SikuliX Documentation

Release 1.1.0-Beta1

Raimund Hocke aka RaiMan

Oct 26, 2017

Contents

Getting Help

Looking for specific information?

- Try the Table of Contents
- Look through the genindex
- Use the search

See other people's questions or ask a question yourself.

If you think you've found bugs, search or report bugs in our bug tracker.

This document is being maintained by Raimund Hocke aka RaiMan.

If you have any questions or ideas about this document, you are welcome to contact him directly via the above link and the email behind.

QuickStart information and other possibilities to get in contact: sikulix.com.

For questions regarding the functions and features of Sikuli itself, that are not answered sufficiently in this documentation, please use the Sikuli Questions and Answers Board.

For hints and links of how to get more information and help, please see the sidebar.

Documentation for previous versions

Might not be available any longer without notice or might not work properly. Feel free to post a bug in case.

The documentation for the versions up to SikuliX-1.0rc3 is still available here.

CHAPTER 1

How to use this document

SikuliX at the top supports **scripting via SikuliX IDE** (a basic script editor to load, edit, save and run scripts including the creation/organization of the needed images for your visual workflow).

Supported scripting languages:

- **Python** (language level 2.7) supported by the Jython interpreter.
- **Ruby** (language level 1.9/2.0) supported by the JRuby interpreter.
- **JavaScript** supported by the Java builtin scripting engine (Java 7: Rhino, Java 8: Nashorn).

If you are new to programming, you can still enjoy using SikuliX to automate simple repetitive tasks without learning one of the supported scripting languages using the SikuliX IDE.

> A good start might be to have a look at the tutorials.

If you plan to write more complex scripts, which might even be structured in classes and modules, you have to dive into the Python Language, the Ruby Language or JavaScript.

NOTE: Since **Jython and JRuby are based on Java**, the modules available for Python or Ruby might not be available in the Sikulix environment. So before trying to use any non-standard modules or extension packages, you have to check, wether they are supported in this SikuliX environment.

NOTE on Java usage The features in SikuliX at the bottom line are implemented with Java. So you might as well use SikuliX at this Java API level in your Java projects or other Java aware environments (*see how to*). Though this documentation is targeted at the scripting people it contains basic information about the Java level API as well at places, where there are major differences between the two API level.

> Additionally you might **look through the JavaDocs** (temporary location).

Each chapter in this documentaton briefly describes a class or a group of methods regarding their basic features. General usage information are provided and hints, that apply to all methods in that chapter. We recommend to read carefully before using the related features.

If you are totally new with Sikuli, it might be a good idea to just read through this documentation sequentially. An alternative way might be to jump to the chapters that you are interested in by scanning the table of contents. A way in the middle would be reading the core classes in this sequence: `Region`, then `Match`, and finally `Screen`.

After that, you can go to any places of interest using the table of contents or use the genindex to browse all classes, methods and functions in alphabetical order.

CHAPTER 2

SikuliX - how does it work and system specific information

SikuliX - the basics

SikuliX allows one to automate visual workflows

Something like that we do every day sitting in front of our PC:

- we want to achieve something
- we use an application for that (e.g. the browser to access web content)
- we click some buttons, links or other visuals
- we wait for the app to react and shows the expected result on the screen
- we fill in some text somewhere and press some functional keys like TAB or ENTER
- again we wait for some expected reaction or result
- we click ...
- we wait ...
- we type ...
- we wait ...
- we ...
- ...

This we call a **visual workflow**.

Some of these workflows everyone repeats more or less often and from time to time we ask ourselves: can that be automated?

There are many tools out there, that can be used for such a workflow automation and who ever made some experiments in this area will have made his experiences and might have favorite tools for automation.

There are basically 3 categories of tools:

- **Recorder** while you manually work along your workflow, a recorder tracks your mouse and keyboard actions. After stopping the recording, you might playback (autorun your worflow). The recordings can usually be edited and augmented with additional features.
- **GUI aware** the tool allows to programmatically operate on GUI elements like buttons. This is based on the knowledge of internal structures and names of the GUI elements and their features. Some of these tools also have a recording feature.
- **Visually** the tool "sees" images (usually retangular pixel areas) on the screen and allows one to act on these images using mouse and keyboard simulation. There might be some recorder feature as well with such a tool.

SikuliX belongs to the 3rd category and currently does not have a recorder feature. While you work through your workflow you capture some images, that you want to act on or that you expect to appear after an action. These images are used by click and type actions or are used to wait for the screen to be ready for the next action.

SikuliX is a **WYSIWYS-Tool**: What You See Is What You Script.

So again taking the above workflow, now using SikuliX commands:

openApp(someApp) # we use an application someApp
click(imageButton) # we click some button
wait(imageExpected1) # we wait for the app to react and show the expected result on the screen
type("some text"); type(Key.ENTER) # we fill in some text and press ENTER
wait(imageExpected2) # again we wait for some expected reaction or result
click(...) # we click ...
wait(...) # we wait ...
type(...) # we type ...
wait(...) # we wait ...
...

Using the SikuliX IDE, you can setup and maintain such visual workflows including capturing and organizing the needed images. Besides knowing the options of the basic commands, you do not need any programming or scripting knowledge. Only in the moment, you want to optimize, repeat or augment such a basic linear workflow, you have to dive deeper into the scripting language of your choice (supported in the IDE: Python, Ruby, JavaScript).

SikuliX can be used for visual testing

In software testing people use tools, to check, whether an application driven by some workflow reveals the expected results. With applications having a visually complex GUI, sooner or later the testers want to check some visual content against their expectation. This might be the presentation of GUI elements on the screen or the visual content of some part of the screen.

SikuliX can be integrated in various ways into such testing environments, either on the scripting level or using the Java based API (where the SikuliX features are implemented). Prominent examples for this approach are RobotFramework or Cucumber.

Also a combination of GUI aware tools and SikuliX are already reality (e.g. together with Selenium for web apps).

The challenge here usually is, that SikuliX's image based features are pixel-aware in the sense, that an image is on the screen exactly pixel-wise or SikuliX fails (with some slight tolerances as long as width and height in pixels match). This usually leads to the neeed to have different image sets for different environments. The version 2 will have some more features, that will make it easier to handle such situations.

Other aspects important for testers:

- SikuliX needs a real screen running the application under test or at least some equivalent virtual solution
- SikuliX is only available on PCs/Workstations running Windows, Mac or Linux and having a Java version 6+

SikuliX - how does it find images on the screen?

SikuliX uses the OpenCV package for finding an image on the screen.

The SikuliX feature is based on OpenCV's method matchTemplate(), which is rather good explained on this example page. If you are not familiar with how it works, you should just have a look there and then come back to here and read further.

A basic feature in Sikulix is to wait for an image to appear in a given region:

```
# some top left part of the screen
aRegion = Region(0, 0, 500, 500)
# a png image file on the file system
# this is the image we want to look for in the given Region
aImage = "someImage.png"
# search and get the result
aMatch = aRegion.find(aImage)
```

To not make it too complicated here, I do not talk about how you create the aImage - we just assume it is there and accessible.

The matchTemplate() expects an even sized or larger image (base), where the given image (target) should be searched. To prepare that, we internally make a screenshot (using Java Robot class) of the screen area defined by the given aRegion. This now is the base image and held in memory. The target image is also created as in memory image read form the image file. Both images then are converted to the needed OpenCV objects (CVMat).

Now we run the matchTemplate() function and get a matrix in the size of the base image, that contains for each pixel a similarity score for the target image compared pixel by pixel with it's top left corner at this pixel location. If this is not clear here now, go back to the above example and try to understand. The score values at each pixel location vary between 0.0 and 1.0: the lower the value, the lower the probability, that the area with it's top left corner at this pixel location contains the target image. Score values above 0.7 - 0.8 signal a high probability, that the image is here.

In the next step, we use another OpenCV method, to get the relevant max value (result score) from the mentioned result matrix, meaning that we are looking for a pixel, that most probable is the top left corner of the target image in the base image.

If nothing else is said, only a result score > 0.7 is taken as found. The other values will signal a FindFailed exception. Depending on various aspects of the target image (mainly how much even background towards the edges in contained in the target image), one usually get result scores > 0.8 or even 0.9. If one follows SikuliX's recommendation how to create target images, one should in most cases get result scores >0.95 or even >0.99 (internally taken as exact match with 1.0).

If the result score is accepted as found, in the next step we create a match object, that denotes the region on the screen, that most probably contains the image (aMatch in the above snippet).

If the image is not found (result score not acceptable), we either terminate the search operation signalling failed or start a new search with a new screenshot of the given region. This is repeated until the image is either found or the given or implicit waiting time (3 seconds in the standard) has elapsed, which also results in a FindFailed signal. The rate of this repetition can be specified, to reduce the cpu usage by SikuliX, since the search process is pure number crunching.

A word on elapsed time for search operations: The larger the base image the longer the search. The smaller the size difference of the 2 images, the faster. On modern systems with large monitors searching a small to medium sized image (up to 10.000 pixels), the elapsed time might be between 0.5 and 1 second or even more. The usual approach, to reduce search time is to reduce the search region as much as possible to the area, one expects the target image to appear. Small images of some 10 pixels in search regions of some 1000 pixels are found within some 10 milliseconds or even faster.

The actual version 1.1.0 of Sikulix implements a still-there-feature: before searching in the search region, it is first checked, whether the image is still in the same place as at the time of the last search (if the search region contains this last match). On success, this preflight operation usually takes some milliseconds, which speeds up workflows enormously if they contain repetitive tasks with the same images.

Not knowing the magic behind SikuliX's search feature and the matchTemplate() function, people always wonder, why images showing up multiple times on the screen, are not found in some regular order (e.g. top left to bottom right). That this is not the case is caused by the implementation of the matchTemplate() function as some statistical numeric matrix calculations. So never expect SikuliX to return the top left appearance of a visual being more than once on the screen at time of search. The result is not predictable in this sense.

If you want to find a specific item of these multiple occurrences, you have to restrict the search region, so that only the one you are looking for is found.

For cases where this is not suitable or if you want to cycle through all appearances, we have the findAll() method, that returns a list of matches in decreasing result score order. You might work through this list according to their position on the screen by using their (x,y) top left corner coordinates. findAll internally evaluates the search result matrix, by repetitively looking for the next max value after having "switched off" some area around the last max.

SikuliX - handling of images

To use images with the features of SikuliX like click(someImage), you need to store these images as image files preferably in the PNG format (someImage.png) somewhere on the file system or somewhere in the internet.

An image in this sense is some rectengular pixel area taken from the screen (captured or made a screenshot - with Sikuli we use the verb capture as the process of taking the image and save it in an image file and the name screenshot for the artifact "visual object = image").

Capturing is supported by the IDE or programmatically via the respective SikuliX features. You might use any capture tool instead to get your images (preferably in PNG format).

To load the images SikuliX has 2 principles:

- **bundle path**: the images are stored together with the script file (.py for Python, .rb for Ruby, .js for JavaScript) in a folder named someScript.sikuli, where the scriptfile must be named the same as the folder (e.g. someScript.py). This all is automatically assured, when working with the SikuliX IDE.
- **image path**: additionally SikuliX supports a list of places as an image path. Possible places are folders in the file system, folders in a jar-file and folders somewhere in the internet. There are functions available to manage your own image path. When an image has to be loaded (exception: the absolute path is given), the places are sequentially checked for the existence of the image. The first matching place wins.

It is strongly recommended, to have a naming scheme for the image files and to not rely on the basic timestamped image file naming of the SikuliX IDE, which is basically for new users with little programming experience.

Version 2 will have a capturing tool as a standalone app, that supports the basic aspects of image handling:

- capture and recapture screeshots along a workflow (some kind of recorder)
- organize your image path
- organize groups of "same" images, that can be switched depending on environment aspects
- organize a group of images, that somehow relate to each other and should be found together
- organize different states of an image (e.g. selected/not selected)
- optimize screenshots to get the highest possible scores at find
- some kind of basic support for transparency (e.g. ignore inner part of button)

SikuliX - system specifics

Some general aspects

A major aspect of SikuliX is to be available on Windows, Mac and Linux with as little differences as possible. This means, that features will only be added to SikuliX as standard, if they can be made available on all these systems. Nevertheless it will be possible beginning with version 2, to add extensions or plugins, that might not be available on all systems from the beginning or forever. Version 2 will have a suitable eco-system for that.

Mainly bacause of this major aspect SikuliX is a Java based application or library. Hence the usable artifacts are delivered as jar-files. Where possible the SikuliX IDE is delivered as an application (currently Mac only, Windows .exe planned for version 2).

To use the SikuliX features you need a valid Java runtime installation (JRE, preferably the Oracle versions) of at least version 6. SikuliX works with version 7 and 8 too and version 2 will need at least Java 7.

With version 1.1.x, there are still vital parts of SikuliX written in C++, which makes a SikuliX artifact system specific in the end. This currently is supported by an initial setup process, that produces the finally usable artifacts for this system environment.

The only exception is Java programming with some Maven compatible build system, that allows to simply start programming without having done a setup. The needed artifacts for this system are dynamically loaded according to the Maven dependency concept.

Beginning with version 1.1.0 the resulting artifacts (currently sikulix.jar and/or sikulixapi.jar) can be moved around as needed (though it is still recommended to have the SikuliX stuff in one well defined place, to avoid update/upgrade problems). Everything else SikuliX needs during runtime is stored either in the system's temp space or in a special system specific area in the user's home space (see the system specific topics below). Missing or outdated things in these areas are created/recreated at runtime by SikuliX automatically (means: you can delete everything at any time, as long as you keep the jars).

The current layout of this space is as follows (we call it **SikulixAppData**):

`Extensions` (place for extensions/Plugins)
`Lib` (the stuff to support Jython/JRuby usage)
`SikulixDownloads` (non SikuliX artefacts like Jython, JRuby, Tesseract support, ...)
`SikulixDownloads_TIMESTAMP` (versioned SikuliX stuff needed for setup)
`SikulixLibs_TIMESTAMP` (the place for the exported native libraries)
`SikulixSetup` (optional: used when the setup is run from the project context)
`SikulixStore` (place for persistent or optional information)
`SikulixTesseract` (place for language specific tessdata files)

Currently there is no need to step into these folders except for debugging purposes.

SikuliX in the standard does not need any environment settings anymore.

SikuliX on Windows

The IDE currently (1.1.0) still is only available as jar-file, that can be double-clicked to start it.

Setup installs a runsikulix.cmd, that can be used to start the IDE from commandline or to run scripts.

For version 2 an application is planned as .exe based on Launch4J.

The SikulixAppData is stored in the folder Sikulix inside the folder the environment variable %APPDATA% points to.

Besides Java there are no prerequisites. All native libraries for 32Bit and/or 64Bit are bundled in the jar-files and exported at runtime as needed. Be aware: the bitness of the native libraries depends on the bitness of the used Java version, which might as well be a 32Bit version on a 64Bit Windows (though not recommended).

SikuliX on Mac

The IDE is available as Sikulix.app after setup and should be moved to the /Applications folder.

Additionally there is a command script runsikulix to run scripts from commandline (Terminal).

The SikulixAppData folder is here ~/Library/Application Support/Sikulix

Besides Java there are no prerequisites. All native libraries (64Bit, since SikuliX needs OSX 10.6+) are bundled in the jar-files and exported at runtime as needed.

SikuliX on Linux

The IDE is only available as jar-file, that can be double-clicked to start it (setting the executable bit of the jar-file might be necessary).

Setup installs a runsikulix command script, that can be used to start the IDE from commandline or to run scripts.

As Java you might use the OpenJDK versions (though the Oracle versions are recommended). A JRE is sufficient, if you do not need a JDK for your own purposes.

Usually on Linux systems SikuliX does not run out of the box, since it is not possible to bundle all native libraries for all possible Linux flavors. It is intended, to have at least a version for the latest Ubuntu systems, that run out of the box.

If setup fails due to library problems, have a look at the setup log to get hints on how to proceed.

About how to get the native libraries working in case look here.

CHAPTER 3

New Features and major changes (version 1.1.0+)

CHAPTER 4

SikuliX scripting and usage in programming scenarios (preferably Java based)

SikuliX - general aspects of scripting

What is a SikuliX script

Where and how can a SikuliX script be stored

How to run a SikuliX script or a series of scripts

Using JavaScript

Using Python

Setup a Jython environment

This only applies to SikuliX 1.1.0+ with Jython 2.7.0+

Note for Mac OSX If you ever encounter an error like `ValueError: unknown locale: UTF-8`, then take care, that your environment at runtime of Jython contains these 2 entries:

- LC_ALL=en_US.UTF-8
- LANG=en_US.UTF-8

You might use `export` or any other appropriate method.

In cases you do not want to run scripts from inside the SikuliX IDE or from command line using the SikuliX command scripts or jar-files, you might setup your own Jython environment and run scripts.

Apply the following steps, to get a Jython environment, that is SikuliX aware:

- download the installer package from Jython Downloads
- install (usually by double-clicking the package) using the standard setup into an empty folder
- test by running `<jython-folder>/bin/jython` from a commandline, which should open an interactive Jython session, that allows, to run Python statements line by line
- make sure, that pip and easy_install are available:
- `<jython-folder>/bin/pip` exists
- `<jython-folder>/bin/easy_install` exists
- if this is not the case run `<jython-folder>/bin/jython -m ensurepip` on a commandline and check again
- if this is still not the case follow the steps further below `Fallback without pip`
- run `<jython-folder>/bin/pip install jip` to install the package `jip`, which allows to add Java libraries easily to your Jython environment
- add any needed Python package (must not depend on C-based stuff) using `pip`, `easy-install` or manual methods into `<jython-folder>/Lib/site-packages` and/or use `jip` for adding Java libraries preferably from Maven Central

Access Python packages from SikuliX scripts run by SikuliX (GUI or commandline)

The following approaches apply to situations, where you want to use Python modules installed somewhere on your system, without the need to manipulate `sys.path`, meaning, that when using `import moduleXYZ` this package is found automatically.

SikuliX uses a central repository (`SikulixRepo` in the following) for internal stuff (native libraries, downloaded artifacts, reso

- Windows: `%APPDATA%\Sikulix`
- Mac: `~/Library/Application Support/Sikulix`
- Linux: `~/.Sikulix`

Basic preparation To `SikulixRepo` add a folder `Lib` (if not already there) and inside add a folder `site-packages`

Approach 1 Since an existing folder `SikulixRepo/Lib/site-packages` will be recognized and added automatically as the `1st entry to sys.path`, modules/packages contained in here will be found when imported without any further preperations. This approach can be used, to "overwrite" modules/packages, that otherwise would be found elsewhere on `sys.path` (e.g. for testing)

Approach 2 In the folder `SikulixRepo/Lib/site-packages` have a file `sites.txt`, that contains absolute paths one per line, that point to other places, where modules packages can be found. These paths will be added automatically at startup to the `end of sys.path` in the given sequence. With this approach, you might for example add the `Lib/site-packages` folder of your own Jython installation.

Prepare and use your own jar files in the Jython environment

You might prepare jar files containing Python scripts/modules/packages, Java classes and other stuff like images, that are intended to be used in the scripting context.

possible use cases

- you want to pack scripted stuff together with other resources into a container ready to be used by yourself or others via import (which is not supported by the .skl packaging method).
- you want to secure your script code against modifications by others, that use your distributed jar.

Later (possibly only with version 2) there will be a feature available, to run such script containers directly from commandline (`java -jar mystuff.jar parameters`) or by double clicking.

typical jar file structure:

```
-- jar rootlevel
module1.py     # Python module
module2.py
- folder1      # Python package
  __init__.py
  stuff1.py
  stuff2.py
- images       # image folder
  img1.png
  img2.png
- org          # Java package
  - mystuff
    class1.class
    class1.class
```

how to pack such a jar

You might use the Java `jar utility` (contained in the JDK).

Or use the **SikuliX provided** feature `Sikulix.buildJarFromFolder(jarpath, folder)`, where jarpath is the absolute path to the jar (the parent folder must exist, the jar is overwritten), that should be created and folder is the absolute path to a folder, containing the stuff to be packed. The content of the folder is copied to the root of the created jar.

Just run `Sikulix.buildJarFromFolder(jarpath, folder)` in an empty tab in the IDE or in a script, that might do some pre- and/or postprocessing.

If the folder contains an `__init__.py` on the first level, the given folder is taken as a Python package and as such copied to the root level of the jar, to preserve the package context:

```
-- packagefolder
  __init__.py
  stuff.py

becomes a jar
-- jar rootlevel
- packagefolder
  __init__.py
  stuff.py
```

how to secure your script code using the jar packaging

- Step 1: prepare a folder as in the previous chapter
- Step 2: compile the folder into a new folder (see below)
- Step 3: pack the new folder into a jar for distribution

Run in an empty IDE tab or as part of a script:

```
Sikulix.compileJythonFolder(sourcefolder, targetfolder)
```

copies the complete content from sourcefolder to targetfolder (the parent folder must exist, the folder is emptied if exists) and then traverses the targetfolder replacing each `foobar.py` with it's compiled version `foobar$py.class`, that contains JVM-byte-code, so your script code cannot be edited anymore in this targetfolder, but still be used with `import foobar`.

Be aware: Be sure, your code compiles without errors, because the compile feature either succeeds or fails (compile errors), but you will not get any information about the cause or even the place of the compile problem.

Using Ruby

Using SikuliX in Java programming

Using SikuliX in non-Java programming scenarios

Using RobotFramework

New in version X1.1.1.

You can run ready Robot scripts out of the box in the Sikulix context (IDE or from commandline). The needed Python module robot (from robotframework 3.0) is bundled with the `sikulixapi.jar`. At runtime and already with setup, the module is exported to the folder `<SikulixAppData>/Lib`, which is on `sys.path` automatically. So there is no need to have anything else available than a suitable setup of SikuliX.

The easiest way is to use the SikuliX IDE with this principal setup

```
runScript("""
robot
*** Variables ***
${USERNAME}               demo
${PASSWORD}               mode
${TESTSITE}               http://test.sikuli.de
*** Settings ***
Library           ./inline/LoginLibrary
Test Setup        start firefox and goto testsite    ${TESTSITE}
Test Teardown     stop firefox
*** Test Cases ***
User can log in with correct user and password
    Attempt to Login with Credentials    ${USERNAME}    ${PASSWORD}
    Status Should Be    Accepted
User cannot log in with invalid user or bad password
    Attempt to Login with Credentials    betty    wrong
    Status Should Be    Denied
""")

class LoginLibrary(object):
  def start_firefox_and_goto_testsite(self, page):
    popup("start_firefox_and_goto_testsite")
  def stop_firefox(self):
    popup("stop_firefox")
  def attempt_to_login_with_credentials(self, username, password):
    popup("attempt_to_login_with_credentials")
  def status_should_be(self, expected):
    popup("status_should_be")
```

the first 2 lines

```
runScript("""
robot
```

signal, that you want to run an inline Robot script, that follows on the next lines terminated by `"""`). This construct is a multiline Python comment, that can be used as a string.

Normally when working with SikuliX features, you have to do some Robot Keyword implementation at the Python level. To Robot you tell where to find these implementation using the `Library setting`.

In this case we have the implementations inline in the same scriptfile according to the Robot rules packed into a Python class having the Keyword methods according to the Robot naming conventions. At runtime this class will be exported to a Python file, whose absolute path is then replacing the Library setting.

If you have the Keyword implementations somewhere outside, then you have to put the correct path specification into the Library setting. Another option is to reference a jar file as a Library again according to the Robot specifications.

If you now run the script in the IDE, internally a `robot.run` will be fired after having setup the script content and the environment. Currently no extra options can be provided for the robot run. As a result you get a folder with the ending `.robot` named as your script in the same folder as your script folder containing inputs to and the results from the robot run

```
# supposing the script is named testrobot.sikuli
# then you get a folder testrobot.sikuli.robot with the content
testrobot.robot # the robot script
LoginLibrary.py # the Python Keyword implementations
# the standard Robot outcome
output.xml
log.html
report.html
```

Still being in the IDE another possible setup would be this way:

```
robotScript = """
robot
*** Variables ***
${USERNAME}               demo
${PASSWORD}               mode
${TESTSITE}               http://test.sikuli.de
*** Settings ***
Library           /some/path/to/LoginLibrary.py
Test Setup        start firefox and goto testsite    ${TESTSITE}
Test Teardown     stop firefox
*** Test Cases ***
User can log in with correct user and password
    Attempt to Login with Credentials    ${USERNAME}    ${PASSWORD}
    Status Should Be    Accepted
User cannot log in with invalid user or bad password
    Attempt to Login with Credentials    betty    wrong
    Status Should Be    Denied
"""

# here you could do some preprocessing and even modify the above robotscript

runScript(robotscript)

# eventually do something with the result
```

BE AWARE for the keyword library, the `Name` in the file name `Name.py` and the statement `class Name()` **must** be the same and start with an uppercase letter.

BE AWARE ON WINDOWS the file path must be escaped with 4 backslashes for each backslash like so `C:\\\\Robot\\\\Libraries\\\\Name.py` (which leads to the needed 2 backslashes for each backslash as escape in the final robot file)

Of course you can use any other method, to fill a string representing a valid Robot script, provided the first line contains the string `robot` and only that (denoting the script type for runScript).

If in such a case you want to provide an inline Keyword implementation: this does the trick:

```
# prepare your script content
runScript("robot\n" + scriptContent)
# eventually do something with the result

# """)

# the rest is taken as inline Keyword implementation
```

If you have the need to specify extra parameters to the `robot.run()`, then you still have the option to stay within the SikuliX context (IDE or from commandline):

```
prepareRobot() # takes care for the correct environment

workdir = getParentFolder()
script = "arobottest/arobottest.robot"
robotscript = os.path.join(workdir, script)

print "*** trying to run:", robotscript
robot.run(robotscript, outputdir=workdir)
```

A library .py file being either in the script folder itself or in the folder containing the script folder is found automatically. So simply the library name is enough in this case. In all other cases you either have to specify the absolute path off the .py script (take care with windows - see above) or use `addImportPath()` to add the folder containing the library .py file to `sys.path`, in which case again only the name is sufficient in the Robot script.

It is strongly recommended, to always specify the *outputdir= parameter* since otherwise the reportfiles will be written to the working folder (from where you are running), which might not always be what you want.

If you want to use any of these variants outside the SikuliX context (some external Jython or in an IDE like PyCharm) you have to add these 2 lines at the beginning of your main script (as always in such cases):

```
import org.sikuli.script.SikulixForJython
from sikuli import *
```

to get the SikuliX context ready.

CHAPTER 5

SikuliX IDE create/run scripts and organize your images

Basic information and quick start

New in version 1.1.0.

Global options - setup and usage

SikuliX now has a basic implementation of a globel options repository, that is recognized and loaded at startup time (IDE and API usage). At runtime, this options store can be accessed to get and set values. Persistence (save) and merge of additional options files is not yet supported.

Since internally the options are stored in a Java Properties object as key-value-pairs in string representation, the options file must conform to the applicable rules (look here).

At startup these places are searched for a file `SikulixOptions.txt` (first appearence wins):

- the working folder
- the user's home folder
- the folder `SikulixStore` in the *Sikulix repository*

Special for the options defined in the Settings class

- see *Controlling Sikuli Scripts and their Behavior*
- any `Settings.option = value` will overwrite the existing option value in the Settings class at startup

Other Options globally recognized and processed on startup

- `Debug.level = n`

  preferably n as 3, debug information produced from beginning of startup

- `Settings.OverwriteImages = yes`

when saving images in the IDE, usually when using a name, that already exists, this is not overwritten, but stored with a name having a suffix -n appended. With this option switched on (default is off) naming images with an existing name will replace the image file without notice.

- `classpath = list of absolute-paths`

  will be added to the end of the classpath in the given sequence. Rules are the same as for Java classpath on that system. On Windows backslashes have to be doubled (escaped).

CHAPTER 6

SikuliX API for scripting and Java programming

General features regarding scripting and image handling

Controlling Sikuli Scripts and their Behavior

setShowActions (*False* | *True*)
If set to *True*, when a script is run, Sikuli shows a visual effect (a blinking double lined red circle) on the spot where the action will take place before executing actions (e.g. `click()`, `dragDrop()`, `type()`, etc) for about 2 seconds in the standard (see *`Settings.SlowMotionDelay`*). The default setting is False.

exit ([*value*])
Stops the script gracefully at this point. The value is returned to the calling environment.

class Settings

`Settings.`**ActionLogs**
`Settings.`**InfoLogs**
`Settings.`**DebugLogs**
see the section about *debug and log messages*.

`Settings.`**MinSimilarity**
The default minimum similiarty of find operations. While using a *`Region.find()`* operation, if only an image file is provided, Sikuli searches the region using a default minimum similarity of 0.7.

`Settings.`**MoveMouseDelay**
Control the time taken for mouse movement to a target location by setting this value to a decimal value (default 0.5). The unit is seconds. Setting it to 0 will switch off any animation (the mouse will "jump" to the target location).

As a standard behavior the time to move the mouse pointer from the current location to the target location given by mouse actions is 0.5 seconds. During this time, the mouse pointer is moved continuosly with decreasing speed to the target point. An additional benefit of this behavior is, that it gives the active application some time to react on the previous mouse action, since the e.g. click is simulated at the end of the mouse movement:

```
mmd = Settings.MoveMouseDelay # save default/actual value
click(image1) # implicitly wait 0.5 seconds before click
Settings.MoveMouseDelay = 3
click(image2) # give app 3 seconds time before clicking again
Settings.MoveMouseDelay = mmd # reset to original value
```

Settings.**DelayBeforeMouseDown**
Settings.**DelayBeforeDrag**
Settings.**DelayBeforeDrop**

DelayBeforeMouseDown specifies the waiting time before mouse down at the source location as a decimal value (seconds).

DelayBeforeDrag specifies the waiting time after mouse down at the source location as a decimal value (seconds).

DelayBeforeDrop specifies the waiting time before mouse up at the target location as a decimal value (seconds).

Usage: When using *Region.dragDrop()*, *Region.drag()* and *Region.dropAt()* you may have situations, where the operation is not processed as expected. This may be due to the fact, that the Sikuli actions are too fast for the target application to react properly. With these settings the waiting time befor and after the mouse down at the source location and before the mouse up at the target location of a dragDrop operation are controlled. The standard settings are 0.3 seconds for each value. The time that is taken, to move the mouse from source to target is controlled by *Settings.MoveMouseDelay*::.

Be aware The given values are only valid for the next following action. The inner timing will be reset to the defaults after the action's completion.

> Settings.DelayBeforeMouseDown = 0.5 Settings.DelayBeforeDrag = 0.2 Settings.DelayBeforeDrop = 0.2 Settings.MoveMouseDelay = 3 dragDrop(source_image, target_image) # time for complete dragDrop: about 4 seconds + search times

Settings.**ClickDelay**

Specify a delay between the mouse down and up in seconds as 0.nnn. This only applies to the next click action and is then reset to 0 again. A value > 1 is cut to 1.0 (max delay of 1 second)

Settings.**TypeDelay**

Specify a delay between the key presses in seconds as 0.nnn. This only applies to the next type action and is then reset to 0 again. A value > 1 is cut to 1.0 (max delay of 1 second)

NOTE: If the internal timing of the compound mouse functions like `click()` or `dragDrop()` is not suitable in your special situation, you might as well build your own functions using the basic mouse functions *Region.mouseDown()*, *Region.mouseMove()* and *Region.mouseUp()*

Settings.**SlowMotionDelay**

Control the duration of the visual effect (seconds).

Settings.**WaitScanRate**
Settings.**ObserveScanRate**

Specify the number of times actual search operations are performed per second while waiting for a pattern to appear or vanish.

As a standard behavior Sikuli internally processes about 3 search operations per second, when processing a *Region.wait()*, *Region.exists()*, *Region.waitVanish()*, *Region.observe()*). In cases where this leads to an excessive usage of system ressources or if you intentionally want to look for the visual object not so often, you may set the respective values to what you need. Since the value is used as a rate per second, specifying values between 1 and near zero, leads to scans every x seconds (e.g. specifying 0.5 will lead to scans every 2 seconds):

```
def myHandler(e):
        print "it happened"
# you may wish to save the actual settings before
Settings.ObserveScanRate = 0.2
onAppear(some_image, myHandler)
observe(FOREVER, background = True)
# the observer will look every 5 seconds ;-)
```

`Settings.`**`ObserveMinChangedPixels`**
The minimum area size in pixels that changes it's content to trigger a change event when using *`Region.onChange()`* when no value is specified. The default value is 50 (a rectangle of about 7x7 Pixels).

New in version 1.1.0.

Writing and redirecting log and debug messages

these are the relevant Settings for user logging showing defaults: (False = switched off, True = switched on)

- `Settings.UserLogs = True` (False: user log calls are ignored)
- `Settings.UserLogPrefix = "user"` (message prefix)
- `Settings.UserLogTime = True`
- `Debug.setUserLogFile("absolute-path-to-file")` (no default)

to write a user log message:

`Debug.user("text with %placeholders", args ...)` where text is a string according to the rules of Java String.format().

