either a 0 based index of the item to select or it can be the string that you want to select

SelectedIndex()
Return the selected index

SelectedText()
Return the selected text

dropped_rect()
Get the dropped rectangle of the combobox

friendlyclassname = u'ComboBox'

get_properties()
Return the properties of the control as a dictionary

has_title = False

item_count()
Return the number of items in the combobox

item_data(*item*)
Returns the item data associated with the item if any

item_texts()
Return the text of the items of the combobox

select (*item*)
Select the ComboBox item

item can be either a 0 based index of the item to select or it can be the string that you want to select

selected_index ()
Return the selected index

selected_text ()
Return the selected text

texts ()
Return the text of the items in the combobox

windowclasses = [u'ComboBox', u'WindowsForms\\d*\\.COMBOBOX\\..*', u'.*ComboBox']

writable_props
Extend default properties list.

class pywinauto.controls.win32_controls.**DialogWrapper** (*hwnd*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Wrap a dialog

ClientAreaRect ()
Return the client area rectangle

From MSDN: The client area of a control is the bounds of the control, minus the nonclient elements such as scroll bars, borders, title bars, and menus.

HideFromTaskbar ()
Hide the dialog from the Windows taskbar

IsInTaskbar ()
Check whether the dialog is shown in the Windows taskbar

Thanks to David Heffernan for the idea: http://stackoverflow.com/questions/30933219/hide-window-from-taskbar-without-using-ws-ex-toolwindow A window is represented in the taskbar if: It has no owner and it does not have the WS_EX_TOOLWINDOW extended style, or it has the WS_EX_APPWINDOW extended style.

RunTests (*tests_to_run=None*, *ref_controls=None*)
Run the tests on dialog

ShowInTaskbar ()
Show the dialog in the Windows taskbar

WriteToXML (*filename*)
Write the dialog an XML file (requires elementtree)

can_be_label = True

client_area_rect ()
Return the client area rectangle

From MSDN: The client area of a control is the bounds of the control, minus the nonclient elements such as scroll bars, borders, title bars, and menus.

force_close ()
Close the dialog forcefully using WM_QUERYENDSESSION and return the result

Window has let us know that it doesn't want to die - so we abort this means that the app is not hung - but knows it doesn't want to close yet - e.g. it is asking the user if they want to save.

friendlyclassname = u'Dialog'

hide_from_taskbar()
Hide the dialog from the Windows taskbar

is_in_taskbar()
Check whether the dialog is shown in the Windows taskbar

Thanks to David Heffernan for the idea: http://stackoverflow.com/questions/30933219/hide-window-from-taskbar-without-using-ws-ex-toolwindow A window is represented in the taskbar if: It has no owner and it does not have the WS_EX_TOOLWINDOW extended style, or it has the WS_EX_APPWINDOW extended style.

run_tests(*tests_to_run=None*, *ref_controls=None*)
Run the tests on dialog

show_in_taskbar()
Show the dialog in the Windows taskbar

write_to_xml(*filename*)
Write the dialog an XML file (requires elementtree)

class pywinauto.controls.win32_controls.**EditWrapper**(*hwnd*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Wrap a windows Edit control

GetLine(*line_index*)
Return the line specified

LineCount()
Return how many lines there are in the Edit

LineLength(*line_index*)
Return how many characters there are in the line

Select(*start=0*, *end=None*)
Set the edit selection of the edit control

SelectionIndices()
The start and end indices of the current selection

SetEditText(*text*, *pos_start=None*, *pos_end=None*)
Set the text of the edit control

SetText(*text*, *pos_start=None*, *pos_end=None*)
Set the text of the edit control

TextBlock()
Get the text of the edit control

friendlyclassname = u'Edit'

get_line(*line_index*)
Return the line specified

has_title = False

line_count()
Return how many lines there are in the Edit

line_length(*line_index*)
Return how many characters there are in the line

select (*start=0*, *end=None*)
Set the edit selection of the edit control

selection_indices ()
The start and end indices of the current selection

set_edit_text (*text*, *pos_start=None*, *pos_end=None*)
Set the text of the edit control

set_text (*text*, *pos_start=None*, *pos_end=None*)
Set the text of the edit control

set_window_text (*text*, *append=False*)
Override set_window_text for edit controls because it should not be used for Edit controls.

