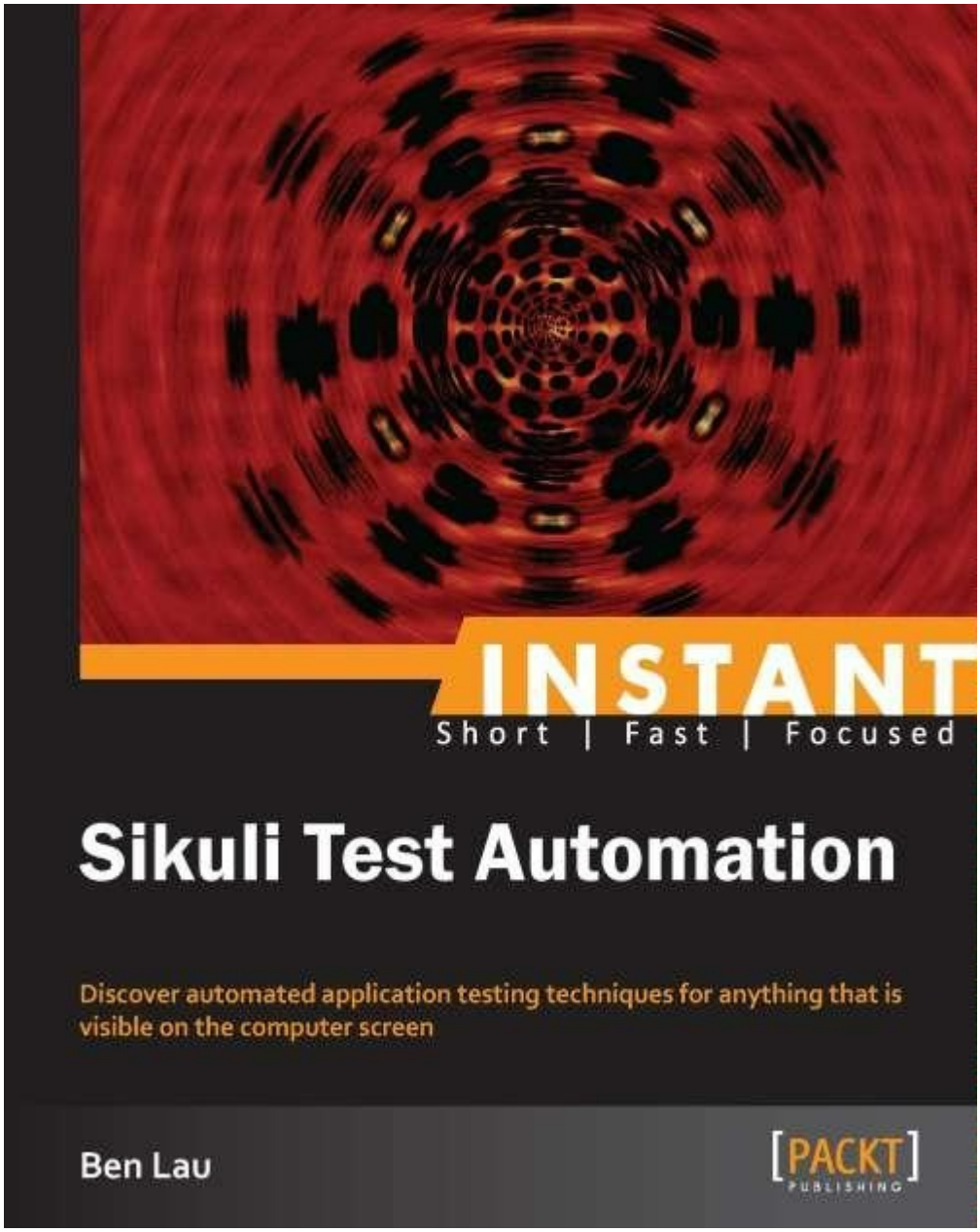

INSTANT
Short | Fast | Focused
Sikuli Test Automation
Discover automated application testing techniques for anything that is visible on the computer screen
Ben Lau
[PACKT]
PUBLISHING

Table of Contents

Instant Sikuli Test Automation

Instant Sikuli Test Automation

First published: July 2013

Production Reference: 1220713

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78216-787-7

www.packtpub.com

Credits

Author

Ben Lau

Reviewer

Purna Chander

Acquisition Editor

Kartikey Pandey

Commissioning Editor

Sruthi Kutty

Technical Editors

Rikita Poojari

Hardik B. Soni

Project Coordinator

Michelle Quadros

Proofreader

Lesley Harrison

Production Coordinator

Pooja Chiplunkar

Cover Work

Pooja Chiplunkar

Cover Image

Sheetal Aute

About the Author

Ben Lau has worked as a software and build engineer since the mid 90s. He began his career as an intern for a major university in Southern California, working on virtual reality training systems, and continued there for almost 10 years, working on a variety of projects. After leaving the university to join a spin-off developing language training applications for the military, he began to specialize in build and release engineering and also branched out into automated testing. He is currently employed at a midsized startup in the Los Angeles area, working as their build and release manager.

I'd like to thank my spouse for her encouragement while producing this book as well as the whole Sikuli community for creating and supporting this wonderful tool.

About the Reviewer

Purna Chander has more than 10 years of experience in software testing the client/server and web-based applications. He has a Bachelor's degree in Mechanical Engineering. His goal is to reduce the manual effort of testing the applications to bring down the time from days to hours and hours to minutes. He has also worked with clients such as Google and Yahoo as part of software testing to build customized tools based on PERL/PYTHON. He holds a certification in PERL from BrainBench and also in CHFI (EC-Council.org) and implemented security testing (http://www.eccouncil.org/).

He lives with his wife, Latha; daughter, Lakshmi; and son, Likhith.

Sikuli is his favorite R&D tool, and he implemented this tool while working with KONY. At that time, there were no commercial or open source tools for testing mobile applications. Sikuli, along with Python, helped him to build a test automation framework that does tests on iPhone, Andriod, BlackBerry (touch and non-touch) and Windows emulators. He has worked with Hypersoft Technologies, Mindtree.com (clients Yahoo and Google), Kony.com, and HostAnalytics.com.

I would like to thank my parents and my wife for supporting me.

www.packtpub.com

Support files, eBooks, discount offers and more

You might want to visit www.packtpub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

Free Access for Packt account holders

If you have an account with Packt at www.packtpub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Chapter 1. Instant Sikuli Test Automation

Welcome to *Instant Sikuli Test Automation*. This book has been specially created to provide you with all the information that you need to get started with Sikuli, and develop automated testing scripts. You will learn the basics of Sikuli, get started with building your first test, and discover a variety of tips, tricks, and techniques for using Sikuli for automated testing.

This book contains the following sections:

So, what is Sikuli? tells you more about Sikuli, which is an automation tool, that uses a combination of screenshots and computer vision techniques to allow control of any application on the screen.

Installation explains how to download and install Sikuli with minimum fuss, and then set it up so that you can use it as soon as possible.

Quick start – writing your first script teaches you how to create your first automated script using Sikuli, and demonstrates how to use its most basic features.

Top 13 features you need to know about helps you explore the most important features of Sikuli, and learn to use it like a pro. By the end of this section you will be able to:

- Write simple tests using Sikuli's basic features
- Create more complex tests using flow control, events, and good methods for structuring tests to allow code reuse
- Build a library of reusable test fragments that will reduce the maintenance burden for a large test suite
- Put all of the previously mentioned techniques together into a framework for running your tests and reporting results

People and places you should get to know provides you with many useful links to the project page and forums, as well as a number of helpful articles, tutorials and blogs, as every open source project is centered around a community.

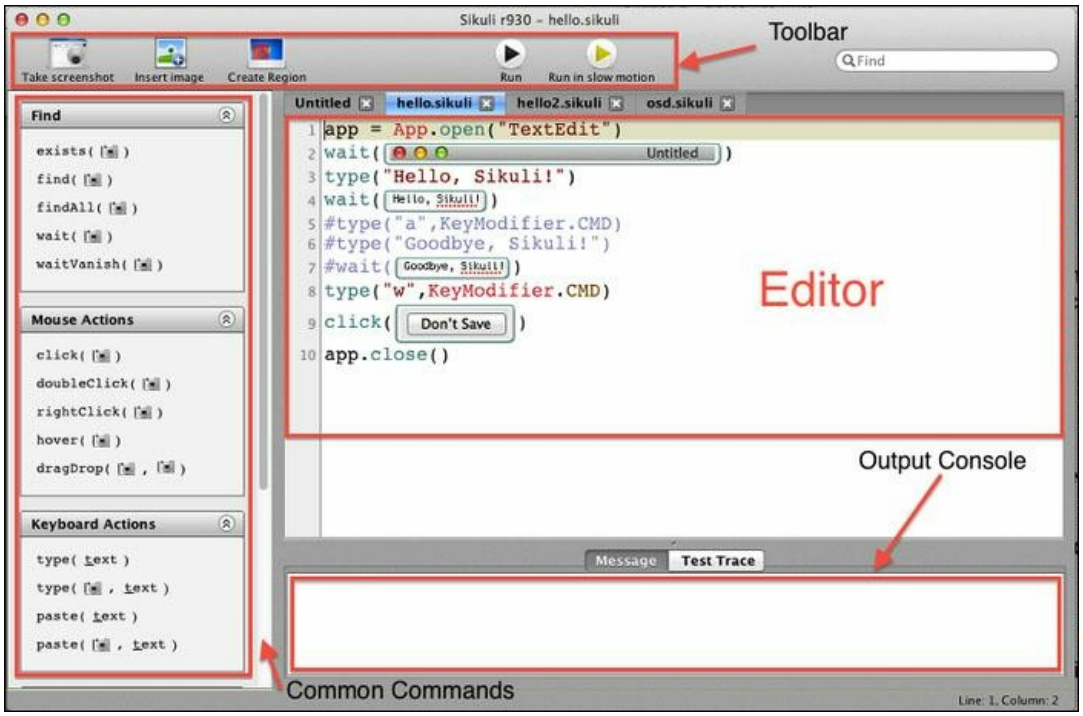
So, what is Sikuli?

Sikuli allows you to automate anything you see on the screen, using screenshots as a means of control. It provides you with an editor for creating scripts using its API in combination with the Python programming

language to give its users a great deal of flexibility in how they construct their scripts.

What can you do with Sikuli? That all depends on how much effort you want to put into the script. You can crawl through websites using it, when tools like **cURL** don't work. You can watch the screen for particular features to appear, and then execute scripts in response. Since Sikuli provides you with a rich programming environment via Python, the only limit is your ability to figure out how to accomplish something.