Information about Java String formatting **can be found here** (rather formal, look for tutorials in the net additionally if this is new for you)

the messages look so: `[prefix optional-timestamp] message-text with filled in arg values`

Being in Jython scripting one might as well use this: `Debug.user("some text with %placeholders" % (list-of-args ...))`

the settings for Sikuli's logging with the defaults: (False = switched off (message type not created), True = switched on)

- `Settings.ActionLogs = True` (message prefix: [log])
- `Settings.InfoLogs = True` (message prefix: [info])
- `Settings.DebugLogs = False` (message prefix: [debug])
- `Settings.LogTime = False`
- `Debug.setLogFile("absolute-path-to-file")` to redirect the Sikuli messages to a file, no default

Debug messages Sikuli internally issues debug messages all over the place, to show, what it is doing. Creating debug messages is dependant on the current DEBUG_LEVEL value:

- if 0, no debug messages are shown
- if >0, debug messages having a level <= DEBUG_LEVEL are created

The initial DEBUG_LEVEL is 0 and can be set with

- the Java command line parameter `-Dsikuli.Debug=n` or

- the command line parameter `-d n` when using SikuliX jars or command scripts.

Currently a suitable DEBUG_LEVEL is 3, that shows enough valuable information about what is going on internally.

If you ever encounter problems, that might have to do with SikuliX's internal processing, switch on debug messaging with level 3.

To avoid tons of not needed messages, you might switch debugging on and off on the fly for only critical sections in your workflow:

- switch on: `Debug.on(n)` setting the DEBUG_LEVEL=n (recommended: 3)
- switch off: `Debug.off()`

Debug messages look so: `[DEBUG optional-timestamp] message-text with filled in arg values`

and can be produced with

`Debug.log(level, "text with %placeholders", args ...)` *Recommendation*: use 1 as level, since this is not used internally by SikuliX and allows you to switch your private debug messaging on `Debug.on(1)` and off.

Logging Callback Currently only for Jython scripting, there is a **logging callback** feature, that redirects the log messages to a given function in your script, where you can finally process the message for example with your own looging concept.

A message, that is redirected to a callback is ignored by the SikuliX log processing. *TAKE CARE:* you should avoid lengthy processing in the callback, since your workflow will wait for the callback to return

This is a basic usage example, where the callback function gets all messages:

```
# a wrapper class is needed for the callback function (name it as you want)
class myLogger():
  # a callback function (name it as you want)
  # you might have more than one for specific handling of message groups
  def callback(self, message):
      print message

# prepare log redirect
Debug.setLogger(myLogger()) # sets the object containing the callback functions

# redirect all logging messages
Debug.setLoggerAll("callback") # the name of the callback function as string
# from now on myLogger.callback will receive the messages
```

Selective log message processing (callback is the name of your specific callback function):

- `Debug.setLoggerUser("callback") # redirect messages [user]`
- `Debug.setLoggerInfo("callback") # redirect messages [info]`
- `Debug.setLoggerAction("callback") # redirect messages [log]`
- `Debug.setLoggerError("callback") # redirect messages [error]`
- `Debug.setLoggerDebug("callback") # redirect messages [debug]`

You might suppress the creation of the message header for all messages, so you only get the message body: use `Debug.setLoggerNoPrefix(myLogger())` instead of the initial `Debug.setLogger(myLogger())`

New in version 1.1.0.

File and Path handling - convenience functions

available for Jython scripting only in the moment

In more complex scripting situations it is often necessary to deal with paths to files and folders. To make this a bit more convenient, the following functions are available (look here for the underlying Python features).

getBundlePath()

returns the path to the current .sikuli folder without trailing separator. (see also *SIKULI_IMAGE_PATH*)

getBundleFolder()
same as *getBundlePath()* but with trailing separator to make it suitable for string concatenation.

getParentPath()
returns the path to the parent folder of the current .sikuli folder without trailing separator.

getParentFolder()
same as *getParentPath()* but with trailing separator to make it suitable for string concatenation.

makePath(*path1*, *path2*, *path3*, ...)
returns a path with the correct path seperators for the system running on by concatenating the given path elements from left to right (given as strings). There is no trailing path seperator.

makeFolder(*path1*, *path2*, *path3*, ...)
same as *makePath()* but trailing path seperator to make it suitable for string concatenation.

NOTE makePath and makeFolder on Windows the first path element can be specified as a drive letter "X:"

unzip(*fromFile*, *toFolder*)
A convenience function to unzip a zipped container to a folder (implemented using the Java builtin support for zip files). The ending of the file does not matter, the content of the file is examined to find out, whether it is a valid zip container. A zipped folder structure is preserved in the target folder. Relative paths would be resolved against the current working folder. This can for example be used, to unpack jar files.

Parameters

- **fromFile** – a file with a zipped content given as path string
- **toFolder** – the folder where to place the unzipped content given as path string

Returns True if it worked, False otherwise

NOTE The complementary feature `zip(fromFolder, toFile)` will follow soon.

NOTE on Java usage:

```
import org.sikuli.basics.FileManager;
FileManager.unzip(fromFile, toFolder);
```

New in version 1.1.0.

Image Search Path - where SikuliX looks for image files

SikuliX maintains a list of locations to search for images when they are not found in the current .sikuli folder (a.k.a. BundlePath). This list is maintained internally but can be inspected and/or modified using the following functions.

GENERAL NOTEs:

- as long as an image file has the ending .png, this might be omitted.
- you might use subfolders as well, to form a relative path to an image file

- an image path might point to a location inside a jar file or a location on the Java classpath
- an image path might point to a folder in the net, that is accessible via HTTP
- SikuliX internally manages a cache for the imagefile content (standard 64 MB), where images are held in memory, thus avoiding a reload on subsequent references to the same image file.

The bundle path can be accessed and modified so:

NOTE: the bundle path can only be on the (local) file system, not in a jar, nor in the net (access via HTTP). If you need places in a jar or in the HTTP net, use the add function.

setBundlePath (*path-to-a-folder*)
Set the base path for searching images. Sikuli IDE sets this automatically to the path of the folder of the script (.sikuli). Therefore, you should use this function only if you really know what you are doing. Using it generally means that you would like to take care of your captured images by yourself.

Additionally images are searched for in the image path, that is a global list of other places to look for images and the bundle path being the first entry. It is implicitly extended by script folders, that are imported (see: *Reuse of Code and Images*).

Currently (will be revised in version 1.2), you should not use a jar file folder, Use *addImagePath()* instead.

getBundlePath ()
Get a string containing the absolute path to a folder containing your images used for finding images and which is set by SikuliX IDE automatically to the script folder (.sikuli). You may use this function for example, to package your private files together with the script or to access the image files in the bundle for other purposes. Be aware of the *convenience functions to manipulate paths*.

NOTE for Java usage: Since there is no default BundlePath, when not running a script, like in the situation, when using the Java API in Java program or other situations with the direct use of Java aware scripting languages, you can use this feature to set the one place, where you have all your images:

```
import org.sikuli.script.ImagePath;
ImagePath.setBundlePath("path to your image folder");
screen.find("image1");
screen.find("imageset1/image2");
```

NOTE: first find omits .png, second find uses a relative path with a subfolder

Other places, where Sikuli looks for images, are stored internally in the image path list.

When searching images, the path's are scanned in the order of the list. The first image file with a matching image file name is used.

Use the following functions to manipulate this list.

NOTE for Java usage: Class of the mentioned functions:

```
import org.sikuli.script.ImagePath
```

getImagePath ()
Get a list of paths where Sikuli will search for images.

```
imgPath = getImagePath() # get the list
# to loop through
for p in imgPath:
        print p
```

Note on Java usage:

```
String[] paths = ImagePath.getImagePath();
for (String path : paths) {
    System.out.println(path)
}
```

addImagePath (*a-new-path*)

Add a new folder path to the end of the current list (avoids double entries) Java API: `ImagePath.add(path)`

As a convenience you might use this function also to add a path to a HTTP net folder like so *sikulix.com:* or *sikulix.com:somefolder/images* (see *addHTTPImagePath*)

addHTTPImagePath (*a-new-path*)

Add a new folder path to the end of the current list (avoids double entries) Java API: `ImagePath.addHTTP(a-new-path)`

a-new-path is a net url like *sikulix.com* optionally with a folder structure attached like so: *sikulix.com/images* (a leading *http://* or *https://* is optional, so one might copy and paste links)

The folder must be accessible via HTTP and must allow HTTP-HEAD requests on the contained image files (this is checked at time of trying to add the path entry).

NOTE on Java usage: images in a jar

It is possible to access images, that are stored inside of jar files. So you might develop a Java app, that comes bundled with the needed images in one jar file.

To support the development cycle in IDE's, you might specify an alternate path, where the images can be found, when running inside the IDE.

Usage in Maven Projects:

Following the conventions of Maven projects you should store your images in a subfolder at `src/main/resources` for example `src/main/resources/images`, which then at jar production will be copied to the root level of the jar. Not following this suggestion you have to work according to the case *other projects*.

`ImagePath.add("someClass/images")` where someClass is the name of a class contained in a jar or folder on the class path containing the images folder.

Usage in other Projects:

`ImagePath.add("someClass/images", alternatePath)` where *someClass* is the name of a class contained in a jar on the class path containing the images folder at the root level of the jar.

where *alternatePath* is a valid path specification, where the images are located, when running from inside an IDE.

Example of a non-Maven project where the images folder `/imgs` in this case is on the same level as the package folder `testAPI` containing the class file `Test.java` so both folders will be side by side at the root level of the runnable jar produced from this project:

```
package testAPI;

import org.sikuli.basics.Debug;
import org.sikuli.script.ImagePath;
import org.sikuli.script.Match;
import org.sikuli.script.Screen;

public class Test {
```

```
        public static void main(String[] args) {
                Screen s = new Screen();
                Debug.info("Screen: %s", s);
                String clazz = "testAPI.Test";
                String imgFolder = "/imgs";
                String img = "test.png";
                String inJarFolder = clazz + imgFolder;
                if (ImagePath.add(inJarFolder)) {
                        Debug.info("Image Folder in jar at: %s", inJarFolder);
                } else {
                        Debug.error("Image Folder in jar not possible: %s",␣
↪inJarFolder);
                }
                Match target = s.exists(img);
                if (null == target) {
                        Debug.error("Not found: ", img);
                } else {
                        Debug.info("Found: %s at %s", img, target);
                        s.hover();
                }
                Debug.info("... leaving");
        }
}
```

Be aware: that you might use the Sikuli IDE, to maintain a script, that only contains the image filenames and then is used as image path in your Java app like `ImagePath.add("myClass/myImages.sikuli")`, which e.g. in the Maven context will assume as image path `src/main/ressources/myImages.sikuli`.

Note for Jython scripting: use *`load()`* without the import to use the feature *images in jars*:

```
from org.sikuli.script import ImagePath
load("absolute path to someJar")
ImagePath.add("someClass/someFolder")
```

`removeImagePath` (*a-path-already-in-the-list*)

Remove the given path from the current list Java API: `ImagePath.remove(path)`

`resetImagePath` (*a-path*)

Clears the current list and sets the first entry to the given path (hence gets the BundlePath). This gets you a fresh image er
Java API: `ImagePath.reset(path)`

Note: paths must be specified using the correct path separators (slash on Mac and Unix and double blackslashes on Windows). The convenience functions in *File and Path handling* might be helpful.

This list is automatically extended by Sikuli with script folders, that are imported (see: *Importing other Sikuli Scripts*), so their contained images can be accessed by only using their plain filenames. If you want to be sure of the results of your manipulations, you can use *`getImagePath()`* and check the content of the returned list.

NOTE: at all time the first entry in the list is internally taken as BundlePath, where appropriate.

Importing other Sikuli Scripts (reuse code and images)

This is possible with SikuliX:

- import other .sikuli in a way that is compatible with Python module import (no module structures)

- **import a python module structure including underlying Java classes from a jar-file,** that is dynamically loaded using the function *load(jar-file)*
- automatically access images contained in the imported .sikuli (no need to use *setBundlePath()*)

Note: .skl cannot be imported. But you might unzip the .skl to a .sikuli, which then can be imported.

The prerequisites:

- the folders containing your .sikuli's you want to import have to be in `sys.path` (see below: Usage)
- Sikuli automatically finds other Sikuli scripts in the same directory, when they are imported
- your imported script **MUST** contain (recommendation: as first line) the following statement:

  `from sikuli import *`
  This is necessary for the Python environment to know the Sikuli classes, methods, functions and global names

Usage:

- **Add the path to the Sikuli module into `sys.path`** *not needed* for modules being in the same directory as the main script

Convenience function to add a path to sys.path:

New in version 1.1.0.

`addImportPath`(*path*)

- **Import your .sikuli using just its name.** For example, to import myModule.sikuli, just write `import myModule`.

A basic example:

```
# the path containing your stuff - choose your own naming
# on Windows
myScriptPath = "c:\\someDirectory\\myLibrary"
# on Mac/Linux
myScriptPath = "/someDirectory/myLibrary"

# all systems (avoids double entries in sys.path)
addImportPath(myScriptPath)

# supposing there is a myLib.sikuli
import myLib

# supposing myLib.sikuli contains a function "def myFunction():"
myLib.myFunction() # makes the call
```

Note on contained images: Together with the import, Sikuli internally uses the feature *SIKULI_IMAGE_PATH* to make sure that images contained in imported .sikuli's are found automatically.

Some comments on general rules for Python import

- An import is only processed once (the first time it is found in the program flow). So be aware:
  - If your imported script contains code outside of any function definitions (`def()`), this code is only processed once at the first time, when the import is evaluated
  - Since the IDE does not reload the modules when running a script the next time, you have to use the Jython's reload() function, if you are changing imported scripts while they are in use:

```
# instead of: import module
import module
reload(module)

# instead of: from module import *
import module
reload(module)
from module import *
```

- Python has a so called namespace concept: names (variables, functions, classes) are only known in it's namespace:
  - your main script has it's own namespace
  - Each imported script has its own namespace. So names contained in an imported script have to be qualified with the module name (e.g. `myLib.myFunction()`)
  - You may use `from myLib import *`, which adds all names from myLib into your current namespace. So you can use `myFunction()` directly. When you decide to use this version, be sure you have a naming convention that prevents naming conflicts.

New in version 1.1.1.

The imports for other .sikuli scripts are now tracked during one IDE session. On rerun of a main script, the respective imports are automatically reloaded, so an extra reload() in these cases is no longer needed.

New in version 1.1.0.

Loading a jar-file containing Java/Python modules and additional resources as needed

load(*jar-file*)
Loads a jar-file and puts the absolute path to it into sys.path, so the Java or Python code in that jar-file can be imported afterwards.

Parameters **jar-file** – either a relative or absolute path to `filename.jar`

Returns `True` if the file was found, otherwise `False`

load(*jar-file*, *image-folder*)
same as load(jar-file), but additionally adds the given folder to the image path. image-folder is assumed to be a foldername available in the jar's rootlevel (not checked though).

Parameters

- **jar-file** – either a relative or absolute path to `filename.jar`
- **image-folder** – a relative path (always use / as path separator, no leading /)

Returns `True` if the file was found, otherwise `False`

Search startegy The given jar is searched as following (first match wins):

- if given as absolute path it is checked for existence and processed (if not exists: no further action)
- if given as relative path:
- the current path (Jython: sys.path, Java: classpath)
- the current folder (Jython only: bundle path)
- the SikuliX Extensions folder
- the SikuliX Lib folder

Note for Java usage at the Java level, this feature is available as `Sikulix.load(jar [, folder])` and adds the given jar to the end of the classpath on the fly. A given folder is added to the image path as mentioned above.

Note on Python usage more details and usage cases are discussed in *Using Python*. After a successful `load()`, you might use the standard `import something`, to make the module *something* available in your scripting context.

New in version 1.1.0.

Running scripts and snippets from within other scripts and run scripts one after the other

What is meant by script and snippet?

- **Script** means, that some code is stored somewhere in a file accessible in this context by giving it's relative or absolute filename or URL.
- **Snippet** means some text stored in a string variable, that represents one or more lines of code in a denoted scripting language, for which an interpreter is available on the running system.

You may call/run **scripts** from a script that is currently running, which saves the startup time for the called script and keeps available the original parameters given and the current image path.

`runScript`(*script_path*, **parameter*)

Runs the script found at the given script-path handing over the given parameters in `sys.argv[1+]`. The called script has it's own bundle path, but the current image path. On exit the bundle path of the calling script is restored.

Param script_path: a path to a script folder (rules see below)

Param parameter: one or more parameters seperated by comma

Returns the return code that the called script has given with exit(n)

Rules for the given script_path

- absolut path to a folder in the file system
- relative path to a folder taken as relative to the working folder
- the path spec can contain leading or intermediate ../
- a path preceded by ./ means the same folder, that the calling script is located
- a pointer to a folder in the HTTP net
- in any case .sikuli can be omitted
- if it is a .skl, then it must be noted as script.skl

Special usage notes for scripts located in the net

- must be accessible via HTTP
- the location specifier can be one of these:
- `base-url:folder/script`
- `http://base-url:folder/script`
- `http://base-url/folder/script`
- where folder is optional and might have more than one level with / as separator
- where script is the folder containing the script file (Python, Ruby or JavaScript) and the images (no .sikuli appended!)

- The contained script file must have the same name as the script folder and a suffix:
- for JavaScript `.js`
- for Python `.py.txt`
- for Ruby `.rb.txt`
- the additional suffixes `.txt` are currently necessary, to avoid download problems (will be addressed in version 2)

This feature allows to create a main script, that contains a row of `runScript() commands`, thus running these scripts one after the other in the same context (no startup delay). Using the return codes and the parameters allows to create medium complex workflows based on smaller reuseable entities.

Another option to run a series of scripts without the startup delay for the second script and following is to run from commandline using option -r (see Running from command line)

You may run **snippets** by simply issuing

runScript (*snippet*)
currently available:

•AppleScript on Mac (script type word: applescript)

•PowerShell on Windows (script type word: powershell)

For version 2 there will be a plugin system to easily add other scripting engines.

Param snippet: a string containing the scripting statements after the word identifying the script type

Returns the return code that was returned by the interpreter running this snippet

Example for Applescript:

```
returnCode = runScript('applescript tell application "Mail" to
activate')
```

or like this for a multiline snippet:

```
cmd = """
applescript
tell application "Mail" to activate
display alert "Mail should be visible now"
"""
returnCode = runScript(cmd)
```

Example for PowerShell:

```
returnCode = runScript('powershell get-process')
```

or like this for a multiline snippet:

```
cmd = """
powershell
get-process
"""
returnCode = runScript(cmd)
```

If the snippet produces some output on stdout and/or stderror, this is accessible after return using:

```
commandOutput = RunTime.get().getLastCommandResult()
```

where the error output comes after a line containing `***** error *****`

Interacting with the User and other Applications

PopUps and input dialogs

In the standard the following dialog boxes are shown in the middle of the screen, where SikuliX (IDE or from commandline) is running (usually the primary screen).

New in version 1.1.1.

As a backport from SikuliX version 2 I added the timed/autoclosing versions of `popup`, `popAsk`, `popError` and `input`. For more information *look here*.

New in version 1.1.1.

They are always on top from the beginning, no matter which application currently is the frontmost. While the dialog is visible, you might move it around and act on other applications, until you work with the dialog box.

New in version 1.1.1.

If you want the dialog to appear in a special location on the screen (even on other screens in multimonitor situations), you can use the function *`popat()`* to define this location. The dialog will be positioned here with the center of its dialog panel. Be aware, that locations near the edge of the screen might make parts of the dialog not accessible (this is not checked). This location will stay in effect until changed by another use of *`popat()`*. A *`popat()`* without parameters will reset it to the standard (center of primary screen).

Note for Java Usage These methods are available in class `org.sikuli.script.Sikuli`

popat (*x*, *y*)

popat (*location*)

popat (*region*)

popat ()

Define the location, where the center of popup dialogs should be positioned from now on.

Parameters

- **x** – x value of the location
- **y** – y value of the location
- **location** – the location as a *`Location`*
- **region** – the location as the center of the given *`Region`*

If no parameter is given, the location will be reset to the center primary screen (default).

popup (*text*[, *title*])

Display a dialog box with an *OK* button and *text* as the message. The script then waits for the user to click the *OK* button.

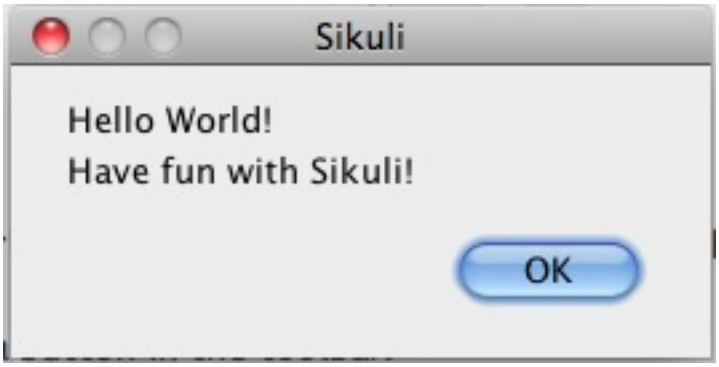
Parameters

- **text** – text to be displayed as message
- **title** – optional title for the messagebox (default: Sikuli Info)

Example:

```
popup("Hello World!\nHave fun with Sikuli!")
```

A dialog box that looks like below will popup *Note:* \n inserts a line break



New in version 1.1.0.

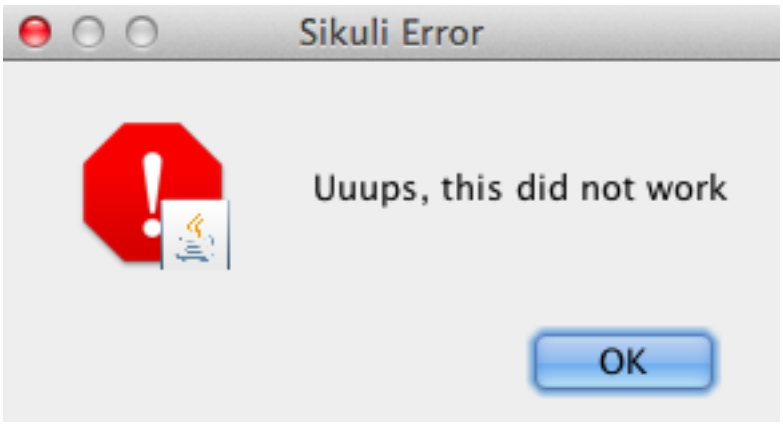
popError (*text*[, *title*])

Same as *popup()* but with a different title (default Sikuli Error) and alert icon.

Example:

```
popError("Uuups, this did not work")
```

A dialog box that looks like below will popup



New in version 1.1.0.

popAsk (*text*[, *title*])

Returns `True` if user clicked `Yes`, `False` otherwise

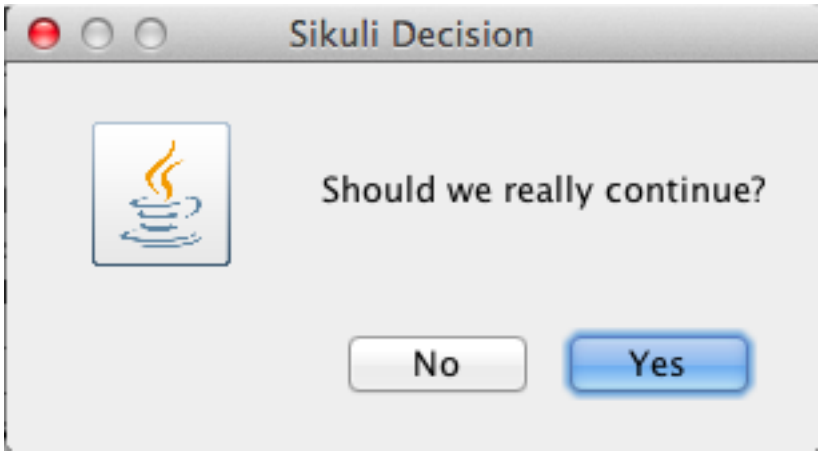
Same as *popup()* but with a different title (default Sikuli Decision) and alert icon.

There are 2 buttons: `Yes` and `No` and hence the message text should be written as an appropriate question.

Example:

```
answer = popAsk("Should we really continue?")
if not answer:
    exit(1)
```

A dialog box that looks like below will popup



New in version 1.1.0.

input ([*msg*][, *default*][, *title*][, *hidden*])

Display a dialog box with an input field, a Cancel button, and an OK button. The optional text *title* is displayed as the messagebox title and the text *msg* as some explanation near the input field. The script then waits for the user to click either the Cancel or the OK button.

Parameters

- **msg** – text to be displayed as message (default: nothing)
- **default** – optional preset text for the input field
- **title** – optional title for the messagebox (default: Sikuli Input)
- **hidden** – (default: False) if true the entered characters are shown as asterisks

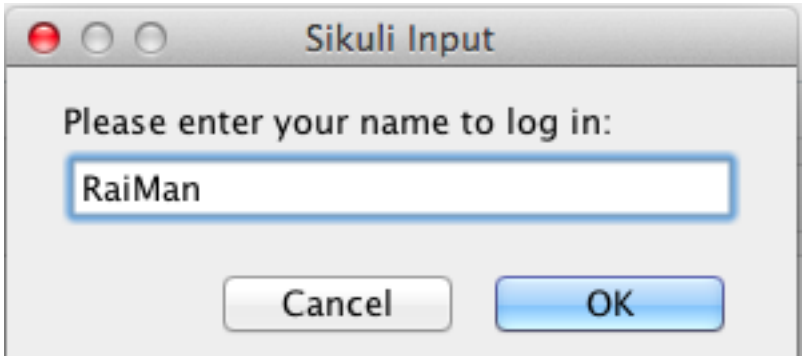
Returns

the text, contained in the input field, when the user clicked `Ok`

None, if the user pressed the `Cancel` button or closed the dialog

Example: plain input:

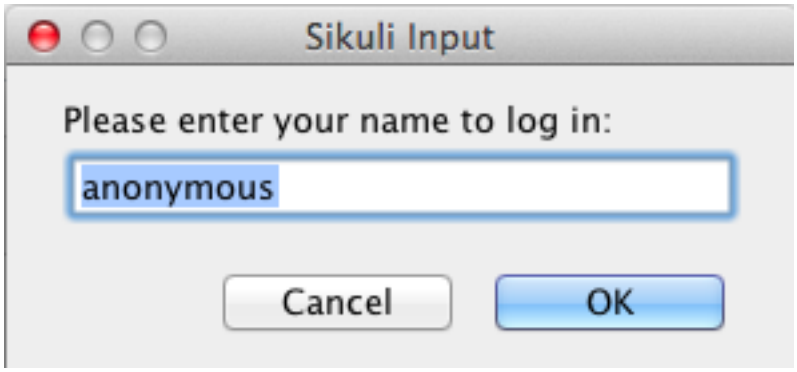
```
name = input("Please enter your name to log in:")
```



A dialog box that looks like above will appear to allow the user to interactively enter some text. This text is then assigned to the variable *name*, which can be used in other parts of the script, such as `paste(name)` to paste the text to a login box.

Example: input with preset:

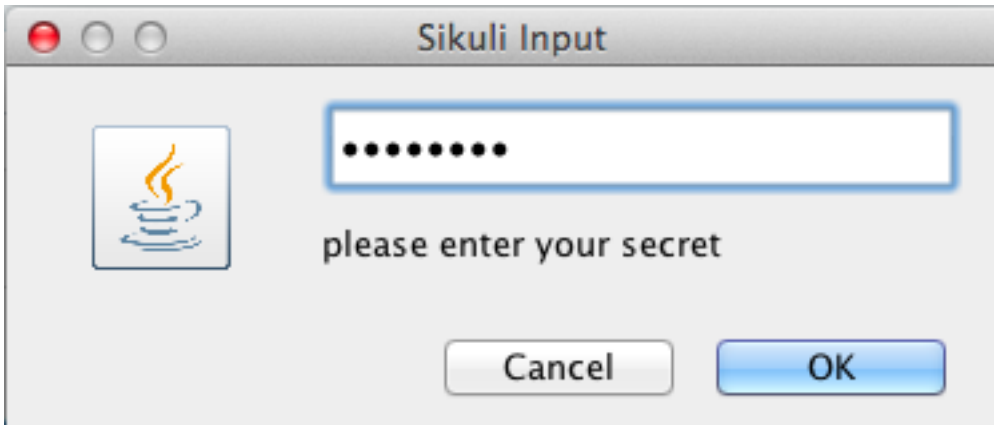
```
name = input("Please enter your name to log in:", "anonymous") # a preset input
↪text
```



When using the parameter `default`, the text input field will be pre-populated with the given text, so the user might just click OK/Cancel or edit the content of the input field.

Example: input with hidden input:

```
password = input("please enter your secret", hidden = True)
```



As the user inputs his secret infoemation, the text is shown as one asterisk per character.

New in version 1.1.0.

inputText (*message*[, *title=""*][, *lines=9*][, *width=20*][, *text=""*])

Parameters

- **message** – text to be displayed as message
- **title** – optional title for the messagebox (default: SikuliX input request)
- **lines** – how many lines the text box should be high (default: 9)
- **width** – how many characters the box should have as width (default: 20)
- **text** – a multiline text, that is preset in the textarea

Returns the multiline text content when user presses `OK` (might be empty) or None if the user presses `CANCEL`

A message box with the given height and width is displayed and allows the user to input as many lines of text as needed. The lines are auto-wrapped at word boundary. A vertical scrollbar is shown if needed.

The default font is the Java AWT Dialog (a sans-serif font) in size 14, which is also the minimum size possible. One might switch to a monospace font using `Settings.InputFontMono=True`. Setting it to `False` switches it back to the standard for the next `inputText()`.

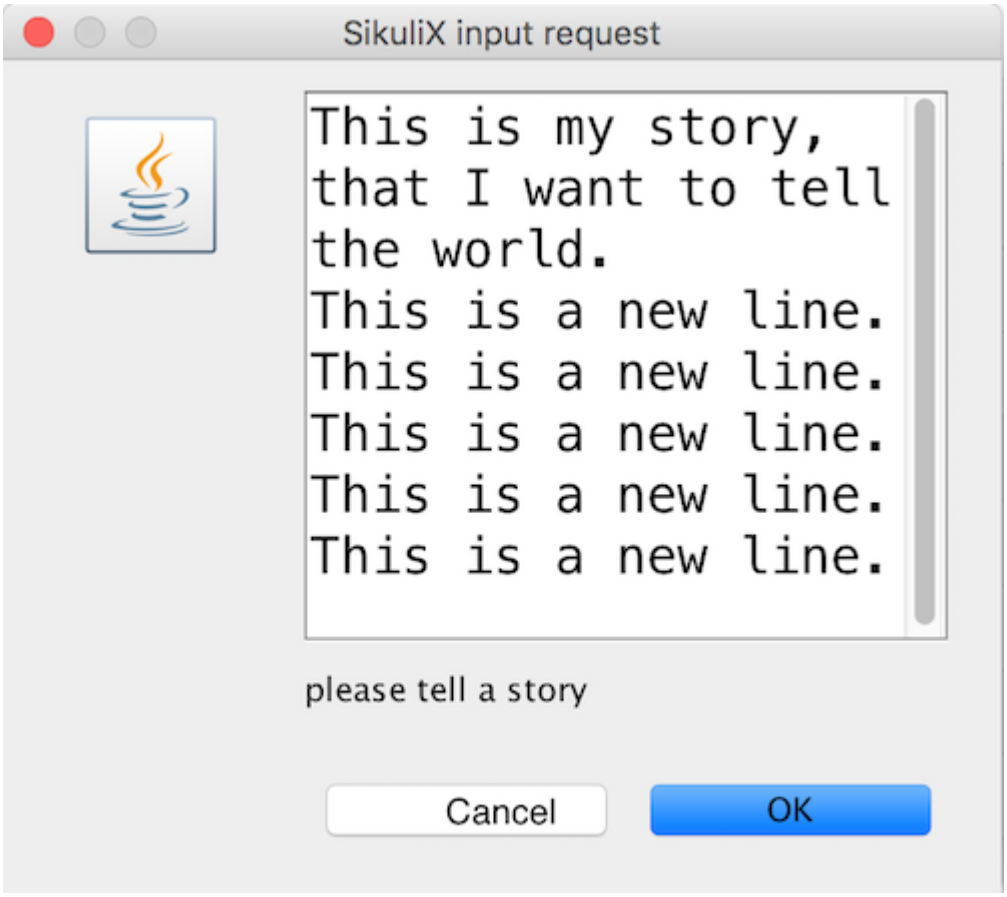
A bigger size than 14 can be set using `Settings.InputFontSize=NN`. Setting it to a value smaller than 14 (e.g. 0) will reset it to 14 again.