Edit Controls should either use set_edit_text() or type_keys() to modify the contents of the edit control.

text_block ()
Get the text of the edit control

texts ()
Get the text of the edit control

windowclasses = [u'Edit', u'.*Edit', u'TMemo', u'WindowsForms\\d*\\.EDIT\\..*', u'ThunderTextBox', u'Thund

writable_props
Extend default properties list.

class pywinauto.controls.win32_controls.**ListBoxWrapper** (*hwnd*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Wrap a windows ListBox control

GetItemFocus ()
Retrun the index of current selection in a ListBox

IsSingleSelection ()
Check whether the listbox has single selection mode.

ItemCount ()
Return the number of items in the ListBox

ItemData (*i*)
Return the item_data if any associted with the item

ItemRect (*item*)
Return the rect of the item

ItemTexts ()
Return the text of the items of the listbox

Select (*item*, *select=True*)
Select the ListBox item

item can be either a 0 based index of the item to select or it can be the string that you want to select

SelectedIndices ()
The currently selected indices of the listbox

SetItemFocus (*item*)
Set the ListBox focus to the item at index

friendlyclassname = u'ListBox'

get_item_focus()
Retrun the index of current selection in a ListBox

has_title = False

is_single_selection()
Check whether the listbox has single selection mode.

item_count()
Return the number of items in the ListBox

item_data(*i*)
Return the item_data if any associted with the item

item_rect(*item*)
Return the rect of the item

item_texts()
Return the text of the items of the listbox

select(*item*, *select=True*)
Select the ListBox item

item can be either a 0 based index of the item to select or it can be the string that you want to select

selected_indices()
The currently selected indices of the listbox

set_item_focus(*item*)
Set the ListBox focus to the item at index

texts()
Return the texts of the control

windowclasses = [u'ListBox', u'WindowsForms\\d*\\.LISTBOX\\..*', u'.*ListBox']

writable_props
Extend default properties list.

class pywinauto.controls.win32_controls.**PopupMenuWrapper**(*element_info*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Wrap a Popup Menu

friendlyclassname = u'PopupMenu'

has_title = False

is_dialog()
Return whether it is a dialog

windowclasses = [u'#32768']

class pywinauto.controls.win32_controls.**StaticWrapper**(*hwnd*)
Bases: *pywinauto.controls.HwndWrapper.HwndWrapper*

Wrap a windows Static control

can_be_label = True

friendlyclassname = u'Static'

windowclasses = [u'Static', u'WindowsForms\\d*\\.STATIC\\..*', u'TPanel', u'.*StaticText']

10.4 Sendkeys

10.4.1 pywinauto.SendKeysCtypes

Check that SendInput can work the way we want it to

The tips and tricks at http://www.pinvoke.net/default.aspx/user32.sendinput is useful!

exception `pywinauto.SendKeysCtypes.`**`KeySequenceError`**
Exception raised when a key sequence string has a syntax error

`pywinauto.SendKeysCtypes.`**`SendKeys`**(*keys*, *pause=0.05*, *with_spaces=False*, *with_tabs=False*, *with_newlines=False*, *turn_off_numlock=True*)
Parse the keys and type them

10.5 Included 3rd party modules

10.5.1 pywinauto.six

Utilities for writing code that runs on Python 2 and 3

class `pywinauto.six.`**`Iterator`**

`next`()

`pywinauto.six.`**`add_metaclass`**(*metaclass*)
Class decorator for creating a class with a metaclass.

`pywinauto.six.`**`assertCountEqual`**(*self*, **args*, ***kwargs*)

`pywinauto.six.`**`assertRaisesRegex`**(*self*, **args*, ***kwargs*)

`pywinauto.six.`**`assertRegex`**(*self*, **args*, ***kwargs*)

`pywinauto.six.`**`b`**(*s*)
Byte literal

`pywinauto.six.`**`byte2int`**(*bs*)

`pywinauto.six.`**`create_bound_method`**(*func*, *obj*)

`pywinauto.six.`**`create_unbound_method`**(*func*, *cls*)

`pywinauto.six.`**`get_unbound_function`**(*unbound*)
Get the function out of a possibly unbound function

`pywinauto.six.`**`indexbytes`**(*buf*, *i*)

`pywinauto.six.`**`iteritems`**(*d*, ***kw*)
Return an iterator over the (key, value) pairs of a dictionary.

