20

CONTROLLING THE KEYBOARD AND MOUSE WITH GUI AUTOMATION

Knowing various Python modules for editing spreadsheets, downloading files, and launching programs is useful, but sometimes there just aren't any modules for the applications you need to work with. The ultimate tools for automating tasks on your computer are programs you write that directly control the keyboard and mouse. These programs can control other applications by sending them virtual keystrokes and mouse clicks, just as if you were sitting at your computer and interacting with the applications yourself.

This technique is known as *graphical user interface automation*, or *GUI automation* for short. With GUI automation, your programs can do anything that a human user sitting at the computer can do, except spill coffee on the keyboard. Think of GUI automation as programming a robotic arm. You can program the robotic arm to type at your keyboard and move your mouse for you. This technique is particularly useful for tasks that involve a lot of mindless clicking or filling out of forms.

Some companies sell innovative (and pricey) "automation solutions," usually marketed as *robotic process automation (RPA)*. These products are effectively no different than the Python scripts you can make yourself with the `pyautogui` module, which has functions for simulating mouse movements, button clicks, and mouse wheel scrolls. This chapter covers only a subset of PyAutoGUI's features; you can find the full documentation at *https://pyautogui.readthedocs.io/*.

INSTALLING THE PYAUTOGUI MODULE

The `pyautogui` module can send virtual keypresses and mouse clicks to Windows, macOS, and Linux. Windows and macOS users can simply use pip to install PyAutoGUI. However, Linux users will first have to install some software that PyAutoGUI depends on. Open a terminal window and enter the following commands:

- `sudo apt-get install scrot`

- `sudo apt-get install python3-tk`

- `sudo apt-get install python3-dev`

To install PyAutoGUI, run `pip install --user pyautogui`. Don't use `sudo` with `pip`; you may install modules to the Python installation that the operating system uses, causing conflicts with any scripts that rely on its original configuration. However, you should use the `sudo` command when installing applications with `apt-get`.

Appendix A has complete information on installing third-party modules. To test whether PyAutoGUI has been installed correctly, run **`import pyautogui`** from the interactive shell and check for any error messages.

> **WARNING**
>
> *Don't save your program as* pyautogui.py. *When you run* `import pyautogui`, *Python will import your program instead of the PyAutoGUI and you'll get error messages like* `AttributeError: module 'pyautogui' has no attribute 'click'`.

SETTING UP ACCESSIBILITY APPS ON MACOS

As a security measure, macOS doesn't normally let programs control the mouse or keyboard. To make PyAutoGUI work on macOS, you must set the program running your Python script to be an accessibility application. Without this step, your PyAutoGUI function calls will have no effect.

Whether you run your Python programs from Mu, IDLE, or the Terminal, have that application open. Then open the System Preferences and go to the Accessibility tab. The currently open applications will appear under the "Allow the apps below to control your computer" label. Check Mu, IDLE, Terminal, or whichever app you use to run your Python scripts. You'll be prompted to enter your password to confirm these changes.

STAYING ON TRACK

Before you jump into a GUI automation, you should know how to escape problems that may arise. Python can move your mouse and type keystrokes at an incredible speed. In fact, it might be too fast for other programs to keep up with. Also, if something goes wrong but your program keeps moving the mouse around, it will be hard to tell what exactly the program is doing or how to recover from the problem. Like the enchanted brooms from Disney's *The Sorcerer's Apprentice*, which kept filling—and then overfilling—Mickey's tub with water, your program could get out of control even though it's following your instructions perfectly. Stopping the program can be difficult if the mouse is moving around on its own, preventing you from clicking the Mu Editor window to close it. Fortunately, there are several ways to prevent or recover from GUI automation problems.

Pauses and Fail-Safes

If your program has a bug and you're unable to use the keyboard and mouse to shut it down, you can use PyAutoGUI's fail-safe feature. Quickly slide the mouse to one of the four corners of the screen. Every PyAutoGUI function call has a 10th-of-a-second delay after performing its action to give you enough time to move the mouse to a corner. If PyAutoGUI then finds that the mouse cursor is in a corner, it raises the `pyautogui.FailSafeException` exception. Non-PyAutoGUI instructions will not have this 10th-of-a-second delay.