Example:

```
# selects a monospaced font
# default is False meaning a SansSerif font
Settings.InputFontMono = True

# default fontsize is 14 (also minimum size)
# use a fontsize of 20
Settings.InputFontSize = 20

story = inputText("please tell a story")
lines = story.split("\n") # split the lines in the list lines
for line in lines:
   print line
```



New in version 1.1.0.

select([*msg*][, *title*][, *options*][, *default*])

Parameters

- **msg** – text to be displayed as message (default: nothing)

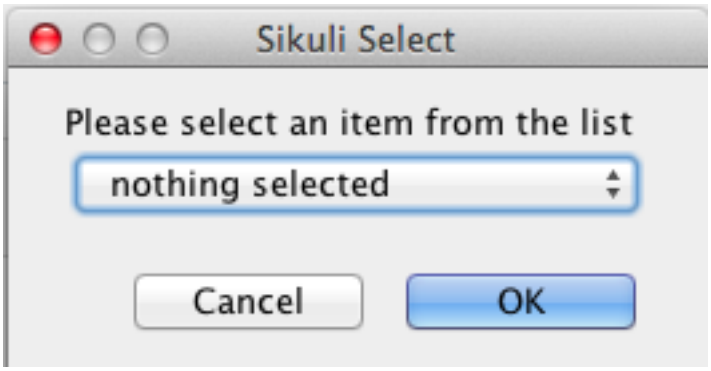
- **title** – optional title for the messagebox (default: Sikuli Selection)
- **options** – a list of text items (default: empty list, nothing done)
- **default** – the preselected list item (default: first item)

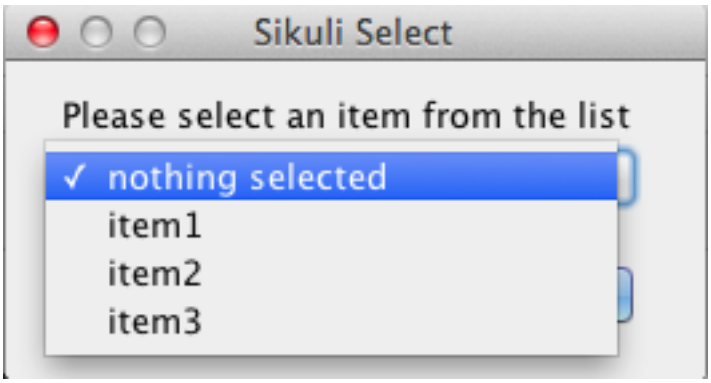
Returns the selected item (might be the default)

Displays a dropdown menu containing the given options list items with the default selected. The user might select one item and click ok.

Example:

```
items = ("nothing selected", "item1", "item2", "item3")
selected = select("Please select an item from the list", options = items)
if selected == items[0]:
   popup("You did not select an item")
   exit(1)
```





New in version 1.1.1.

popFile ([*title*])

Display a file open dialog, that lets the user select a folder or file.

Parameters **title** – optional title for the dialogbox (default: Select a file or folder)

Returns the absolute path of the selected file or folder as a string

Timed (autoclosing) popups

New in version 1.1.1.

As a backport from SikuliX version 2 I added the timed/autoclosing versions of `popup`, `popAsk`, `popError` and `input`.

General information on using these features:

- the respective methods are implemented in the class `org.sikuli.script.Do` (this corresponds to the implementation of all toplevel features in SikuliX version 2 in `com.sikulix.api.Do`). Hence the usage in scripts is upward compatible. At the Java level at least the import would have to be changed.
- **the usage in all cases** is `returnValue = Do.function()`. Do not try any other usage, since this might clash with existing version 1 implementations.
- since the implementation is only on the Java level, there are no named parameters (unlike the known non-timed versions of `popup`, `popAsk`, ...). Nevertheless it is possible to only give a subset of parameters, as long as the defined sequence is obeyed. In doubt use `None/null` for a parameter, to get the default value. See the given examples for use cases.
- if the dialog is autoclosed by intention, the return value is `None/null` in all cases.
- the dialogs can only be displayed on the primary screen

These are the **possible parameters and their defined sequence**:

- **message** a declarative text to be shown in the dialog (`all methods`, default `"not set"`)
- **title** the dialog box title (`all methods`, default `"SikuliX"`)
- **preset** a prefilled input text (`input`, default empty)
- **hidden** a boolean value, `True/true` will show the input text as dots ((`input`, default `False/false`)
- **timeout** an integer as seconds after that the dialog will autoclose (`all methods`, default stay open)
- **location** a Region object, over which the dialog will be displayed centered (`all methods`, default screen center) which allows to place the dialog anywhere on the screen. As a convenience you can use `Region(x, y)` if you want to specify a point. Hence no need to use `popAt()`.

New in version 1.1.1.

`Do.`**`popup`**`(`[*parameters*]`)`
Display an informational message with an `OK` button.

Parameters **`parameters`** – see above

Returns always `True`, `None/null` if autoclosed

Example:

```
result = popup("autoclosed after 3 seconds", 3)
if not result:
  print "user did not click ok"
```

New in version 1.1.1.

`Do.`**`popAsk`**`(`[*parameters*]`)`
Display an informational message with `YES` and `NO` button.

Parameters **`parameters`** – (see above)

Returns `True` if `YES` was clicked, `False` otherwise, `None/null` if autoclosed

Example:

```
result = popAsk("Nothing done if not\nclicked within 3 seconds", "Your decision", 3)
if None == result:
  print "nothing to do"
elif result:
  print "user said yes"
```

```
else:
  print "user said no"
```

New in version 1.1.1.

Do.**popError**([*parameters*])
Display an error message with an OK button.

Parameters **parameters** – see above

Returns always True, None/null if autoclosed

Example:

```
result = popError("autoclosed after 3 seconds", "Severe Error", 3, Region(300,300))
# the dialog will display somewhere in the upper left of the screen
# with a box title as "Severe Error"
if not result:
  print "user did not click ok"
```

New in version 1.1.1.

Do.**input**([*parameters*])
Display an informational message and ask for a text input with a possible preset text in the input field. The dialog has an OK and a Cancel button. With the hidden parameter as True/true the text in the input field will be shown as dots (not readable).

Parameters **parameters** – see above

Returns the text in the input field, when clicked OK, False/false otherwise, None/null if autoclosed

Example:

```
result = input("please fill in", "A filename", "someImage.png", Region(300,300))
# the dialog will display somewhere in the upper left of the screen
# with a box title as "A filename"
# and a preset input field containing "someImage.png"
if not result:
  # input field was left empty
  print "we will use a default file name"
else:
  print "we will use as filename: " + result
```

Example for hidden input:

```
password = input("please enter your secret", "Secret", "defaultSecret", True, 10)
# the dialog's input field displays the text as dots per character
if not password:
  # password is empty or dialog autoclosed
  print "not allowed - exiting"
  exit(1)
# we can proceed
```

Listening to Global Hotkeys

Sikuli can listen to global hotkeys that you register with Env.addHotkey and call the corresponding handler (sikuli functions) when the user presses the hotkeys.

BE AWARE Be sure, that the key combination you use is free and not used by the system or any other application. The hotkey feature may not report an error in such situations and as a consequence your hotkey definition simply does not work as expected. An example is the F12 key on Windows alone or with SHIFT, which in the standard is occupied by the system as global debugging key (might be released by hacking the registry).

`Env.`**`addHotkey`**(*key*, *modifiers*, *handler*)
Register the specified *key* + *modifiers* as a global hotkey. When the hotkey is pressed, the specified function *handler* will be called.

Parameters

- **`key`** – a character or a constant value defined in *`Key`*.
- **`modifiers`** – Key modifiers, which can be one or multiple constants defined in `KeyModifier`.

Returns True if success.

`Env.`**`removeHotkey`**(*key*, *modifiers*)
Unregister the registered global hotkey *key* + *modifiers*.

Parameters

- **`key`** – a character or a constant value defined in *`Key`*.
- **`modifiers`** – Key modifiers, which can be one or multiple constants defined in `KeyModifier`.

Returns True if success.

A more generic example

It keeps the handlers free from processing code, just signals the keypress using a global variable to the main loop. The main loop simply permanently scans the global variables and then does what has to be done.

The whole process is blocking in the sense, that hotkeys are processed one after the other in the sequence they appear in the main loop and each hotkey is only recognized again, after its current press is processed in the main loop.

This setup keeps things more transparent and straightforward. Other setups even with threading are possible, but need much more effort to correctly synchronize the processing especially when mouse or keyboard actions are involved.

Example:

```
# hotkey to stop the script
hotKeyX = False; # global to communicate with main loop
def xHandler(event):
  global hotKeyX
  hotKeyX = True # tell main loop that hotkey was pressed
# add the hotkey with its handler
Env.addHotkey("x", KeyModifier.CTRL + KeyModifier.SHIFT, xHandler)

# function hotkey: something to do when pressed
hotKeyN = False;
def nHandler(event):
  global hotKeyN
  hotKeyN = True
Env.addHotkey("n", KeyModifier.CTRL + KeyModifier.SHIFT, nHandler)

# the main loop, that simply waits for pressed hotkeys
# which are then processed
count = 0;
while True:
  if (hotKeyX):
```

```
    popup("processing ctrl+shift+x: stopping")
    exit()
  if (hotKeyN):
    hotKeyN = False # reset the hotkey variable
    # and now do something
    count += 1
    popup("processing ctrl+shift+n: %d" % count)
  wait(1)
```

Starting and stopping other applications and bring them to front

New in version 1.1.0.

Completely revised in version 1.1.0

Here we talk about the basic features of opening or closing other applications and switching to them (bring them to front).

For the more sophisticated usages including some basic handling of application windows look class *App*.

You can use the feature run(someCommand) to delegate something, you can do on a commandline, to a seperate process. The script waits for completion and you have acces to the return code and the output the command has produced.

NOTE on Java usage At the Java level only the features of the App class are available (class *App*).

General hint for Windows users on backslashes \ and double apostrophes "

In a Sikuli script in normal strings enclosed in " (double apostrophes), these special characters \ and " have to be escaped using a backslash, when you have them inside the string. So for one backslash you need \\ and for one " you need \". In a string enclosed in ' (single apostrophes), a ' has to be \' and a " is taken as such.

To avoid any problems, it is recommended to use the raw string `r'some text with \ and " ...'`, since there is no need for escaping (but no trailing \ is allowed here).

> This is especially useful, when you have to specify Windows path's containing blanks or want to setup command lines for use with openApp(), App.open(), run(), os.popen() or Jythons Subprocess module.

NOTE for Mac users As application name use the name, that is displayed with the program symbol on the taskbar, which might differ from what is displayed in the top left of the menu bar.

Example: The Chrome browser displays "Chrome" in the menu bar, but the application name is "Google Chrome". So openApp("chrome") will fail, whereas openApp("google chrome") will do the job. Same goes for switchApp() and closeApp().

openApp(*application*)

Open the specified application, or swith to it, if it is already open.

> **Parameters** **application** – a string containing the name of an application (case-insensitive), that can be found in the path used by the system to locate applications. Or it can be the full path to an application.

:return None if an error occured, on success a new App class object (look *App*)

This function opens the specified application and brings it to front. It might switch to an already opened application, if this can be identified in the process list.

Windows: A running instance will be ignored in any case and hence in most cases a new instance of the program will be started.

Examples:

```
# Windows: run a batch file in a new command window:
`òpenApp("cmd.exe /c start path-to-some.bat")``

# Windows: opens Firefox (full path specified)
``openApp("c:\\Program Files\\Mozilla Firefox\\firefox.exe")`` or
``openApp(r"c:\Program Files\Mozilla Firefox\firefox.exe")``

# Mac: opens Safari
``openApp("Safari")``
```

switchApp(*application*)

Bring the matching application or window to front (make it the active/focused application/window). If no matching application/window can be found, it is tried to open an application using the given string as program name or location.

Parameters **application** – the name of an application (case-insensitive) or (part of) a window title (Windows/Linux) (case-sensitive).

:return None if an error occured, on success a new App class object (look *App*)

This function switches the input focus to the specified application (brings it to front).

Windows: In the first step, the given text is taken as part of a program name (not case sensitive). If it is found in the process list, it will be switched to front, if it has a main window (registered in the process list). Otherwise the text will be used to search for a matching window title.

Windows/Linux: the window is identified by scanning the titles of all accessible windows for the occurence of the *application* string. The first window in the system specific order, whose title contains the given text, is given focus.

Mac: the string `application` is used to identify the application. If the application has multiple windows opened, all these windows will be brought to the front with unchanged z-order, which cannot be influenced currently.

Examples:

```
# Windows: switches to an already opened Firfox or opens it otherwise
switchApp("c:\\Program Files\\Mozilla Firefox\\firefox.exe")

# Windows: switches to the frontmost opened browser window (or does nothing
# if no Firefox window is currently opened)
# works, because all Firefox window titles contain "Mozilla Firefox"
switchApp("Mozilla Firefox")

# Mac: switches to Safari or starts it
switchApp("Safari")
```

closeApp(*application*)

Close the specified application.

Parameters **application** – the name of an application (case-insensitive) or (part of) a window title (Windows/Linux)

:return None if an error occured, on success a new App class object (look *App*)

This function closes the application indicated by the string *application* (Mac) or the windows whose titles contain the string *application* (Windows/Linux). On Windows/Linux, the application itself may be closed if the main window is closed or if all the windows of the application are closed.

Example:

```
# Windows: closes Firefox if it is running, does nothing otherwise
closeApp("c:\\Program Files\\Mozilla Firefox\\firefox.exe")

# Windows: stops firefox including all its windows
closeApp("Mozilla Firefox")

# Mac: closes Safari including all its windows
closeApp("Safari")
```

run (*command*)

Run *command* in the command line

Parameters **command** – a command that can be run from the command line.

Returns a multiline string containing the result of the execution

This function executes the command and the script waits for its completion.

structure of the result (comments after #, not part of the result)

Multiline string:

```
N # a number being the return code
text
text
text
text # no, one or more lines execution output (stdout)
*****error***** # if the execution ended with an error
error text # or the return code was not 0
error text
error text # no, one or more lines error output (stderr)
```

NOTE for usage variants of the command run() and for the Java usage see class *App*

General Settings and Access to Environment Information

Java Level

Java maintains a global storage for settings (key/value pairs), that can be accessed by the program/script. Sikuli uses it for some of it's settings too. Normally it is not necessary to access these settings at the Java level from a Sikuli script, since Sikuli provides getter and setter methods for accessing values, that make sense for scripting. One example is the list of paths, that Sikuli maintains to specify additional places to search for images (please refer to *Importing other Sikuli Scripts* for more information).

If needed, you may access the java settings storage as shown in the following example:

```
import java.lang.System

# get a value
val = System.getProperty("key-of-property")

# set a property's value
System.setProperty("key-of-property", value)
```

Jython/Python Level

You may use all settings, that are defined in standard Python/Jython and that are available in your system environment.

The modules `sys` and `time` are already imported, so you can use their methods without the need for an import statement.

`sys.path` may be one of the most valuable settings, since it is used by Python/Jython to locate modules, that are referenced using `import module`. It is a list of path's, that is e.g. maintained by Sikuli to implement *Importing other Sikuli Scripts* as a standard compliant feature (exception: .Sikuli scripts cannot form a module tree).

If you want to use `sys.path`, it is recommended to do it as shown in the following example, to avoid appending the same entry again:

```
myPath = "some-absolute-path"
if not myPath in sys.path:
        sys.path.append(myPath)
```

Sikuli Level

Sikuli internally uses the class *`Settings`* to store globally used settings. Publicly available attributes may be accessed by using `Settings.[name-of-an-attribute]` to get it's value and `Settings.attribute = value` to set it. It is highly recommended to only modify attributes, that are described in this document or when you really know, what you are doing.

Actually all attributes of some value for scripting are described in the topic *Controlling Sikuli Scripts and their Behavior*.

To store some **settings across SikuliX IDE sessions**, SikuliX utilizes the Java feature Preferences.

As persistent storage Java uses:

- on Windows the registry branch HKCUSoftwareJavaSoftPrefsorgsikuli...
- on Mac a plist file in ~/Library/Preferences/org.sikuli.....plist
- on Linux usually at ~/.java/.userPrefs/org/sikuli/prefs.xml

The content is *controlled by the IDE's Preferences panel*. It is safe to delete this branch/file, to get a default setup and might help in some situations, wher the startup of the IDE does not work or crashes.

Store some of your information persistently and reload it later

You can have a so called **Property File** somewhere on the file system, that you can prefill with key-value-pairs representing information, that can be used by your scripts for whatever purpose at runtime. So it can be used instaed of commandline parameters, for some kind of data-driven approach or for any other solution, that needs information to be persistent over time.

```
# this is a property file
key1 = value1
key2 = value2
...
```

At runtime in your script, you first load such a property file into an in-memory store and then access the values using their keys (both basically are strings). You might change existing values, add new values and remove values. At any time you might save the store content back to the originating file or to another file.

Currently there is no Auto-Save feature, so that your changes are lost in case of crashes before you saved the store back to a file. The feature might not be fully thread safe.

Features that operate on the store as entity

`loadOpts` (*filePath*)
load a property file into an internal store

Parameters **filePath** – absolute or relative to the working folder

Returns the reference to the internal store to be used with the store functions

saveOpts(*store*)
save the store back to the file it was loaded from

Parameters **store** – the reference to a loaded store

Returns true if it worked, false otherwise

saveOpts(*store*, *filepath*)
save the store to the given file, overwritten without notice

Parameters

- **store** – the reference to a loaded store
- **filePath** – absolute or relative to the working folder

Returns true if it worked, false otherwise

makeOpts()
make a new, empty internal store, that might be saved to a file later

Returns the reference to the internal store to be used with the store functions

delOpts(*store*)
purge all key-value-pairs from the store (make it empty)

Parameters **store** – the reference to a loaded store

Returns true if it worked, false otherwise

hasOpts(*store*)
count the key-value-pairs in the store

Parameters **store** – the reference to a loaded store

Returns a positive number (0 means empty)

getOpts(*store*)
load the key-value-pairs into a dictionary (Java: Map(String, String)), to be able to use more powerful features on the store information

Parameters **store** – the reference to a loaded store

Returns the dictionary filled with the key-value-pairs

setOpts(*store*, *map*)
store the key-value-pairs from a dictionary (Java: Map(String, String)) to the given store

Parameters

- **store** – the reference to a loaded store
- **map** – the dictionary/map containing the key-value-pairs

Returns the number of stored key-value-pairs (0 might signal a problem)

Features that operate on individual entries in a loaded store

hasOpt(*store*, *key*)
check the existence of a key-value-pair

Parameters

- **store** – the reference to a loaded store

- **key** – the key as string of a stored key-value-pair

Returns true if the key exists, false otherwise

getOpt(*store*, *key*[, *default*])
read the value of a specific key and get the default value, if the key does not exist. If the key does not exists and no default is given, an empty string is returned.

Parameters

- **store** – the reference to a loaded store
- **key** – the key as string of a stored key-value-pair
- **default** – an optional value in case the key does not exist in the store

Returns the stored value or the default

setOpt(*store*, *key*, *value*)
set the value of a specific key. if the key does not exist, the key-value-pair is added, otherwise the value is overwritten.

Parameters

- **store** – the reference to a loaded store
- **key** – the key as string of a stored key-value-pair
- **value** – a string value to be stored with the given key

Returns the stored value before the change, an empty string if the key did not exist yet

delOpt(*store*, *key*)
delete the key-value-pair from the store

Parameters

- **store** – the reference to a loaded store
- **key** – the key as string of a stored key-value-pair

Returns the stored value before the deletion, an empty string if the key did not exist yet

Convenience functions for number values

Since the values in the store are strings only, the following functions take care for the necessary conversions. All returned numbers are of format double.

getOptNum(*store*, *key*[, *default*])
read the value of a specific key and get the default value, if the key does not exist. If the key does not exists and no default is given, a 0.0 is returned.

Parameters

- **store** – the reference to a loaded store
- **key** – the key as string of a stored key-value-pair
- **default** – an optional number value in case the key does not exist in the store

Returns the stored value as double or the default

setOptNum(*store*, *key*, *value*)
set the value of a specific key. if the key does not exist, the key-value-pair is added, otherwise the value is overwritten.

Parameters

- **store** – the reference to a loaded store
- **key** – the key as string of a stored key-value-pair
- **value** – an valid number value to be stored with the given key

Returns the stored value before the change, a 0.0 if the key did not exist yet

The following feature only works on the same machine

... and has nothing to do with the above feature, but can of course be combined.

You might use SikuliX's persistent storage, to **store and reload your own information** accross SikuliX sessions or only across different runs of same or different scripts/programs.

There is no feature to preload the store before the first run nor to export your information.

`Sikulix.`**`prefStore`**(*key*, *value*)
Store a key-value-pair in Javas persistent preferences store

Parameters

- **key** – an item name as string
- **value** – a string value to be stored as the item's content

`Sikulix.`**`prefLoad`**(*key*[, *value*])
Retrieve the value of a previously stored key-value-pair using key as the item's name

Parameters

- **key** – an item name as string
- **value** – an optional string value to be returned, if the item was not yet stored like some default

Returns the item's content if the item exists, otherwise an empty string or the given default

`Sikulix.`**`prefRemove`**(*key*)
Permanently remove the key-value-pair using key as the item's name

Parameters **key** – an item name as string

Returns the item's content if the item exists, otherwise an empty string

`Sikulix.`**`prefRemove`**()
Permanently remove all key-value-pairs stored before using *`Sikulix.prefStore()`*

Parameters **key** – an item name as string

Returns the item's content if the item exists, otherwise an empty string

Get Information about the runtime environment

The `class Env` is deprecated and should not be used anymore. The contained features are moved to other places and redirected from inside `class Env` to be downward compatibel.

NOTE: In the following the **non-Env methods** are the replacements, that should be used instead.

`Settings.`**`getOS`**()
`Env.`**`getOS`**()
`Settings.`**`getOSVersion`**()

`Env.`**`getOSVersion`**`()`
Get the type (`getOS()`) and version string (`getOSVersion()`) of the operating system your script is running on.

An example using these methods on a Mac is shown below:

```
# on a Mac
myOS = Settings.getOS()
myVer = Settings.getOSVersion()

if myOS == OS.MAC:
        print "Mac " + myVer # e.g., Mac 10.6.3
else:
        print "Sorry, not a Mac"

myOS = Settings.getOS()
if myOS == "MAC" or myOS.startswith("M"):
        print "Mac " + myVer # e.g., Mac 10.6.3
else:
        print "Sorry, not a Mac"
```

There are convenience functions, to **check wether we are running on a specific system**:

`Settings.`**`isWindows`**`()`
`Settings.`**`isMac`**`()`
`Settings.`**`isLinux`**`()`

Returns True if we are running on this system, False otherwise

`Settings.`**`getSikuliVersion`**`()`
`Env.`**`getSikuliVersion`**`()`
Get the version of Sikuli.

Returns a string containing the version string

```
if not Settings.getSikuliVersion().contains("1.0.1"):
        print "This script needs SikuliX 1.0.1"
        exit(1)
```

`App.`**`getClipboard`**`()`
`Env.`**`getClipboard`**`()`
Get the content of the clipboard if it is text, otherwise an empty string.

NOTE: Be careful, when using `Env.getClipboard()` together with `paste()`, since paste internally uses the clipboard to transfer text to other applications, the clipboard will contain what you just pasted. Therefore, if you need the content of the clipboard, you should call `Env.getClipboard()` before using `paste()`.

Tipp: When the clipboard content was copied from a web page that mixes images and text, you should be aware, that there may be whitespace characters around and inside your text, that you might not have expected. In this case, you can use `Env.getClipboard().strip()` to at least get rid of surrounding white spaces.

`Key.`**`isLockOn`**`(`*key-constant*`)`
`Env.`**`isLockOn`**`(`*keyConstant*`)`
Get the current status (on / off) off the respective key. Only one key can be specified.

Parameters **`keyConstant`** – one of the key constants `Key.CAPS_LOCK`, `Key.NUM_LOCK`, `Key.SCROLL_LOCK`

Returns True if the specified key is on, False otherwise

Further information about key constants can be found in Class *Key*.

Mouse.**at**()
Env.**getMouseLocation**()
Get the current location of the mouse cursor.

Returns a *Location* object of the position of the mouse cursor on the screen.

Advanced Settings for Speed and Robustness

Note It is not recommended, to use this.

With version 1.2 the matching process will be revised anyway and might bring other options. If you really want to speed up the search process, take care, that you are searching in a region being as small as possible.

Here you get more information about how to make your workflow fast and robust.

Vision.**setParameter**(*param*, *value*)
Set the parameter *param* of the vision algorithm to *value*.

Parameters

- **param** – a string that indicates the parameter to set.
- **value** – a float value.

Vision.**getParameter**(*param*)
Get the parameter *param* of the vision algorithm.

Parameters **param** – a string that indicates the parameter to get.

Returns the float value of the specified parameter.

MinTargetSize

MinTargetSize is the minimum image size to which Sikuli can resize.

If you feel that Sikuli is running too slow, you might try a smaller value than 12. On the other hand, if you see Sikuli returns a match that is not what you expect, i.e. a false match, try to increase MinTargetSize to make Sikuli be more robust to small details.

You can tune this parameter using the following Jython code:

```
from org.sikuli.natives import Vision
Vision.setParameter("MinTargetSize", 6) # the default is 12.
```

Setting the size to a smaller value would make the matching algorithm be faster.

Region (rectangular pixel area on a screen)

class Region

The Region is besides Images/Patterns (called Visuals) and Matches (where a Visual was found and how) the **basic element in the SikuliX concept**. So be sure, you have understood all aspects of a Region.

A Region is a rectangular area on a *Screen* and is defined by

1. its upper left corner (x, y) being the pixel with this offset relative to the upper left corner of the screen (usually (0, 0)) and
2. its dimension (w, h) as its width and height in pixels.

x, y, w, h are integer numbers counting a distance in pixels.

A Region **does not know anything about it's visual content** (windows, pictures, graphics, text, ...). It only knows *the position on the screen and its dimension*.

A `Match`, being the result of a `Region.find()` operation, basically is a Region in all aspects, just having a few additional attributes specific for a find result.

New Regions can be created in various ways:

- specify their position and dimension
- extend a given Region in all directions (expand or shrink)
- based on adjacent rectangles up to the bounds of the screen horizontally or vertically.
- based on their corners
- as subregions being rows, columns or cells of a regular grid
- combine different Regions or use their intersection

You can use `Region.find()`, to **search a given Visual being a rectangular pixel pattern** (given as an Image (filename or `Image`) or a `Pattern` object) within this Region. If this Visual is found in the Region, the resulting `Match` object has a similarity score between >0 and 1. The lower the similarity score, the higher the chance, that you got a false positive (found something else). To make your scripts robust against false positives, you should take care, to get similarity scores >0.85 or even >0.9.

If the Visual is given for the search as an Image , Sikuli uses a **minimum similarity of 0.7**, which only returns Matches with a score >0.7. This default value can be changed in `Settings.MinSimilarity`. A Pattern is searched with the optionally given minimum similarity using `Pattern.similar()`.

Find operations return a `Match` object, which has all attributes and methods of a Region and can be used in the same way as a Region (e.g. find something or click another target inside). A `Match` has the size in pixels of the Visual used for searching, the position where it was found, the similarity score and the elapsed time.

Look here for more detailed information on *How SikuliX finds images on the screen*.

Be aware: every mouse or keyboard action, that specifies a Visual to search for, will internally do the respective find operation first, to evaluate the action target.

A Region **remembers the match of the last successful find operation**, all matches of the last successful `Region.findAll()` and the elapsed time. With `Region.getLastMatch()`, `Region.getLastMatches()` and `Region.getLastTime()` you can get these objects/value.

You can **wait for a Pattern** to appear using `Region.wait()` or wait for it to vanish using `Region.waitVanish()`

Every not successful find operation (even those done internally with a click() ...) will raise a *FindFailed exception*, that has to be handled in your script. If you do not do that, your script will simply stop here with an error.

If you do not want to handle these FindFailed exceptions, you might search for a Pattern using `exists`, which just returns nothing (None/null) in case of not found. So you simply check the return value for being a Match.

For other options to handle FindFailed situations see *FindFailed exception*.

During a find operation internally the search is repeated with a scan rate (standard 3 per second) **until success or an optionally given timeout** (standard 3 seconds) is reached, which then results in a *FindFailed exception*.

Sikuli supports **visual event driven programming**: You can tell a Region *to observe that something appears, vanishes or changes*. It is possible to wait for the completion of an observation or let it run in the background, while your script continues running. When one of the visual events happens, a handler in your script is called. Each Region can only have one observer, but each observer can observe multiple visual events in that Region. You might also check the

status of a background observation later in your workflow, to handle events inline. Generally it is your responsibility to stop observations, but at termination of your script or Java program all observations are terminated automatically.

NOTE: For **hints and tips how to get robust and fast acting workflows** look into the Best Practices.

Create a Region, Set and Get Attributes

NOTES

In any case a newly created Region will be restricted to the boundaries of the screen containing the largest part of the new Region.

It displays an **error, if no part of the newly created Region is contained by any of the available screens**. Subsequent usages of such a Region might result in errors, exceptions or even crashes, if features are used, that access the screen.

Use `Region.isValid()` to check, wether a Region is contained by a screen.

Create a new Region based on top left corner and size

class `Region`

`Region`(*x*, *y*, *w*, *h*)
`Region`(*region*)
`Region`(*Rectangle*)
Create a region object

Parameters

- **`x`** – x position of top left corner
- **`y`** – y position of top left corner.
- **`w`** – width of the region.
- **`h`** – height of the region.
- **`region`** – an existing Region object.
- **`rectangle`** – an existing object of class java.awt.Rectangle

Returns a new Region object.

For **other ways to create new Regions** see: *Extend Regions ...*.

NOTE: The position and dimension attributes are named x, y representing the top left corner and w, h being width and height. You might access/change these values directly or use the available getter/setter methods.

```
topLeft = Location(reg.x, reg.y) # equivalent to
topLeft = reg.getTopLeft()

theWidth = reg.w # getting the width equivalent to
theWidth = reg.getW()

reg.w = theWidth # setting the width equivalent to
reg.setW(theWidth)

# same is available for the height: reg.h, reg.getH(), reg.setH()
```

Note: Additionally you might use *`selectRegion()`* to interactively create a new region at runtime.

NOTE: Using *Region(someOtherRegion)* just duplicates this region (creates a new object). This can be useful, when you **need the same Region with different attributes**, such as another *observation loop* or

another setting for *Region.setThrowException()* to control whether throwing an exception or not when find ops fail.

Change a Regions position and/or size

class Region

setX(*number*)
setY(*number*)
setW(*number*)
setH(*number*)
Set the respective attribute of the region to the new value. This effectively moves the region around and/or changes its dimension.

Parameters **number** – the new value

moveTo(*location*)
Set the position of this region regarding it's top left corner to the given location (the x and y values are modified).

Parameters **location** – location object becomes the new top left corner

Returns the modified region object

```
reg.moveTo(anotherLocation) # equivalent to
reg.setX(anotherLocation.x); reg.setY(anotherLocation.y)
```

setROI(*x*, *y*, *w*, *h*)
setROI(*rectangle*)
setRect(*x*, *y*, *w*, *h*)
setRect(*rectangle*)

All these methods are doing exactly the same: setting position and dimension to new values. The motivation for two names is to make scripts more readable: `setROI()` is intended to restrict the search to a smaller area to speed up processing searches (region of interest), whereas `setRect()` should be used to change a region (move and/or shrink or enlarge).

Parameters

- **x, y, w, h** (*all*) – the attributes of a rectangle
- **rectangle** – a rectangle object

Returns None

morphTo(*region*)
Set the position and dimension of this region to the corresponding values of the region given as parameter. (see: *setRect()*)

Parameters **region** – a region object

Returns the modified region object

```
reg.morphTo(anotherRegion) # equivalent to
r = anotherRegion; reg.setX(r.x); reg.setY(r.y); reg.setW(r.w); reg.setH(r.h)
```

Access a Region's attributes and settings

class Region

getX()
getY()
getW()
getH()
Get the respective attribute of the region.

Returns integer value

getCenter()
Get the center of the region.

Returns an object of *Location*

getTopLeft()
getTopRight()
getBottomLeft()
getBottomRight()
Get the location of the region's respective corner

Returns Location object

getScreen()
Returns the screen object that contains this region.

Returns a new *Screen* object

See *Multi Monitor Environments*.

getLastMatch()
getLastMatches()
To access the Matches returned by the last find op in this Region.