`pywinauto.six.`**`iterkeys`**(*d*, ***kw*)
Return an iterator over the keys of a dictionary.

`pywinauto.six.`**`iterlists`**(*d*, ***kw*)
Return an iterator over the (key, [values]) pairs of a dictionary.

`pywinauto.six.`**`itervalues`**(*d*, ***kw*)
Return an iterator over the values of a dictionary.

`pywinauto.six.`**`python_2_unicode_compatible`**(*klass*)

A decorator that defines __unicode__ and __str__ methods under Python 2. Under Python 3 it does nothing.

To support Python 2 and 3 with a single code base, define a __str__ method returning text and apply this decorator to the class.

`pywinauto.six.`**`u`**(*s*)

Text literal

`pywinauto.six.`**`with_metaclass`**(*meta, *bases*)

Create a base class with a metaclass.

`pywinauto.six.`**`wraps`**(*wrapped, assigned=('__module__', '__name__', '__doc__'), updated=('__dict__',)*)

10.6 Pre-supplied tests

10.6.1 pywinauto.tests.allcontrols

Get All Controls Test

What is checked This test does no actual testing it just returns each control.

How is it checked A loop over all the controls in the dialog is made and each control added to the list of bugs

When is a bug reported For each control.

Bug Extra Information There is no extra information associated with this bug type

Is Reference dialog needed No,but if available the reference control will be returned with the localised control.

False positive bug reports Not possible

Test Identifier The identifier for this test/bug is "AllControls"

`pywinauto.tests.allcontrols.`**`AllControlsTest`**(*windows*)

Returns just one bug for each control

10.6.2 pywinauto.tests.asianhotkey

Asian Hotkey Format Test

What is checked

This test checks whether the format for shortcuts/hotkeys follows the standards for localised Windows applications. This format is {localised text}({uppercase hotkey}) so for example if the English control is "&Help" the localised control for Asian languages should be "LocHelp(H)"

How is it checked

After checking whether this control displays hotkeys it examines the 1st string of the control to make sure that the format is correct. If the reference control is available then it also makes sure that the hotkey character is the same as the reference. Controls with a title of less than 4 characters are ignored. This has been done to avoid false positive bug reports for strings like "&X:".

When is a bug reported

A bug is reported when a control has a hotkey and it is not in the correct format. Also if the reference control is available a bug will be reported if the hotkey character is not the same as used in the reference

Bug Extra Information

This test produces 2 different types of bug: BugType: "AsianHotkeyFormat" There is no extra information associated with this bug type

BugType: "AsianHotkeyDiffRef"

There is no extra information associated with this bug type

Is Reference dialog needed

The reference dialog is not needed. If it is unavailable then only bugs of type "AsianHotkeyFormat" will be reported, bug of type "AsianHotkeyDiffRef" will not be found.

False positive bug reports

There should be very few false positive bug reports when testing Asian software. If a string is very short (eg "&Y:") but is padded with spaces then it will get reported.

Test Identifier

The identifier for this test/bug is "AsianHotkeyTests"

`pywinauto.tests.asianhotkey.`**`AsianHotkeyTest`**`(`*windows*`)`
Return the repeated hotkey errors

10.6.3 pywinauto.tests.comboboxdroppedheight

ComboBox dropped height Test

What is checked It is ensured that the height of the list displayed when the combobox is dropped down is not less than the height of the reference.

How is it checked The value for the dropped rectangle can be retrieved from windows. The height of this rectangle is calculated and compared against the reference height.

When is a bug reported If the height of the dropped rectangle for the combobox being checked is less than the height of the reference one then a bug is reported.

Bug Extra Information There is no extra information associated with this bug type

Is Reference dialog needed The reference dialog is necessary for this test.