The Sikuli IDE provides you with your primary interface for working with and executing Sikuli scripts. Here, we can get a quick overview of the components of the UI:



The toolbar contains buttons for common actions in Sikuli. From left to right these are:

- **Take screenshot**: This is used when capturing portions of the screen for use in your scripts
- **Insert image**: This lets you select an existing image to search for on the screen
- **Create Region**: This lets you create a region on the screen to reference in your scripts
- **Run**: This will execute your script
- **Run in slow motion**: This will execute your script in slow motion, which can be extremely handy for debugging Sikuli scripts
- **Find**: This lets you search for text in your Sikuli script code

The bar on the left provides a list of common commands and their arguments. You can double-click on any of these commands, and they will be inserted into your script. For example, if you double-click on the `exists()` command, it will insert it into your script, and immediately put Sikuli into the capture mode to make a screen selection. This selection will then be inserted into the `exists()` command in your script.

The editor provides you with a way to create and change your scripts. For every open script there is a tab. Generally speaking, you'll want to use Sikuli's editor for writing scripts, since it provides you a good way to see what your scripts are doing and interact with the screenshots. The editor has a few quirks, but considering the additional features it provides, specific to Sikuli (and the need to synchronize multiple files in the underlying file format), it is an important tool for working with Sikuli.

Finally, there's a small console at the bottom of the UI, which provides feedback about running scripts. If your script generates console output or fails due to an error, it will be noted here.

Sikuli provides an excellent platform for automated testing of applications. Through the use of screenshots and simple commands, it makes it much easier for non-programmers to create tests, and provides a good introduction for testers to the greater power that scripting provides. Testers can start out simply, and add more complexity as they become familiar with the tool and need to expand their ability to do more complex testing. Since Sikuli uses a common programming language instead of something unique to the tool, there is a wealth of information available on the Internet about learning how to program using Python.

Sikuli can integrate with your existing testing tool pipeline. Using Python, it is possible to have your Sikuli tests do things like automatically submit test results to an existing test tracking system such as JIRA, Zephyr, or Testopia. This would require some additional scripting in Python, but with a little bit of effort, it is possible to integrate Sikuli with any tools of this kind that support access via some sort of web-based API.

Installation

Let's look at the steps involved for installing Sikuli on your computer. We'll be reviewing how to install Sikuli on Mac OS X and Windows. The examples for the book are available for both Mac OS X and Windows.

Installation on Mac OS X

Let us have a look at the steps involved in installing Sikuli on systems with a Mac OS X.

Step 1 – what do I need?

Download and install Java if you don't already have it. Java is available from Oracle at http://java.com.

Step 2 – downloading Sikuli

Download Sikuli for Mac OS X using the following link:

https://launchpad.net/sikuli/sikuli-api/1.0.0/+download/Sikuli-IDE-1.0.0-Mac.zip

You can get a complete list of downloads available here:

https://launchpad.net/sikuli/+download

Step 3 – extracting the Sikuli IDE

Open a terminal and extract the `Sikuli-IDE-1.0.0-Mac.zip` file into the `Applications` directory (this is where the scripts in this book are expecting it to live) using the following command:

```
unzip ~/Downloads/Sikuli-IDE-1.0.0-Mac.zip
-d /Applications/Sikuli-IDE
```

If you've already got a copy of Sikuli installed, you might want to back it up first, as follows:

```
mv /Applications/Sikuli-
IDE.app /Applications/Sikuli-IDE.app.bak
```

Step 4 – starting Sikuli

Now, check if the Sikuli IDE starts. In your terminal, run the Sikuli IDE using the following command:

```
/Applications/Sikuli-IDE.app/sikuli-ide
```

There is also a version of **Automator** application available if you only want the IDE. But, this will prevent the testing framework examples, later in this book, from being able to function properly.

Some of the examples in this book use Firefox to show things which are not easily demonstrated within TextEdit (which is used for most of the books, since it is included with the OS). Firefox can be downloaded from http://firefox.com.

Installing Sikuli on Windows 7

If you are a Windows 7 user, perform the following steps for installing Sikuli on your system.

Step 1 – what do I need?

Download and install Java if you don't already have it. Java is available from Oracle at http://java.com.

Step 2 – downloading Sikuli

Download the Sikuli IDE for Windows, and be sure to get the 32-bit version from:

https://launchpad.net/sikuli/sikuli-api/1.0.0/+download/Sikuli-IDE-1.0.0-Win32.zip

You can get a complete list of downloads available here:

https://launchpad.net/sikuli/+download

Step 3 – extracting the Sikuli IDE

Unzip your copy of Sikuli (right-click and select **Extract files…**). The examples that go along with this book assume that this is located your `Downloads` directory, using the folder name `Sikuli-IDE-1.0.0-Win32`.

Step 4 – starting Sikuli

Check that the Sikuli IDE is working correctly by opening the directory that Sikuli is installed in, and then double-clicking on the `sikuli-ide.cmd` file to start it.

For the testing framework examples later on, you will need to install a copy of **Cygwin** from http://cygwin.com. Also, some of the examples in the book make use of Firefox (http://firefox.com), Chrome (https://www.google.com/chrome), and VLC (http://www.videolan.org/vlc/index.html) that must also be downloaded and installed, or the copies on your computer can also be used if you have these programs already.

These applications are used to demonstrate some of the Sikuli features that can't be shown when using WordPad (which is used for most of the examples, since it comes preinstalled with the OS). VLC is used to show how to capture video of your test runs as a debugging aid.

Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.PacktPub.com. If you purchased this book elsewhere, you can visit http://www.PacktPub.com/support and register to have the files e-mailed directly to you.

Quick start – writing your first script

We'll now show you how to write your first script. The complete process will be shown for both Mac OS X and Windows. The rest of the book will focus on OS X, but a complete set of examples is available for Windows as well. Along with each example, there will be a note indicating which example file to refer to.

Your first script with Sikuli on Mac OS X

In this section, you will be introduced to Sikuli through a simple example. In the end, you should be ready to start using Sikuli for simple test automation.

So let's jump in! To get started, we're going to build an extremely simplistic testing script based around an application that ships with OS X called TextEdit. This will act as a simple "Hello World" example, and show what a simple script would look like, and how you might go about assembling it. Writing Sikuli scripts is a highly iterative process, and you will often end up executing your scripts dozens or hundreds of times while developing them (depending on their complexity). Here's the complete script (`hello.sikuli`), so that you can see what the entire thing looks like before we begin:

Step 1 – starting an application

This is the very first thing that most scripts will need to do. This is already utilizing some of the Python capabilities in Sikuli, so we can retain a reference to the application for later use:

1. Start Sikuli, and select the editor.
2. Start the TextEdit application using Sikuli.

```
1 app = App.open("TextEdit")
```

Try running this, and see what happens. You should see a new TextEdit window appear.

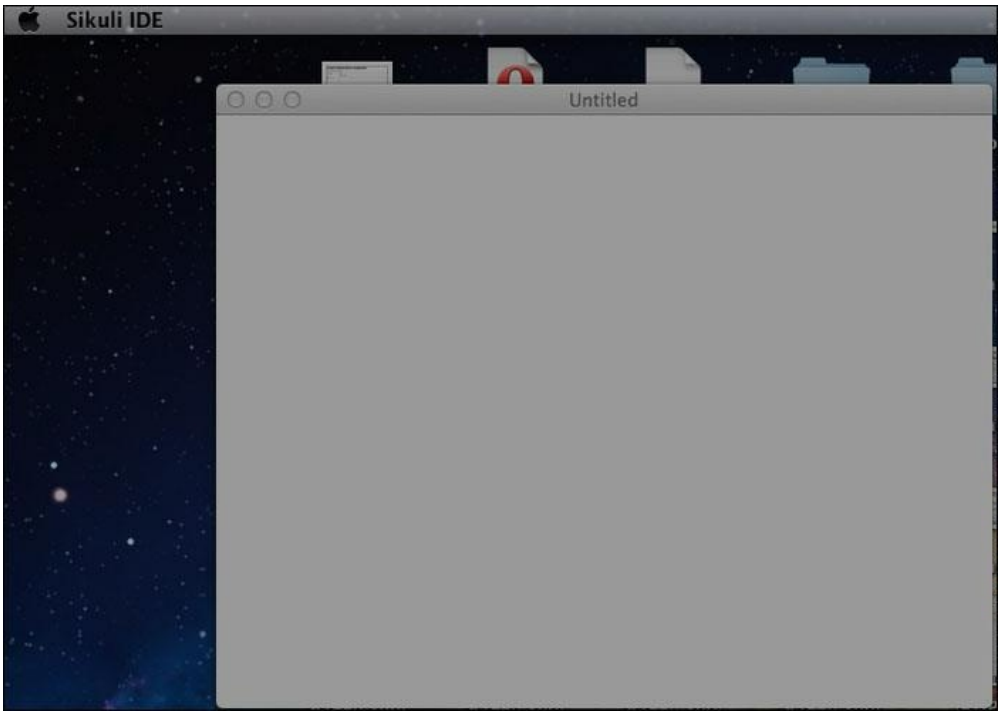
There are a couple of things to be noted:

- If you ever need to stop the Sikuli script from running, use *Command* + *Shift* + *C* (on Windows, use *Alt* + *Shift* + *C*). Sometimes, scripts can get stuck in a loop, and you will need to be able to exit from them even though they have not yet completed.
- On Windows, you will often need to provide the full path to an application to start it. On Mac OS X, you can start Firefox using `App.open("Firefox")`. However, on Windows, you will need `App.open(r"C:\Program Files\Firefox\firefox.exe")`. The character `r` before the string is important, if you do not use it, you'll have to escape the spaces in the path.

Step 2 – checking whether the application has started

When writing scripts with Sikuli, you will want to check that the results on the screen are similar to what you expect. For this, we use the `wait()` method as follows with a screenshot (your first of many):

1. Run the part of the script we created in *Step 1 – starting an application*.
2. Type `wait(` in the editor.
3. Click on the **Take Screenshot** button (the screen will turn gray). You can escape this mode using the *Esc* key.



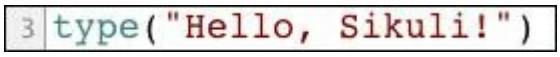
4. Select the area of the screen you'd like Sikuli to wait to appear.
5. Finish your `wait` method call with a `)`.