Returns a *Match* object or a list of Match objects

All basic find operations (explicit like *Region.find()* or implicit like *Region.click()*) store the match in `lastMatch` and multi-find ops (like *Region.findAll()*) all found matches into `lastMatches` of the Region that was searched.

How to go through the Matches returned by getLastMatches().

TIPP: The `LastMatch` can be used to avoid a second search for the same Visual in sequences like:

```
wait(someVisual)
click(someVisual)
# or
if exists(someOtherVisual):
        click(someOtherVisual)
```

To avoid the second search with the `click()` you can use:

```
wait(someVisual)
click(getLastMatch())
# or
if exists(someOtherVisual):
        click(getLastMatch())
```

There are convenience shortcuts for this:

```
wait(someVisual)
click()
# or
```

```
if exists(someOtherVisual):
        click()
```

See *Region.click()* for the usage of these convenience shortcuts. A `someRegion.click()` will either click the center of the given Region or the `lastMatch`, if any is available.

getTime()

Returns the elapsed time in number of milli-seconds of the last find op in this Region

Attributes influencing the behavior of features of a Region

class Region

NOTE For settings influencing the handling of Visual-not-found situations in this Region look here: *FindFailed Exceptions*.

isRegionValid()

Returns False, if the Region is not contained by a Screen and hence cannot be used with faetures, that need a Screen (find, capture, ...), otherwise True.

setAutoWaitTimeout(*seconds*)

Set the maximum waiting time for all subsequent find operations in that Region.

Parameters **seconds** – a number, which can have a fraction. The internal granularity is milli-seconds.

All subsequent find ops will be run with the given timeout instead of the current value of `Settings.AutoWaitTimeout`, to which the region is initialized at time of creation (default 3 seconds).

getAutoWaitTimeout()

Get the current value of the maximum waiting time for find ops in this region.

Returns timeout in seconds

setWaitScanRate(*rate*)

Set this Region's value: A find op should repeat the search for the given Visual `rate` times per second until found or the maximum waiting time is reached. At time of Region creation the value is initialized from *Settings.WaitScanRate*, which has a default of 3.

Parameters **rate** – a value > 0. values < 1 will lead to scans every x seconds and hence longer pauses between the searches (reduces cpu load).

TIPP Since on average the shortes search times are some milli seconds, `rate` > 100 will lead to a continous search under all circumstances.

getWaitScanRate()

Get the current value of this Region's `WaitScanRate`.

Returns the rate number

Get evenly sized parts of a Region (as rows, columns and cells based on a raster)

In many cases, one has parts of a GUI, that are to some extent evenly structured, having some virtual raster (rows, columns and/or cells), that one wants to use for restricting searches or walk through this parts for other reasons.

Typical examples are tables like in an Excel sheet, boxes in some GUI or on a webpage or dropdown lists and menues.

A given Region can be set to have some evenly sized raster, so that one can access these subregions and create new Regions.

Convenience functions, to get a subregion from a specified raster in one step

class Region

get (*somePart*)

Select a part of the given Region based on the given part specifier.

Parameters **somePart** – a constant as Region.CONSTANT or an integer between 200 and 999 (see below)

Returns a new Region created from the selected part

Usage based on the javadocs:

```
Constants for the top parts of a region (Usage: Region.CONSTANT)
shown in brackets: possible shortcuts for the part constant
NORTH (NH, TH) - upper half
NORTH_WEST (NW, TL) - left third in upper third
NORTH_MID (NM, TM) - middle third in upper third
NORTH_EAST (NE, TR) - right third in upper third
... similar for the other directions:
right side: EAST (Ex, Rx)
bottom part: SOUTH (Sx, Bx)
left side: WEST (Wx, Lx)

specials for quartered:
TT top left quarter
RR top right quarter
BB bottom right quarter
LL bottom left quarter

specials for the center parts:
MID_VERTICAL (MV, CV) half of width vertically centered
MID_HORIZONTAL (MH, CH) half of height horizontally centered
MID_BIG (M2, C2) half of width / half of height centered
MID_THIRD (MM, CC) third of width / third of height centered

Based on the scheme behind these constants there is another possible usage:
specify part as e 3 digit integer where the digits xyz have the following meaning
1st x: use a raster of x rows and x columns
2nd y: the row number of the wanted cell
3rd z: the column number of the wanted cell
y and z are counting from 0
valid numbers: 200 up to 999 (< 200 are invalid and return the region itself)
example: get(522) will use a raster of 5 rows and 5 columns and return the cell␣
↪in the middle
special cases:
if either y or z are == or > x: returns the respective row or column
example: get(525) will use a raster of 5 rows and 5 columns and return the row in␣
↪the middle
```

Internally this is based on *Region.setRaster()* and *Region.getCell()*.

If you need only one row in one column with x rows or only one column in one row with x columns you can use *Region.getRow()* or *Region.getCol()*

getRow (*whichRow*, *numberRows*)

Parameters

- **numberRows** – in how many evenly sized rows should the region be devided

- **whichRow** – the row to select counting from 0, negative counts backwards from the end

Returns a new Region created from the selected row

getCol (*whichColumn*, *numberColumns*)

Parameters

- **numberColumns** – in how many evenly sized columns should the region be devided
- **whichColumn** – the column to select counting from 0, negative counts backwards from the end

Returns a new Region created from the selected column

The basic functions for any raster setup

class Region

setRows (*numberRows*)
setCols (*numberColumns*)
Define a rows or columns only raster, by dividing the Region's respective dimension into even parts. The corresponding Regions will only be created, when the respective access methods are used later.

Parameters

- **numberRows** – the number of rows the Region should be devided in
- **numberColumns** – the number of columns the Region should be devided in

Returns the first element as new Region if successful or the region itself otherwise

setRaster (*numberRows*, *numberColumns*)
Define a raster, by deviding the Region's height in `numberRows` even sized rows and it's width into `numberColumns` even sized columns.

Parameters

- **numberRows** – the number of rows the Region should be devided in
- **numberColumns** – the number of columns the Region should be devided in

Returns the top left cell (`getCell(0, 0)`) if success, the Region itself if not

getRow (*whichRow*)
getCol (*whichColumn*)
Get the Region of the `whichRow` row or `whichColumn` column in the Region's valid raster counting from 0. Negative values will count backwards from the end. Invalid indexes will return the last or first element respectively.

Parameters

- **whichRow** – the number of the row to create a new Region from
- **whichColumn** – the number of the column to create a new Region from

Returns a new Region representing the selected element or the Region if no raster

getCell (*whichRow*, *whichColumn*)
Get the cell with the coordinates (`whichRow`, `whichColumn`) in the Region's valid raster counting from 0. Negative values will count backwards from the end. Invalid indexes will return the last or first element respectively. If the current raster only has rows or columns, the element of the corresponding index will be returned.

Parameters

- **whichRow** – the number of the row
- **whichColumn** – the number of the column

Returns a new Region representing the selected element or the Region itself if no raster

getting information about the current raster

class Region

isRasterValid()
Can be used to check, wether the Region currently has a valid raster

Returns True if it has a valid raster (either getCols or getRows or both would return > 0)

getRows()
getCols()

Returns the current raster setting (0 means not set) as number of rows/columns

getRowH()
getColW()

Returns the current raster setting (0 means not set) as height of one row or width of one column.

Extend Regions and create new Regions based on existing Regions

NOTES:

Except otherwise noted

- these methods **return new Region objects**, whose location and size are based on the specified region.
- the given **base Region remains unchanged**.

In any case the new Region will be restricted to the boundaries of the screen containing the largest part of the new Region.

It displays an **error, if no part of the new Region is contained by any of the available screens**. Subsequent usages of such a Region object might result in errors, exceptions or even crashes, if features are used, that acces the screen.

Use `Region.isValid()` to check, wether a Region is contained by a screen.

class Region

offset(*location*)
offset(*x*, *y*)
Creates a new Region object, whose upper left corner is relocated adding the given x and y values to the respective values of the given Region. Width and height are the same.

Parameters

- **location** – a *Location* object providing the relocating x and y values
- **x** – a number being the offset horizontally (< 0 to the left, > 0 to the right)
- **y** – a number being the offset vertically (< 0 to the top, > 0 to the bottom)

Returns the new *Region* object

```
new_reg = reg.offset(Location(xoff, yoff)) # same as
new_reg = Region(reg.x + xoff, reg.y + yoff, reg.w, reg.h)
```

inside()
Returns the same object. Retained for upward compatibility. `region.inside().find()` is totally equivalent to `region.find()`.

Returns Region itself

NOTE: Besides the individual methods like `nearby`, `left`, `right`, `above`, `below` there is one new method `grow` with some more options and different signatures. Where documented together, they are fully equivalent. The reason behind is some better compatibility to the usage of Java Rectangle.

class Region

grow([*range*])
nearby([*range*])
The new region is defined by extending (>0) or shrinking (<0) the current region's dimensions in all directions by `range` number of pixels. The center of the new region remains the same.

The default is taken from `Settings.DefaultPadding` (standard value 50)

Parameters **range** – an integer indicating the number of pixels or the current default if omitted.

Returns a new *Region* object

above([*range*])
below([*range*])
left([*range*])
right([*range*])
Returns a new *Region* that is defined with respect to the given region:

•above: new bottom edge next pixel row above given region's top edge

•below: new top edge next pixel row below given region's bottom edge

•left: new right edge next pixel clomn left of given region's left edge

•right: new left edge next pixel column right of given region's right edge

It does not include the current region. If *range* is omitted, it reaches to the corresponding edge of the screen.

Parameters **range** – a positive integer defining the new demension aspect (width or height)

Returns a new *Region* object

grow(*width*, *height*)

Finding inside a Region and Waiting for a Visual Event

Besides *acting on visual objects*, finding them is one of the core functions of Sikuli.

PS: means, that either a *Pattern* or a string (path to an image file or just plain text) can be used as parameter. A find operation is successful, if the given image is found with the given minimum similarity or the given text is found exactly. Similarity is a value between 0 and 1 to specify how likely the given image looks like the target. By default, the similarity is 0.7 if an image rather than a pattern object with a specific similarity is given to *Region.find()*.

Normally all these region methods are used as `reg.find(PS)`, where reg is a region object. If written as `find(PS)` it acts on the default screen, which is an implicit region in this case (see: *SCREEN as Default Region*). But in most cases it is a good idea to use `region.find()` to restrict the search to a smaller region in order to speed up processing.

If a find operation is successful, the returned match is additionally stored internally with the region that was used for the search. So instead of using a variable to store the match (`m = reg.find()`), you can use `reg.getLastMatch()` to access it afterwards. Unsuccessful find operations will leave these values unchanged. By default, if the **visual object (image or text) cannot be found**, Sikuli will stop the script by raising an *Exception FindFailed*. This follows the standards of the Python language, so that you could handle such exceptions using `try: ... except: ...`.

If you are not used to programming using the Python language or because of other reasons, you might just want to bypass the exception handling, which means just ignoring it (None is returned in that case). Or you might interactively react on a FindFailed situation (e.g. optionally repeat the find). Read more about concepts and options at: *Exception FindFailed*.

If you have **multiple monitors**, please read *Multi Monitor Environments*.

Note on IDE: Capturing is a tool in the IDE, to quickly set up images to search for. These images are named automatically by the IDE and stored together with the script, at the time it is saved (we call the location in the file system bundle-path). Behind the curtain, the images itself are specified simply by using a string containing the file name (path to an image file).

class `Region`

`find`(*PS*)

Parameters **`PS`** – a `Pattern` object or a string (path to an image file or just plain text)

Returns a `Match` object that contains the best match or fails if *not found*

Find a particular GUI element, which is seen as the given image or just plain text. The given file name of an image specifies the element's appearance. It searches within the region and returns the best match, which shows a similarity greater than the minimum similarity given by the pattern. If no similarity was set for the pattern by `Pattern.similar()` before, a default minimum similarity of 0.7 is set automatically.

If autoWaitTimeout is set to a non-zero value, find() just acts as a wait().

Side Effect *lastMatch*: the best match can be accessed using `Region.getLastMatch()` afterwards.

`findAll`(*PS*)

Parameters **`PS`** – a `Pattern` object or a string (path to an image file or just plain text)

Returns one or more `Match` objects as an iterator object or fails if *not found*

How to iterate through is *documented here*.

Repeatedly find ALL instances of a pattern, until no match can be found anymore, that meets the requirements for a single `Region.find()` with the specified pattern.

By default, the returned matches are sorted by the similiarty. If you need them ordered by their positions, say the Y coordinates, you have to use Python's sorted function. Here is a example of sorting the matches from top to bottom.

Side Effect *lastMatches*: a reference to the returned iterator object containing the found matches is stored with the region that was searched. It can be accessed using getLastMatches() afterwards. How to iterate through an iterator of matches is *documented here*.

`wait`(*[PS][, seconds]*)

Parameters

- **`PS`** – a `Pattern` object or a string (path to an image file or just plain text)
- **`seconds`** – a number, which can have a fraction, as maximum waiting time in seconds. The internal granularity is milliseconds. If not specified, the auto wait timeout value set

by `Region.setAutoWaitTimeout()` is used. Use the constant *FOREVER* to wait for an infinite time.

Returns a `Match` object that contains the best match or fails if *not found*

If *PS* is not specified, the script just pauses for the specified amount of time. It is still possible to use `sleep(seconds)` instead, but this is deprecated.

If *PS* is specified, it keeps searching the given pattern in the region until the image appears (would have been found with `Region.find()`) or the specified amount of time has elapsed. At least one find operation is performed, even if 0 seconds is specified.)

Side Effect *lastMatch*: the best match can be accessed using `Region.getLastMatch()` afterwards.

Note: You may adjust the scan rate (how often a search during the wait takes place) by setting `Settings.WaitScanRate` appropriately.

waitVanish (*PS*[, *seconds*])

Wait until the give pattern *PS* in the region vanishes.

Parameters

- **PS** – a `Pattern` object or a string (path to an image file or just plain text)
- **seconds** – a number, which can have a fraction, as maximum waiting time in seconds. The internal granularity is milliseconds. If not specified, the auto wait timeout value set by `Region.setAutoWaitTimeout()` is used. Use the constant *FOREVER* to wait for an infinite time.

Returns *True* if the pattern vanishes within the specified waiting time, or *False* if the pattern stays visible after the waiting time has elapsed.

This method keeps searching the given pattern in the region until the image vanishes (can not be found with `Region.find()` any longer) or the specified amount of time has elapsed. At least one find operation is performed, even if 0 seconds is specified.

Note: You may adjust the scan rate (how often a search during the wait takes place) by setting `Settings.WaitScanRate` appropriately.

exists (*PS*[, *seconds*])

Check whether the give pattern is visible on the screen.

Parameters

- **PS** – a `Pattern` object or a string (path to an image file or just plain text)
- **seconds** – a number, which can have a fraction, as maximum waiting time in seconds. The internal granularity is milliseconds. If not specified, the auto wait timeout value set by `Region.setAutoWaitTimeout()` is used. Use the constant *FOREVER* to wait for an infinite time.

Returns a `Match` object that contains the best match. None is returned, if nothing is found within the specified waiting time

Does exactly the same as `Region.wait()`, but no exception is raised in case of FindFailed. So it can be used to symplify scripting in case that you only want to know wether something is there or not to decide how to proceed in your workflow. So it is typically used with an if statement. At least one find operation is performed, even if 0 seconds is specified. So specifying 0 seconds saves some time, in case there is no need to wait, since its your intention to get the information "not found" directly.

Side Effect *lastMatch*: the best match can be accessed using `Region.getLastMatch()` afterwards.

Note: You may adjust the scan rate (how often a search during the wait takes place) by setting `Settings.WaitScanRate` appropriately.

Observing Visual Events in a Region

This feature is completetely revised in version 1.1.x

> **Note** Some features have a changed behavior, are no longer available or differ in usage compared to prior versions. This break of downward compatibility is by intention, since the complexity of changes could not be hidden. In some cases it forces the revision of scripts, that use the observe feature and are run with version 1.1.x. Watch the notes with a specific feature that changed.
>
> **Main areas of change:**
>
> - onAppear, onVanish are stopped after first event - use repeat in handler
> - observe in background now is observeInBackground()
> - SikuliEvent now is ObserveEvent and uses getters instead of direct access to attributes

You can tell a region to observe that something appears or vanishes, or something changes in that region. Using the methods *Region.onAppear()*, *Region.onVanish()* and *Region.onChange()*, you register an event to be observed, while the observation is running for that Region. The observation in a Region is started using *Region.observe()* and stopped again using *Region.stopObserver()*.

Each Region can have exactly one observer. For each observer, you can register as many events as needed. So you can think of it as grouping some `wait()` and `waitVanish()` together and have them processed simultanouesly, while you are waiting for one of these events to happen.

It is possible to let the script wait for the completion of an observation or let the observation run in background (meaning in parallel), while your script is continuing. With a timing parameter you can tell *Region.observeInBackground()* to stop observation after the given time.

When one of the visual events happens, an event handler (callback function) provided by you is called, handing over a *ObserveEvent* object as a parameter, that contains all relevant information about the event and that has features to act on the events or change the behavior of the observation. During the processing in your handler, the observation is paused until your handler has ended. Information between the main script and your handlers can be given forward and backward using global variables or other appropriate measures.

Another option to handle events, that are observed in the background, is to check the status of the observation inline in your workflow. Each registered event has a unique name, that later can be used, to check, wether it already happened or not. Furthermore you can inactivate registered events, so that they are ignored until activated again (*see: Named Events*).

It's your responsibility to stop the observation. This can either be done by calling *Region.stopObserver()* (in the main workflow or in the handler) or by starting the observation with a timing parameter. All running observations are stopped automatically, when the script or Java program (in fact the JVM) terminates.

Since you can have as many region objects as needed and each region can have one observation active and running, theoretically it is possible to have as many visual events being observed at the same time as needed. But in reality, the number of observations is limited by the system resources available to Sikuli at that time.

Be aware, that every observation is a number of different find operations that are processed repeatedly. So to speed up processing and keep your script acting, you should define a region for observation as small as possible. You may adjust the scan rate (how often a search during the observation takes place) by setting *Settings.ObserveScanRate* appropriately.

PS: as a parameter in the following methods you have to specify a Pattern or a String (path to an image file or just plain text). **handler**: as a parameter in the following methods you have to specify the **name of a function**, which will be called by the observer, in case the observed event happens. The function name (and usually the function itself) has to be defined in your script before using the appropriate functions to register an observe event. The existance of the function will be checked after starting the script, but before running it.

So to get your script running, you have to have at least the following statements in your script:

```
def myHandler(event): # you can choose any valid function name
        # event: can be any variable name, it references the ObserveEvent object
        pass # add your statements here

onAppear("path-to-an-image-file", myHandler) # or any other onXYZ()
observe(10) # observes for 10 seconds
```

Note for Java And this is how you setup a handler in your Java program and run the observation:

```
// one has to combine observed event and its handler
// overriding the appropriate method
someRegion.onAppear("path-to-an-image-file",
        new ObserverCallBack() {
                @Override
                public void appeared(ObserveEvent event) {
                        // here goes your handler code
                }
        }
);
// run observation in foreground for 10 seconds
someRegion.observe(10)
```

Here `ObserverCallBack` is a class defining the possible callback funtions `appeared`, `vanished` and `changed` as well as `findfailed` and `missing` as noop-methods, that have to be overwritten as needed in your implementation of the ObserverCallBack. You only need to overwrite the one method, that corresponds to your event.

Read *`ObserveEvent`* to know what is contained in the event object and what its features are.

NOTE ON CONCURRENCY with ObserveInBackground, the callback concept and Mouse/Keyboard usage In Sikuli version prior to 1.1.0 it could happen, that mouse actions in the handler callback could interfere with mouse actions in the main workflow or other callback handlers, since these threads work in parallel without any automatic synchronization.

Beginning with 1.1.0 mouse actions like click are safe in the way, that they always are completed, before any other click operation can be started (internally handled like a transaction).

So parallel clicks in main workflow and handler should do their job correctly, but might be run in a sequence, that cannot be foreseen. Look here, if you want to have more control over mouse and keyboard usage in parallel processes.

class `Region`

`onAppear`(*PS*, *handler*)

With the given region you register an `APPEAR` event, whose pattern/image/text is looked for to be there or to appaear while running an observation with the next call of `observe()`. In the moment the observation is successful for that event, your registered handler is called and the observation is paused until you return from your handler.

With the first appearence, the observation for this event is terminated. If you want the observation for this event to be continued, you have to use `ObserveEvent.repeat()` before leaving the handler.

Parameters

- **`PS`** – a *`Pattern`* object or a string (path to an image file or just plain text)
- **`handler`** – the name of a handler function in the script

Returns a string as unique name of this event *to identify this event later*

onVanish (*PS*, *handler*)

With the given region you register a `VANISH` event, whose pattern/image/text is looked for to not be there or to vanish while running an observation with the next call of `observe()`. In the moment the observation is successful for that event, your registered handler is called and the observation is paused until you return from your handler.

With the first vanishing, the observation for this event is terminated. If you want the observation for this event to be continued, you have to use `ObserveEvent.repeat()` before leaving the handler.

Parameters

- **PS** – a *Pattern* object or a string (path to an image file or just plain text.
- **handler** – the name of a handler function in the script

Returns a string as unique name of this event *to identify this event later*

onChange ([*minChangedSize*], *handler*)

With the given region you register a `CHANGE` event. While running an observation with the next call of `observe()`, it is looked for changes in that region. A change is, if some non-overlapping rectengular areas of the given minimum size changes its pixel content from one observation step to the next. In the moment the observation is successful for that event, your registered handler is called and the observation is paused until you return from your handler.

Parameters

- **minChangedSize** – the minimum size in pixels of a change to trigger a change event (see *Settings.ObserveMinChangedPixels*, default 50).
- **handler** – the name of a handler function in the script

Returns a string as unique name of this event *to identify this event later*

Here is a example that highlights all changes in an observed region.

```
def changed(event):
        print "something changed in ", event.region
        for ch in event.getChanges():
                ch.highlight() # highlight all changes
        wait(1)
        for ch in event.getChanges():
                ch.highlight() # turn off the highlights

r = selectRegion("select a region to observe")
# any change in r larger than 50 pixels would trigger the changed function
r.onChange(50, changed)
# another way to observe for 30 seconds
r.observeInBackground(); wait(30)
r.stopObserver()
```

observe ([*seconds*])

Begin observation within the region. The script waits for the completion of the observation (meaning until the observation is stopped by intention or timed out).

Parameters **seconds** – a number, which can have a fraction, as maximum observation time in seconds. Omit it or use the constant FOREVER to tell the observation to run for an infinite time (or until stopped by `stopObserve()`).

Returns True, if the observation could be started, False otherwise

For each region object, only one observation can be running at a given time, meaning, that a call to observe(), while an observe for that region is running, is ignored with an error message, returning False.

Note: You may adjust the scan rate (how often a search during the observation takes place) by setting `Settings.ObserveScanRate` appropriately.

observeInBackground([*seconds*])
The observation is run in the background, meaning that the observation will be run in a subthread and processing of your script is continued immediately.

Take care, that your script continues with some time consuming stuff. Additionally *Named Events* might be of interest.

The over all behavior and the features are the same as `Region.observe()`.

observe([*seconds*], *background* = *True*)
DEPRECATED (will not be in version 2+) Only available in Python scripts for some limited backward compatibility, with the impact, that the Region object *must* be a Python level Region. In case you have to cast a Java level Region using `Region(someRegion)`.

It is strongly recommended to revise your scripts using this observe feature as soon as possible.

stopObserver()
Stop observation for this region.

The source region of an observed visual event is available from the *event* that is passed as parameter to the handler function.

Additionally there is a convenience feature to stop observation within a handler function: simply call `event.stopObserver()` inside the handler function.:

```
def myHandler(event):
        event.stopObserver() # stops the observation
        # instead of
        # event.getRegion().stopObserver()

onAppear("path-to-an-image-file", myHandler)
observe(FOREVER) # observes until stopped in handler
```

class ObserveEvent
When processing an *observation in a region*, a *handler function is called*, when one of your registered events `Region.onAppear()`, `Region.onVanish()` or `Region.onChange()` happen.

The one parameter, you have access to in your handler function is an instance of `ObserveEvent`. You have access to the following features of the event, that might help to identify the cause of the event, act on the resulting matches and optionally modify the behavior of the observation.

New in version X1.1.0: **Note on versions prior to 1.1.0** The event class was `SikuliEvent` and it allowed to directly access the attributes like type, match, region, ... This class no longer exists and its follow up is the class `ObserveEvent`. This break of downward compatibility is by intention, to force the revision of scripts, that use the observe feature and are run with version 1.1.0+

New in version X1.1.1.

The feature `FindFailed` and/or `ImageMissing` handler allows to specify functions that are visited in the case of these failures happening. The handler gets an ObserveEvent object as parameter, that can be used to analyse the situation and define how the situation should be handled finally (for details see: *comments on FindFailed / ImageMissing*)

getType()
get the type of the event

Returns a string containing APPEAR, VANISH, CHANGE, GENERIC, FINDFAILED, MISSING

isAppear(), *isVanish()*, *isChange()*, *isGeneric()*, *isFindFailed()*, *isMissing()*
convenience methods, to check the type

Returns True or False

getRegion()
The observing region of this event.

Returns the region object

getPattern()
Get the pattern that triggered this event. A given image is packed into the pattern. This is only valid for APPEAR and VANISH events as well as for FINDFAILED and MISSING.

Returns the pattern object (which allows to access the given image if needed)

getImage()
Directly access the given image in case of FINDFAILED and MISSING.

Returns the image object

getMatch()
For an APPEAR you get the *Match* object that appeared in the observed region (same as with `wait()`).

For a VANISH event, you get the last *Match* object that was found in the observed region before it vanished.

This method is not valid in a CHANGE event.

Returns the match object

getChanges()
Get a list of *Match* objects that represent the rectengular areas that changed their content. Their sizes are at least *minChangedSize* pixels.

This attribute is valid only in a CHANGE event.

Returns an unsorted list of match objects

getCount()
Get the count how often the handler was visited.

Returns the count as number

getTime()
Get the time, when the event happened.

Returns a long integer value according to the Java feature `new Date().getTime()`

repeat([*waitTime*])
Specify the time in seconds, that the observation of this event should pause after returning from the handler.

Remember APPEAR and VANISH events are stopped after the first occurence. You have to use an approriate `repeat()`, to continue the observation.

Parameters **waitTime** – seconds to pause, taken as 0 if not given

getResponse()
In case of FINDFAILED or MISSING get the current setting of the FindFailedResponse of the event region

Returns PROMPT, RETRY, SKIP or ABORT

setResponse(*response*)
In case of FINDFAILED or MISSING set the FindFailedResponse of the event region. This will be the option, that is used after return from the handler for the final reaction.

Parameters **response** – PROMPT, RETRY, SKIP or ABORT

getName()
Get the unique name of this event for use with the appropriate features (see *working with named events*)

Returns a string containing the name

stopObserver()
Stop observation for this region (shortcut for `event.getRegion().stopObserver()`).

Working with named observe events

Additionally to the callback-concept of the observation feature, it is possible, to start one or more observations in background, having registered events without handlers. When these events happen, the event is stored in a list and its observation is paused until the event is taken from the list. Both concepts can be combined per observation.

Events without handlers are registered by omitting the handler parameter in the methods *Region.onAppear()*, *Region.onVanish()* and *Region.onChange()* and storing the returned name for later use.

After having started the observation the usual way using *Region.observe()*, you can check, wether any events have happened until now, you can access the events using their name or get a list of all events that happened until now. With the events themselves you can work exactly like in the handler concept (see: *ObserveEvent*).

The following methods are bound to the region under observation.

class Region

hasObserver()
Check wether at least one event is registered for this region. The observation might be running or not.

Returns True or False

isObserving()
Check wether currently an observation is running for that region

Returns True or False

hasEvents()
Check wether any events have happened for that region

Returns True or False

getEvents()
Get the events, that have happened until this moment. The events are purged from the internal event list.

Returns a list of *ObserveEvent* (might be empty)

getEvent(*name*)
Get the named event and purge it from the internal event list

Parameters **name** – the name of the event (string)

Returns the named event or None/null if it is not on the internal event list

setInactive(*name*)
The named event is paused during the running observation until activated again or the observation is restarted.

Parameters **name** – the name of the event (string)

setActive(*name*)
The named event is activated, so it is observed during the running observation.

Parameters **name** – the name of the event (string)

Acting on a Region

Besides *finding visual objects* on the screen, acting on these elements is one of the kernel operations of Sikuli. Mouse actions can be simulated as well as pressing keys on a keyboard.

The place on the screen, that should be acted on (in the end just one specific pixel, the click point), can be given either as a `pattern` like with the find operations or by directly referencing a pixel `location` or as center of a `region` object (`match` or `screen` also) or the target offset location connected with a `match`. Since all these choices can be used with all action methods as needed, they are abbreviated and called like this:

PSMRL: which means, that either a `Pattern` object or a string (path to an image file or just plain text) or a `Match` or a `Region` or a `Location` can be used as parameter, in detail:

- **P**: *pattern*: a `Pattern` object. An implicit find operation is processed first. If successful, the center of the resulting matches rectangle is the click point. If the pattern object has a target offset specified, this is used as click point instead.
- **S**: *string*: a path to an image file or just plain text. An implicit find operation with the default minimum similarity 0.7 is processed first. If successful, the center of the resulting match object is the click point.
- **M**: *match:* a `match` object from a previous find operation. If the match has a target specified it is used as the click point, otherwise the center of the match's rectangle.
- **R**: *region:* a `region` object whose center is used as click point.
- **L**: *location:* a `location` object which by definition represents a point on the screen that is used as click point.

It is possible to simulate pressing the so called *key modifiers* together with the mouse operation or when simulating keyboard typing. The respective parameter is given by one or more predefined constants. If more than one modifier is necessary, they are combined by using "+" or "|".

Normally all these region methods are used as `reg.click(PS)`, where reg is a region object. If written as `click(PS)` the implicit find is done on the default screen being the implicit region in this case (see: *SCREEN as Default Region*). But using `reg.click(PS)` will restrict the search to the region's rectangle and speed up processing, if region is significantly smaller than the whole screen.

Generally all aspects of *find operations* and of `Region.find()` apply.

If the find operation was successful, the match that was acted on, can be recalled using `Region.getLastMatch()`.

As a default, if the **visual object (image or text) cannot be found**, Sikuli will stop the script by raising an *Exception FindFailed* (details see: *not found*).

Note on IDE: Capturing is a tool in the IDE, to quickly set up images to search for. These images are named automatically by the IDE and stored together with the script, at the time it is saved (we call the location in the file system bundle-path). Behind the curtain the images itself are specified by using a string containing the file name (path to an image file).

Note: If you need to implement more sophisticated mouse and keyboard actions look at *Low Level Mouse and Keyboard Actions*.

Note: In case of having more than one Monitor active, refer to *Multi Monitor Environments* for more details.

Note on Mac: it might be necessary, to use `switchApp()` before, to prepare the application for accepting the action.

class Region

click (*PSMRL*[, *modifiers*])
Perform a mouse click on the click point using the left button.

Parameters

- **PSMRL** – a pattern, a string, a match, a region or a location that evaluates to a click point.
- **modifiers** – one or more key modifiers

Returns the number of performed clicks (actually 1). A 0 (integer null) means that because of some reason, no click could be performed (in case of *PS* may be *not Found*).

Side Effect if *PS* was used, the match can be accessed using `Region.getLastMatch()` afterwards.

Example:

doubleClick (*PSMRL*[, *modifiers*])
Perform a mouse double-click on the click point using the left button.

Parameters

- **PSMRL** – a pattern, a string, a match, a region or a location that evaluates to a click point.
- **modifiers** – one or more key modifiers

Returns the number of performed double-clicks (actually 1). A 0 (integer null) means that because of some reason, no click could be performed (in case of *PS* may be *not Found*).

Side Effect if *PS* was used, the match can be accessed using `Region.getLastMatch()` afterwards.

rightClick (*PSMRL*[, *modifiers*])
Perform a mouse click on the click point using the right button.

Parameters

- **PSMRL** – a pattern, a string, a match, a region or a location that evaluates to a click point.
- **modifiers** – one or more key modifiers

Returns the number of performed right clicks (actually 1). A 0 (integer null) means that because of some reason, no click could be performed (in case of *PS* may be *not Found*).

Side Effect if *PS* was used, the match can be accessed using `Region.getLastMatch()` afterwards.

highlight ()
Highlight the region, showing a red colored frame around it, until the effect is stopped by another parameterless highlight call with the same region. The script continues after switching highlight on with the first call.

highlight (*color*)

Parameters **color** – see **Note** below

Highlight the region, showing a frame with the given color around it, until the effect is stopped by another parameterless highlight call with the same region. The script continues after switching highlight on with the first call.

highlight (*seconds*)
Highlight the region for the given time in seconds, showing a red colored frame around it.