False positive bug reports No false bugs should be reported. If the font of the localised control has a smaller height than the reference then it is possible that the dropped rectangle could be of a different size.

Test Identifier The identifier for this test/bug is "ComboBoxDroppedHeight"

`pywinauto.tests.comboboxdroppedheight.`**`ComboBoxDroppedHeightTest`**`(`*windows*`)`
Check if each combobox height is the same as the reference

10.6.4 pywinauto.tests.comparetoreffont

`pywinauto.tests.comparetoreffont.`**`CompareToRefFontTest`**`(`*windows*`)`
Compare the font to the font of the reference control

10.6.5 pywinauto.tests.leadtrailspaces

Different Leading and Trailing Spaces Test

What is checked Checks that the same space characters (<space>, <tab>, <enter>, <vertical tab>) are before and after all non space characters in the title of the control when compared to the reference control.

How is it checked Find the 1st non-space character, and the characters of the title up to that are the leading spaces. Find the last non-space character, and the characters of the title after that are the trailing spaces. These are then compared to the lead and trail spaces from the reference control and if they are not exactly the then a bug is reported.

When is a bug reported When either the leading or trailing spaces of the control being tested does not match the equivalent spaces of the reference control exactly.

Bug Extra Information The bug contains the following extra information

- **Lead-Trail** Whether this bug report is for the leading or trailing spaces of the control, String

  This will be either:

  - "Leading" bug relating to leading spaces
  - "Trailing" bug relating to trailing spaces
- **Ref** The leading or trailings spaces of the reference string (depending on Lead-Trail value), String
- **Loc** The leading or trailings spaces of the local string (depending on Lead-Trail value), String

Is Reference dialog needed This test will not run if the reference controls are not available.

False positive bug reports This is usually not a very important test, so if it generates many false positives then we should consider removing it.

Test Identifier The identifier for this test/bug is "LeadTrailSpaces"

`pywinauto.tests.leadtrailspaces.`**`GetLeadSpaces`**(*title*)
Return the leading spaces of the string

`pywinauto.tests.leadtrailspaces.`**`GetTrailSpaces`**(*title*)
Return the trailing spaces of the string

`pywinauto.tests.leadtrailspaces.`**`LeadTrailSpacesTest`**(*windows*)
Return the leading/trailing space bugs for the windows

10.6.6 pywinauto.tests.miscvalues

Miscellaneous Control properties Test

What is checked This checks various values related to a control in windows. The values tested are class_name The class type of the control style The Style of the control (GetWindowLong) exstyle The Extended Style of the control (GetWindowLong) help_id The Help ID of the control (GetWindowLong) control_id The Control ID of the control (GetWindowLong) user_data The User Data of the control (GetWindowLong) Visibility Whether the control is visible or not

How is it checked After retrieving the information for the control we compare it to the same information from the reference control.

When is a bug reported If the information does not match then a bug is reported.

Bug Extra Information The bug contains the following extra information Name Description ValueType What value is incorrect (see above), String Ref The reference value converted to a string, String Loc The localised value converted to a string, String

Is Reference dialog needed This test will not run if the reference controls are not available.

False positive bug reports Some values can change easily without any bug being caused, for example User Data is actually meant for programmers to store information for the control and this can change every time the software is run.

Test Identifier The identifier for this test/bug is "MiscValues"

`pywinauto.tests.miscvalues.`**`MiscValuesTest`**`(`*windows*`)`
Return the bugs from checking miscelaneous values of a control

10.6.7 pywinauto.tests.missalignment

Missalignment Test

What is checked This test checks that if a set of controls were aligned on a particular axis in the reference dialog that they are all aligned on the same axis.

How is it checked A list of all the reference controls that are aligned is created (ie more than one control with the same Top, Left, Bottom or Right coordinates). These controls are then analysed in the localised dialog to make sure that they are all aligned on the same axis.

When is a bug reported A bug is reported when any of the controls that were aligned in the reference dialog are no longer aligned in the localised control.