You should end up with something similar to this:

Step 3 – providing input to the app with type()

Now, we're going to add some input to the app using `type()`, and check that the input was correct using `wait()` again:

1. Add a `type()` command to the end of your script.



2. Check that the text appeared as expected using `wait()`.

Step 4 – using the modifier key with type()

As soon as you know how to type, you'll want to know how to use modifier keys, so that you can control an application using its keyboard binding:

1. Select all the text using *Command* + *A*.

```
5 type("a",KeyModifier.CMD)
```

2. Replace it with something new, as shown in the following screenshot:

```
6 type("Goodbye, Sikuli!")
```

3. Now, check that it was replaced correctly.

```
7 wait( Goodbye, Sikuli! )
```

Step 5 – clicking on buttons with click()

The next thing we want to do is close up the app, but before we can do that we have to show how you click on buttons:

1. Close the TextEdit document using *Command* + *W*.

```
8 type("w",KeyModifier.CMD)
```

2. This will pop up a dialog confirming whether you want to save, and we wait for it to appear by looking for the button we're interested in clicking.

```
9 wait( Don't Save )
```

3. Then, we click on the button using the `click()` method.

```
10 click( Don't Save )
```

Step 6 – closing the application

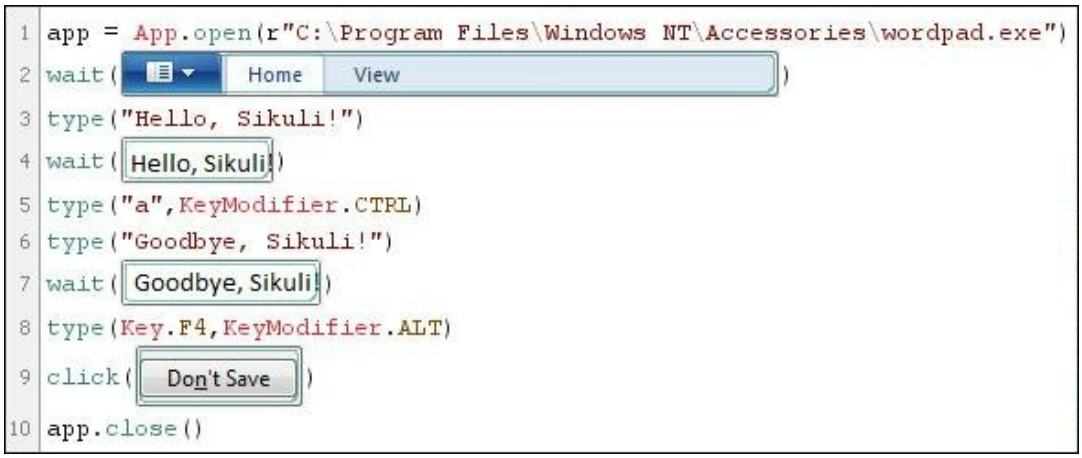
Finally, we can close TextEdit using the `close()` method of the app we saved a reference to in the first line of our script:

```
11 app.close()
```

Your first script with Sikuli on Windows

So, let's jump in! To get started, we're going to build an extremely simplistic testing script based around an application that ships with Windows called WordPad. This will act as a simple "Hello World" example, and show what a

simple script would look like, and how you might go about assembling it. Writing Sikuli scripts is a highly iterative process, and you will often end up executing your script dozens or hundreds of times while developing them (depending on their complexity). Here's the complete script (`hello.sikuli`), so that you can see what the entire thing looks like before we begin:



Step 1 – starting an application

This is the very first thing that most scripts will need to do. This is already utilizing some of the Python capabilities in Sikuli, so we can retain a reference to the application for later use;

1. Start Sikuli and select the editor.
2. Start the WordPad application using Sikuli.

```
app = App.open(r"C:\Program Files\Windows NT\Accessories\wordpad.exe")
```

Try running this, and see what happens. You should see a new WordPad window appear.

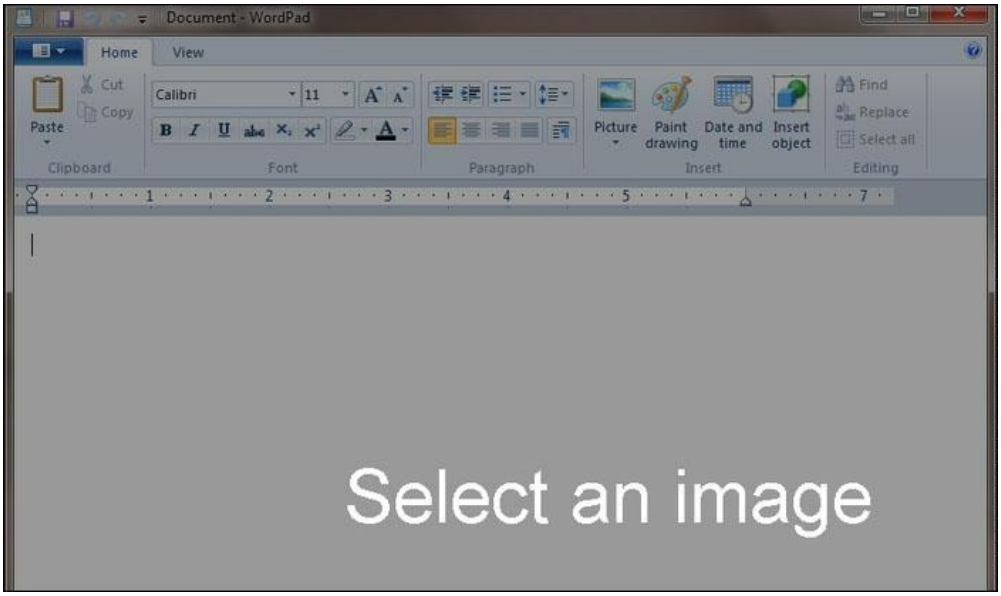
There are a couple of things to be noted:

- If you ever need to stop the Sikuli script from running, use *Alt* + *Shift* + *C* on Windows to stop execution. Sometimes, scripts can become stuck in a loop, and you will need to be able to exit from them even though they have not yet completed.
- On Windows, you will often need to provide the full path to an application to start it. On Mac OS X, you can start Firefox using `App.open("Firefox")`. But on Windows, you will need `App.open(r"C:\Program Files\Firefox\firefox.exe")`. The character `r` before the string is important, if you do not use this character, you'll have to escape the spaces in the path.

Step 2 – checking whether the application has started

When writing scripts with Sikuli, you will want to check that the results on the screen are similar to what you expect. For this, we use the `wait()` method with a screenshot (your first of many):

1. Run the part of the script we created in *Step 1 – starting an application*.
2. Type `wait(` into the editor.
3. Click on the **Take Screenshot** button (screen will turn gray). You can escape this mode with the *Esc* key.



4. Select the area of the screen you'd like Sikuli to wait to appear.
5. Finish your `wait` method call with a `)`.

You should end up with something similar to this:

Step 3 – providing input to the app with type()

Now, we're going to add some input to the app using `type()`, and check that the input was correct using `wait()` again:

1. Add the `type()` command to the end of your script.

```
3 type("Hello, Sikuli!")
```

2. Check that the text appeared as expected using `wait()`.

```
4 wait( Hello, Sikuli! )
```

Step 4 – using the modifier key with type()

As soon as you know how to type, you'll want to know how to use modifier keys, so that you can control an application using its keyboard binding:

1. Select all the text using *Ctrl* + *A*.

```
5 type("a",KeyModifier.CTRL)
```

2. Replace it with something new.

```
6 type("Goodbye, Sikuli!")
```

3. Now, check that it was replaced correctly.

```
7 wait( Goodbye, Sikuli! )
```

Step 5 – clicking on buttons with click()

The next thing we want to do is to close the app, but before we can do that we have to show how you can click on buttons:

1. Close the WordPad document using *Alt* + *F4*.

```
8 type(Key.F4,KeyModifier.ALT)
```

2. This will pop up a dialog confirming whether you want to save, and then we click on the button using the `click()` method.

```
9 click( Don't Save )
```

Step 6 – closing the application

Finally, we can close WordPad using the `close()` method of the app we saved a reference to in the first line of our script:

```
10 app.close()
```

Top 13 features you need to know about

As you start to use Sikuli, you will realize that there are a wide variety of things that you can do with it. This section will teach you all about the most commonly performed tasks and most commonly used features found in Sikuli.

The basics

Firstly, let's run through some of the basic capabilities found in Sikuli, and then expand on our first script. These types of activities are the ones that you'll use every day with Sikuli.

Waiting is one of the most common tasks you'll find in a Sikuli script. Let's go to some of the other options for waiting, and how to use them. Sikuli provides two modes for waiting, we've already seen one using the `wait()` method in the *Quick start – writing your first script* section, which shows how to wait for something to appear. Sikuli also provides a mechanism to wait for something to disappear, which is invoked using `waitVanish()`. The following snippet is waiting for the title bar of our document to disappear (see `hello2.sikuli`):

```
11 waitVanish( Untitled ,FOREVER)
```

`wait()` and `waitVanish()` both take an optional duration parameter. This can be number of seconds (the default is 5), or the global keyword `FOREVER`, which will wait until something happens.

Another important feature is typing and keyboard input. We've already seen some examples of this in the *Quick start – writing your first script* section as well, but now we'll expand on that some more. In many instances, it is simpler and faster to drive your application through the use of the keyboard.

The basic `type()` command is quite simple. As you saw in the *Quick start – writing your first script* section, you can just pass it a string.

```
6 type("Goodbye, Sikuli!")
```

If you need to provide modifier keys, Sikuli also makes this easy by providing a collection of constants to make them easy to use. Here's an example of using multiple modifier keys at the same time:

```
3 type("t",KeyModifier.CMD+KeyModifier.ALT)
```

Here's a complete list of the available `KeyModifiers`:

Control	`KeyModifier.CTRL`
Shift	`KeyModifier.SHIFT`
Alt	`KeyModifier.ALT`
Meta	`KeyModifier.META`
Command	`KeyModifier.CMD`
Windows Key	`KeyModifier.WIN`

There are also lower-level functions for working with the keyboard. Here's a snippet of code, which shows how these work. This script types the letter `a`, and then uses the left arrow key and a carriage return to move it down a line (see `keyupdown.sikuli`).

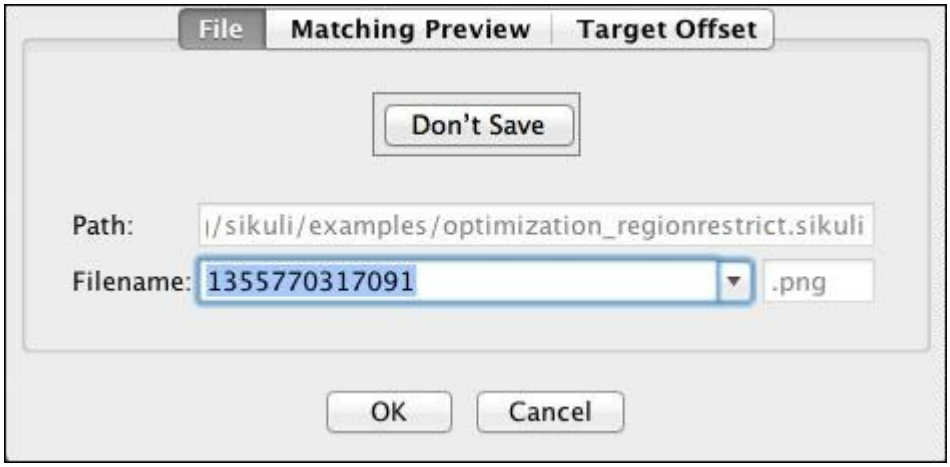
```
3 keyDown("a")
4 keyUp("a")
5 keyDown(Key.LEFT)
6 keyUp(Key.LEFT)
7 keyDown(Key.ENTER)
8 keyUp(Key.ENTER)
```

You can even operate modifiers using `keyDown` and `keyUp`, which allows you to do more complicated operations when needed:

```
9 keyDown(Key.CMD)
10 keyDown("w")
11 keyUp()
```

Clicking is the last introductory skill we need to cover. Again, we showed its most basic usage as part of the *Quick start – writing your first script* section. You can also specify a bunch of details about the search image.