Parameters **seconds** – a decimal number taken as duration in seconds

The region is highlighted showing the frame around it for the given time, while the script is suspended for this time.

highlight (*seconds*, *color*)
Highlight the region for the given time in seconds, showing a frame with the given color around it.

Parameters

- **seconds** – a decimal number taken as duration in seconds
- **color** – see **Note** below

The region is highlighted showing a frame with the given color around it for the given time, while the script is suspended for the this time.

Note on parameter color: There are these options to specify the color

- name of a color as string. The following names are accepted: black, blue, cyan, gray, green, magenta, orange, pink, red, white, yellow (lowercase and uppercase can be mixed, internally transformed to all uppercase)
- the following color names exactly so: lightGray, LIGHT_GRAY, darkGray, DARK_GRAY
- a string containing a hex value like in HTML: #XXXXXX (6 hex digits) specifying an RGB value
- a string containing digits #rrrgggbbb, where rrr, ggg, bbb are integer values in range 0 - 255 padded with leading zeros if needed (hence exactly 9 digits) and so specifying an RGB value

Example:

```
m = find(some_image)

# the red frame will blink for about 7 - 8 seconds
for i in range(5):
        m.highlight(1)
        wait(0.5)

# a second red frame will blink as an overlay to the first one
m.highlight()
for i in range(5):
        m.highlight(1)
        wait(0.5)
m.highlight()

# the red frame will grow 5 times
for i in range(5):
        m.highlight(1)
        m = m.nearby(20)
```

Note: The coloured frame is just an overlay in front of all other screen content and stays in its place, independently from the behavior of this other content, which means it is not "connected" to the *content* of the defining region. But it will be adjusted automatically, if you change position and/or dimension of this region in your script, while it is highlighted.

Note: Due to the implementation of this function, the target application might loose focus and opened menus or lists get closed again. In other cases the highlight frame is not or not completely visible (not getting to the front). In these cases the highlight feature cannot be used for tracking the search results.

A possible workaround is to use hover(), to move the mouse over the match or even use a function like this:

```
def hoverHighlight(reg, loop = 1):
  for n in range(loop):
    hover(reg.getTopLeft())
    hover(reg.getTopRight())
```

```
hover(reg.getBottomRight())
hover(reg.getBottomLeft())
hover(reg.getTopLeft())
```

Using this function instead of highlight will let the mousepointer visit the corners of the given region clockwise, starting and stopping top left. With the standard move delay of 0.5 seconds this will last about 2 seconds for one loop (second parameter, default 1).

hover (*PSMRL*)
Move the mouse cursor to hover above a click point.

Parameters

- **PSMRL** – a pattern, a string, a match, a region or a location that evaluates to a click point.
- **modifiers** – one or more key modifiers

Returns the number 1 if the mousepointer could be moved to the click point. A 0 (integer null) returned means that because of some reason, no move could be performed (in case of *PS* may be *not Found*).

Side Effect if *PS* was used, the match can be accessed using *Region.getLastMatch()* afterwards.

dragDrop (*PSMRL*, *PSMRL*[, *modifiers*])
Perform a drag-and-drop operation from a starting click point to the target click point indicated by the two PSMRLs respectively.

Parameters

- **PSMRL** – a pattern, a string, a match, a region or a location that evaluates to a click point.
- **modifiers** – one or more key modifiers

If one of the parameters is *PS*, the operation might fail due to *not Found*.

Sideeffect: when using *PS*, the match of the target can be accessed using *Region.getLastMatch()* afterwards. If only the first parameter is given as *PS*, this match is returned by *Region.getLastMatch()*.

If the operation does not perform as expected (usually caused by timing problems due to delayed reactions of applications), you may adjust the internal timing parameters *Settings.DelayBeforeMouseDown*, *Settings.DelayBeforeDrag* and *Settings.DelayBeforeDrop* (default value is 0.3 seconds) for the next action (timing is reset to default after the operation is completed). In case this might be combined with the internal timing parameter *Settings.MoveMouseDelay*.

Note: If you need to implement more sophisticated mouse and keyboard actions look at *Low Level Mouse and Keyboard Actions*.

drag (*PSMRL*)
Start a drag-and-drop operation by starting the drag at the given click point.

Parameters **PSMRL** – a pattern, a string, a match, a region or a location that evaluates to a click point.

Returns the number 1 if the operation could be performed. A 0 (integer null) returned means that because of some reason, no move could be performed (in case of *PS* may be *not Found*).

The mousepointer is moved to the click point and the left mouse button is pressed and held, until the button is released by a subsequent mouse action. (e.g. a *Region.dropAt()* afterwards).

If the operation does not perform as expected (usually caused by timing problems due to delayed reactions of applications), you may adjust the internal timing parameters *Settings.DelayBeforeMouseDown*, and *Settings.DelayBeforeDrag* (default value is 0.3 seconds) for the next action (timing is reset to default after the operation is completed).

Side Effect if *PS* was used, the match can be accessed using *Region.getLastMatch()* afterwards.

dropAt (*PSMRL*[, *delay*])

Complete a drag-and-drop operation by dropping a previously dragged item at the given target click point.

Parameters **PSMRL** – a pattern, a string, a match, a region or a location that evaluates to a click point.

Returns the number 1 if the operation could be performed. A 0 (integer null) returned means that because of some reason, no move could be performed (in case of *PS* may be *not Found*).

The mousepointer is moved to the click point and the left mouse button is released. If it is necessary to visit one ore more click points after dragging and before dropping, you can use *Region.mouseMove()* or *Region.hover()* inbetween and `dropAt` only for the final destination.

If the operation does not perform as expected (usually caused by timing problems due to delayed reactions of applications), you may adjust the internal timing parameter *Settings.DelayBeforeDrop* (default value is 0.3 seconds) for the next action (timing is reset to default after the operation is completed).

Side Effect if *PS* was used, the match can be accessed using *Region.getLastMatch()* afterwards.

type ([*PSMRL*], *text*[, *modifiers*])

Type the text at the current focused input field or at a click point specified by *PSMRL*.

Parameters

- **PSMRL** – a pattern, a string, a match, a region or a location that evaluates to a click point.
- **modifiers** – one or more modifier keys (*Class Key*)

Returns the number 1 if the operation could be performed, otherwise 0 (integer null), which means, that because of some reason, it was not possible or the click could be performed (in case of *PS* may be *not Found*).

This method simulates keyboard typing interpreting the characters of text based on the layout/keymap of the **standard US keyboard (QWERTY)**.

Special keys (ENTER, TAB, BACKSPACE, ...) can be incorporated into text using the constants defined in *Class Key* using the standard string concatenation +.

If *PSMRL* is given, a click on the clickpoint is performed before typing, to gain the focus. (Mac: it might be necessary, to use *switchApp()* to give focus to a target application before, to accept typed/pasted characters.)

If *PSMRL* is omitted, it performs the typing on the current focused visual component (normally an input field or an menu entry that can be selected by typing something).

Side Effect if *PS* was used, the match can be accessed using *Region.getLastMatch()* afterwards.

Note: If you need to type international characters or you are using layouts/keymaps other than US-QWERTY, you should use *Region.paste()* instead. Since type() is rather slow because it simulates each key press, for longer text it is preferrable to use *Region.paste()*.

Best Practice: As a general guideline, the best choice is to use `paste()` for readable text and `type()` for action keys like TAB, ENTER, ESC, ... Use one `type()` for each key or key combination and be aware, that in some cases a short `wait()` after a `type()` might be necessary to give the target application some time to react and be prepared for the next Sikuli action.

SPECIAL macOS Sierra 10.12+ If type does not behave as expected (characters like e or s are not typed) then look here for explanation and workaround.

paste ([*PSMRL*], *text*)
Paste the text at a click point.

Parameters

- **PSMRL** – a pattern, a string, a match, a region or a location that evaluates to a click point.
- **modifiers** – one or more key modifiers

Returns the number 1 if the operation could be performed, otherwise 0 (integer null), which means, that because of some reason, it was not possible or the click could be performed (in case of *PS* may be *not Found*).

Pastes *text* using the clipboard (OS-level shortcut (Ctrl-V or Cmd-V)). So afterwards your clipboard contains *text*. `paste()` is a temporary solution for typing international characters or typing on keyboard layouts other than US-QWERTY.

If *PSMRL* is given, a click on the clickpoint is performed before typing, to gain the focus. (Mac: it might be necessary, to use *`switchApp()`* to give focus to a target application before, to accept typed/pasted characters.)

If *PSMRL* is omitted, it performs the paste on the current focused component (normally an input field).

Side Effect if *PS* was used, the match can be accessed using *`Region.getLastMatch()`* afterwards.

Note: Special keys (ENTER, TAB, BACKSPACE, ...) cannot be used with `paste()`. If needed, you have to split your complete text into two or more `paste()` and use `type()` for typing the special keys inbetween. Characters like \n (enter/new line) and \t (tab) should work as expected with `paste()`, but be aware of timing problems, when using e.g. intervening \t to jump to the next input field of a form.

Extracting Text from a Region

class Region

text()
Extract the text contained in the region using OCR.

Returns the text as a string. Multiple lines of text are separated by intervening 'n'.

Note: Since this feature is still in an **experimental state**, be aware, that in some cases it might not work as expected. If you face any problems look at the Questions & Answers / FAQ's and the Bugs.

Low-level Mouse and Keyboard Actions

class Region

mouseDown (*button*)
Press the mouse *button* down.

Parameters **button** – one or a combination of the button constants `Button.LEFT`, `Button.MIDDLE`, `Button.RIGHT`.

Returns the number 1 if the operation is performed successfully, and zero if otherwise.

The mouse button or buttons specified by *button* are pressed until another mouse action is performed.

mouseUp ([*button*])
Release the mouse button previously pressed.

Parameters **button** – one or a combination of the button constants `Button.LEFT`, `Button.MIDDLE`, `Button.RIGHT`.

Returns the number 1 if the operation is performed successfully, and zero if otherwise.

The button specified by *button* is released. If *button* is omitted, all currently pressed buttons are released.

mouseMove (*PSRML*)
Move the mouse pointer to a location indicated by PSRML.

Parameters **PSMRL** – a pattern, a string, a match, a region or a location that evaluates to a click point.

Returns the number 1 if the operation could be performed. If using *PS* (which invokes an implicity find operation), find fails and you have switched off FindFailed exception, a 0 (integer null) is returned. Otherwise, the script is stopped with a FindFailed exception.

Sideeffects: when using *PS*, the match can be accessed using *Region.getLastMatch()* afterwards

mouseMove (*xoff*, *yoff*)
Move the mouse pointer from it's current position to the position given by the offset values (<0 left, up >0 right, down)

Parameters

- **xoff** – horizontal offset
- **yoff** – vertical offset

Returns 1 if possible, 0 otherwise

wheel (*PSRML*, *direction*, *steps*)
Move the mouse pointer to a location indicated by PSRML and turn the mouse wheel in the specified direction by the specified number of steps.

Parameters

- **PSMRL** – a pattern, a string, a match, a region or a location that evaluates to a click point.
- **direction** – one of the button constants `Button.WHEEL_DOWN` or `Button.WHEEL_UP` denoting the wheeling direction.
- **steps** – an integer indicating the amoung of wheeling.

Sideeffects: when using *PS*, the match can be accessed using *Region.getLastMatch()* afterwards

keyDown (*key | list-of-keys*)
Press and hold the specified key(s) until released by a later call to *Region.keyUp()*.

Parameters **key|list-of-keys** – one or more keys (use the consts of class Key). A list of keys is a concatenation of several key constants using "+".

Returns the number 1 if the operation could be performed and 0 if otherwise.

keyUp ([*key | list-of-keys*])
Release given keys. If no key is given, all currently pressed keys are released.

Parameters **key|list-of-keys** – one or more keys (use the consts of class Key). A list of keys is a concatenation of several key constants using "+".

Returns the number 1 if the operation could be performed and 0 if otherwise.

Exception FindFailed

As a default, find operations (*explicit* and *implicit*) when not successful raise an `Exception FindFailed`, that will stop the script immediately.

To implement some checkpoints, where you want to asure your workflow, use *`Region.exists()`*, that reports a not found situation without raising FindFailed (returns False instead).

To run all or only parts of your script without FindFailed exceptions to be raised, use *`Region.setThrowException()`* or *`Region.setFindFailedResponse()`* to switch it on and off as needed.

For more sophisticated concepts, you can implement your own exception handling using the standard Python construct `try: ... except: ...`.

New in version X1.1.1.

Generally a FindFailed situation is also signalled (besides that the image could not befound on the screen), if the image could not be found on the current image path and hence could not be loaded for the find process.

New in version X1.1.1.

To implement even more sophisticated concepts, it is possible to declare **handler functions**, that are visited in case of a FindFailed and/or ImageMissing situations and *allow to take corrective actions*. Before leaving the handler you can specify how the case should finally be handled (ABORT, SKIP, RETRY or PROMPT). If specified, a handler is always visited before any other action is taken. Handlers can be specified for a single Region object and/or globally with class FindFailed, so that each new Region object afterwards would call this handler in case.

The PROMPT response now allows to recapture the image on the fly or just to capture an image, that is not loadable.

New in version X1.0-rc2.

These are the possibilities to handle "not found" situations:

- **generally abort a script, if not handled with `try: ... except: ...`** (the default setting or using *`setThrowException(True)`* or *`setFindFailedResponse(ABORT)`*)
- **generally ignore all "not found" situations** (using *`setThrowException(False)`* or *`setFindFailedResponse(SKIP)`*),
- **want to be prompted in such a case** (using *`setFindFailedResponse(PROMPT)`*)
- **advise Sikuli to wait forever (be careful with that!)** (using *`setFindFailedResponse(RETRY)`*)

  New in version X1.1.1:

- **advise Sikuli to visit the specified handler before taking any other action** (using *`setFindFailedHandler(handler)`*)

Comment on using PROMPT:

This feature is helpful in following situations:

- you are developing something, that needs an application with it's windows to be in place, but this workflow you want to script later. If it comes to that point, you get the prompt, arrange the app and click *Retry*. Your workflow should continue.
- you have a workflow, where the user might do some corrective actions, if you get a FindFailed
- guess you find more ;-)

In case of a FindFailed, you get the following prompt:

New in version X1.1.1.

Clicking *Retry* would again try to find the image. *Capture/Skip* would allow to (re)capture the image and *Abort* would end the script. In case of clicking *Capture* you get another similar prompt, that allows you to either do the capture, finally skip or advise SikuliX to abort the script immediately.

Examples: 4 solutions for a case, where you want to decide how to proceed in a workflow based on the fact that a specific image can be found. (**pass** is the python statement, that does nothing, but maintains indentation to form the blocks):

```
# --- nice and easy
if exists("path-to-image"): # no exception, returns None when not found
        pass # it is there
else:
        pass # we miss it

# --- using exception handling
# every not found in the try block will switch to the except block
try:
        find("path-to-image")
        pass # it is there
except FindFailed:
        pass # we miss it

# --- using setFindFailedResponse
setFindFailedResponse(SKIP) # no exception raised, not found returns None
if find("path-to-image"):
        setFindFailedResponse(ABORT) # reset to default
        pass # it is there
else:
        setFindFailedResponse(ABORT) # reset to default
        pass # we miss it

# --- using setThrowException
setThrowException(False) # no exception raised, not found returns None
if find("path-to-image"):
        setThrowException(True) # reset to default
        pass # it is there
else:
        setThrowException(True) # reset to default
```

```
        pass # we miss it
```

Comment on using a handler function:

setFindFailedHandler (*functionname*)

> **Parameters** **functionname** – the name of a function, that should handle FindFailed situations (no apostrophes!)

setImageMissingHandler (*functionname*)

> **Parameters** **functionname** – the name of a function, that should handle ImageMissing situations (no apostrophes!)

To specify the respective handlers globally for all new Regions use `FindFailed.setFindFailedHandler` (or as a shortcut for that: `FindFailed.setHandler`) and `FindFailed.setImageMissingHandler` respectively.

Both methods might name the same handler function, since it is possible to differentiate the situation to handle by inspecting the type of the event, that is the parameter, when the handler is called. On the other hand with 2 handlers it is easier and more transparent to handle both situations completely different.

This is a basic handler:

```
def handler(event):
    print "handler entered for", event.getType()
    # type here might be FINDFAILED or MISSING
    # do something
    event.setResponse(PROMPT) # now go back and prompt the user
    # or use RETRY, SKIP or ABORT
```

For more information on the possibilities in a handler see `ObserveEvent`.

Note for Java And this is how you setup a handler in your Java program:

```
someRegion.setFindFailedHandler(new ObserveCallback() {
        @Override
        public void findfailed(ObserveEvent event) {
                // here goes your handler code
        }
});
```

... and to set globally:

```
FindFailed.setFindFailedHandler(new ObserveCallback() {
        @Override
        public void findfailed(ObserveEvent event) {
                // here goes your handler code
        }
});
```

... for the image missing situation combine `setImageMissingHandler` with overriding `missing`.

class Region

Reminder If used without specifying a region, the default/primary screen (default region SCREEN) is used.

New in version X1.0-rc2.

setFindFailedResponse (*ABORT* | *SKIP* | *PROMPT* | *RETRY*)

For the specified region set the option, how Sikuli should handle "not found" situations. The option stays in effect until changed by another `setFindFailedResponse()`.

Parameters

- **ABORT** – all subsequent find failed operations (explicit or implicit) will raise exception FindFailed (which is the default when a script is started).
- **SKIP** – all subsequent find operations will not raise exception FindFailed. Instead, explicit find operations such as *Region.find()* will return *None*. Implicit find operations (action functions) such as *Region.click()* will do nothing and return 0.
- **PROMPT** – all subsequent find operations will not raise exception FindFailed. Instead you will be prompted for the way to handle the situation (see *using PROMPT*). Only the current find operation is affected by your response to the prompt.
- **RETRY** – all subsequent find operations will not raise exception FindFailed. Instead, Sikuli will try to find the target until it gets visible. This is equivalent to using `wait( ... , FOREVER)` instead of `find()` or using setAutoWaitTimeout(FOREVER).

New in version X1.1.1.

setFindFailedHandler(*handler*)
For all subsequent find failed operations (explicit or implicit) the specified handler should be visited in case of FindFailed or image not loadable. (see *using a FindFailed handler*)

Parameters **handler** – the name of the handler function that should be visited.

getFindFailedResponse()
Get the current setting in this region.

Returns ABORT or SKIP or PROMPT or RETRY

Usage:

```
val = getFindFailedResponse()
if val == ABORT:
        print "not found will stop script with Exception FindFailed"
elif val == SKIP:
        print "not found will be ignored"
elif val == PROMPT:
        print "when not found you will be prompted"
elif val == RETRY:
        print "we will always wait forever"
else:
        print val, ": this is a bug :-("
```

Note: It is recommended to use `set/getFindFailedResponse()` instead of `set/getThrowException()` since the latter ones might be deprecated in the future.

setThrowException(*False* | *True*)
By using this method you control, how Sikuli should handle not found situations in this region.

Parameters

- **True** – all subsequent find operations (explicit or implicit) will raise exception FindFailed (which is the default when a script is started) in case of not found.
- **False** – all subsequent find operations will not raise exception FindFailed. Instead, explicit find operations such as *Region.find()* will return *None*. Implicit find operations (action functions) such as *Region.click()* will do nothing and return 0.

getThrowException()

Returns `True` or `False`

Get the current setting as True or False (after start of script, this is True by default) in this region.

Grouping Method Calls (with Region:)

Instead of:

```
# reg is a region object
if not reg.exists(image1):
        reg.click(image2)
        reg.wait(image3, 10)
        reg.doubleClick(image4)
```

you can group methods applied to the same region using Python's `with` syntax:

```
# reg is a region object
with reg:
        if not exists(image1):
                click(image2)
        wait(image3, 10)
        doubleClick(image4)
```

All methods inside the `with` block (mind indentation) that have the region omitted are redirected to the region object specified at the `with` statement.

IMPORTANT Usage Note

This only works without problems for region objects created on the scripting level using one of the constructors `Region()`.

Region objects created with Region methods, that return new region objects, might not work though in some cases.

If you get strange results or errors in the with block (e.g. syntax error `__enter__ is not defined for this region`) cast your Region object to a scripting level Region object using

`castedRegion = Region(regionNotWorking)` and use `castedRegion` in the with clause

or do it like this:

```
# reg is a scripting level region object
regNew = reg.left() # returns a non-scripting-level region object
with Region(regNew):
        if not exists(image1):
                click(image2)
        wait(image3, 10)
        doubleClick(image4)
```

Location

This class is there as a convenience, to handle single points on the screen directly by its position (x, y). It is mainly used in the actions on a region, to directly denote the click point. It contains methods, to *move* a point around on the screen.

class `Location`

`Location` (*x*, *y*)

Parameters

- **`x`** – x position

- **y** – y position

Returns a new location object representing the position (x,y) on the screen

getX()
getY()
Get the x or y value of a location object

It is possible to get the values directly by `location.x` or `location.y`. It is also possible to set these values directly by `location.x = value` or `location.y = value`.

Note: `getX()``and ``getY()` currently (versions 0.10.2 and X 1.0rc2) return float values (Java: double), whereas `location.x` and `location.y` return integer values.

setLocation(*x*, *y*)
Set the location of this object to the specified coordinates.

offset(*dx*, *dy*)
Get a new location which is *dx* and *dy* pixels away horizontally and vertically from the current location.

above(*dy*)
Get a new location which is *dy* pixels vertically above the current location.

below(*dy*)
Get a new location which is *dy* pixels vertically below the current location.

left(*dx*)
Get a new location which is *dx* pixels horizontally to the left of the current location.

right(*dx*)
Get a new location which is *dx* pixels horizontally to the right of the current location.

Screen

class Screen

Class Screen is there, to have a representation for a pysical monitor where the capturing process (grabbing a rectangle from a screenshot, to be used for further processing with find operations is implemented. For *Multi Monitor Environments* it contains features to map to the relevant monitor.

Since Screen extends class *Region*, all methods of class Region can be used with a screen object.

Of special interest might be the grouping of region method calls using `with:` in Multi Monitor Environments: use it for other screens, than the default/primary screen, where you have this feature by default.

Be aware, that using the whole screen for find operations may have an impact on performance. So if possible either use `setROI()` or restrict a find operation to a smaller region object (e.g. `reg.find()`) to speed up processing.

Screen: Setting, Getting Attributes and Information

class Screen

Screen([*id*])
Create a new Screen object

Parameters **id** – an integer number indicating which monitor in a multi-monitor environment.

Returns a new screen object.

It creates a new screen object, that represents the default/primary monitor (whose id is 0), if id is omitted. Numbers 1 and higher represent additional monitors that are available at the time, the script is running (read for details).

Using numbers, that do not represent an existing monitor, will stop the script with an error. So you may either use getNumberScreens() or exception handling, to avoid this.

Note: If you want to access the default/primary monitor (Screen(0)) without creating a new screen object, use the constant reference SCREEN, that is initiated when your script starts: SCREEN=Screen(0).

getNumberScreens ( )
Get the number of screens in a multi-monitor environment at the time the script is running

getBounds ( )
Get the dimensions of monitor represented by the screen object.

Returns a rectangle object

The width and height of the rectangle denote the dimensions of the monitor represented by the screen object. These attributes are obtained from the operating system. They can not be modified using Sikuli script.

Screen as (Default) Region

Normally all region methods are used as reg.find(PS), where reg is a region object (or a screen or a match object). If written as find(PS) it acts on the default screen being the implicit region in this case (mapped to the constant reference SCREEN). In Multi Monitor Environments this is the primary monitor (use the constant reference SCREEN, to access it all the time), that normally is Screen(0), but might be another Screen() object depending on your platform.

So its a convenience feature, that can be seen as an implicit use of the python construct ‘’‘with object:’‘’.

On the other hand this may slow down processing speed, because of time consuming searches. So to speed up processing, saying region.find() will restrict the search to the specified rectangle. Another possibility is to say setROI() to restrict the search for all following find operations to a smaller region than the whole screen. This will speed up processing, if the region is significantly smaller than the whole screen.

Capturing

Capturing is the feature, that allows to grab a rectangle from a screenshot, to save it for later use. At each time, a capturing is initiated, a new screenshot is taken.

There are two different versions: the first one *Screen.capture()* saves the content of the selected rectangle in a file and returns its file name, whereas the second one *Screen.selectRegion()* just returns the position and dimension of the selected rectangle.

Both features are available in the IDE via the buttons in the toolbar.

class Screen

capture ([*region* | *rectangle* | *text*])
capture (*x*, *y*, *w*, *h*)

Parameters

- **region** – an existing region object.
- **rectangle** – an existing rectangle object (e.g., as a return value of another region method).

- **text** – text to display in the middle of the screen in the interactive mode.
- **x** – x position of the rectangle to capture
- **y** – y position of the rectangle to capture
- **w** – width of the rectangle to capture
- **h** – height of the rectangle to capture

Returns the path to the file, where the captured image was saved. In interactive mode, the user may cancel the capturing, in which case *None* is returned.

Interactive Mode: The script enters the screen-capture mode like when clicking the button in the IDE, enabling the user to capture a rectangle on the screen. If no *text* is given, the default "Select a region on the screen" is displayed.

If any arguments other than text are specified, capture() automatically captures the given rectangle of the screen. In any case, a new screenshot is taken, the content of the selected rectangle is saved in a temporary file. The file name is returned and can be used later in the script as a reference to this image. It can be used directly in cases, where a parameter PS is allowed (e.g. *Region.find()*, *Region.click()*, ...).

selectRegion([*text*])

Select a region on the screen interactively

Parameters **text** – Text to display in the middle of the screen.

Returns a new *Region* object or None, if the user cancels the capturing process.

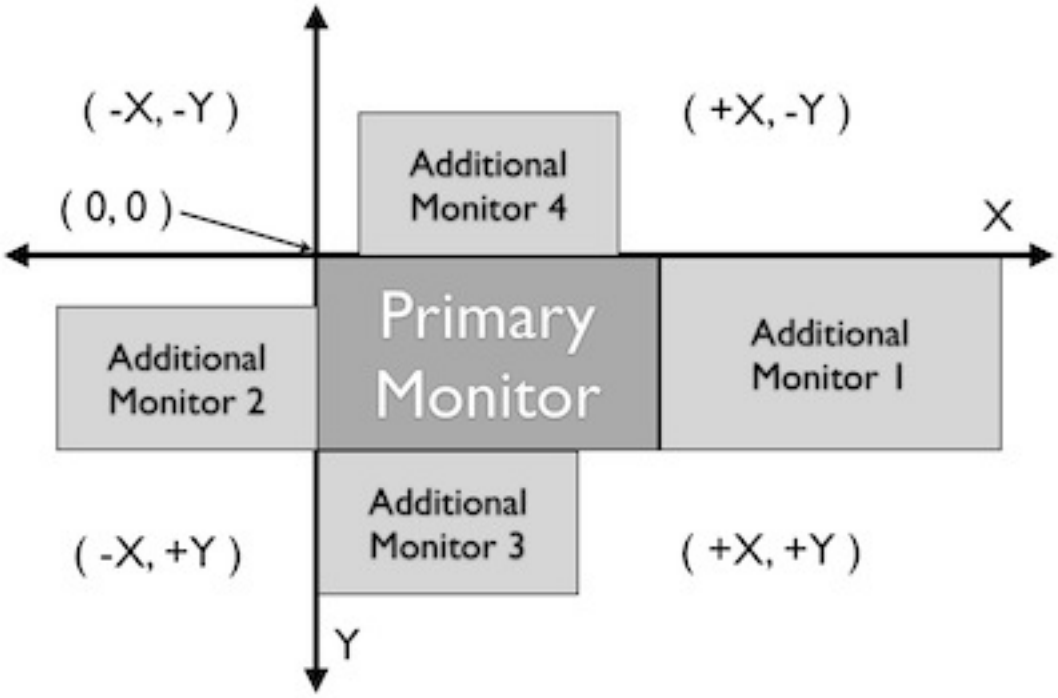
text is displayed for about 2 seconds in the middle of the screen. If **text** is omitted, the default "Select a region on the screen" is displayed.

The interactive capture mode is entered and allows the user to select a region the same way as using the selection tool in the IDE.

Note: You should check the result, since the user may cancel the capturing.

Multi-Monitor Environments

If more than one monitor is available, Sikuli is able to manage regions and click points on these monitors.



The base is the coordinate system (picture above), that positions the primary monitor with its upper left corner at (0,0) extending the x-direction to the right and the y-direction towards the lower boundary of the screen. The position of additional monitors can be configured in the operating system to be on either side of the primary monitor, with different positions and sizes. So monitors left of the primary will have pixels with negative x-values and monitors above will have negative y-values (left and above both x and y are negative).

At script start, Sikuli gets the relevant information from the operating system and creates respective screen objects, that have an ID (0 for the first or primary monitor, 1 and higher for additional monitors with a maximum of one less than the number of screens) and know the rectangle, they cover in the coordinate system. These informations are readonly for a script.

These predefined screen objects can be accessed with Screen(0), Screen(1), ... and are normally used to create your own screen objects. The possibility to use the region methods on a default region mapped to the primary monitor is implemented with the constant reference SCREEN. This concept is only available for the primary monitor.

How to get the relevant information:

- `getNumberScreens()` returns the number of available screens.
- `getBounds()` returns the rectangle covered by the default/primary monitor.
- `Screen.getBounds()` returns the rectangle covered by a screen object created using `Screen(id)`.

Be aware: Changes in your system settings are only recognized by the IDE, when it is started.

Windows: The monitor, that is the first one based on hardware mapping (e.g. the laptop monitor), will always be Screen(0). In the Windows settings it is possible to place the taskbar on one of the secondary monitors, which makes it the primary monitor getting the base coordinates (0,0). The other available monitors are mapped around based on your settings. But the Sikuli internal mapping is not changed, so the primary monitor might be any of your Screen() objects. Sikuli takes care for that and maps SCREEN always to the primary monitor (the one with the (0,0) coordinates). So for example you have a laptop with an external monitor, that shows the taskbar (is primary monitor):

- SCREEN maps to Screen(1)
- Screen(0) is your laptop monitor

Mac: The monitor, that has the System Menu Bar, is always Screen(0) and mapped to the default SCREEN.

Linux (Under construction)

With its rectangle, a screen object is always identical with the monitor it was created using `Screen(id)`. Using `Region.setROI()` to restrict the region of interest for find operations has no effect on the base rectangle of the screen object.

On the other hand region objects and location objects can be positioned anywhere in the coordinate system. Only when a find operation or a click action has to be performed, the objects rectangle or point has to be inside the rectangle of an existing monitor (basically repersented by Screen(0), Screen(1), ...). When this condition is met, everything works as expected and known from a single monitor system.

With finding and acting there are the following exceptions:

- **Point Outside:** a click point is outside any monitor rectangle. The clickpoint will be mapped to the edges or corners of the primary monitor according to the relative position:
  - to the edges if its x or y value is in the range of the respective edge (right, left, above, below)i
  - to the corners, if x and y are outside any range of any edge (left/above -> upper left corner, ...)
- **Region Outside:** a region is completely outside any monitor
  - a click action is handled in thesame way as **Point Outside**
  - a find operation will always fail
- **Region Partially Outside:** a region is partially outside a monitor but not overlapping another monitor
  - a click action is handled in the same way as **Point Outside**
  - a find operation will be carried out only in the part of region within the bounds of the monitor, excluding the area outside the monitor.
- **Region Across Monitors:** a region lies across multiple monitors:
  - a click action is handled in the same way as **Point Outside**
  - a find operation will be restricted to the region within the bounds of the monitor that has a smaller *id*.

An interactive capture (the user is asked to select an image or a rectangle via `Screen.capture()` or `Screen.selectRegion()`) will automatically be restricted to the monitor, where it was started.

A scripted capture using a rectangle or a region (i.e. `Screen.capture( region | rectangle )`), will be handled accordingly:

- **Region Outside:** no image is captured, *None* is returned
- **Region Partially Outside:** the returned image will only cover the part inside the monitor
- **Region Across Monitors:** the returned image will only cover the part inside the monitor with the smallest id.

Based on the knowledge of your monitor configuration, you can now start some further evaluations using e.g. `Region.hover()` together with `setShowActions(True)` and highlighting using `Region.highlight()`.

Connecting to a VNC Server (VNCScreen)

New in version 1.1.1.

The implementation is based on the TigerVNC Viewer package and was initially contributed by Pepijn Van Eeckhoudt https://github.com/pepijnve.

The intention of the following information is to only describe what is officially supported by a VNCScreen object aquired using vncStart(). For usage of the classes in the package itself you have to read the javadocs or look into the sources.