Bug Extra Information The bug contains the following extra information Name Description Alignment-Type This is either LEFT, TOP, RIGHT or BOTTOM. It tells you how the controls were aligned in the reference dialog. String AlignmentRect Gives the smallest rectangle that surrounds ALL the controls concerned in the bug, rectangle

Is Reference dialog needed This test cannot be performed without the reference control. It is required to see which controls should be aligned.

False positive bug reports It is quite possible that this test reports false positives: 1. Where the controls only just happen to be aligned in the reference dialog (by coincidence) 2. Where the control does not have a clear boundary (for example static labels or checkboxes) they may be miss-aligned but it is not noticeable that they are not.

Test Identifier The identifier for this test/bug is "Missalignment"

`pywinauto.tests.missalignment.`**`MissalignmentTest`**`(`*windows*`)`
Run the test on the windows passed in

10.6.8 pywinauto.tests.missingextrastring

Different number of special character sequences Test

What is checked This test checks to make sure that certain special character sequences appear the in the localised if they appear in the reference title strings. These strings usually mean something to the user but the software internally does not care if they exist or not. The list that is currently checked is: ">>", ">", "<<", "<", ":"(colon), "...", "&&", "&", ""

How is it checked For each of the string to check for we make sure that if it appears in the reference that it also appears in the localised title.

When is a bug reported

- If the reference has one of the text strings but the localised does not a bug is reported.
- If the localised has one of the text strings but the reference does not a bug is reported.

Bug Extra Information The bug contains the following extra information

MissingOrExtra Whether the characters are missing or extra from the controls being check as compared to the reference, (String with following possible values)

- "MissingCharacters" The characters are in the reference but not in the localised.
- "ExtraCharacters" The characters are not in the reference but are in the localised.

MissingOrExtraText What character string is missing or added, String

Is Reference dialog needed This test will not run if the reference controls are not available.

False positive bug reports Currently this test is at a beta stage filtering of the results is probably necessary at the moment.

Test Identifier The identifier for this test/bug is "MissingExtraString"

`pywinauto.tests.missingextrastring.`**`MissingExtraStringTest`**(*windows*)
    Return the errors from running the test

10.6.9 pywinauto.tests.overlapping

Overlapping Test

What is checked The overlapping test checks for controls that occupy the same space as some other control in the dialog.

- If the reference controls are available check for each pair of controls:
  - If controls are exactly the same size and position in reference then make sure that they are also in the localised.
  - If a reference control is wholly contained in another make sure that the same happens for the controls being tested.
- If the reference controls are not available only the following check can be done
  - If controls are overlapped in localised report a bug (if reference is available it is used just to say if this overlapping happens in reference also)

How is it checked Various tests are performed on each pair of controls to see if any of the above conditions are met. The most specific tests that can be performed are done 1st so that the bugs reported are as specific as possible. I.e. we report that 2 controls are not exactly overlapped when they should be rather than jut reporting that they are overlapped which contains less information.

When is a bug reported A bug is reported when:

- controls are overlapped (but not contained wholly, and not exactly overlapped)
- reference controls are exactly overlapped but they are not in tested dialog
- one reference control is wholly contained in another but not in tested dialog

Bug Extra Information This test produces 3 different types of bug: BugType: "Overlapping" Name Description OverlappedRect <What this info is>, rectangle

BugType - "NotContainedOverlap" There is no extra information associated with this bug type

BugType - "NotExactOverlap" There is no extra information associated with this bug type

Is Reference dialog needed For checking whether controls should be exactly overlapped and whether they should be wholly contained the reference controls are necessary. If the reference controls are not available then only simple overlapping of controls will be checked.

False positive bug reports If there are controls in the dialog that are not visible or are moved dynamically it may cause bugs to be reported that do not need to be logged. If necessary filter out bugs with hidden controls.

Test Identifier The identifier for this test is "Overlapping"

class `pywinauto.tests.overlapping.`**`OptRect`**

`pywinauto.tests.overlapping.`**`OverlappingTest`**`(`*windows*`)`
Return the repeated hotkey errors

10.6.10 pywinauto.tests.repeatedhotkey

Repeated Hotkeys Test

What is checked This test checks all the controls in a dialog to see if there are controls that use the same hotkey character.