Try clicking on one of the search images, and you will get the Pattern Settings dialog. Here's the one for our **Don't Save** click action.



The **File** tab shows you a preview of the image, the path of the image file being used for matching, and the filename:

The **Matching Preview** tab shows you where your search is found on screen. The matches are indicated using a red transparent highlight. You can also adjust the similarity, which will allow you to match more or less exactly (by default Sikuli looks for a 70 percent match). If your script is clicking on places you don't expect, that often means you need to adjust the

similarity, so that the script will only find the correctly matching items. You can use the **Matching Preview** tab to make sure it's going to select the items you expect. Though, sometimes the items on the screen you're interested in will be transient, and you'll need to use a process of trial and error to figure out the correct similarity value. If you've changed the similarity, this will be indicated in the editor by a number in the upper-right corner of the screenshot. For the **Don't Save** button, this is what it should look like:

The maximum number of matches limits the number of items that `findAll()` will locate. The default maximum value is 10.

Finally, we have the **Target Offset** tab. This lets you manage where on the screen you want to click. By default the target offset is set to the center of the image, but this can be changed. What this lets you do is to change the click position to someplace other than the center of the image. This is extremely useful when you want to click on something that won't be easy to find, say an arbitrary spot in a large blank area of the screen, or on a particular button which is similar to many others. What you can do instead is find a unique element on the screen, and use an offset to click on the one you want instead. Here, we've just set the offset a bit off to the right of the button. If the offset is set, it will be indicated by a little red cross inside the editor.

The offset marker will be positioned related to where the offset is set. So, if the offset was to the left, the cross would be on the left. In a large image, the offset can also appear where the target actually will be.

There are also convenience functions for performing the `doubleClick()` or `rightClick()` actions. Similar to the `type` command, these can take a modifier option. So, say you needed to press *Command* and `rightClick()` at the same time, you can do that as well.

You can get pretty far with Sikuli before you start getting into its more advanced features. Through the simple combination of starting up application, waiting for things to appear/vanish from the screen, being able to type, and being able to click on objects, you can produce a rich collection of interactions with applications. In the next section, we'll start introducing

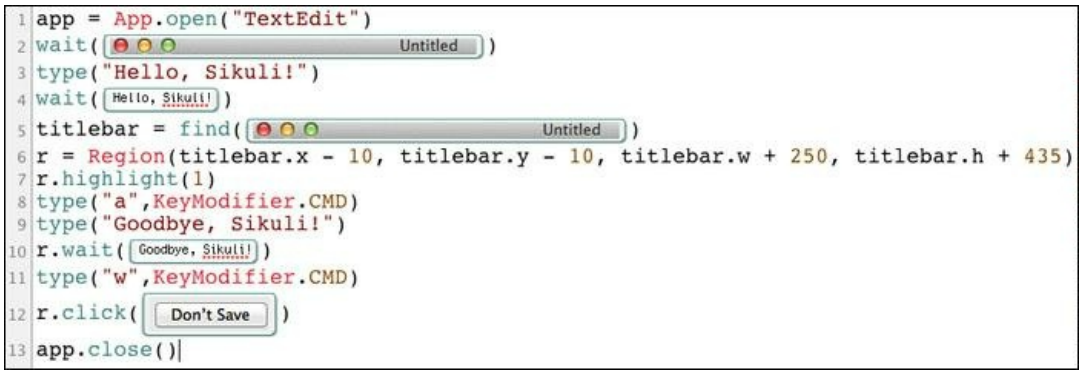
some of Sikuli's more advanced capabilities, which will let you handle more variation in your programs.

Finding things on the screen and how to use regions

Finding things on the screen and interacting with them is another common operation in Sikuli. We've already seen this to some extent with the `click()` and `wait()` commands. Behind the scenes, these are actually combined with a `find` operation. But, sometimes you need to look for things explicitly.

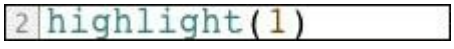
`find()` and `findAll()` let you find matches on the screen. Generally speaking, we won't use `find` very often, but it is handy for doing things like setting up regions to allow for optimization of your scripts. `findAll()` is used a lot when trying to operate on a bunch of similar items on the screen, and we'll discuss it further in a moment.

To show how to use `find`, we'll add a region to our existing script to limit the area of the screen where Sikuli is scanning when running our script (see `optimization_regionrestric.sikuli`).

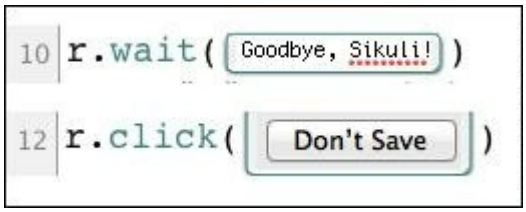


So, let's discuss this for a moment. If you look at the overall script, you'll see that it's pretty similar to our original script from the *Quick start – writing your first script* section. The new elements are the use of a `find()` call to get a reference to the title bar and the creation of a region based on the title bar's location. We also make some use of variables (`titlebar =` and `r =`), so we can re-use some of our collected data.

Another useful tidbit is the `highlight()` method of `Region`. This allows you to draw a box around a portion of the screen corresponding to a region. You can try this out in any of our scripts up to this point as well by calling `highlight()` directly.

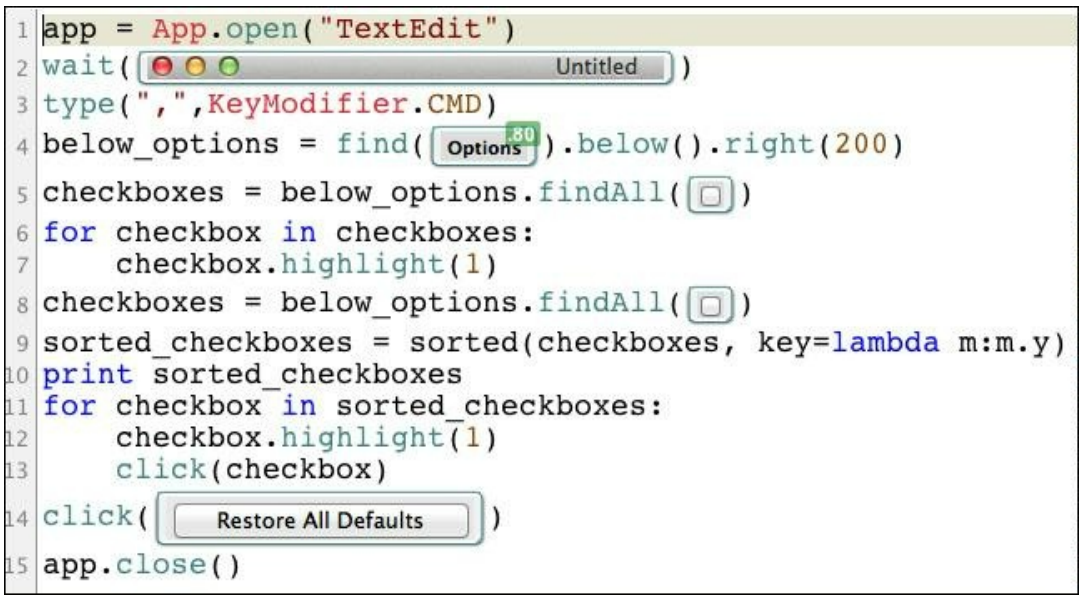


Notice that the entire screen is briefly marked with a red box showing the current region (the whole screen). All of the commands (`wait()`, `find()`, `click()`, and others) can be used on a variable containing a region (`r`) as it will restrict the search area, and help speed up your scripts. Setting up regions is also helpful if there are similar items found on the screen, but you only want to work with the ones in a particular place. Note that we're using the `r` region instead of calling `click()` and `wait()` directly.