To make the package more useable there are now highlevel wrappers, that hide the logic to create, start and stop the socket based connection. More than one connection can be used at one time, each represented by a different VNCScreen object.

vncStart (*[ip="127.0.0.1",] [port=5900,] [password=None,][connectionTimeout=10,] [timeout=1000]*)

Start a VNC session to the given (usually remote) running VNC server and on success get a VNCScreen object, that can be used like a Screen object. About the restrictions and special features see the comments below.

Parameters

- **ip** – the server IP (default: 127.0.0.1 loopback/localhost)
- **port** – the port number (default 5900)
- **password** – for password protected connections as plain text
- **connectionTimeout** – seconds to wait for a valid connection (default 10)
- **timeout** – the timout value in milli-seconds during normal operation (default 1000)

Returns a new VNCScreen object useable like a Screen object

stop()

Stop the referenced VNC session, which closes the underlying socket connection and makes the VNCScreen object unuseable.

mandatory usage `someVNCScreen.stop()`, where `someVNCScreen` is a VNCScreen object aquired before using `someVNCScreen = vncStart(...)`.

In basic operation environments there is no need to issue the `vnc.stop()` explicitely, because all active VNC connections are auto-stopped at the end of a script run or at termination of a Java run.

USAGE IN JAVA as being a static method in class VNCScreen, `vncStart()` has to be used as:

```
VNCScreen vnc = VNCScreen.start(ip, port, connectionTimeout, timeout) // or
VNCScreen vnc = VNCScreen.start(ip, port, password, connectionTimeout, timeout)
// the parameters are mandatory with values as mentioned above
// password can be null for unprotected connections
// do something with the vnc object
vnc.stop() // optional - see above
```

Some general information and comments

Due to the current implementation concept of VNCScreen, **Region or Location objects intended to be on a remote screen have to know this fact**. Otherwise they are simply Regions and Locations on a local screen with fitting coordinates. This knowledge of being on a remote screen is internally propagated from one object to a new object created by a feature of the existing object. Hence in the beginning only the created VNCScreen object knows about being on a remote screen. So to create Regions and Locations on the remote screen from scratch, you have to use features of VNCScreen.

These are the rules:

- the VNCScreen object itself is a remote Region in this sense
- each `Match/Region/Location` created using a `VNCScreen` object knows about being remote
- each `Region/Location` object created using a feature of a `remote Region/Location` also knows about being remote

- to create a `new Region/Location` from scratch use the `newRegion()/newLocation()` methods of VNCScreen
- all mouse and keyboard actions using remote Regions/Locations are going to the remote screen

Methods to create new remote Regions and Locations:

```
# someRegion/someLocation may be normal Region/Location objects
# someRectangle/somePoint are normal java.awt.Rectangle/java.awt.Point objects
# remoteRegion/remoteLocation/remoteMatch know about being remote

vnc = vncStart("192.168.2.25") # some VNC Server in the local net

# create from scratch
remoteRegion = vnc.newRegion(x, y, w, h)
remoteRegion = vnc.newRegion(someRegion)
remoteRegion = vnc.newRegion(someRectangle)
remoteLocation = vnc.newLocation(x, y)
remoteLocation = vnc.newLocation(someLocation)
remoteLocation = vnc.newLocation(somePoint)

# propagate remote aspect
remoteRegion = remoteRegion.right(200)
remoteMatch = vnc.find("someImage.png")
remoteLocation = remoteMatch.getCenter()
```

BE AWARE

- Due to the correct RFB protocol implementaion in TigerVNC Viewer, it may take some time (up to few seconds depending on line speed and remote screen size) to initialize the frame buffer content after connection start. So if you get problems with the first access to the remote screen content (capture, userCapture, find operations explicit or implicit), you should simply add an appropriate wait() after the vncStart(). Experiences in local environment with large screens: 2 - 3 seconds are sufficient.
- Not all documented Screen/Region/Location methods might work as expected due to implementation quirks. In case, feel free to report a bug.
- The current implementation only supports a **limited set of RemoteFrameBuffer protocols**. The above described level of usage is successfully tested from a Mac OSX 10.10+ against a TightVNC server running on a Windows 10 64-Bit in the local network or both client and server on the mentioned Windows machine using the loopback IP (127.0.0.1).

Connecting to an Android device or emulator (ADBScreen)

New in version 1.1.1.

Based on the ideas and first implementation of Gergő Törcsvári https://github.com/tg44 it is now possible to capture images from an Android device in the IDE and run scripts the same way as with a local screen. You can wait for images to appear on the Android screen and act with taps, swipes and text input. Since the avarage latency for a search operation is about 1 second (varies with screen resolution, region size and device processor speed), this solution only makes sense for basic automation and testing, where speed does not matter.

The implementation uses adb (Android Debugging Bridge) and the Java wrapper jadb. It can be used with devices and emulators with minimum Android version 4 and does not need rooting. A device has to be attached via USB (first tests using a WiFi connection where discouraging). Currently only one connection is supported. If more than one device is available, then the one is connected, that shows up in first place on the device list.

If you want to use this feature, you should be familiar at least with the basics of adb.

The usage is similar to VNCScreen: you work with an ADBScreen object, that represents the device's screen and in SikuliX terms is a Region that provides all search and action features. As far as possible the actions are transformed to Android actions: a click gets a tap and type/paste result in an input text. Most mouse and keyboard actions will do nothing but produce an error log. Furthermore Android typical actions are now available with class Region: tap, swipe, input, ... and it is possible to issue device commands via exec. If used with a local screen, these features silently do nothing.

Pattern

class `Pattern`

A pattern is used, to associate an image file with additional attributes used in find operations and when acting on a match object.

Minimum Similarity:

While using a *`Region.find()`* operation, if only an image file is provided, Sikuli searches the region using a default minimum similarity of 0.7. This default value can be changed in *`Settings.MinSimilarity`*.

Using *`similar()`* you can associate a specific similarity value, that will be used as the minimum value, when this pattern object is searched. The IDE supports adjusting the minimum similarity of captured images using the Preview Pane (internally in the script, the images are turned into a pattern object automatically).

Click Point:

Normally when clicking on a match, the center pixel of the associated rectangle is used. With a pattern object, you can define a different click point relative to the center using *`targetOffset()`*.

class `Pattern`

`Pattern` (*string*)

> **Parameters** **`string`** – a path to an image file
>
> **Returns** a new pattern object

This will initialize a new pattern object without any additional attributes. As long as no pattern methods are used additionally, it is the same as just using the image file name itself in the find operation.

`similar` (*similarity*)

Return a new Pattern object containing the same attributes (image, click point) with the minimum similarity set to the specified value.

> **Parameters** **`similarity`** – the minimum similarity to use in a find operation. The value should be between 0 and 1.
>
> **Returns** a new pattern object

`exact` ()

Return a new Pattern object containing the same attributes (image, click point) with the minimum similarity set to 1.0, which means exact match is required.

> **Returns** a new pattern object

`targetOffset` (*dx*, *dy*)

Return a new Pattern object containing the same attributes (image, similarity), but a different definition for the click. By default, the click point is the center of the found match. By setting the target offset, it is possible to specify a click point other than the center. *dx* and *dy* will be used to calculate the position relative to the center.

Parameters

- **dx** – x offset from the center
- **dy** – y offset from the center

Returns a new pattern object

getFilename()
Get the filename of the image contained in the Pattern object.

Returns a filename as a string

getTargetOffset()
Get the target offset of the Pattern object.

Returns a *Location* object as the target offset

Match

class Match

An object of class Match represents the result of a successful find operation. It has the rectangle dimension of the image, that was used to search. It knows the point of its upper left corner on an existing monitor, where it was found.

Since class Match extends class Region, all methods of class *Region* can be used with a match object.

Creating a Match, Getting Attributes

A match object is created as the result of an explicit *find operation*. It can be saved in a variable for later use with actions like *click()*.

It has the rectangle dimension of the image, that was used to search. It knows the point of its upper left corner on an existing monitor, where it was found. It knows the similarity it was found with and a click point to be used, if set by a pattern.

For all other aspects, the features and attributes of class *Region* apply.

class Match

getScore()
Get the similarity score the image or pattern was found. The value is between 0 and 1.

getTarget()
Get the `location` object that will be used as the click point.

Typically, when no offset was specified by *Pattern.targetOffset()*, the click point is the center of the matched region. If an offset was given, the click point is the offset relative to the center.

Iterating over Matches after findAll()

A find operation *Region.findAll()* returns an iterator object that can be used to fetch all found matches as match objects one by one. A reference to the iterator is stored in the respective region and can be accessed using *Region.getLastMatches()*.

Important to know:

- per definition, an iterator can be stepped through only once - it is empty afterwards

You can read more about the basics of operations with iterators from the description of *Finder* class. To save contained matches for later use, you can convert them to list.

Example: using `while:` with default screen

Example: using `with:` with default screen

Finder

class `Finder`

New in version 1.1.0.

The behavior is changed compared to previos versions, to be consistent with Region.find()/findAll().

A Finder object implements an iterator of matches and allows to search for a visual object in an image file that you provide (e.g. a screenshot taken and saved in a file before). After setting up the finder object and doing a find operation, you can iterate through the found matches if any.

Important to know:

- per definition, an iterator can be stepped through only once - it is empty afterwards
- it has to be destroyed using `finder.destroy()`, especially when used with `for:` or `while:`
- when used in a `with:` construct, it is destroyed automatically

Compared with the region based find/findAll operation, no exception FindFailed is raised in case nothing is found at all (use `hasNext()` to check).

The result using a finder object can be compared to what you get with `region.getLastMatch()` when using *find()* or with `region.getLastMatches()` when using *findAll()*.

Note: There is no chance, to get the number of matches in advance. If you iterate through to count, afterwards your finder is empty. So you have to save your matches somehow while counting, if you need them later (one possible solution see example below).

The workflow always is:

- setup a Finder
- do a find or findAll operation
- check with hasNext(), wether anything was found at all
- get the available matches with next() if hasNext() says more available
- After a complete iteration, the finder object is empty.
- You can start a new find or findAll operation at any time.

class `Finder`

`Finder` (*path-to-imagefile*)
Create a new finder object.

Parameters **`path-to-imagefile`** – filename to a source image to search within

`find` (*path-to-imagefile*[, *similarity*])
Find a given image within a source image previously specified in the constructor of the finder object. If more than one match is possible, yet only one is returned and which one is not predictible.

Parameters

- **path-to-imagefile** – the target image to search for
- **similarity** – the minimum similarity a match should have. If omitted, the default is used.

New in version 1.1.0.

findAll (*path-to-imagefile*[, *similarity*])
Find all occurences of the given image within a source image previously specified in the constructor of the finder object.

Parameters

- **path-to-imagefile** – the target image to search for
- **similarity** – the minimum similarity a match should have. If omitted, the default is used.

hasNext ()
Check whether there are more matches available that satisfy the minimum similarity requirement. A *False* returned after the first hasNext() signals, that nothing was found at all.

Returns *True* if more matches exist.

next ()
Get the next match.

Returns a *Match* object or None, if empty or no more matches.

The returnd reference to a match object is no longer available in the finder object afterwards. So if it is needed later, it has to be saved to another variable.

Example 1: basic operations using a Finder

Example 2: we want to know how many matches in advance and want to save the matches for later use (based on the previous example).

Key Constants

class Key

Applicable usage situations for these predefined constants of special keys and key modifiers can be found in *Acting on a Region* and *Low Level Mouse and Keyboard Actions*.

Special Keys

The methods supporting the use of special keys are

- *type()*,
- *keyDown()*
- *keyUp()*.

Usage: Key.CONSTANT (where CONSTANT is one of the following key names, uppercase mandatory).

String concatenation with with other text or other key constants is possible using ‘‘ + ‘‘.

```
type("some text" + Key.TAB + "more text" + Key.TAB + Key.ENTER)
# or eqivalent
type("some text\tmore text\n")
```

miscellanous keys

```
ENTER, TAB, ESC, BACKSPACE, DELETE, INSERT
```

miscellanous keys

```
SPACE
```

function keys

```
F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F13, F14, F15
```

navigation keys

```
HOME, END, LEFT, RIGHT, DOWN, UP, PAGE_DOWN, PAGE_UP
```

special keys

```
PRINTSCREEN, PAUSE, CAPS_LOCK, SCROLL_LOCK, NUM_LOCK
```

Note: The status (on / off) of the keys `Key.CAPS_LOCK`, `Key.NUM_LOCK` and `Key.SCROLL_LOCK` can be evaluated with the method *`Env.isLockOn()`*.

numpad keys

```
NUM0, NUM1, NUM2, NUM3, NUM4, NUM5, NUM6, NUM7, NUM8, NUM9
SEPARATOR, ADD, MINUS, MULTIPLY, DIVIDE
```

Key Modifiers (modifier keys)

modifier keys

```
ALT, CMD, CTRL, META, SHIFT, WIN, ALTGR
```

A key is called a modifier key, when it is used in conjunction with other keys by pressing and holding it while typing the other keys. The keys used most commonly as modifier keys are SHIFT, ALT (left alt key, CTRL (might be strg on Windows) and the CMD key (Apple key on OS X). On Windows you additionally have the Windows key and the alt key on the right.

the plain *`type()`* command has 2 parameters:

- parameter 1: the keys, that should be pressed/released one after the other
- parameter 2: the modifier keys, that alltogether should be pressed and held during the typing and released at the end.

For parameter 2 there are 2 options:

- new version: (a key is a key ;-): `Key.XXX`
- old version (kept for upwards compatibility): `KeyModifier.XXX`

... and these early versions should not be used anymore `KEY_ALT, KEY_CTRL, KEY_SHIFT, KEY_WIN, KEY_CMD, KEY_META`

The modifier keys can be combined using ‘‘ + ‘‘, if more than one key modifier is needed.

```
# example with the recommended Key.XXX version
type(Key.ESC, Key.CTRL + Key.ALT)
```

***Note for Java programming*:** These constants are mapped to the according constants of `java.awt.event.InputEvent`.

The Application Class (App)

class App

Using class or instance methods

Generally you have the choice between using the class methods (e.g. `App.open("application-identifier")`) or first create an App instance and use the instance methods afterwards (e.g. `myApp = App("application-identifier")` and then later on `myApp.open()`). There is no recomendation for a preferred usage. The only real difference is, that you might save some ressources, when using the instance approach, since using the class methods produces more intermediate objects. So if you plan to act on the same app or window more often, it might be more transparent, to use the instance approach **How to create an App instance**

The basic choice is to just say `someApp = App("some-app-identifier")` and you have your app instance, that you can later use together with its methods, without having to specify the string again.

Generally all class methods return an app instance, that you might save in a variable to use it later in your script.

At time of instance creation the process list is scanned for the name of the executable using the given text. If this is found, the app instance is initialized with the respective information (PID, executable, window title of main/frontmost window). So you could directly ask the app instance, wether it is running (`isRunning()`), has a main window (`hasWindow()`), get the title of that main window (`getWindow()`) and get the process id (PID) (`getPID`).

The string representation of an app instance looks like this: `[nPID:executableName (main/frontmost window title)] given text`

(Windows/Linux) If it is not found, the titles of the currently known windows are scanned, wether they contain the given text. The first matching window found evaluates to the application, that initializes the app instance.

If neither the executable is found in the process list, nor a matching window, the app instance is initialized as not running (the PID is set to -1). The given text is remembered.

If you specify the exact window title of an open window, you will get exactly this one. But if you specify some text, that is found in more than one open window title, you will get the first in the row of all these windows. So if you want exactly one specific window, you either need to know the exact window title or at least some part of the title text, that makes this window unique in the current context (e.g. save a document with a specific name, before accessing it's window).

(Mac OS X) not yet possible, to identify a running app by part of the title of one of it's windows. The window title you get by ‘‘getWindow()‘‘is the one of the currently frontmost window of that application.

NOTE Currently the information, wether a window is hidden or minimized, is not available and it is not possible yet, to bring such a window to front with a compound SikuliX feature.

class App

classmethod pause(*waitTime*)

Usage: `App.pause(someTime)` (convenience function)

Just do nothing for the given amount of time in seconds (integer or float).

classmethod App(*application*)

Usage: `someApp = App(application)`

Create an App instance, to later use with the instance methods (*see above*)

Parameters **application** – The name of an application (case-insensitive), that can be found in the path used by the system to locate applications, or the full path to an application. Optionally you might add parameters, that will be given to the application at open (see *setUsing()*).

Returns an App object, that can be used with the instance methods

isRunning([*waitTime*])

Usage: `if not someApp.isRunning(): someApp.open()` where App instance `someApp` was *created before*.

Parameters **waitTime** – optional: seconds as integer, that should be waited for the app to get running

Returns True if the app is running (has a process ID), False otherwise

Windows It is common, to identify an app by (part of) it's window title. If it is not open yet, one has to use methods, to open it first before proceeding.

So this is a typical example, how to deal with that:

Example:

```
# we want to act in a VirtualBox VM window and use the VM's name
# which is always part of the Window title when running
vb = App("VM-name")
if not vb.isRunning():
        App.open(r'"C:\Program Files\Oracle\VirtualBox\VBoxManage.exe" startvm VM-name')
        while not vb.isRunning():
                wait(1)
vb.focus()
appWindow = App.focusedWindow()
```

hasWindow()

Usage: `if not someApp.hasWindow(): openNewWindow() # some private function` where App instance `someApp` was *created before*.

Returns True if the app is running and has a main window registered, False otherwise

getWindow()

Usage: `title = someApp.getWindow()` where App instance `someApp` was *created before*.

Returns the title of the frontmost window of this application, might be an empty string

getPID()

Usage: `pid = someApp.getPID()` where App instance `someApp` was *created before*.

Returns the process ID as number if app is running, -1 otherwise

getName()

Usage: `appName = someApp.getName()` where App instance `someApp` was *created before*.

Returns the short name of the app as it is shown in the process list

setUsing(*parametertext*)

Usage: `appName = someApp.setUsing("parm1 x parm2 y parm3 z")` where App instance `someApp` was *created before*.

Parameters **parametertext** – a string, that is given to the applications startup as you would give it, if you would start the app from a commandline.

Returns the app instance

classmethod **open**(*application*)
Usage: `App.open(application)`

Open the specified application, if it is not yet opened and bring it to front

Parameters **application** – The name of an application (case-insensitive), that can be found in the path used by the system to locate applications, or the full path to an application (Windows: use double backslash \ in the path string to represent a backslash)

Returns an App object, that can be used with the instance methods, None in case of failing

This method is functionally equivalent to *openApp()*.

open([*waitTime*])
Usage: `someApp.open()` where App instance `someApp` was *created before*.

Open this application.

Parameters **waitTime** – optional: seconds as integer, that should be waited for the app to get running

Returns the app instance or null/None if open failed

classmethod **focus**(*application*)
Usage: `App.focus(application)`

Switch the input focus to an application/window.

Parameters **application** – The name of an application (case-insensitive) or (part of) a window title (Windows/Linux) (case-sensitive).

Returns an App object, that can be used with the instance methods, , None in case of failing

focus()
Usage: `someApp.focus()` where App instance `someApp` was *created before*.

Switch the input focus to this application/window.

classmethod **close**(*application*)
Usage: `App.close(application)`

It closes the given application or the matching windows (Windows/Linux). It does nothing if no running application or opened window (Windows/Linux) can be found. On Windows/Linux, whether the application itself is closed depends on weather all open windows are closed or a main window of the application is closed, that in turn closes all other opened windows.

Parameters **application** – The name of an application (case-insensitive) or (part of) a window title (Windows/Linux)(case-sensitive).

This method is functionally equivalent to *closeApp()*.

close()
Usage: `someApp.close()` where App instance `someApp` was *created before*.

Close this application.

Dealing with Application windows

classmethod `focusedWindow`()

Usage: `App.focusedWindow()`

Identify the currently focused or the frontmost window and switch to it. Sikuli does not tell you, to which application this window belongs.

Returns a *Region* object representing the window or *None* if there is no such window.

On Mac, when starting a script, Sikuli hides its window and starts processing the script. In this moment, no window has focus. Thus, it is necessary to first click somewhere or use `App.focus()` to focus on a window. In this case, this method may return *None*.

On Windows, this method always returns a region. When there is no window opened on the desktop, the region may refer to a special window such as the task bar or an icon in the system tray.

Example:

```
# highlight the currently fontmost window for 2 seconds
App.focusedWindow().highlight(2)

# save the windows region before
firstWindow = App.focusedWindow()
firstWindow.highlight(2)
```

`window`([*n*])

Usage 1: `App(application).window([n])` an App instance is created on the fly.

Usage 2: `someApp.window([n])` where App instance `someApp` was *created before*.

Get the region corresponding to the n-th window of this application (Mac) or a series of windows with the matching title (Windows/Linux).

Parameters **`n`** – 0 or a positive integer number. If ommitted, 0 is taken as default.

Returns the region on the screen occupied by the window, if such window exists and *None* if otherwise.

Below is an example that tries to open a Firefox browser window and switches to the address field (Windows):

```
# using an existing window if possible
myApp = App("Firefox")
if not myApp.window(): # no window(0) - Firefox not open
        App.open("c:\\Program Files\\Mozilla Firefox\\Firefox.exe")
        wait(2)
myApp.focus()
wait(1)
type("l", KEY_CTRL) # switch to address field
```

Afterwards, it focuses on the Firefox application, uses the `window()` method to obtain the region of the frontmost window, applies some operations within the region, and finally closes the window:

```
# using a new window
firefox = App.open("c:\\Program Files\\Mozilla Firefox\\Firefox.exe");
wait(2)
firefox.focus()
wait(1)
# now your just opened new window should be the frontmost
with Region(firefox.window()): # see the general notes below
```

```
        # some actions inside the window(0)'s region
        click("somebutton.png")
firefox.close() # close the window - stop the process
```

Below is another example that highlights all the windows of an application by looping through them (Mac):

```
# not more than 100 windows should be open ;-)
myApp = App("Safari")
for n in range(100):
        w = myApp.window(n)
        if not w: break # no more windows
        w.highlight(2) # window highlighted for 2 second
```

General aspects, hints and tipps

- Be aware, that especially the window handling feature is experimental and under further development.
- Especially on Windows be aware, that there might be many matching windows and windows, that might not be visible at all. Currently the `window()` function has no feature to identify a special window besides returning the region. So you might need some additional checks to be sure you are acting on the right window.
- Windows/Linux: The `close()` function currently kills the application, without closing it's windows before. This is an abnormal termination and might be recognized by your application at the next start (e.g. Firefox usually tries to reload the pages).
- Even if the windows are hidden/minimized, their region that they have in the visible state is returned. Currently there is no Sikuli feature, to decide wether the given window(n) is visible or not or if it is currently the frontmost window. The only guarentee: `window()`/`window(0)` is the topmost window of an application (Mac) or a series of matching windows (Windows/Linux).
- Currently there are no methods available to act on such a window (resize, bring to front, get the window title, ...).

Some tips:

- Check the position of a window's returned region: some apps hide there windows by giving them "outside" coordinates (e.g. negative)
- Check the size of a window's returned region: normally your app windows will occupy major parts of the screen, so a window's returned region of e.g. 150x30 might be some invisible stuff or an overlay on the real app window (e.g. the "search in history" input field on the Safari Top-Sites page, which is reported as `windows(0)`)
- If you have more than one application window, try to position them at different coordinates, so you can decide which one you act on in the moment.
- It is sometimes possible to use the OCR text extraction feature *`Region.text()`* to obtain the window title.

CHAPTER 7

Miscellanous

Can I do X or Y or Z in SikuliX?

If you are wondering if Sikuli can do X or Y, these two general rules apply:

- If you can do X with Java, you can also do it in SikuliX ...

  ... by simply adding the respective Java resources to the classpath (the standard Java classes are already there).

  For example, you know how to create a GUI with Java Swing, you can do it in the same way in SikuliX.

- If you can do X with Python, you probably can do it in SikuliX as well.

  This actually depends on what Python modules you want to use. SikuliX is on Python language level 2.5 and 2.7. So everything available in the respective Python base package is available in SikuliX too.

  If modules are written in pure Python, you can use them in SikuliX as well. A typical example are the Excel access modules xlrd and xlwt.

  If modules are written in C or depend on C-based stuff, unfortunately, you can't use them in SikuliX (e.g. the support for Win32API calls).

  But it always is a good idea to check in the net, wether there either is a feature compatible module already available for Jython or you might use an appropriate Java library package.

- To Ruby the same rule applies accordingly.

For any specific question it is always a good idea to first consult SikuiX's FAQs and Questions

Then search with Google for the topic mentioning Jython, JRuby or even Sikuli - you will get tons of hints and snippets.

Can I write a loop in SikuliX?

Yes. Check the Python or Ruby language specs and/or tutorials.

Can I create a GUI in SikuliX?

Yes. You can create GUIs with Java Swing or any other Java/Jython/JRuby GUI toolkit.

Can I connect to MySQL/MS SQL/PostgreSQL or any database systems in SikuliX?

You can use JDBC or zxJDBC.

Can I read/write files in SikuliX?

Yes. Check the Python or Ruby language specs and/or tutorials.

How to run SikuliX from Command Line

SikuliX can be used on command line to run a Sikuli script or open the IDE.

The usage on each platform:

Windows

```
PATH-TO-SIKULIX/runsikulix.cmd [options]
```

Mac OS X

```
PATH-TO-SIKULIX/runsikulix [options]
```

Linux

```
PATH-TO-SIKULIX/runsikulix [options]
```

runsikulix(.cmd) without any options simply starts SikuliX IDE.

PATH-TO-SIKULIX is the folder containing the Sikuli stuff after having run setup.

Command Line Options (generally, debug output related)

PATH-TO-SIKULIX/runsikulix(.cmd)

-d, `--debug <value> (up to 3 makes sense)`
raise the verbosity level of SikuliX's internal debug messages - might be useful to clarify odd situations

-f, `--logfile [<path to log file>]`
a valid filename (if omitted: WorkingDir/SikuliLog.txt) where SikliX's log messages should be written to

-u, `--userlog [<path to log file>]`
a valid filename (if omitted: WorkingDir/UserLog.txt) where user log messages created using Debug.user() should be written to

Command Line Options (special)

PATH-TO-SIKULIX/runsikulix(.cmd)

-h, `--help`
print a help message showing the available options and exit

-i, `--interactive`
open an interactive Jython session that is prepared for the usage of the SikuliX features

Command Line Options (intention: IDE should open)

PATH-TO-SIKULIX/runsikulix(.cmd)

-c, `--console`
all output goes to stdout

Command Line Options (intention: run a script without opening the IDE)

PATH-TO-SIKULIX/runsikulix(.cmd)

-r, `--run <sikuli-folder/file>  (one or more entries seperated by space)`
run one or more .sikuli or .skl files one after the other

***<sikuli-folder/file>* can be**

- a relative or absolute path with or without dotted parts (e.g. ../some-script)
- a pointer to a location in the HTTP net (*for details look here*). The contained script file is downloaded and run, while the image files are downloaded when used in the script at runtime.

Having more than one script to run, the folder containing the script folder is remembered and applied to a following entry, that has a preceding ./ - example

sikulix.com:scripts/test1 ./test2 ./test2 will reuse the location sikulix.com:scripts/ for test2 and test3

Having more than one script specified: a return code of -1 will stop the complete execution.

Having more than one script specified: the next script can get the return code of the script run before using *ScriptingSupport.getLastReturnCode()*

Command Line Options (intention: run the experimental scriptrun server)

PATH-TO-SIKULIX/runsikulix(.cmd)

-s, `--server [<port>]  (optional port not yet supported,` **`50001`** `is used as default)`
start a scriptrun server (**'more information<http://www.sikulix.com/support.html>'_**)

Command Line Options (intention: provide user parameters for running scripts)

PATH-TO-SIKULIX/runsikulix(.cmd)

-- <arguments>
: the space delimited and optionally quoted arguments (only apostrophes are supported) are passed to Jython's sys.argv and hence are available in your script. A parameter containing intermediate blanks MUST be quoted to get it into one sys.argv entry.

  This option must go after all the other options mentioned above.

How to use SikuliX API in your JAVA programs or Java aware scripting

The core of SikuliX is written in Java, which means you can use the SikuliX API as a standard JAVA library in your program.

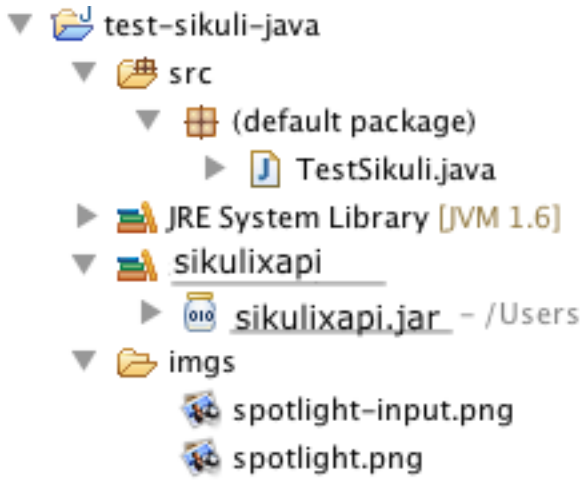
This applies to any Java aware scripting environment like Jython, JRuby, Scala, Groovy, Clojure and more, where you write your scripts in other IDE's and run them using the respective runtime support directly.

NOTE: When using the scripting support provided by the SikuliX IDE and when running Jython/JRuby scripts from command
: For information what you can do when using the Java API to have this convenience too, look here *SIKULI_IMAGE_PATH*.

After having setup SikuliX on your system, as recommended at Getting started, you have to do the following:

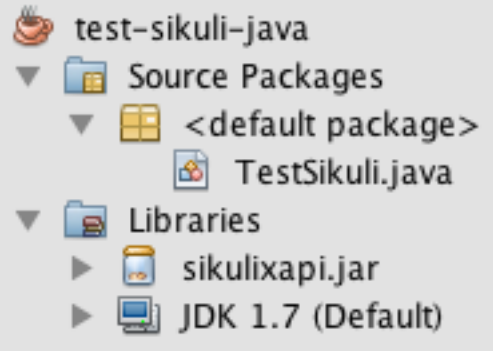
1. Include sikulixapi.jar in the CLASSPATH of your Java project.

After adding sikulixapi.jar as a library reference into your project, the project hierarchy in Eclipse might look like this:

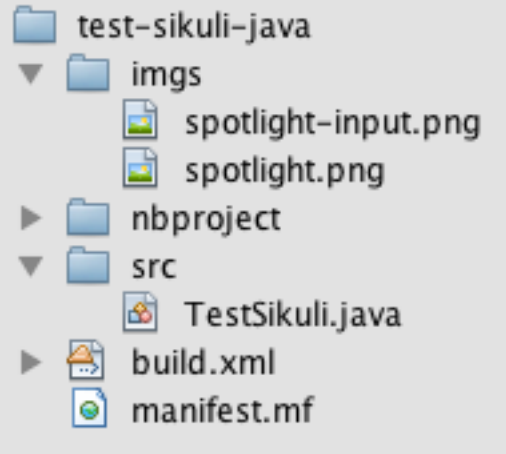


The same in Netbeans:

Project view



Files view



2. Import the Sikuli classes you need

You can simply use

```
import org.sikuli.script.*;
```

or import the classes you need:

```
import org.sikuli.script.Screen;
```

In most cases, you would need at least *Screen* and/or *Region*.

Other candidates are *Pattern*, *Match*, *Location*, *App* and some more.

3. Write code!

More basic usage information is available here.

Here is a hello world example on Mac. The program clicks on the spotlight icon on the screen, waits until spotlight's input window appears, activates it by clicking and then writes "hello world" into the field and hits ENTER.

```
import org.sikuli.script.*;

public class TestSikuli {

        public static void main(String[] args) {
```

```
                Screen s = new Screen();
                try{
                        s.click("imgs/spotlight.png");
                        s.wait("imgs/spotlight-input.png");
                        s.click();
                        s.write("hello world#ENTER.");
                }
                catch(FindFailed e){
                        e.printStackTrace();
                }
        }

}
```

A comment on projects using Maven

It is planned, to publish sikulixapi.jar version 1.1.0+ on MavenCentral, so having a dependency in your project pom would be sufficient.

The coordinates:

<groupId>com.sikulix</groupId>
<artifactId>sikulixapi</artifactId>
<version>1.1.0</version>

Snapshots of developement head can be loaded from OSSRH they are created daily (most of the time ;-)

use this repository setting:

<repository>
<!– OSSRH: com.sikulix –>
<id>com.sikulix</id>
<name>com.sikulix</name>
<url>https://oss.sonatype.org/content/groups/public</url>
<layout>default</layout>
<snapshots>
<enabled>true</enabled>
<updatePolicy>always</updatePolicy>
</snapshots>
</repository>

and as version use: <version>1.1.0-SNAPSHOT</version>

Other valuable information

Be aware, that some method signatures in the Java API differ from the scripting level.

- Javadoc of SikuliX (temporary location).