How is it checked A list of all the hotkeys (converted to uppercase) used in the dialog is created. Then this list is examined to see if any hotkeys are used more than once. If any are used more than once a list of all the controls that use this hotkey are compiled to be used in the bug report.

When is a bug reported If more than one control has the same hotkey then a bug is reported.

Bug Extra Information The bug contains the following extra information Name Description Repeated-Hotkey This is the hotkey that is repeated between the 2 controls converted to uppercase, String Char-sUsedInDialog This is a list of all the hotkeys used in the dialog, String AllCharsInDialog This is a list of all the characters in the dialog for controls that have a hotkeys, String AvailableInControlS A list of the available characters for each control. Any of the characters in this list could be used as the new hotkey without conflicting with any existing hotkey.

Is Reference dialog needed The reference dialog does not need to be available. If it is available then for each bug discovered it is checked to see if it is a problem in the reference dialog. NOTE: Checking the reference dialog is not so exact here! Only when the equivalent controls in the reference dialog all have the hotkeys will it be reported as being in the reference also. I.e. if there are 3 controls with the same hotkey in the Localised software then those same controls in the reference dialog must have the same hotkey for it to be reported as existing in the reference also.

False positive bug reports There should be very few false positives from this test. Sometimes a control only has one or 2 characters eg "X:" and it is impossible to avoid a hotkey clash. Also for Asian languages hotkeys should be the same as the US software so probably this test should be run on those languages.

Test Identifier The identifier for this test/bug is "RepeatedHotkey"

`pywinauto.tests.repeatedhotkey.`**`GetHotkey`**`(`*text*`)`
Return the position and character of the hotkey

`pywinauto.tests.repeatedhotkey.`**`ImplementsHotkey`**`(`*win*`)`
checks whether a control interprets & character to be a hotkey

`pywinauto.tests.repeatedhotkey.`**`RepeatedHotkeyTest`**`(`*windows*`)`
Return the repeated hotkey errors

10.6.11 pywinauto.tests.translation

Translation Test

What is checked This checks for controls which appear not to be translated.

How is it checked It compares the text of the localised and reference controls.

If there are more than string in the control then each item is searched for in the US list of titles (so checking is not order dependent). The indices for the untranslated strings are returned in a comma separated string. Also the untranslated strings themselves are returned (all as one string). These strings are not escaped and are delimited as "string1","string2",..."stringN".

When is a bug reported

> If the text of the localised control is identical to the reference control (in case, spacing i.e. a binary compare) then it will be flagged as untranslated. Otherwise the control is treated as translated.

Note: This is the method to return the least number of bugs. If there are differences in any part of the string (e.g. a path or variable name) but the rest of the string is untranslated then a bug will not be highlighted

Bug Extra Information The bug contains the following extra information Name Description Strings The list of the untranslated strings as explained above StringIndices The list of indices (0 based) that are untranslated. This will usually be 0 but if there are many strings in the control untranslated it will report ALL the strings e.g. 0,2,5,19,23

Is Reference dialog needed The reference dialog is always necessary.

False positive bug reports False positive bugs will be reported in the following cases. - The title of the control stays the same as the US because the translation is the same as the English text(e.g. Name: in German) - The title of the control is not displayed (and not translated). This can sometimes happen if the programmer displays something else on the control after the dialog is created.

Test Identifier The identifier for this test/bug is "Translation"

`pywinauto.tests.translation.`**`TranslationTest`**`(`*windows*`)`

> Returns just one bug for each control

10.6.12 pywinauto.tests.truncation

Truncation Test

What is checked Checks for controls where the text does not fit in the space provided by the control.

How is it checked There is a function in windows (DrawText) that allows us to find the size that certain text will need. We use this function with correct fonts and other relevant information for the control to be as accurate as possible.

When is a bug reported When the calculated required size for the text is greater than the size of the space available for displaying the text.

Bug Extra Information The bug contains the following extra information Name Description Strings The list of the truncated strings as explained above StringIndices The list of indices (0 based) that are truncated. This will often just be 0 but if there are many strings in the control untranslated it will report ALL the strings e.g. 0,2,5,19,23

Is Reference dialog needed The reference dialog does not need to be available. If it is available then for each bug discovered it is checked to see if it is a problem in the reference dialog.