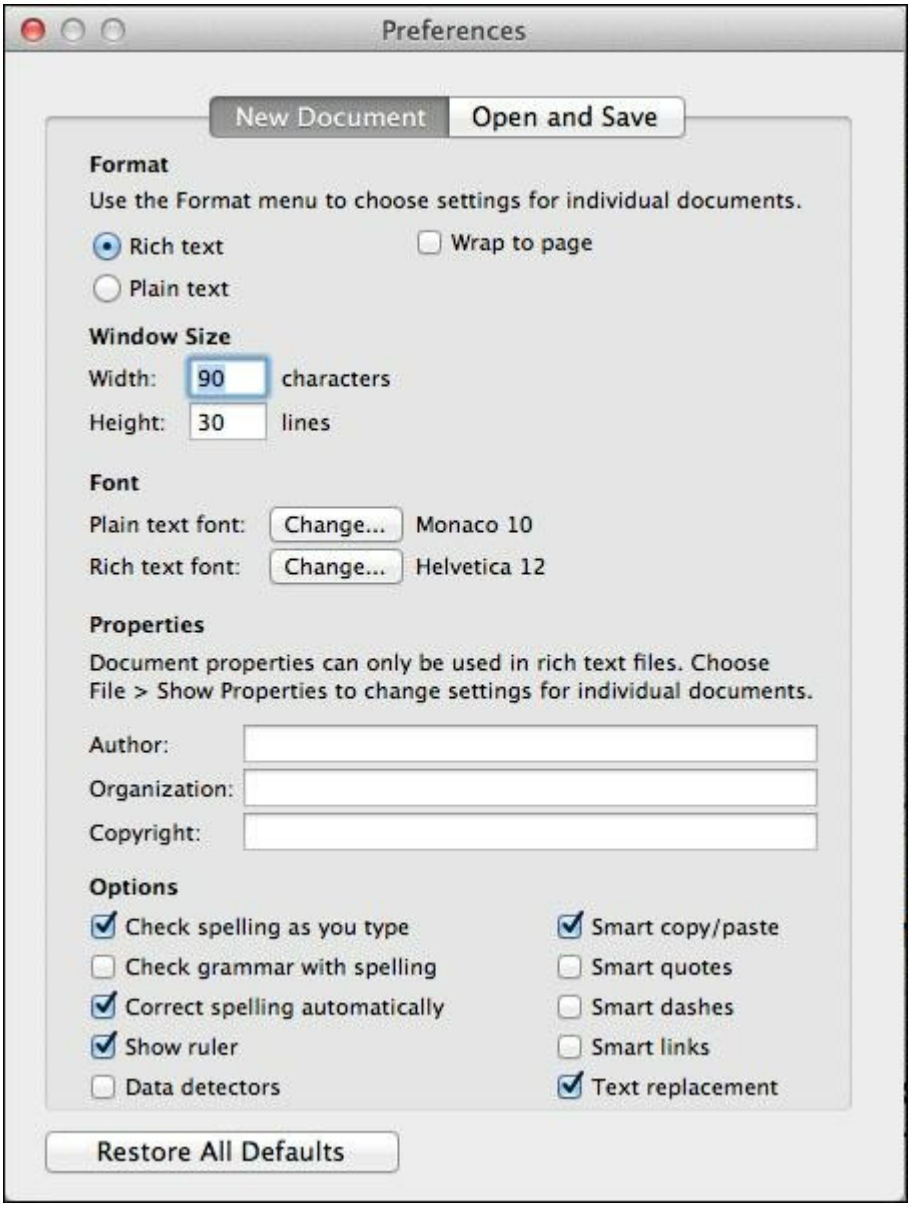


Flow control in Python using loops

We'll finish our discussion of `find` by talking about `findAll()` and combining that with our first discussion of flow control. Generally speaking, `findAll()` isn't so useful without combining it with some sort of flow control structure like a `for` loop. The `for` loop lets you iterate through the results and perform an action with each one. To facilitate the discussion, take a look at the following script (see `findallcheckall.sikuli`):



Most of this should now be familiar to us, but let's discuss the new bits. Firstly, we use `find()` to set up a region. This makes use of the helper functions `below()` and `right()`, allowing us to select the items below the options label (with an adjusted match percentage) and 200 pixels to the right (the ones in the right column). Here's a screenshot of the dialog we're working with to give you more context:



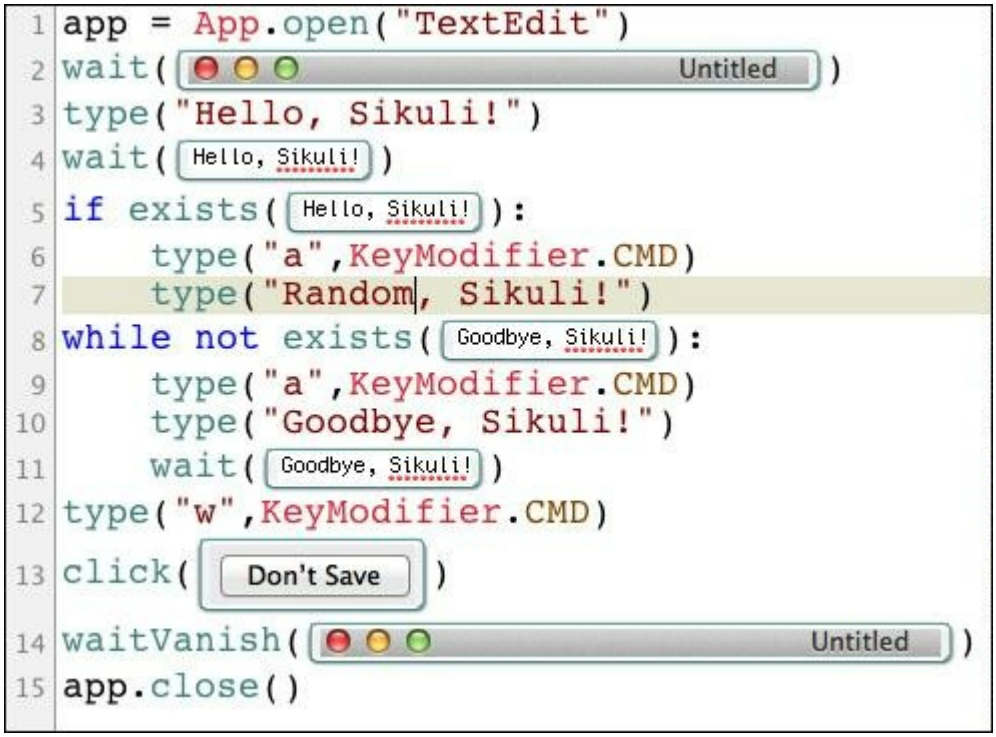
The script is looking for the three checkboxes for **Smart quotes**, **Smart dashes**, and **Smart links**. We then use the results to highlight each of the checkboxes. There is no guarantee that the items will be found in a particular order, so to show a handy trick we find them again, and order the result using the Python's `sorted` and `lambda` commands (we'll learn more on Python later). This allows us to order the checkboxes in vertical order from top to bottom. We then highlight and click on them. Before we close TextEdit, we reset the environment.

More flow control techniques and interacting with scrollbars

Sikuli uses Jython for all of its language constructs. This gives you access to the Python language inside your scripts for performing operations along with the portions of the Python standard library, which Jython supports.

We've just shown how to use a `for` loop. Now, let's look at two other flow control mechanisms.

The following script shows how to use the `if` and `while` flow control mechanisms in Python to allow for more complex interactions with the `TextEdit` application (see `flowcontrol.sikuli`):



Again, most of the script is the same except the introduction to the `exists()` method, which checks for whether something exists on the screen, and the two flow control methods. `if` is used to check whether the text `Hello, Sikuli!` is on the screen, and if it exists, replaces it with `Random, Sikuli!`. The `while` control runs in a loop checking if `Goodbye, Sikuli!` exists on the screen and replaces the text with that, if it does not exist.

With these tools, you can now do something interesting with the application window, which is to scroll the screen. So, now we'll show you how to work with scroll bars.

For this example we're going to use a browser instead of the text editor. Working with the scroll bars within the text editor on Mac OS X can be a little bit difficult, since the scroll bar is invisible except when being used. In Firefox, if the page is long enough to warrant scroll bars, they are visible by default. So, let's look at an example.

Here's a script which opens Firefox and navigates to search results for the search term `sikuli` (see `scrolling.sikuli`).

Let's discuss the scrolling portion of the script first, and we'll come back to a couple of additional useful elements in a moment. On line **6**, we start by creating a variable that points to a screenshot matching the bottom of the scroll bar. The match percentage has been increased to 90 percent from the default to prevent matching of other portions of the scroll bar. If you open the **Matching Preview** tab for the image, you can see how it initially matches multiple positions on the scroll bar. We then wait for the scroll bar to appear, and then loop through waiting for the Google results pager to appear.

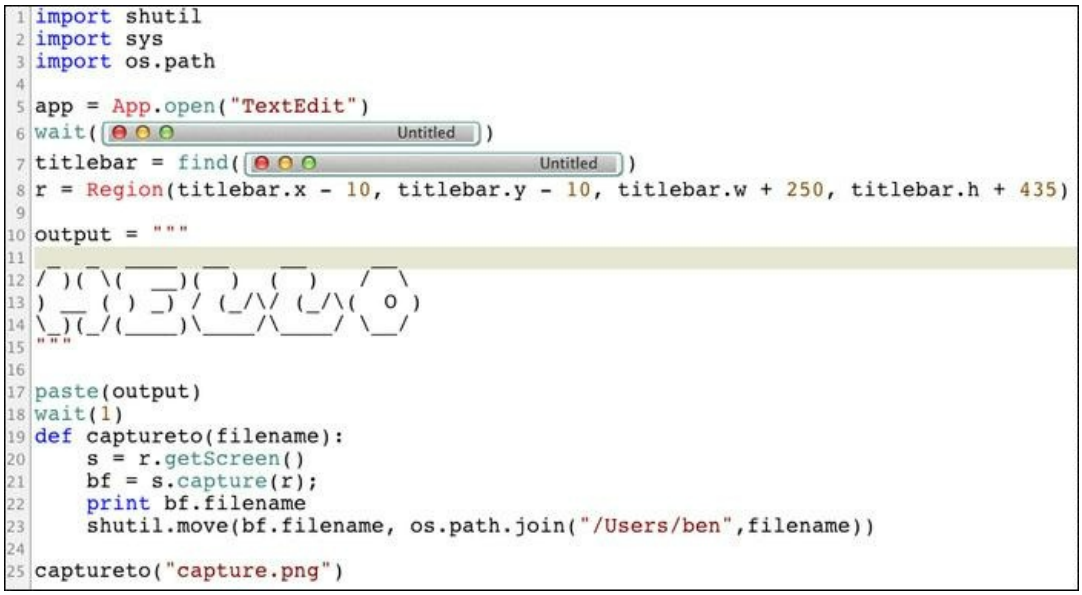
On each pass through the loop, we are highlighting the bottom of the scroll bar, and then dragging it down by 200 pixels. This is the first time we've used the `dragDrop()` command. This allows you to perform the GUI's drag-and-drop operation using two patterns. In this case, we use the scroll bar's bottom and a position 200 pixels below it.

A handy technique in Sikuli is to use positions relative to items. A bunch of helper methods are provided to make this easier. Here, we're using the `below()` method to create a reference to a position below the bottom of our scroll bar. There are three other methods for specifying relative positions: `above()`, `left()`, and `right()`, in which all take the number of pixels as an argument. You can also combine them. For example, `find(scrollbar_bottom).above(100).left(200)` to reference a point 100 pixels above and 200 pixels to the left.

Utilizing Python in your Sikuli scripts

Sikuli provides access to the entire Python language as a way to provide rich scripting capabilities. These can be used for a variety of tasks and we'll show a couple of them here. You are pretty much only limited by your imagination in this area.

To give you a better idea of the possibilities, let's look at an example. A handy trick when writing Sikuli scripts is to save images for debugging later. This is especially valuable when you want to run your scripts unattended, but you might need to determine why a script failed later (see `capture.sikuli`).



The script just shown shows how to achieve this. The first three lines imports some of the Python libraries that we'll need, in particular `shutil`, which deals with file operations and `os.path`, which provides tools to help with working with file paths. As previously shown, we then start the `TextEdit` application and set up a region (`r`) corresponding to the area of the window.