How to use images stored in jar files

For the basic ImagePath features see: *SIKULI_IMAGE_PATH*

The prereqs:

- the jar file containing the images must be on classpath at runtime
- it must contain at least one valid java class file

This is always fulfilled, if the running jar itself contains the images to be used.

An example:

lets assume:

- the class file is mypackage.ImageContainer
- the images are stored in the folder images, being at the jar's root folder level

the jar file should look like this

mypackage (optional)
    ImageContainer.class
images
    image1.png
    image2.png

the standard Maven project setup would be

pom.xml
src
    main
        java
            mypackage
                ImageContainer.class
        resources
            images
                image1.png
                image2.png

In other non-Maven project setups in NetBeans, Eclipse and other IDE's this might look different. But what counts is the final position of the images folder in the resulting jar file relative to it's root.

This would try to load image files from the jar file of the given class name, supposing the jar packaging during project build produces a jar like this:

TestSikuli.class
images
    spotlight.png
    spotlight-input.png

```
import org.sikuli.script.*;

public class TestSikuli {

        public static void main(String[] args) {
                Screen s = new Screen();
                ImagePath.add("TestSikuli/images")
                try{
                        s.click("spotlight.png");
                        s.wait("spotlight-input.png");
                        s.click();
                        s.write("hello world#ENTER.");
                }
```

```
                catch(FindFailed e){
                        e.printStackTrace();
                }
        }

}
```

As a goody you might use the SikuliX IDE to manage your image folders inside your java projects.

Just capture the images and save the "script" appropriately (e.g. as images.sikuli) in your project structure in a suitable place and add the path to your java program having the jar file on the class path at runtime.

Taking the above ImageContainer example ina Maven context:

project structure:

pom.xml
src
main
java
mypackage
ImageContainer.class
resources
images.sikuli
images.py
spotlight.png
spotlight-input.png

The ImageContainer class need not have any features, but must be valid (compileable).

Have the resulting jar on classpath at runtime and access the images just using their base names:

```
import org.sikuli.script.*;

public class TestSikuli {

        public static void main(String[] args) {
                Screen s = new Screen();
                ImagePath.add("ImageContainer/images.sikuli")
                try{
                        s.click("spotlight.png");
                        s.wait("spotlight-input.png");
                        s.click();
                        s.write("hello world#ENTER.");
                }
                catch(FindFailed e){
                        e.printStackTrace();
                }
        }

}
```

Extensions

General Information About Sikuli Extensions

This feature is currently under developement and might only be available on request.

Extensions allow to implement new Sikuli features by adding packages to your current Sikuli installation.

They are maintained by the developers (see **Technical Details** below).

If you want to contribute a new extension or a modified exisiting one, please look at **How to contribute an extension** below.

How to Download and use an Extension

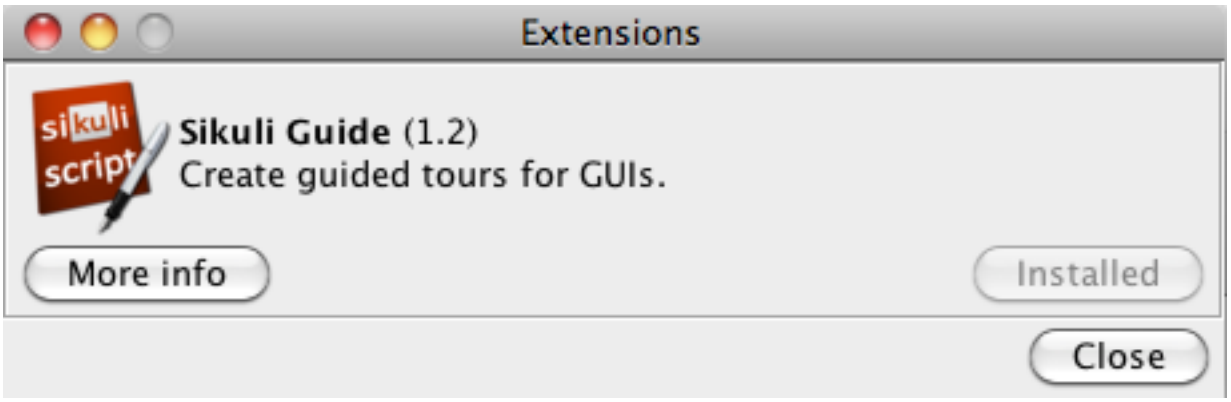
The download of an extension is supported by the IDE through the menu *Tools -> Extensions*. You get a popup, that lists the available and already installed extensions and allows to download new packages or updates for installed ones.

This popup shows a new **package not yet installed**:

If you need more information about the features of the extension, just click *More Info* - this will open the related documentation from the web in a browser window.

If you want to install the extension, just click the *Install...* button. The package will be downloaded and added to your extensions repository.

This popup shows an **installed package**:



If a new version would be available at that time, the *Install...* button would be active again, showing the new version number. Now you could click and download the new version.

How to Use an Extension

To use the features of an installed extension in one of your scripts, just say `from extension-name import *`. For an usage example read *Sikuli Guide*.

For information about features, usage and API use menu *Tools* -> *Extensions* -> *More Info* in the IDE.

Technical Details

Extensions are Java JAR files containing some Java classes (usually the core functions) and/or Python modules, which define the API to be used in a script.

Sikuli maintains a local extensions directory, where downloaded extensions are stored together with a hidden list of the installed extensions (Windows: `%APPDATA%\Sikuli\extensions`, Mac: `~/Library/Application Support/Sikuli/extensions`, Linux: `~/.sikuli/extensions`).

Once an extension is imported using `import extension-name`, Sikuli automatically searches and loads the JAR file of that extension into the current context with *`load(path-to-jar-file)`*.

How to develop an extension

The **source structure** of an extension named `extension-name` looks like this:

```
Java
- org/com
-- your-organization-or-company
--- extension-name
---- yourClass1.java
---- yourClass2.java
---- .... more classes
python
- extension-name
-- __init__.py
-- extension-name.py
```

The **final structure of a JAR** (filename `extension-name-X.Y` where X.Y is the version string) looks like this:

```
org/com
- your-organization-or-company
-- extension-name
--- yourClass1.class
--- yourClass2.class
--- .... more classes
extension-name
- __init__.py
- extension-name.py
META-INF
- MANIFEST.MF
```

The file `__init__.py` contains at least `from extension-name import *` to avoid one qualification level. So in a script you might either use:

```
import extension-name
extension-name.functionXYZ()
```

or:

```
from extension-name import *
functionXYZ()
```

The second case requires more investement in a naming convention, that avoids naming conflicts.

The file `extension-name.py` contains the classes and methods, that represent the API, that one might use in a Sikuli script.

As an example you may take the source of the extension Sikuli Guide.

Name your extensions properly

Sikuli extensions can be Python/Jython modules or Java classes.

For Java classes, following the reverse URL convention of Java is a good idea (for example, org.foo.your-extension). However, **DO NOT use Java's convention for Python/Jython modules**. You need to come up with a unique extension name that does not conflict with existing Python modules and other Sikuli extensions.

Please read Naming Python Modules and Packages to learn the details for naming a Python module.

How to test your extension

While developing your extensions, you can put the JAR file in Sikuli's extension directory or in the same .sikuli folder as your test script. The JAR file should not have a version number in its file name, e.g. `extension-name.jar`. Because Sikuli starts to search extensions in the .sikuli folder of the running script and then in the Sikuli extensions folder, it is usually a good idea to put your developing extensions in the .sikuli folder of your test script.

Another option is to use the *`load()`* function with an absolute path to your `extension-name.jar`. If this fails, Sikuli goes on searching in the current .sikuli folder and then in the Sikuli extensions folder. If load() succeeds, it returns `True` and puts `absolute-path-to-your-extension-name.jar` into `sys.path`, so you can use `import extension-name` afterwards.

How to contribute your extension

Currently you have to contact the developers of Sikuli and agree on how to proceed.

Sikuli Guide

New in version X1.0-rc2.

Sikuli Guide is an extension to Sikuli that provides a revolutionary way to create guided tours or tutorials for GUI applications. The revolutionary aspect is that the content of the tours or tutorials can be displayed right on the **actual interface**, rather than in a video or a series of screenshots on a web page. All this can be combined with guided user activities directly in the respective GUI applications using all the other Sikuli features.

Quick Start

First Example

In our first example, suppose we want to create a guided tour of this very documentation page you are currently reading. We want to bring your attention to the logo picture to the right. Using the functions provided by Sikuli Guide, we can write the following script to accomplish this:

When you run this script, Sikuli Guide will search for the logo's image on the screen, highlight it, and display the text ''This is Sikuli's logo" below the image, like the figure below:

Again, this happens in the **actual interface**, rather than in a video or a screenshot. The logo image that is highlighted is the actual interface element users can click on.

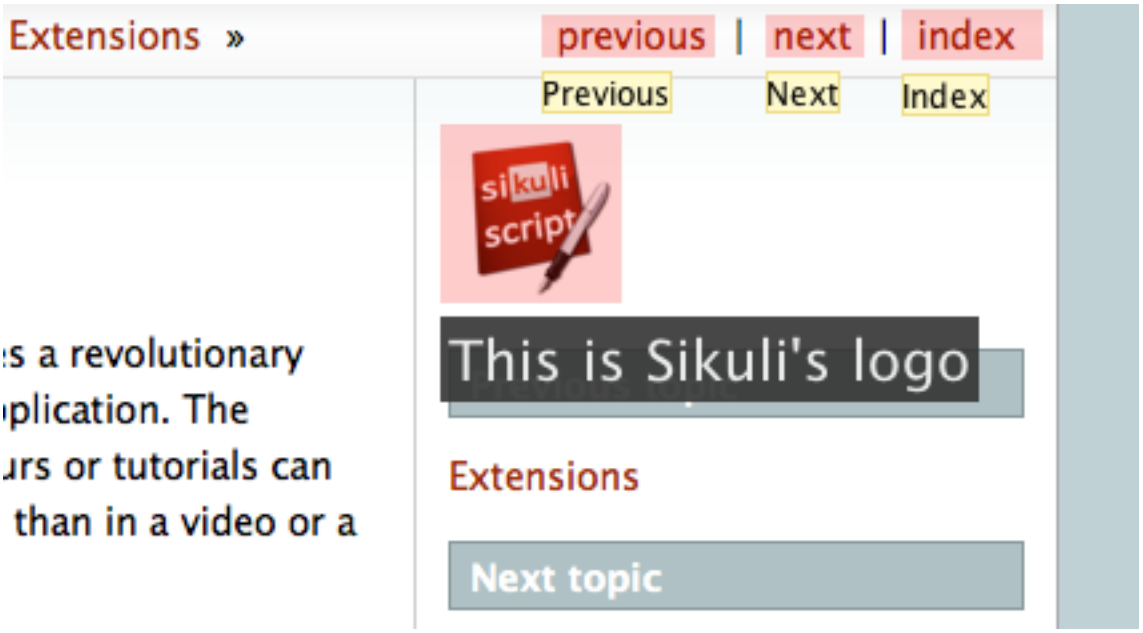
Let's explain the script line by line. The first line is an `import` statement that tells Sikuli to load the Sikuli Guide extension. The secod line uses the *`text(pattern, text)`* function to add `text` next to a given `pattern`, in this case, the logo image. Note that by default the text added is not displayed immediately, it is only internally added to the visual element. In the third line, we call *`show(secs)`* to explicitly tell Sikuli Guide to now display all registerd annotation elements (in this case only the text) for the duration specified by `secs`.

Below is a YouTube video of this example.

Adding Multiple Annotations

It is possible to add text or other annotations to multiple visual elements before calling *`show(secs)`* in order to show them on the screen at the same time.

The script above uses the function *`tooltip(text)`* to add tooltips to three links in addition to the text annotation. The result of running this script is shown below:



Rather than showing the annotations all at once, we can also show them one by one using separate `show()` statements. Below is an example where we cycle through the three links and show the tooltip of each link one at a time.

The result of running this script is shown below (sorry, no animation):

Adding Interaction

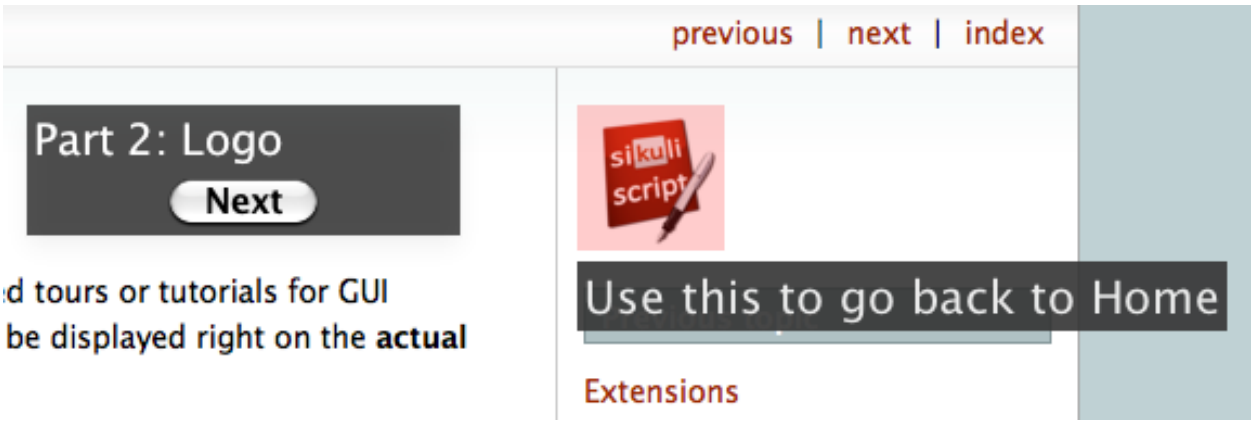
Another way to control the flow of a guided tour is to display a dialog box and let users click on a button to continue to the next part of the tour. Sikuli Guide provides a function `dialog(message)` to accomplish this easily. Below is an example using this function to create a two-part guided tour.

The tour presented by the script above introduces the navigation links above and the Sikuli's logo as a shortcut to go back to the documentation's HOME page. The function call `dialog("Part 1")` indicates the tour will show a

dialog that displays the message specified by the string argument (i.e., Part 1: Navigation Links). The following call to `show()` will actually display the dialog along with the text elements spcified earlier.

The figure below shows what happens after Line 3:

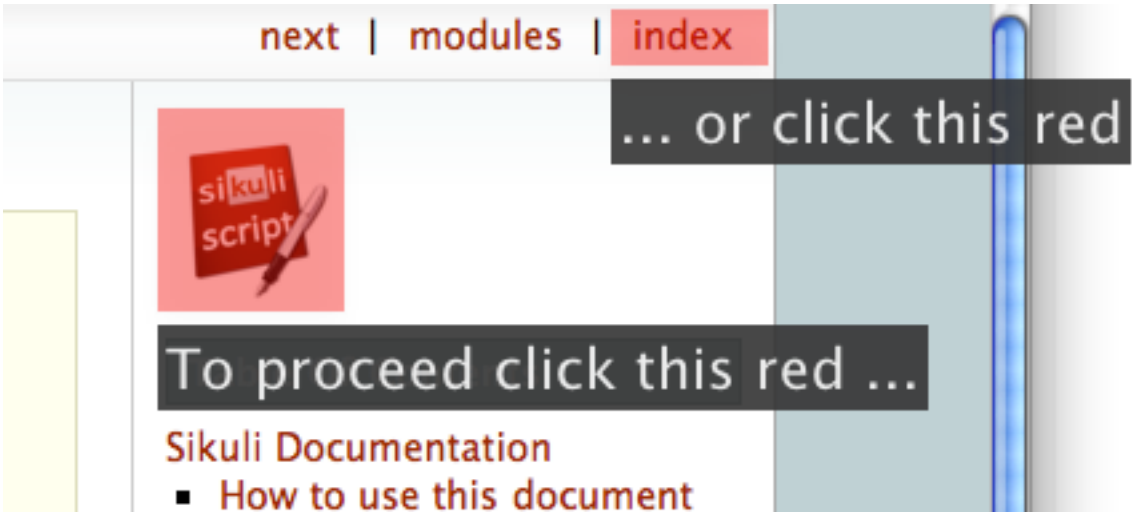
After users click on the **Next** button, the tour moves to the next part. The screen will look like below:



Making a Region clickable

You might use the feature `clickable(PSRM)`, to make a region sensitive for clicks.

The script waits until the user clicks one of the two highlighted areas.



Function References

element: when used as a parameter, it can either be something that can be used with a find() (Pattern or string as image file name or just plain text), a Region or Match object or another Guide element.

All funtions return the created element, so later the layout can be changed by specific setters or they can be used as target elements for other elements

Static Annotations

`guide.`**`rectangle`**(*element*)
Add a rectangular overlay as frame on the specified element's region.

Parameters **`element`** – a suitable

`guide.`**`circle`**(*element*)
Add a circle around the specified target's region.

Parameters **`element`** – a pattern, string, region or match

`guide.`**`text`**(*element*, *txt*)
Add some text to one edge of the specified element

Parameters

- **`element`** – a suitabel element
- **`txt`** – a string as text to display

`guide.`**`tooltip`**(*element*, *txt*)
Add a tooltip (small font in a light yellow box). same as text(), but with predefined layout. As usual for tooltips: the text should be a short oneliner

Parameters

- **`element`** – a suitabel element
- **`txt`** – a string as text to display

Interactive Elements

guide.**button**(*element*, *name*)
A clickable button showing it's name as the button text.

Parameters **name** – a string as text to display, later used as reference to check how the button was used

Control

guide.**show**([*seconds*])
Show static and interactive components added so far for the specified amount of time.

Parameters **seconds** – a decimal number as display duration in seconds

The default duration is 10 seconds. If interactive elements (either one or more clickable elements or a dialog box) were previously added, it waits until the user interacts with one of these elements. At this time all elements vanish and are discarded.

CHAPTER 8

Tutorials (not yet revised for version 1.1.0+)

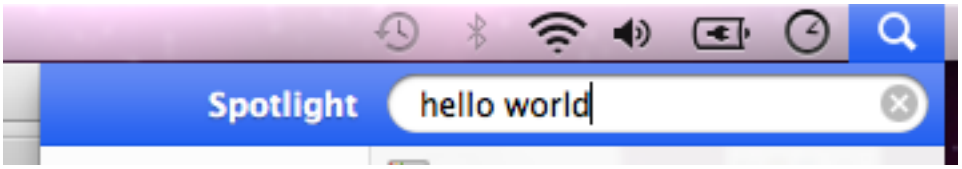
Hello World (Mac)

Let us begin with a customary Hello World example!

You will learn how to capture a screenshot of a GUI element and write a Sikuli Script to do two things:

1. Click on that element
2. Type a string in that element

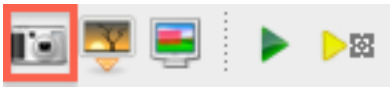
The goal of the Hello World script is to automatically type "Hello World" into the spotlight search box, like this:



Now, open the Sikuli IDE. We begin by taking the screenshot of our target, the Spotlight symbol in the upper-right corner, so that we can tell Sikuli Script what to look for and click on.

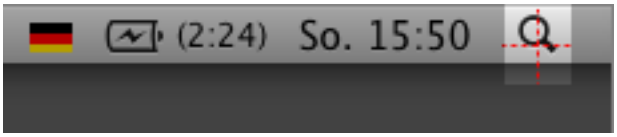
To simulate a mouse click on the Spotlight symbol, we are going to use the `click` function. To tell Sikuli how the target looks like, we need to capture the target's image on the screen.

Sikuli IDE provides two methods to capture screen images. The first method is to click on the camera button in the toolbar. This will bring you to the screen capturing mode.

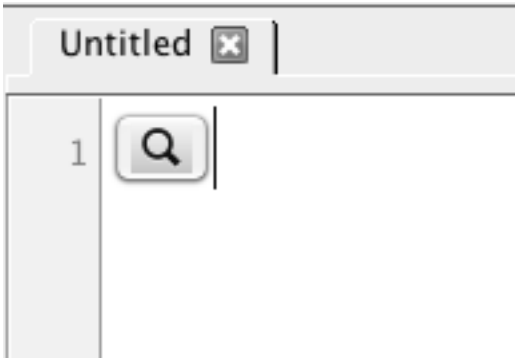


The second method is to press a hot-key (Command + Shift + 2). Often the target whose image you wish to capture may be covered by the Sikuli IDE's window. You can minimize the IDE's window and use this hot-key to switch to the capturing mode.

In the screen capturing mode, the screen will look darker and freeze momentarily. The entire desktop becomes like a canvas where you can draw a rectangle around the target you want to capture an image of. In this case, the target is the spotlight symbol. The cross of red dotted lines shows the center of the rectangle you just drew.

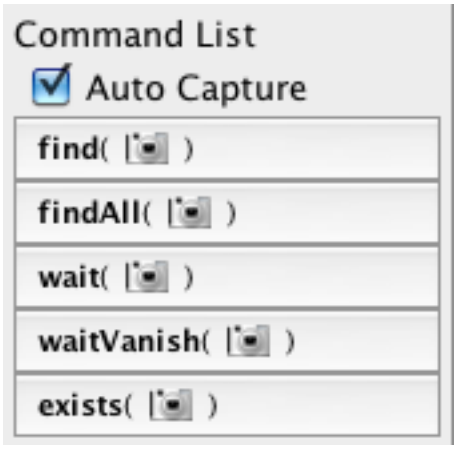


After you have drawn (or selected) a rectangle, the image within the rectangle will be captured and inserted into the script editor at the current cursor position.



Now, you can write the click function using this image as an argument to tell Sikuli to click on spotlight symbol.

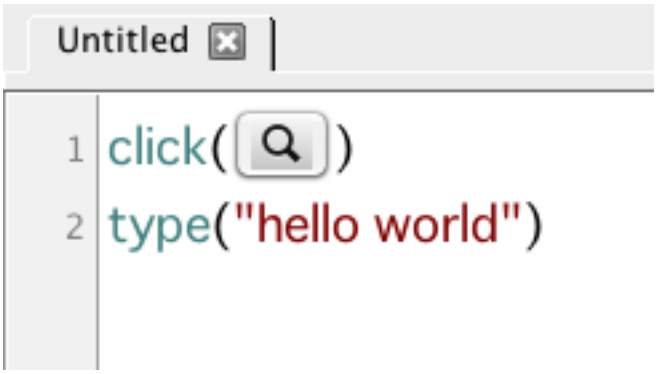
As a convenience, Sikuli IDE provides a *Command List* on the left panel. It shows a list of the most often used functions. Camera icons in the functions indicate these functions expect a captured image as an argument.



Locate the click() function in the list and click on it. If **Auto Capture** is on (default), you will be directed to the screen capturing mode in which you can capture an image of an interface target to be inserted into the click() function as an

argument.

The next step is to tell Sikuli to enter the string "Hello World" into spotlight's search box, which can be done with a simple `type` function.



This function will type the string given in the argument into whichever input control that has the focus. After clicking on the spotlight symbol, we can expect the spotlight search box will be the input that has the focus.

Congratulations! You have just completed your first Sikuli Script. Press the run button to see this script in action!

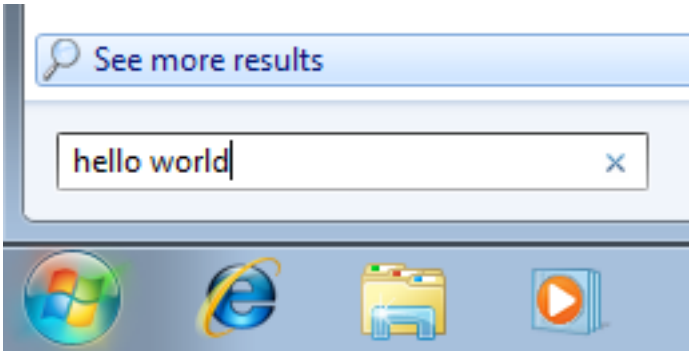
Hello World (Windows)

Let us begin with a customary Hello World example!

You will learn how to capture a screenshot of a GUI element and write a Sikuli Script to do two things:

1. Click on that element
2. Type a string in that element

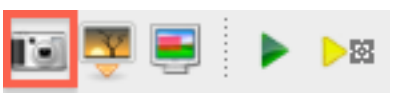
The goal of the Hello World script is to automatically type "Hello World" into the *Start* menu search box, like this:



Now, open the Sikuli IDE. We begin by taking the screenshot of our target, the *Start* menu symbol that is usually located in the lower-left corner of the desktop. Using this screenshot, we can tell Sikuli script what to click on.

To simulate a mouse click on the *Start* symbol, we are going to use the `click` function. To tell Sikuli how the *Start* symbol look like, we need to capture its image on teh screen.

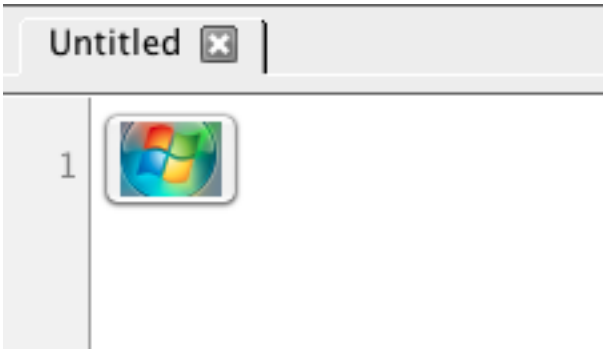
Sikuli IDE provides two methods to capture screen images. The first method is to click on the camera button in the toolbar. This will bring you to the screen capturing mode.



The second method is to press a hot-key (Ctrl + Shift + 2). Often the target whose image you wish to capture may be covered by the Sikuli IDE's window. You can minimize the IDE's window and use this hot-key to switch to the capturing mode.

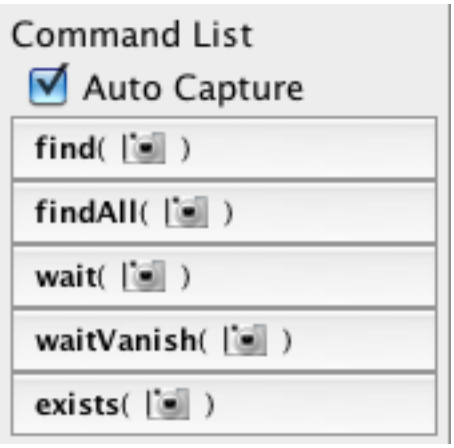
In the screen capturing mode, the screen will look darker and freeze momentarily. The entire desktop becomes like a canvas where you can draw a rectangle around the target you want to capture an image of. In this case, the target is the *Start* symbol. The cross of red dotted lines shows the center of the rectangle you just drew.

After you have drawn (or selected) a rectangle, the image within the rectangle will be captured and inserted into the script editor at the current cursor position.



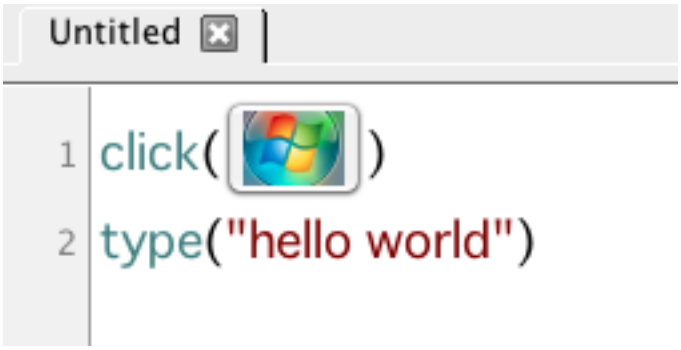
Now, you can write the click function using this image as an argument to tell Sikuli to click on start symbol.

As a convenience, Sikuli IDE provides a *Command List* on the left panel. It shows a list of the most often used functions. Camera icons in the functions indicate these functions expect a captured image as an argument.



Locate the click() function in the list and click on it. If **Auto Capture** is on (default), you will be directed to the screen capturing mode in which you can capture an image of an interface target to be inserted into the click() function as an argument.

The next step is to tell Sikuli to enter the string "Hello World" into search box, which can be done with a simple `type` function.



This function will type the string given in the argument into whichever input control that has the focus. After clicking on the *Start* symbol, we can expect the search box will be the input that has the focus.

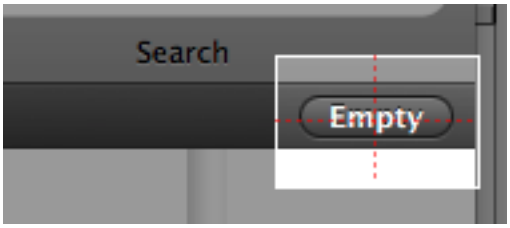
Congratulations! You have just completed your first Sikuli Script. Press the run button to see this script in action!

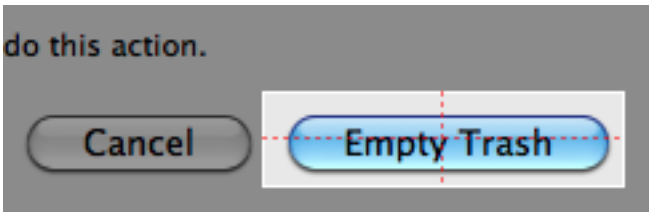
Goodbye Trash (Mac)

In this tutorial, we will write a Sikuli Script to automate the operation of clearing the content of the trash bin. On Mac OS X, this operation takes three steps:

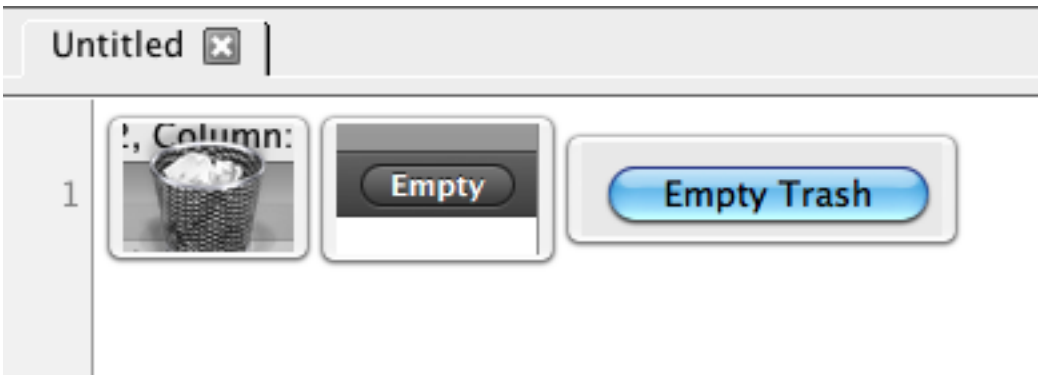
1. Click on the Trash icon in the dock
2. Click on the Empty button in the container window
3. Click on the Empty Trash button to confirm

To automate this operation, first we need to capture the images of the GUI elements that need to be clicked.

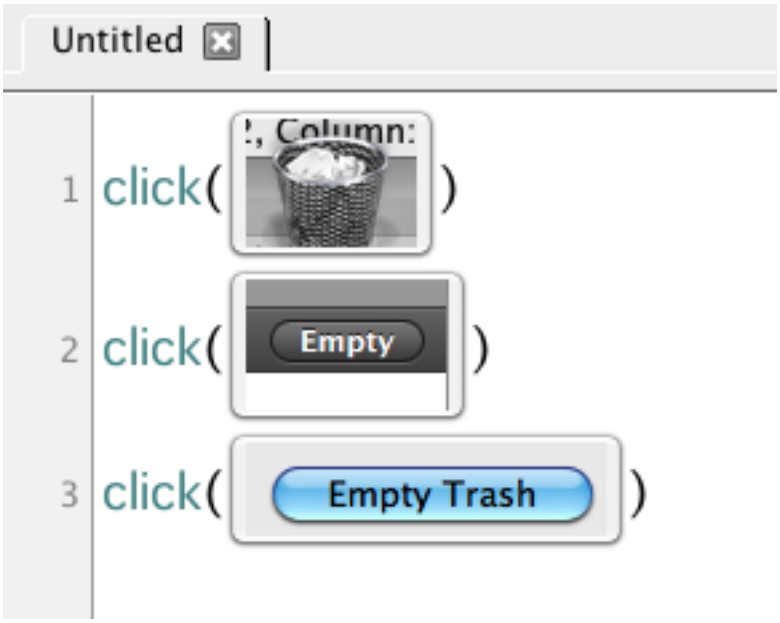




These captured images will be inserted into the Script editor.



Then, we can write a sequence of three `click()` statements to click on the three elements that need to be clicked in order.