False positive bug reports Certain controls do not display the text that is the title of the control, if this is not handled in a standard manner by the software then DLGCheck will report that the string is truncated.

Test Identifier The identifier for this test/bug is "Truncation"

`pywinauto.tests.truncation.`**`TruncationTest`**`(`*windows*`)`

> Actually do the test

10.7 Internal modules

10.7.1 pywinauto.controlproperties

Wrap

class `pywinauto.controlproperties.`**`ControlProps`**`(`**args*, ***kwargs*`)`
Wrap controls read from a file to resemble hwnd controls

`HasExStyle``(`*exstyle*`)`

`HasStyle``(`*style*`)`

`WindowText``()`

`window_text``()`

class `pywinauto.controlproperties.`**`FuncWrapper`**`(`*value*`)`
Little class to allow attribute access to return a callable object

`pywinauto.controlproperties.`**`GetMenuBlocks`**`(`*ctrls*`)`

`pywinauto.controlproperties.`**`MenuBlockAsControls`**`(`*menuItems*, *parentage=None*`)`

`pywinauto.controlproperties.`**`MenuItemAsControl`**`(`*menuItem*`)`
Make a menu item look like a control for tests

`pywinauto.controlproperties.`**`SetReferenceControls`**`(`*controls*, *refControls*`)`
Set the reference controls for the controls passed in

This does some minor checking as following:

- test that there are the same number of reference controls as controls - fails with an exception if there are not
- test if all the ID's are the same or not

10.7.2 pywinauto.handleprops

Functions to retrieve properties from a window handle

These are implemented in a procedural way so as to to be useful to other modules with the least conceptual overhead

`pywinauto.handleprops.`**`children`**`(`*handle*`)`
Return a list of handles to the children of this window

`pywinauto.handleprops.`**`classname`**`(`*handle*`)`
Return the class name of the window

`pywinauto.handleprops.`**`clientrect`**`(`*handle*`)`
Return the client rectangle of the control

`pywinauto.handleprops.`**`contexthelpid`**`(`*handle*`)`
Return the context help id of the window

`pywinauto.handleprops.`**`controlid`**`(`*handle*`)`
Return the ID of the control

`pywinauto.handleprops.`**`dumpwindow`**`(`*handle*`)`
Dump a window to a set of properties

`pywinauto.handleprops.`**`exstyle`**`(`*handle*`)`
Return the extended style of the window

`pywinauto.handleprops.`**`font`**`(`*handle*`)`
Return the font as a LOGFONTW of the window

`pywinauto.handleprops.`**`has_exstyle`**`(`*handle*, *tocheck*`)`
Return True if the control has extended style tocheck

`pywinauto.handleprops.`**`has_style`**`(`*handle*, *tocheck*`)`
Return True if the control has style tocheck

`pywinauto.handleprops.`**`is64bitbinary`**`(`*filename*`)`
Check if the file is 64-bit binary

`pywinauto.handleprops.`**`is64bitprocess`**`(`*process_id*`)`
Return True if the specified process is a 64-bit process on x64

Return False if it is only a 32-bit process running under Wow64. Always return False for x86.

`pywinauto.handleprops.`**`is_toplevel_window`**`(`*handle*`)`
Return whether the window is a top level window or not

`pywinauto.handleprops.`**`isenabled`**`(`*handle*`)`
Return True if the window is enabled

`pywinauto.handleprops.`**`isunicode`**`(`*handle*`)`
Return True if the window is a Unicode window

`pywinauto.handleprops.`**`isvisible`**`(`*handle*`)`
Return True if the window is visible

`pywinauto.handleprops.`**`iswindow`**`(`*handle*`)`
Return True if the handle is a window

`pywinauto.handleprops.`**`parent`**`(`*handle*`)`
Return the handle of the parent of the window

`pywinauto.handleprops.`**`processid`**`(`*handle*`)`
Return the ID of process that controls this window