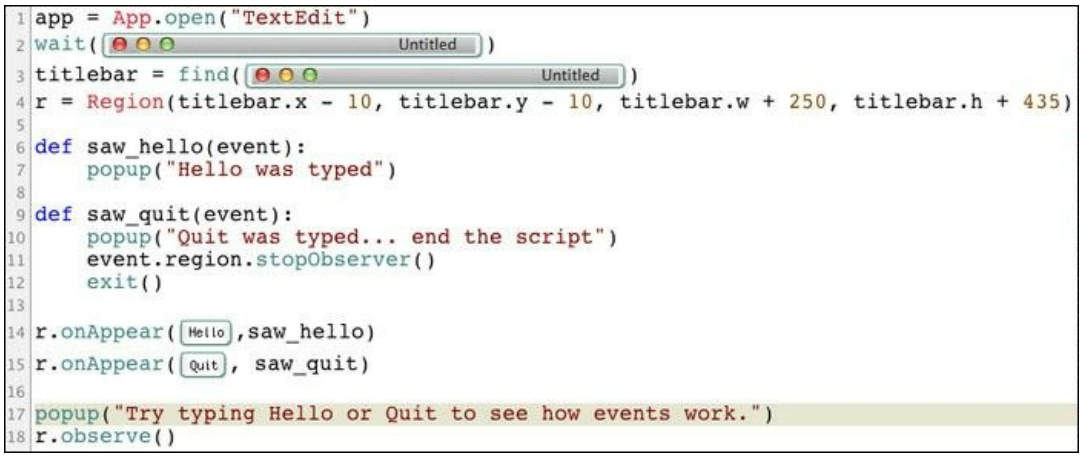
We then inject some content using the `paste()` method (this is similar to `type()`, but pushes all of the content very quickly instead of emulating a user's typing). Following that, the `captureto()` method is defined, which lets us perform a capture of the region by providing a filename. For this example, things are greatly simplified and the files are stored to a preset directory, but it would be a trivial matter to make the `captureto()` method able to take a provided region and path to save the files.

If you'd like to learn more about Python in general, you should check out these sites:

- Zed Shaw's *Learn Python the Hard Way* at http://learnpythonthehardway.org/
- Mark Pilgrim's *Dive into Python* at http://www.diveintopython.net/
- The Jython website at http://jython.org

Event-driven workflows

Sikuli also supports an event-driven workflow. This allows Sikuli to respond to changes on the screen for more complex interaction patterns such as waiting for one of the multiple things to happen. We'll stick with a fairly simple example for now, but this approach to test writing can be extremely useful (see `events.sikuli`).



The script is pretty simple. It sets up some event handlers to respond to the word `Hello` or `Quit` being typed into the text area of the `TextEdit` application. As we've been doing, start TextEdit and set up a region to work on. We then set up two event handlers (`saw_hello` and `saw_quit`) to perform actions when Sikuli sees what we're looking for. We then add the handlers to the region using `onAppear()`, and finally call `observe()` to start looking for these strings in the text area. This script also makes use of the `popup()` method to show messages to the user.

There is also a collection of configurable settings for the timing and behavior of the mouse and scanning rate. Generally, you won't need to adjust the defaults, but sometimes it is necessary. You can find a complete list of these configuration options here:

http://doc.sikuli.org/globals.html#Settings

The most common ones that you might want to modify are `Settings.MoveMouseDelay`, `Settings.DelayBeforeDrag`, and `Settings.DelayAfterDrop`, which allow you to tweak how long it takes to move the mouse. To change a value you just need to set it. All the values are in seconds. For example:

- `Settings.MoveMouseDelay = 0 # jump the cursor`
- `Settings.MoveMouseDelay = 3 # take 3 seconds to move it`

And, with that, we've pretty much covered all of the core features that you'll need to create rich automated tests using Sikuli. You should be able to

operate most aspects of an application and have a variety of techniques at your disposal to approach various testing challenges. In the next section, we'll start discussing how to organize your test scripts and manage their execution to create a complete testing solution. See the *People and places you should get to know* section for links to additional reference material, or to make use of your favorite search engine.

Test automation

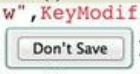
Now, we get to the good stuff! Everything up to this point has been to prepare you to understand the much more complex examples found in the actual test automation process. Using Sikuli as a testing tool is greatly enhanced by leveraging its Python capabilities to allow you to build up libraries of common actions, construct test harnesses, and aggregate results. We'll be going over how to actually do these things beginning in this section.

Building a library of common actions is the place where we'll start. This is critical to building and maintaining an effective collection of tests. In most applications, you'll have a bunch of common tasks such as starting up and shutting down the application, or doing some sort of common action with one of the dialogs (which you may have noticed in a bunch of our examples). Without a library, you need to keep all of these actions in your scripts and this poses a major problem. What do you do when the UI is changed and it breaks your scripts? Without a library, the answer is that you go through and update images in every Sikuli script you've written. With a library, you go through one script and fix it. And with variables defined for all of your UI elements in your library, you can limit this further, though it is the author's experience that this reduces the readability of scripts (we won't be showing them this way).

The library is just another Sikuli script, but there are some things you'll need to do to make it work. To create it, open a new script and add the following to the top:

```
1 from sikuli import *
```

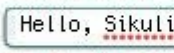
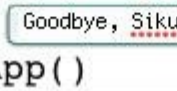
This is a fairly common bit of Python. Earlier, we showed how to use Python libraries using `import`. This is another way to use a Python library but brings the classes and methods of the Sikuli library in a special way, so you don't have to type `sikuli.App` (for example) when trying to use the `App` class. This makes sure that when you bring your library into your tests scripts, it will be able to find the parts of Sikuli that it needs. So, let's start building the library (see `library.sikuli`):

```
from sikuli import *

def startApp():
    global app
    app = App.open("TextEdit")
    wait(  )
    titlebar = find(  )
    r = Region(titlebar.x - 10, titlebar.y - 10, titlebar.w + 250, titlebar.h + 435)
    setROI(r)

def stopApp():
    global app
    app.focus()
    type("w",KeyModifier.CMD)
    click(  )
    app.close()
```

Here, we have defined two functions, one to start the app and another to stop it. To share the reference to the app, we use a Python global variable. If a function doesn't need to make use of the app, you can leave out that particular line (we'll show some examples without it later). If you go back and look at our original "Hello World" example, you can see similarities with it in these functions. In the start function, we also make use of the region technique, and use another Sikuli function, `setROI()`, to make this region default. This makes all the scripts use the area inside the region to restrict their search. This can also be dropped if you don't feel you need it. In the stop function, we use another method of the application, `focus()`, to make sure focus is directed at the right app when performing the action.

The library should be saved like any other Sikuli script. I call mine `library.sikuli`, and this name will be assumed in the rest of the examples.

So, how do we use the library to write a test you're probably asking at this point? Let's go back to our original `Hello, Sikuli!` example, and re-write it using the library (see `hellolibrary.sikuli`).

```
import sys
if "library" in sys.modules:
   del sys.modules["library"]
exec("from library import *")

startApp()
type("Hello, Sikuli!")
wait(  )
type("a",KeyModifier.CMD)
type("Goodbye, Sikuli!")
wait(  )
stopApp()
```

You can now see that the script only contains the bits specific to the "Hello World" example, typing the text, checking that it exists, and so on. And, all of the utility portions are now using the start and stop functions we defined

in the library. The four lines at the top are what that let us accomplish this magic. They are a bit of Python library magic and you don't need to fully understand what they're doing to use them, but let's go over it in the event you want to do something more complicated.

Firstly, we import the Python `sys` library. This provides access to commands for working with the Python system like looking at a list of the loaded modules (`sys.modules`). The second line looks to see if our library is already loaded, and then deletes it if it is (as shown in the next line). This is very important to make sure that changes to your library are reloaded each time you run your Sikuli script in the IDE. Let's not even talk about the hair-pulling and teeth gnashing involved when trying to figure this out in the first place. The fourth line dynamically executes (using `exec`) an import of the library. Again, this was to deal with making sure the library loads properly when running the script via the IDE.

Assembling a testing framework

To tie this all together, we now move out of Sikuli. While Sikuli provides an internal mechanism for managing tests, these have been buggy and difficult to use. To work around this issue, we will make use of some shell scripts. This gives the tester a much better level of control than trying to manage everything from within Sikuli.

On Mac and Linux systems, this is very simple, since both the operating systems include a Unix-style shell. For Windows systems, you will want to install a copy of Cygwin (http://www.cygwin.com/).

The shell script is fairly complicated, but to start with, let's take a look at it and try to understand how it works. We're going to start with a somewhat simplistic approach to testing. For each of the tests we write, we add an entry to a test list called `testlist.txt`. Each test is on a line and looks like this:

```
hellolibrary.sikuli
checkalllibrary.sikuli
```

We then have a test runner script which iterates through the tests in the list and records the results. Each test is executed by Sikuli and the output logged. The output is then checked for errors to determine whether the test passed or failed. At the end of the script, a summary of the test results is generated.

One of the issues encountered when trying to build up a library of tests is when tests get stuck. To compensate for this problem, we also have a utility script to provide a timeout capability in `bash`, and allow us to detect when a

test has gotten stuck without breaking the entire testing cycle. Now, let's review the test runner script (see `runtests_withoutcleanup.sh`):

```
TESTS=$(cat testlist.txt)
TESTCOUNT=0
TESTFAILED=0
TESTPASSED=0
TIMEOUT=720 #seconds
SIKULI="/Applications/Sikuli-IDE.app/Contents/Resources/Java/sikuli-script.jar"
for test in $TESTS
do
  TESTCOUNT=$(( $TESTCOUNT + 1 ))
  (("./timeout3.sh" -t $TIMEOUT java -jar "$SIKULI" -s "$test" 2>&1 1>&3 |
    tee "${test}.errors.log") 3>&1 1>&2 |
    tee "${test}.output.log") > "${test}.final.log" 2>&1
  RETURN=$?
  ISTERMINATED=$(grep TERMINATED "${test}.final.log")
  if [ "TERMINATED" == "$ISTERMINATED" ]
  then
    echo "TERMINATED" >> "${test}.errors.log"
  fi
  ERRORLOGLEN=$(wc -l "${test}.errors.log" | perl -pe "s#.*?([0-9]+).*#\1#g")
  if [ $ERRORLOGLEN -gt 0 ]
  then
    RETURN="FAIL"
  else
    RETURN="PASS"
  fi
  echo ${test}:$RETURN
  if [ "$RETURN" == "FAIL" ]
  then
    TESTFAILED=$(( $TESTFAILED + 1 ))
  else
    TESTPASSED=$(( $TESTPASSED + 1 ))
  fi
done
echo SUMMARY TESTS: $TESTCOUNT PASSED: $TESTPASSED FAILED: $TESTFAILED
```

Firstly, we define a bunch of variables to hold the list of tests, counts of the test results (number of tests, number of passing tests, and number of failing tests), a default timeout (12 minutes), and the location of the Sikuli application JAR to execute the test (which will need to be adjusted on Linux or Windows systems).