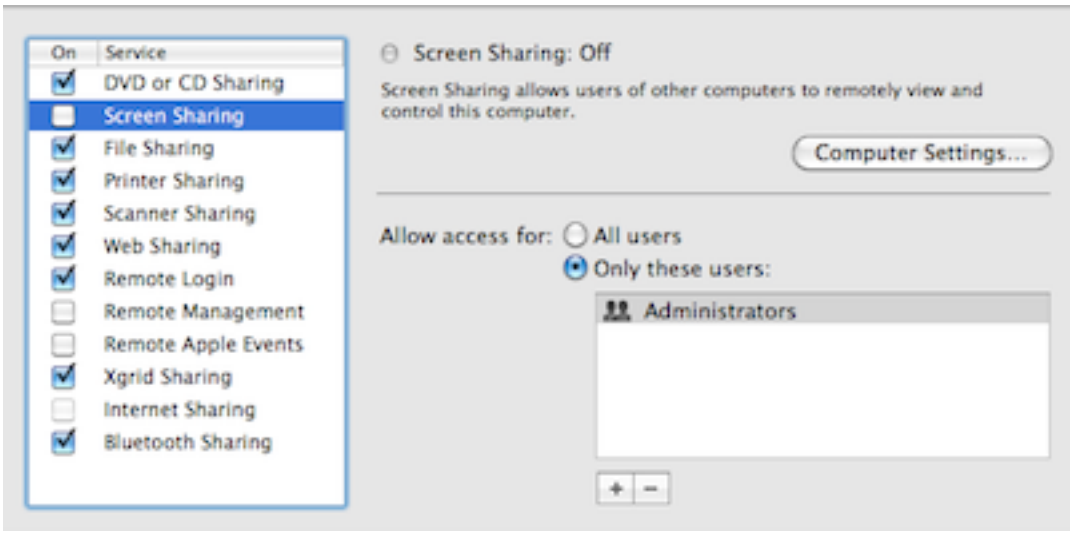


Notice how closely these three lines of code are mapped to the text description of the three-step operation earlier. We simply replace the description of each GUI element (e.g., Trash bin) by its image directly. How intuitive it is!

Before running the script, make sure the Trash icon is visible on the screen, otherwise Sikuli Script can not find it.

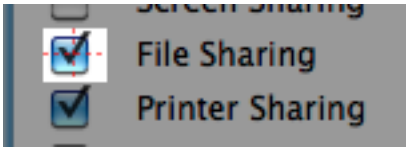
Uncheck All Checkboxes

In this tutorial, we will demonstrate how to use a `for` loop to interact with multiple instances of a GUI component. Suppose we want to uncheck all the check boxes in a window, such as the Sharing preferences window shown below:

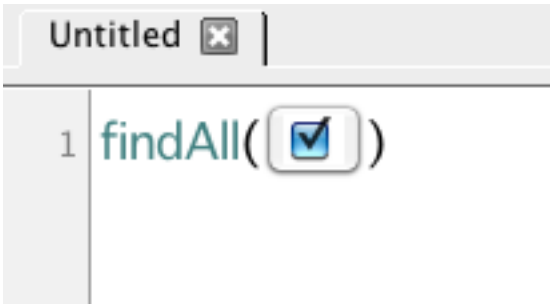


Unfortunately, there is no "uncheck all" function available. The solution? Write a Sikuli Script to look for ALL the checked items and uncheck them automatically. The function needed for this operation is *`findAll()`*.

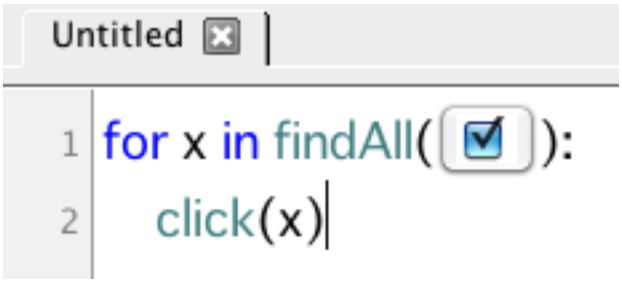
First, let's capture the screenshot image of a checked item.



Then, we can insert the image into the `findAll()` function.



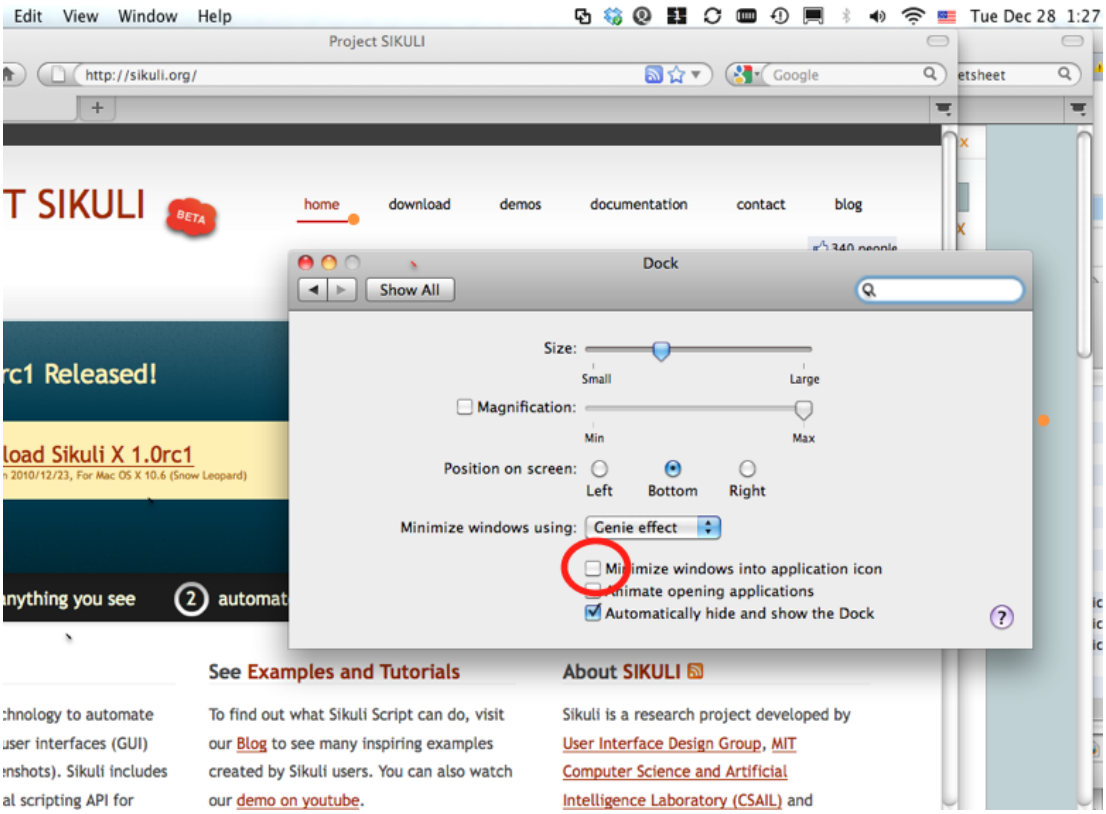
`findAll()` searches the entire screen for all the matching visual patterns and returns a list of locations of those similar patterns. This capability allows us to obtain all the checked items are on the screen. Then, we can simply write a for loop in standard Python syntax and call `click()` on each element in the list.



When this script is executed, Sikuli will find all the items that are currently checked and click on each item one by one in the loop.

Check the Right Checkbox

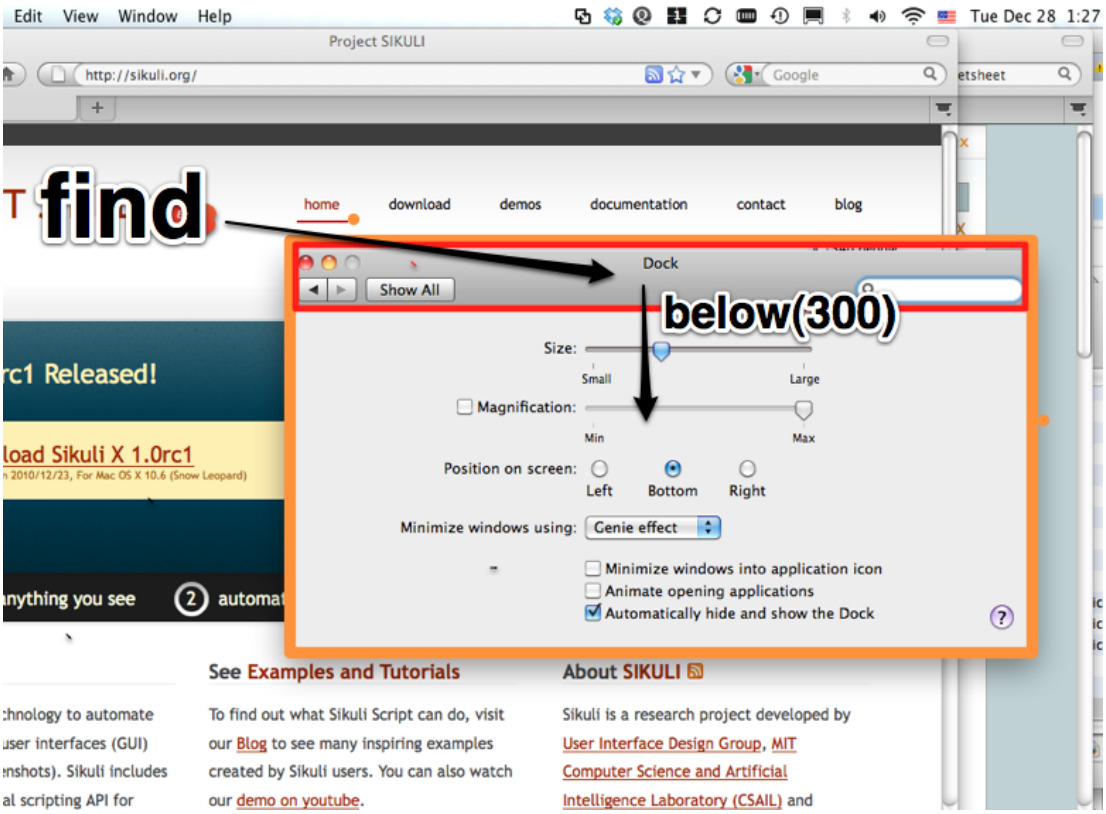
This tutorial demonstrates how to check a particular checkbox when there are multiple checkboxes in a window. Consider the following window, which is the window for setting the preferences for *Dock* on Mac, and we want to check the checkbox indicated by the circle that reads "Minimize Windows into Application Icons."



Simply looking for the image of the checkbox like below will not work.

Sikuli does not know which checkbox we are referring to; it simply clicks on the first one it finds. Below is what we will do instead.

First we want to obtain the region covering the content area of the *Dock* pane. One way to accomplish this is to use a spatial operator to obtain a region below the title bar of the *Dock* pane. The figure below illustrates this strategy.



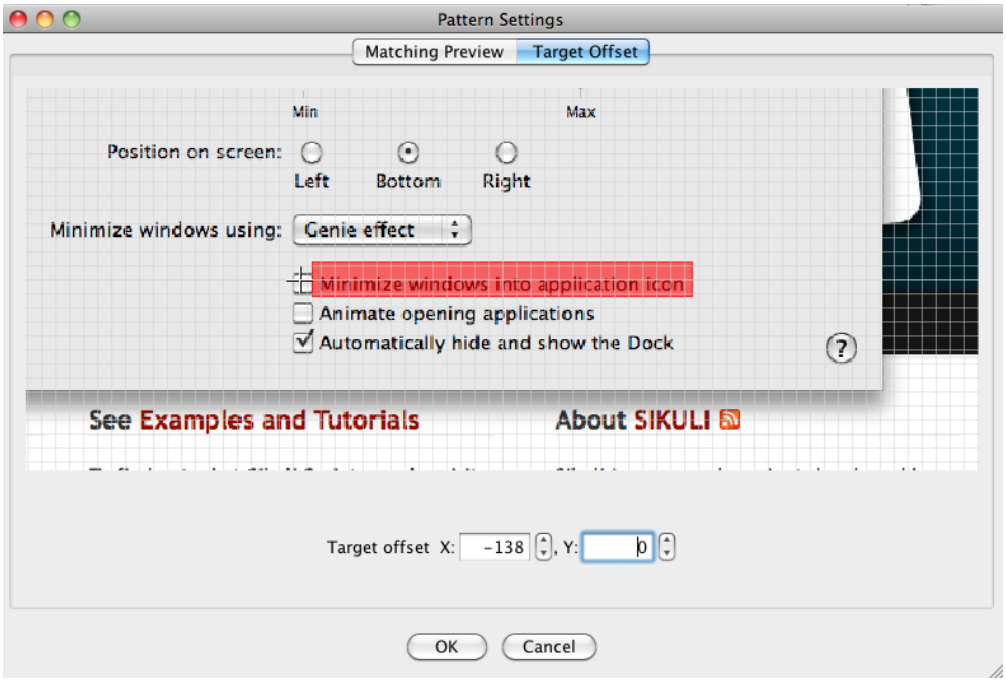
The Sikuli script to do this is:

It finds the title bar and then extend the matched region 300 pixels below, using the spatial operator `below`. The resulting region is assigned to the variable *r*, which is the orange rectangle in the figure above.

Next, we can search within the content region *r* for the label text of the checkbox we want to check and click on it.

If we do `click(t)`, Sikuli will click on the center of the label. However, what we want is to click on the right of the label where the check box is.

Sikuli IDE provides a convenient interface for specifying where to click relative to the center of a pattern. This is known as the *target offset*. The interface is shown below.



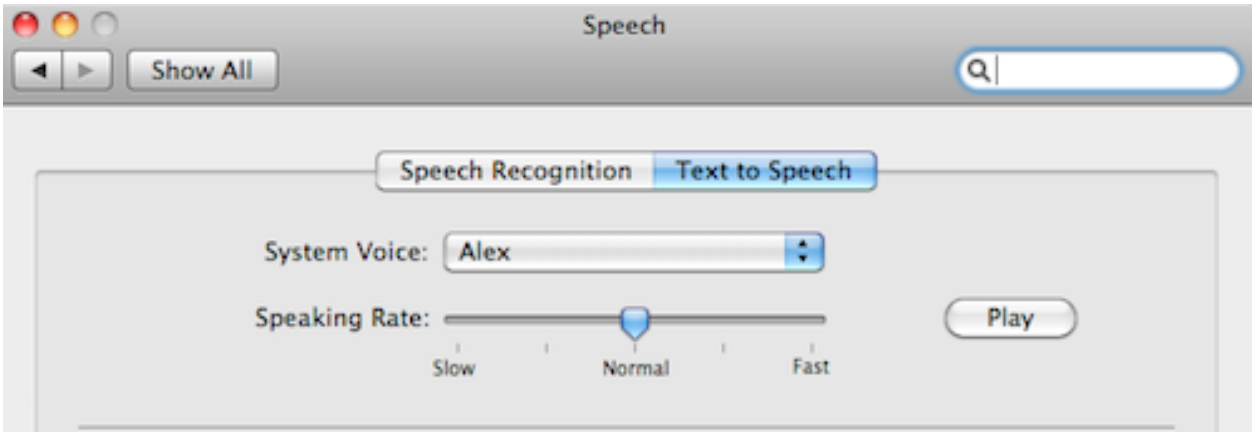
In this interface, we can click on the checkbox to indicate the desired location of the click point. In this example, the offset is then determined to be -137 in x, which means 137 pixels to the left of the center of the text label. After selecting the offset, the thumbnail in the script editor will be updated with a small red cross to indicate the new click point.

Then, the call `click(t)` will do the right thing, clicking on the checkbox instead of the center of the text label.

Working with Sliders

In this tutorial, we will learn how to use *`dragDrop()`* and *spatial operators* by writing a number of scripts to manipulate sliders.

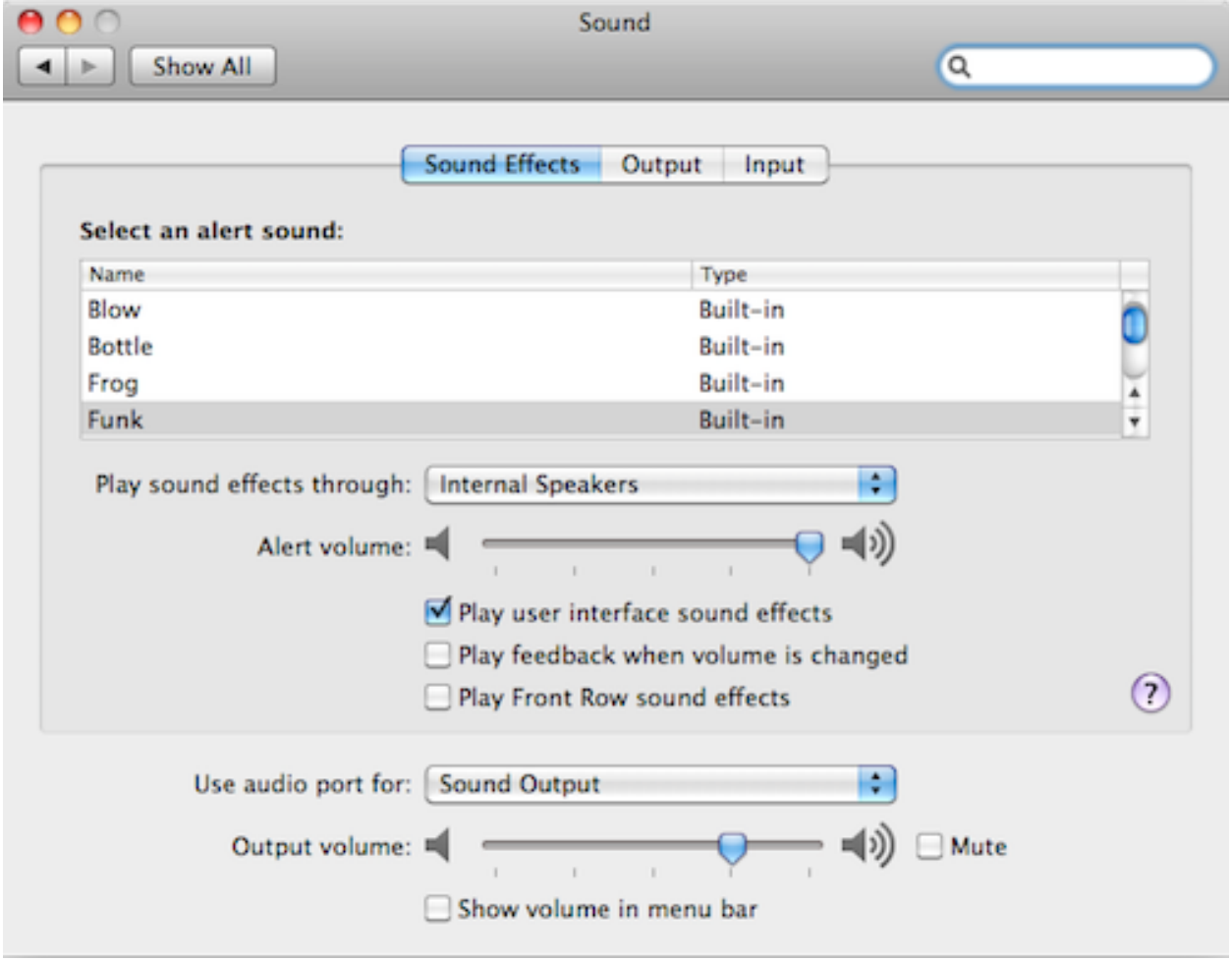
Suppose we wish to lower the speaking rate of the Text to Speech function. We want to drag the slider to the slow side in the Speech preferences window shown below.



The function that can perform the dragging is *`dragDrop()`*. This function takes two images as arguments. The first image describes the source GUI object to drag and the second image describes the appearance of the destination location where the GUI object should be dragged to and dropped.

Let us capture the source and destination images respectively.

What happen if there are more than two sliders. How can we make sure the right slider is dragged? The above example works because the particular window only has one slider. How can we deal with cases when there are several instances of similar looking GUI components? Let us consider the Sound preferences window shown below.



Suppose we wish to lower the Alert volume. To make sure Sikuli Script drags the right slider, we need a way to tell Sikuli Script to look for the one that is to the right of the Alert volume label, not the one next to the Output volume label. Sikuli Script provides a set of spatial operators to do exactly this. Here we will apply the `right()` operator as follows.

This statement tells Sikuli Script to first find the Alert volume label and then find the slider thumb only within the region strictly to the right of the result of the first find. The slider thumb found is then stored in the variable t. Now that we have identified the desired slider thumb, we can call `dragDrop()` to drag it to the left by giving the image of the Alter volume as the target.

In the above example, we use the image of the Alert volume label to implicitly guide the direction of dragging to the left. It is also possible to use relative coordinates to explicitly drag to the left, as shown below.

Here, the (x,y) coordinates of the slider thumb are accessible as the attributes of t. We can thus calculate the position 200 pixels to the left of t and ask Sikuli Script to drag the thumb to that position.

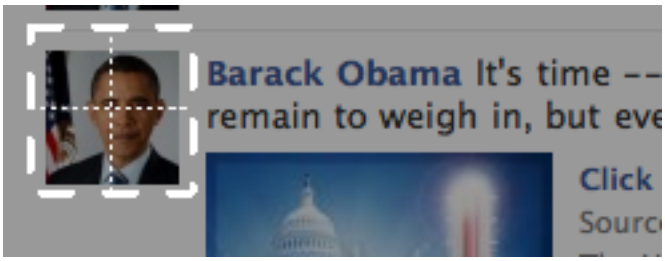
Desktop Surveillance

Sikuli Script can be used to perform **desktop surveillance**. In this tutorial, we will go through a number of exercises to create scripts that can monitor the screen and notify us when certain interesting visual event happens.

Facebook App

The first exercise is to create a Facebook app to periodically check our Facebook homepage visually and see if a particular friend has recently updated the status message. One easy way to detect this event is to look for the friend's face image on our Facebook homepage. If found, the friend must have posted a new status message. If not found, we should check back again in a few moments.

Let's implement this operation using a `while:` loop. First we need to capture a screenshot of the friend's face image.



To check if the friend's face can be seen on the screen, we use *`exists()`*, which returns True when the face image is found. We set the looping condition to Not Found so that the while loop will terminate only when the face image is found. We add a `sleep(n)` statement in the body of the loop to introduce a 5 second interval between attempts to look for the face image on the screen.

Alternatively, Sikuli provides a convenient *`wait()`* function that periodically checks the screen to wait for a given image pattern to appear. Using wait(), the above code can be rewritten as:

The constant *FOREVER* means we want Sikuli to wait indefinitely. If we do not want to wait forever, we can replace *FOREVER* with a number to indicate the number of seconds to wait until Sikuli should giveup.

After the while loop exits or the wait function returns, we can call *`popup()`* to display a notification message.

This will display a popup message that looks like below:

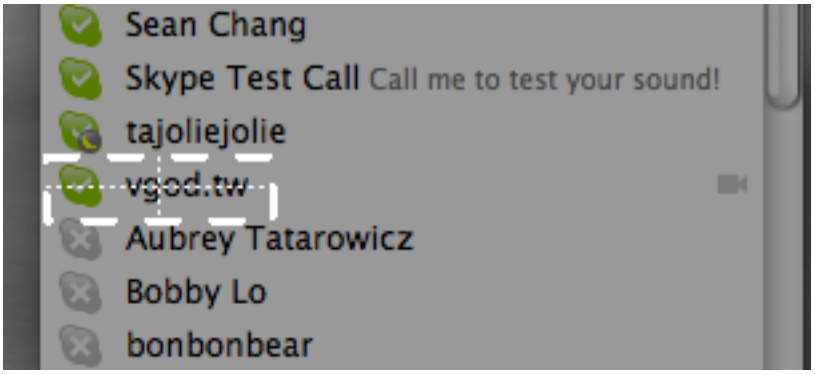
Now we can run this Sikuli Facebook App, sit back and relax, and get notified when our friend updates his message.

Skype App

In the previous exercise, we wrote a script to detect an image's appearance. In this exercise, we will do the opposite — *detecting the disappearance of a visual pattern*.

Skype is a great tool that allows us to stay in close contact with our friends even if they are in remote parts of the world. However, there might be some unfortunate circumstances we may want to avoid being seen online by a certain individual. Perhaps the individual talks too much. Perhaps we owe the individual some money. It would be nice to know when the individual is offline so that it is safe to get online. While Skype does provide a feature to notify us when an individual is online, there is no notification feature when the opposite happens.

An automatic logoff notifier would be desirable to deal with this situation. Let us build this tool using Sikuli Script. Notice that if an individual is no longer online, the combined visual pattern of the green status icon and the individual's screen name will disappear. Thus, we can take a screenshot that includes both the green icon and the screen name as follows.



Then, we can write a Sikuli Script to watch for the disappearance of the screenshot image we just captured.

This script is very similar to the one in the previous exercise. The only difference is the removal of the NOT operator from the condition statement for the while loop, since we are trying to do the opposite.

Another way to wait for the disappearance of an image is to use the `waitVanish()` function. The above script can be rewritten as follows:

Bus Arrival Notifier

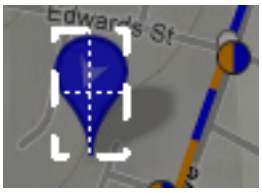
The third exercise is to build a bus arrival notification tool. For many bus riders, online GPS-based tracking services are very useful. Instead of patiently standing outside at a bust stop, braving the freezing wind in the winter or scorching sun in the summer, riders can sit comfortably inside in front of their computers, checking emails, updating Facebook status, or watching YouTube? videos, or what have you. They only need to look at the map every few moments to check the location of the bus symbol on the map. Only when the bus is close enough do they have to finally get out and walk to the bus top.

Since we care about whether the bus is getting close to the stop, we only need to look at the neighborhood around the stop. Thus, we can resize the browser to show just that portion of the map, while leaving a lot of screen space to do something else, in this case, reading CNN news.

Let us write a Sikuli Script to do the bus tracking for us. It is possible to define a region and ask Sikuli Script to focus only on that region to search for a particular visual pattern. This way, Sikuli don't have to waste precious computing cycles scanning the whole screen. To do so, let us click on the "select a region" button in the toolbar as indicated below.

The entire screen will freeze and turn darker, similar to what happen in the screen capture mode. Simply draw a rectangle to cover the entire neighborhood map. The region covered by this rectangle is visually represented as a thumbnail image of the entire desktop where the region is shaded in red.

Next, we capture the screenshot image of the bus symbol so that we can tell Sikuli Script to watch for its appearance.



Now. we can write the following script to check the map and watch for the appearance of a bus symbol:

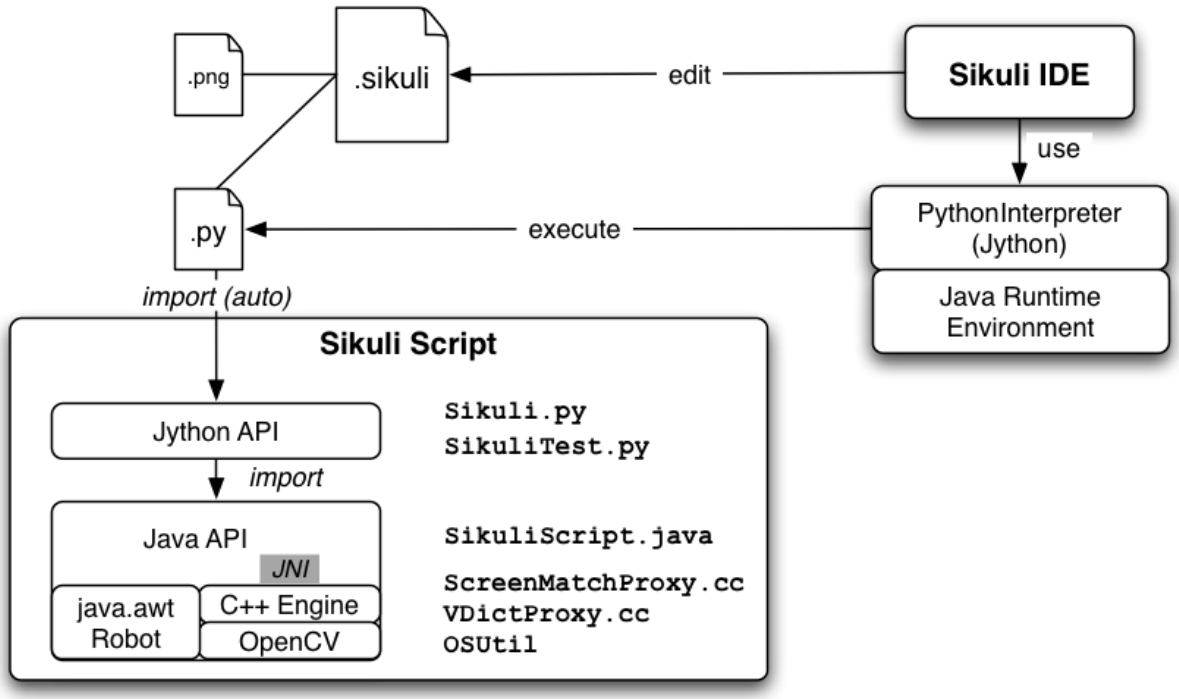
Interpreting the meaning of the while loop is straightforward—while it is not the case that a bus symbol can be found inside the region, sleep for 5 seconds. In other words, the while loop will exit only when the bus symbol is found inside the region. Then, the popup statement will be executed to notify us the bus has arrived.

Again, the same effect can be achived using `wait()`. The script can be rewritten as:

CHAPTER 9

For Developers

How Sikuli Works



Sikuli Script

Sikuli Script is a Jython and Java library that automates GUI interaction using image patterns to direct keyboard/mouse events. The core of Sikuli Script is a Java library that consists of two parts: java.awt.Robot, which delivers keyboard

and mouse events to appropriate locations, and a C++ engine based on OpenCV, which searches given image patterns on the screen. The C++ engine is connected to Java via JNI and needs to be compiled for each platform. On top of the Java library, a thin Jython layer is provided for end-users as a set of simple and clear commands. Therefore, it should be easy to add more thin layers for other languages running on JVM, e.g. JRuby, Scala, Javascript, etc.

The Structure of a Sikuli source folder or zipped file (.sikuli, .skl)

A Sikuli script (.sikuli) is a directory that contains a Python source file (.py) representing the automation workflow or the test cases and all the image files (.png) used by the source file. All images used in a Sikuli script are simply a path to the .png file in the .sikuli bundle. Therefore, the Python source file can also be edited by any text editor.

While saving a script using Sikuli IDE, an extra HTML file may optionally be created in the .sikuli directory so that users can share a visual copy of the scripts on the web easily.

A Sikuli zipped script (.skl) is simply a zipped file of all files in the .sikuli folder. It is intended for distribution via mail or web upload, can also be run from command line and reopened in the Sikuli IDE. (The previous naming as "Sikuli executable" is deprecated, since this is misleading: people most often thought, it is something like a self-contained and self-running package comparable to a Windows EXE, but it is not).

How to get involved

Have fun working with SikuliX? You can do more than just being a user! There are many ways you can help SikuliX's development:

- **Blog or tweet** about SikuliX. Share your cool scripts and ideas to the world and let more people know how cool SikuliX is.

  Don't forget to integrate a link to the official homepage of SikuliX **http://sikulix.com**

- **Report bugs or request new features** in our bug tracker.
- Visit the question board and the bug tracker regularly and **answer people's questions** there or comment posted bugs. (You may want to subscribe to the bug tracker or subscribe to all questions.) Many people have questions that you may know how to deal with. Help them to get through the obstacles and they may help you in the future.
- *Submit patches* to fix bugs or add features.
- **Translate SikuliX** into your language and help more people who speak different languages to access SikuliX. You can help us to
  - translate SikuliX IDE's interface and SikuliX's messages, or
  - translate SikuliX's documentation (this site you are reading).

  Read *Documentation, internationalization and localization* for more details.

Submit Patches

If you are interested in making SikuliX better: Any patches to SikuliX are always welcome

1. Fork the SikuliX source tree from GitHub

- Version 1.1.x
- Version 1.2.x

1. Fix bugs or add new features.

2. Post pull requests
3. In any case please leave documentation in the comment area.

Documentation, internationalization and localization

We hope SikuliX can be used by anyone from anywhere in the world. You can help us to translate the user interface of SikuliX IDE, any SikuliX messages or to improve and translate this documentation.

SikuliX Translation

If you find an incorrect or missing translation, or if you would like to add a language that isn't yet translated, here's you can do:

When ready, Visit our translations page on transifex and use the provided tools to translate the relevant items.

It is also possible to post pull requests against the sources on GitHub.

Documentation and it's Translation

This documentation is created using Sphinx, and written in the reStructuredText format. You can view how the source code looks like using the link `Show Source` in the side bar.

The docs are hosted on readthedocs with the sources being on Github.

Changes or additions to the docs should be posted as pull requests on GitHub.

If you want to contribute to the docs or it's translation on a regular base, you can be added as a maintainer on readthedocs. In case use one of the contacts on sikulix.com.

Python Module Index

g

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W