`pywinauto.handleprops.`**`rectangle`**`(`*handle*`)`
Return the rectangle of the window

`pywinauto.handleprops.`**`style`**`(`*handle*`)`
Return the style of the window

`pywinauto.handleprops.`**`text`**`(`*handle*`)`
Return the text of the window

`pywinauto.handleprops.`**`userdata`**`(`*handle*`)`
Return the value of any user data associated with the window

10.7.3 pywinauto.XMLHelpers

Module containing operations for reading and writing dialogs as XML

`pywinauto.XMLHelpers.`**`ReadPropertiesFromFile`**`(`*filename*`)`
Return a list of controls from XML file filename

`pywinauto.XMLHelpers.`**`WriteDialogToFile`**`(`*filename*, *props*`)`
Write the props to the file

props can be either a dialog or a dictionary

exception `pywinauto.XMLHelpers.`**`XMLParsingError`**
Wrap parsing Exceptions

10.7.4 pywinauto.fuzzydict

Match items in a dictionary using fuzzy matching

Implemented for pywinauto.

This class uses difflib to match strings. This class uses a linear search to find the items as it HAS to iterate over every item in the dictionary (otherwise it would not be possible to know which is the 'best' match).

If the exact item is in the dictionary (no fuzzy matching needed - then it doesn't do the linear search and speed should be similar to standard Python dictionaries.

```
>>> fuzzywuzzy = FuzzyDict({"hello" : "World", "Hiya" : 2, "Here you are" : 3})
>>> fuzzywuzzy['Me again'] = [1,2,3]
>>>
>>> fuzzywuzzy['Hi']
2
>>>
>>>
>>> # next one doesn't match well enough - so a key error is raised
...
>>> fuzzywuzzy['There']
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "pywinauto
uzzydict.py", line 125, in __getitem__
    raise KeyError(
KeyError: "'There'. closest match: 'hello' with ratio 0.400"
>>>
>>> fuzzywuzzy['you are']
3
>>> fuzzywuzzy['again']
[1, 2, 3]
>>>
```

class `pywinauto.fuzzydict.`**`FuzzyDict`**(*items=None*, *cutoff=0.6*)
Provides a dictionary that performs fuzzy lookup

10.7.5 pywinauto.actionlogger

`pywinauto.actionlogger.`**`ActionLogger`**
alias of *`StandardLogger`*

class `pywinauto.actionlogger.`**`CustomLogger`**(*logFilePath=None*)

`log`(**args*)

`logSectionEnd`()

`logSectionStart`(*msg*)

class `pywinauto.actionlogger.`**`StandardLogger`**(*logFilePath=None*)

log(**args*)

logSectionEnd()

logSectionStart(*msg*)

pywinauto.actionlogger.**disable**()
Disable logging pywinauto actions

pywinauto.actionlogger.**enable**()
Enable logging pywinauto actions

pywinauto.actionlogger.**reset_level**()
Reset pywinauto logging level to default one (logging.NOTSET)

pywinauto.actionlogger.**set_level**(*level*)
Set pywinauto logging level for default logger. Use logging.WARNING (30) or higher to disable pywinauto logging.

10.7.6 pywinauto.sysinfo

pywinauto.sysinfo.**is_x64_OS**()

pywinauto.sysinfo.**is_x64_Python**()

pywinauto.sysinfo.**os_arch**()

pywinauto.sysinfo.**python_bitness**()

10.7.7 pywinauto.RemoteMemoryBlock

Module containing wrapper around VirtualAllocEx/VirtualFreeEx Win32 API functions to perform custom marshalling

exception pywinauto.RemoteMemoryBlock.**AccessDenied**
Raised when we cannot allocate memory in the control's process

class pywinauto.RemoteMemoryBlock.**RemoteMemoryBlock**(*ctrl*, *size=4096*)
Class that enables reading and writing memory in a different process

Address()
Return the address of the memory block

CheckGuardSignature()
read guard signature at the end of memory block

CleanUp()
Free Memory and the process handle

Read(*data*, *address=None*, *size=None*)
Read data from the memory block

Write(*data*, *address=None*, *size=None*)
Write data into the memory block

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search