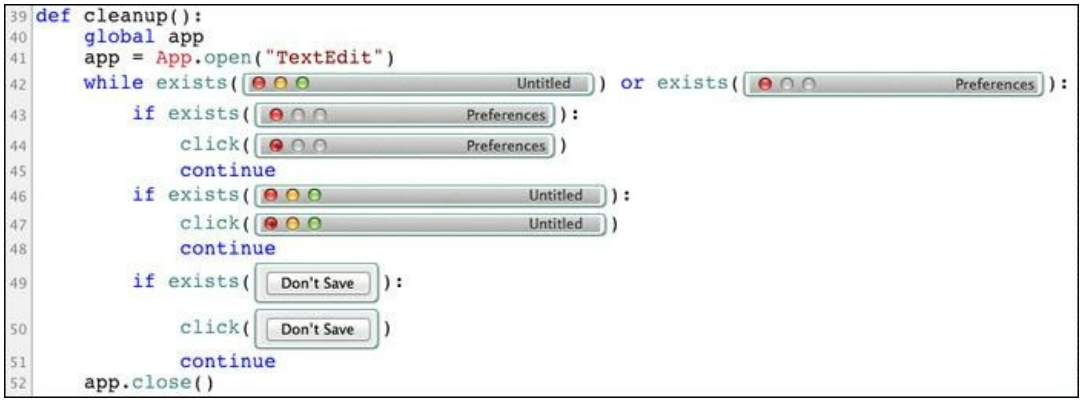
We then iterate through the list of tests adding to the test count, running the tests, and saving the console output to separate files for the standard and error output. We then check that the test wasn't terminated due to a timeout (which we log to the error log), and check that the error log is empty. If the error log is empty, the test has passed; otherwise, we assume that it has failed.

Once we finish running through all the tests in the `testlist.txt`, we print a summary of the results to the console and the script ends.

Everything will work well until you encounter some sort of failure condition. In a testing scenario, it's important to be able to execute all of your tests in sequence without them affecting one another. This is just a good practice when writing tests. Next, we'll extend the script for running the tests in two

ways. Firstly, we'll create a Sikuli script to clean up after test failures, and then we'll integrate this script with our test harness.

To support this, we start by adding a new cleanup method to our library. This looks like this (see `library.sikuli`):



To deal with cleanup, we first check that the application is open. This will make sure that the focus is set correctly. It then checks whether the various windows that our tests use in the application exist, and tries to close them. It does this by iterating through the various elements that might appear on the screen that need to be closed to get the application back to its default state.

Note the use of a collection of `if` blocks with `continue` statements. This ensures that all of the items are iterated through quickly to get the application back to its default state.

We also have to create a Sikuli script to make use of this library method. This script is very simple and makes use of the library loader we introduced earlier (see `cleanup.sikuli`):

```
import sys
if "library" in sys.modules:
   del sys.modules["library"]
exec("from library import *")

cleanup()
```

We also make a small addition to our test runner script to execute our cleanup script (see `runtests.sh`).

```
if [ $ERRORLOGLEN -gt 0 ]
then
  RETURN="FAIL"
  echo -n "CLEANING UP..."
  (("./timeout3.sh" -t $TIMEOUT java -jar "$SIKULI" -s "cleanup.sikuli" 2>&1 1>&3 |
  tee "cleanup_${test}.errors.log") 3>&1 1>&2 |
  tee "cleanup_${test}.output.log") > "cleanup_${test}.final.log" 2>&1
  echo "DONE"
else
  RETURN="PASS"
fi
```

Debugging Sikuli scripts

The last topic with test automation is the debugging of scripts. A portion of all time working on script development will be spent running the script trying to debug problems with the scripts to get them to run reliably. Once you have a collection of scripts that you run on a regular basis without supervision, identifying causes of errors can become much more difficult.

There are two main techniques for debugging Sikuli scripts when running them in the test harness presented here. The first method is to look at the logs. If you look back over the test runner script, you can see that it logs a complete record of the console output to a file. These files end in `.final.log`. You can open these in your text editor, and see what your script did and get feedback about the errors. The errors in the logs will tell you what happened. For example, you might get something like this:

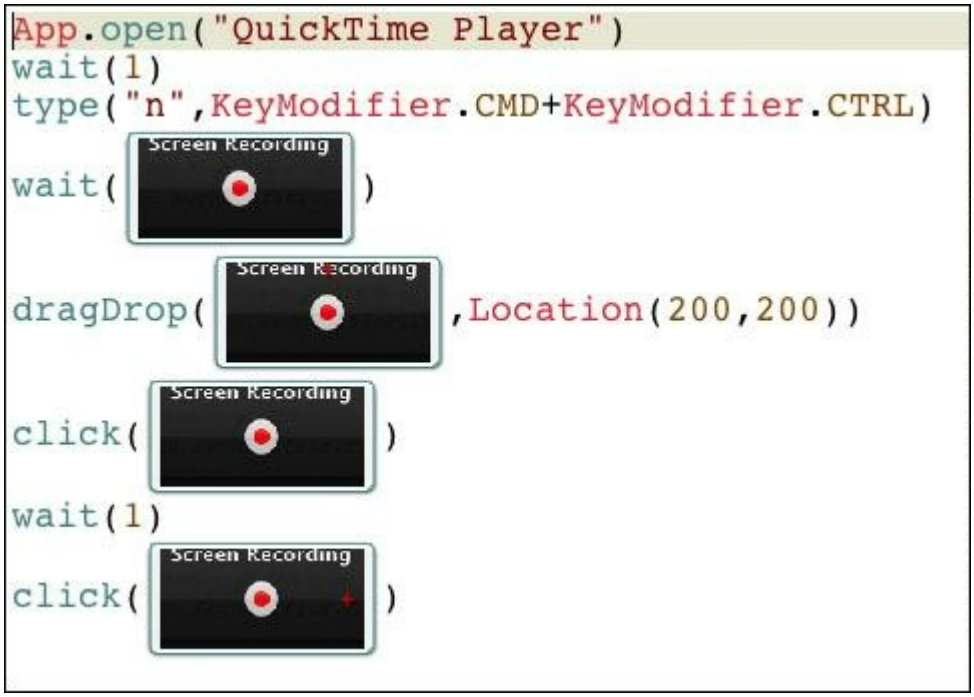
```
org.sikuli.script.FindFailed: FindFailed: can not find Untitled-1.png on the screen.
  Line 12, in file /Users/ben/tmp/writing/sikuli/examples/hellolibrary.sikuli/hellolibrary.py
```

This one is telling us that Sikuli couldn't find the requested image on the screen. Or, you might see errors with your Python code.

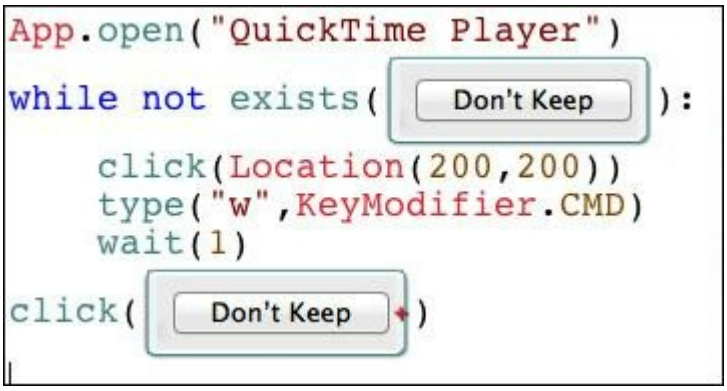
In situations like this, it's handy to know that Sikuli scripts are just a collection of files in a directory. You can actually open it up and look at the images and Python code within it.

Another handy technique is to record videos of your test runs. This allows you to review what happened during a test (passing or failing) to see what went wrong, or analyze the execution for possible improvements to execution speed. For Mac OS X, this can be done using QuickTime Player, which is included with the OS. For Windows or Linux, you will need to investigate a similar solution solution (the examples prepared for this book contain a working example for Windows), but the general technique should still apply. Let's see how this would work in practice.

Firstly, we need to create two additional scripts, one to start recording and another to stop it. The script is broken into two parts, so they can be executed independently. Here's the startup script (see `startcapture.sikuli`):



And here's the script to stop recording (see `stopcapture.sikuli`):



These are then pretty easy to integrate with our test runner scripts (see `runtests_withrecording.sikuli`):

```
for test in $TESTS
do
  TESTCOUNT=$(( $TESTCOUNT + 1 ))
  echo "START RECORDING"
  java -jar "$SIKULI" -s "startcapture.sikuli"

  echo "STOP RECORDING"
  java -jar "$SIKULI" -s "stopcapture.sikuli"
  echo ${test}:$RETURN
  if [ "$RETURN" == "FAIL" ]
```

Depending on your machine, you may also encounter some performance degradations when recording video along with your tests. To compensate,

you can adjust the default amount of time that Sikuli will wait to find something from 5 seconds to 10 seconds, or more (you may need to experiment), by adding the following line to the end of your `library.sikuli` script:

```
75 setAutoWaitTimeout(10)
```

Capturing images using capture()

There are a couple of additional areas that might be of interest to readers of this book that will flesh out your collection of techniques for writing testing scripts using Sikuli.

Renaming scripts can cause trouble. Since a Sikuli file is actually a directory, renaming the file will not rename the Python and HTML files within it. To prevent these sorts of issues, it is easier to open up the Sikuli script you want to rename in the application, create a new file, copy over the contents, and save it out again. By renaming scripts this way, you can ensure that the internal files of the script are named correctly and will continue to function.

When using regions to speed up performance, it is sometimes necessary to reset the region of interest, using the `setROI()` method, back to its default of the whole screen. This is a good candidate to execute on the start of every test, especially when testing in the Sikuli interface. The `resetROI()` function is included with the `library.sikuli` sample included with the book, but it also provides a starting point for discussion of how Sikuli works with the screen (see `library.sikuli`):

```
55 def resetROI():
56     s = Screen()
57     r = Region(s.x,s.y,s.w,s.h)
58     setROI(r)
```

The `Screen` class provides access to data about the screen. On systems with multiple screens, you need to provide it a numerical argument with the screen number (for example, `Screen(0)` for the first display, `Screen(1)` for the second, and so on). As you have seen in the previous example, we are getting a reference to the screen, and then using it to create a region corresponding to the screen's height and width in pixels. We then reset the region of interest back to the complete screen.

```
61 def screenshot(output):
62     import shutil
63     s = Screen()
64     r = Region(s.x,s.y,s.w,s.h)
65     filename = s.capture(r)
66     shutil.move(filename, output)
```

The `Screen` class provides an additional handy method called `capture()`, which captures the screen as an image and dump it into a file. You can use it like this:

```
83 screenshot("/Users/ben/output.png")
```

Accessing Java from Sikuli

Let's have one final discussion about the use of Python in Sikuli. Since Sikuli integrates Jython, not only do you have access to large portions of the standard Python library, but also you have complete access to Java's class library as well. If you need to do odd things where you want to provide additional GUI elements as part of your scripts, this can be extremely handy. The following example shows how to use Java classes from Sikuli to create an on-screen display method. This can be extremely handy in combination with the video recording feature, so you can display on screen and record notes about what a test is doing at particular points (see `osd.sikuli`).

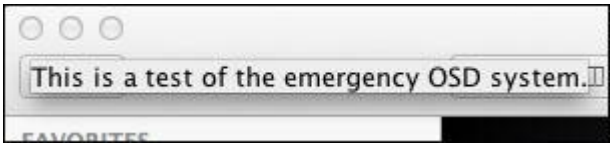
```
68 def osd(region,text,seconds=1):
69     import javax.swing
70     win = javax.swing.JWindow()
71     win.setLocation(region.x,region.y)
72     conn = javax.swing.JLabel(text)
73     win.contentPane.add(conn)
74     win.setSize(conn.getPreferredSize().width, conn.getPreferredSize().height)
75     win.setAlwaysOnTop(1)
76     win.setVisible(1)
77     sleep(seconds)
78     win.dispose()
```

The method takes a region, text, and optionally number of seconds to display as arguments. It starts by importing the swing classes which are one of Java's GUI libraries, and then uses swing to show a borderless window with the designated text over everything on the screen for the specified number of seconds (default is 1).

This is how you can call it:

```
84 osd(r,"This is a test of the emergency OSD system.", 10)
```

And this is what it looks like when running:



Searching for and decoding text on the screen

Sikuli contains optical character recognition (OCR) capabilities based on Tesseract, an open source OCR library. The OCR feature is disabled by default in the 1.0 release of Sikuli. If you try and use the feature (`Region.text()` or `find("<string>")` method), you will get this error if you have not enabled OCR:

```
[error] Region.find(text): text search is currently switched off
```

OCR can be turned on in the more options section of the Preferences. Look for the TextSearch and OCR section. To enable finding text on the screen (`find("string")` method), enable the allow searching for text option. To allow for text decoding (the `Region.text()` method), enable the allow OCR option. This can also be done in your Sikuli scripts. If you are using the test harness, this is required. Just add the following to your script (or add it to the library if you want it in all of your scripts):

```
6 Settings.OcrTextSearch=True
7 Settings.OcrTextRead=True
```

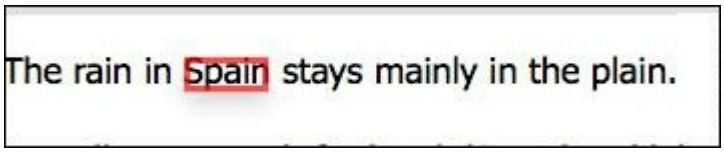
To show you the capability and some of the limitations, we have the following example (see `ocrlibrary.sikuli`):

```
import sys
if "library" in sys.modules:
   del sys.modules["library"]
exec("from library import *")

Settings.OcrTextSearch=True
Settings.OcrTextRead=True

startApp()
for font in ["Arial", "Times New Roman", "Tahoma", "Monaco"]:
    if font == "Monaco":
        switchFont(font,10)
    else:
        switchFont(font,18)
    type("\n")
    type("The rain in Spain stays mainly in the plain.\n")
    type("\n")
    paste("we will now search for 'rain', 'Spain' and 'plain'.\n")
    paste("Below are the decoded the following characters using Tesseract:\n")
    for word in ["rain","Spain","plain"]:
        r = find(word)
        r.highlight(3)
        output = r.text()
        type("%s: " % word)
        try:
            type(output)
        except:
            type("couldn't decode all the characters see log")
        type("\n")
    wait(3)
    type("a",KeyModifier.CMD)
    type(Key.BACKSPACE)
stopApp()

```

This script is built on top of our library, so that we could re-use the `startApp()` and `stopApp()` functions. We've also added a `switchFont()` function based on a portion of the `checkalllibrary` example we saw earlier. This is a good example of how your library will grow. As you expand the breadth of your testing, you'll start to find that parts of your individual tests are actually useful when building others.

The script iterates through a series of fonts, writes out a string, and then uses the OCR library's two primary functions to find text on the screen and to decode text from the screen.

Here is the expected behavior of the `find()` portion of our example:



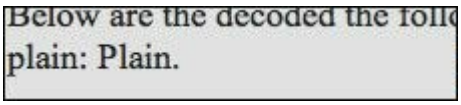
And here's the bit of script that accomplishes this:

```
        r = find(word)
        r.highlight(20)
```

When working with OCR, we also use the `find()` method, but instead of giving it an image to work with, we give it a string of text. In this instance, we're using a variable called `word` which is currently set to `Spain` (`word =`

`"Spain"`). If you look at the complete example, you can see that these lines are part of the `for` loop, which is iterating through a series of words (**rain**, **Spain**, and **plain**).

The other aspect of character recognition is decoding text on the screen. Here is some of Sikuli's decoding (in this case, somewhat successfully):



Decoding text is also pretty straight forward, and we'll also introduce another Python technique here:

```
20          output = r.text()
21          type("%s: " % word)
22          try:
23              type(output)
24          except:
25              type("couldn't decode all the characters see log")
```

To decode the text, all we have to do is call the `text()` method on the region (`r`). This tells Tesseract to give us its interpretation of the area of the screen indicated by the region. One issue that you'll run into though is that you won't always get back results which can be displayed easily, or are interpretable. To compensate for this we make use of Python's exception handling using `try:` and `except:`. You put the statements you think might cause an exception in the `try:` portion of the code, and put whatever you want to do in response in the `except:` portion. In this case, we just print out the message that we couldn't decode the text if there was a problem.

You'll often run into the limitations of OCR. Like its cousin speech recognition, OCR is not 100 percent accurate. It can get tripped up by similarities between letter shapes. This example is structured to highlight these limitations because it is hard to make effective use of a technique like OCR unless you work within the limits it expects. Watch this example execute and you'll see that it often makes mistakes. The mistakes it makes can vary based on the font or the size of the characters being recognized. This doesn't make OCR unuseful, but you've got to be aware of the limitations and work within them. It might be possible to improve OCR accuracy by training your own recognition models. There is detailed documentation about how to do this on the Tesseract wiki at http://code.google.com/p/tesseract-ocr/wiki/TrainingTesseract3. The necessary files to replace in Sikuli are found in the `libs/tessdata` directory of your Sikuli installation.

Extracting text using the clipboard() method

Sometimes you need to get the actual text off the screen to do something. As we discussed in the previous section about OCR, it is not always accurate. So, what can we do? The answer to this is to use `clipboard()`.

Sikuli provides access to the system's clipboard via the `clipboard()` method. So, if you can select the text you want on the screen, you can get access to it within your Sikuli script. Let's show a small example (see `clipboardlibrary.sikuli`). This example is based on http://www.youtube.com/watch?v=8vtMArVWw8A, which is one of the first Sikuli scripts the author saw, and started his more serious exploration of the technology.

```
1  import sys
2  if "library" in sys.modules:
3     del sys.modules["library"]
4  exec("from library import *")
5
6  def copy(begin,end):
7      dragDrop(begin,end)
8      type("c",KeyModifier.CMD)
9
10 startApp()
11 output = """
12   *** This is some text we're not interested in
13   *** We would love this one!
14   *** Not interested in this one.
15   *** We like this one too!
16 """
17 type(output)
18 lines = list(findAll(***0))
19 lines.sort(lambda a,b: a.y-b.y)
20 for line in lines:
21     copy(line.right(1),line.right(550))
22     linetext = Env.getClipboard().strip()
23     sleep(0.2)
24     if linetext.startswith("We"):
25         osd(getROI().right(1),"we want that one!")
26 stopApp()
```

Once again, we're building on our library. We generate some text (similar to the `capture` example). We look for all of the `***` patterns on the screen and sort the results (similar to the `checkall` example). And then we get to the good part. Using the location of the `***`, we use the `copy()` function to select the text, and put it into the system clipboard.

```
6  def copy(begin,end):
7      dragDrop(begin,end)
8      type("c",KeyModifier.CMD)
```

We then extract the value from the clipboard and sleep for a moment (0.2 seconds). This sleep step is important, since if you don't do that, the clipboard will not contain the data you copied, frequently. The Python

`strip()` method is used to remove whitespace from the beginning and end of the clipboard contents. This is a good habit when working with an input like this, where it might contain extra spaces or newline characters that you do not want to process.

```
21      copy(line.right(1),line.right(550))
22      linetext = Env.getClipboard().strip()
23      sleep(0.2)
24      if linetext.startswith("We"):
25          osd(getROI().right(1),"we want that one!")
```

To show an example of what you can do, we perform a simple comparison of the extracted text looking for lines that begin with `We`. In response, we make use of our `osd()` library method, which we created in the *Accessing Java from Sikuli* feature of the *Top 13 features you need to know about* section.

People and places you should get to know

If you need help with Sikuli, here are some people and places which will prove invaluable.

Official sites

- Homepage: http://sikuli.org
- Sikuli examples: http://www.sikuli.org/videos.html
- Manual and documentation: http://doc.sikuli.org/
- Project hosting: https://launchpad.net/sikuli
- Source code: https://code.launchpad.net/sikuli
- Bug tracker: https://bugs.launchpad.net/sikuli

Articles and tutorials

- Automated Regression Testing Mac OS X Clients Using Sikuli: http://www.krypted.com/?p=7760
- Basics of using Sikuli to automate tasks: http://helpdeskgeek.com/how-to/sikuli-commands-tutorial/
- Practical Sikuli: http://www.slideshare.net/vgod/practical-sikuli-using-screenshots-for-gui-automation-and-testing

Community

- Official mailing list: https://lists.csail.mit.edu/mailman/listinfo/sikuli-dev
- User FAQs: https://answers.launchpad.net/sikuli

Blogs

- 3QiLabs Sikuli-related posts: http://3qilabs.com/tag/sikuli-3/
- Author's Sikuli-related posts: http://netjunki.org/category/sikuli.html

Twitter

- For more information on Sikuli, follow the Sikuli at https://twitter.com/sikuli
- For more open source information, follow Packt at http://twitter.com/#!/packtopensource

Summary

Hopefully, this book has provided you with a good introduction to using Sikuli for test automation along with sufficient examples to help you solve complex problems using Sikuli. See the *People and places you should get to know* section for starting points on getting deeper into automated testing with Sikuli.

As this book was being written, the final release of Sikuli 1.0 was announced. The 1.0 release involved some major changes to the way Sikuli is started and run, but generally, things remain the same. The author has tried to align the text and examples with these changes where possible. The examples have all been run and tested with the 1.0 release, but the old versions developed using the 1.0RC3 (Maltipoo) release will also be made available for download with the examples for the book.