If you find yourself in a situation where you need to stop your PyAutoGUI program, just slam the mouse toward a corner to stop it.

Shutting Down Everything by Logging Out

Perhaps the simplest way to stop an out-of-control GUI automation program is to log out, which will shut down all running programs. On Windows and Linux, the logout hotkey is CTRL-ALT-DEL. On macOS, it is ⌘-SHIFT-OPTION-Q. By logging out, you'll lose any unsaved work, but at least you won't have to wait for a full reboot of the computer.

CONTROLLING MOUSE MOVEMENT

In this section, you'll learn how to move the mouse and track its position on the screen using PyAutoGUI, but first you need to understand how PyAutoGUI works with coordinates.

The mouse functions of PyAutoGUI use x- and y-coordinates. Figure 20-1 shows the coordinate system for the computer screen; it's similar to the coordinate system used for images, discussed in Chapter 19. The *origin*, where *x* and *y* are both zero, is at the upper-left corner of the screen. The x-coordinates increase going to the right, and the y-

coordinates increase going down. All coordinates are positive integers; there are no negative coordinates.
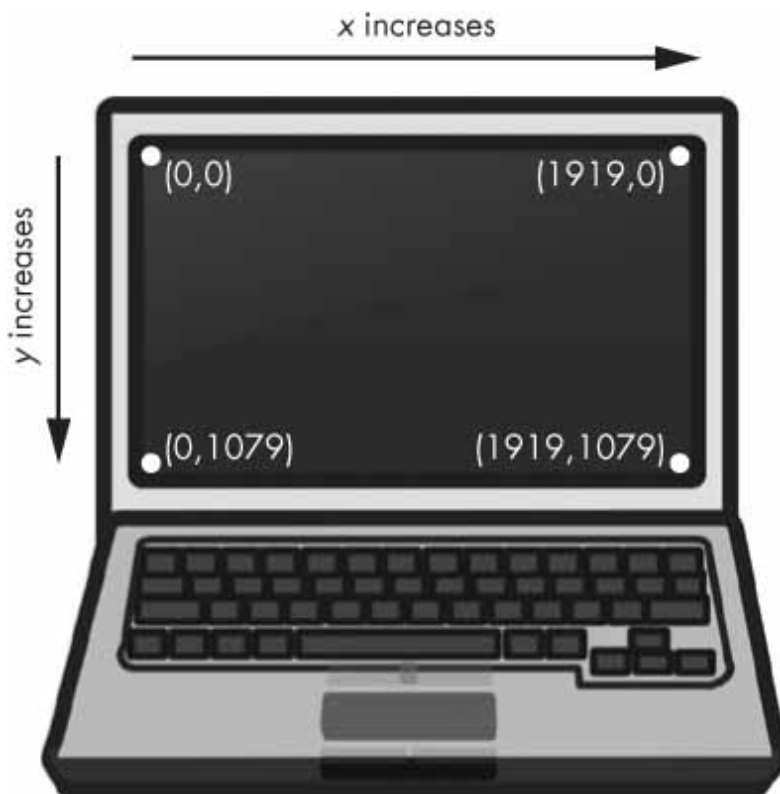

Figure 20-1: The coordinates of a computer screen with 1920×1080 resolution

Your *resolution* is how many pixels wide and tall your screen is. If your screen's resolution is set to 1920×1080, then the coordinate for the upper-left corner will be (0, 0), and the coordinate for the bottom-right corner will be (1919, 1079).

The `pyautogui.size()` function returns a two-integer tuple of the screen's width and height in pixels. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> wh = pyautogui.size() # Obtain the screen resolution.
>>> wh
Size(width=1920, height=1080)
>>> wh[0]
1920
>>> wh.width
1920
```

The `pyautogui.size()` function returns `(1920, 1080)` on a computer with a 1920×1080 resolution; depending on your screen's resolution, your return value may be different. The `size` object returned by `size()` is a named tuple. *Named tuples* have numeric indexes, like regular tuples, and attribute names, like objects: both `wh[0]` and `wh.width` evaluate to

the width of the screen. (Named tuples are beyond the scope of this book. Just remember that you can use them the same way you use tuples.)

Moving the Mouse

Now that you understand screen coordinates, let's move the mouse. The `pyautogui.moveTo()` function will instantly move the mouse cursor to a specified position on the screen. Integer values for the x- and y-coordinates make up the function's first and second arguments, respectively. An optional `duration` integer or float keyword argument specifies the number of seconds it should take to move the mouse to the destination. If you leave it out, the default is `0` for instantaneous movement. (All of the `duration` keyword arguments in PyAutoGUI functions are optional.) Enter the following into the interactive shell:

```
>>> import pyautogui
>>> for i in range(10): # Move mouse in a square.
...         pyautogui.moveTo(100, 100, duration=0.25)
...         pyautogui.moveTo(200, 100, duration=0.25)
...         pyautogui.moveTo(200, 200, duration=0.25)
...         pyautogui.moveTo(100, 200, duration=0.25)
```

This example moves the mouse cursor clockwise in a square pattern among the four coordinates provided a total of 10 times. Each movement takes a quarter of a second, as specified by the `duration=0.25` keyword argument. If you hadn't passed a third argument to any of the `pyautogui.moveTo()` calls, the mouse cursor would have instantly teleported from point to point.

The `pyautogui.move()` function moves the mouse cursor *relative to its current position*. The following example moves the mouse in the same square pattern, except it begins the square from wherever the mouse happens to be on the screen when the code starts running:

```
>>> import pyautogui
>>> for i in range(10):
...         pyautogui.move(100, 0, duration=0.25)    # right
...         pyautogui.move(0, 100, duration=0.25)    # down
...         pyautogui.move(-100, 0, duration=0.25)   # left
...         pyautogui.move(0, -100, duration=0.25)   # up
```

The `pyautogui.move()` function also takes three arguments: how many pixels to move horizontally to the right, how many pixels to move vertically downward, and (optionally)

how long it should take to complete the movement. A negative integer for the first or second argument will cause the mouse to move left or upward, respectively.

Getting the Mouse Position

You can determine the mouse's current position by calling the `pyautogui.position()` function, which will return a `Point` named tuple of the mouse cursor's x and y positions at the time of the function call. Enter the following into the interactive shell, moving the mouse around after each call:

```
>>> pyautogui.position() # Get current mouse position.
Point(x=311, y=622)
>>> pyautogui.position() # Get current mouse position again.
Point(x=377, y=481)
>>> p = pyautogui.position() # And again.
>>> p
Point(x=1536, y=637)
>>> p[0] # The x-coordinate is at index 0.
1536
>>> p.x # The x-coordinate is also in the x attribute.
1536
```

Of course, your return values will vary depending on where your mouse cursor is.

CONTROLLING MOUSE INTERACTION

Now that you know how to move the mouse and figure out where it is on the screen, you're ready to start clicking, dragging, and scrolling.

Clicking the Mouse

To send a virtual mouse click to your computer, call the `pyautogui.click()` method. By default, this click uses the left mouse button and takes place wherever the mouse cursor is currently located. You can pass x- and y-coordinates of the click as optional first and second arguments if you want it to take place somewhere other than the mouse's current position.

If you want to specify which mouse button to use, include the `button` keyword argument, with a value of `'left'`, `'middle'`, or `'right'`. For example, `pyautogui.click(100, 150, button='left')` will click the left mouse button at the coordinates (100, 150), while `pyautogui.click(200, 250, button='right')` will perform a right-click at (200, 250).

Enter the following into the interactive shell:

```
>>> import pyautogui
>>> pyautogui.click(10, 5) # Move mouse to (10, 5) and click.
```

You should see the mouse pointer move to near the top-left corner of your screen and click once. A full "click" is defined as pushing a mouse button down and then releasing it back up without moving the cursor. You can also perform a click by calling pyautogui.mouseDown(), which only pushes the mouse button down, and pyautogui.mouseUp(), which only releases the button. These functions have the same arguments as click(), and in fact, the click() function is just a convenient wrapper around these two function calls.

As a further convenience, the pyautogui.doubleClick() function will perform two clicks with the left mouse button, while the pyautogui.rightClick() and pyautogui.middleClick() functions will perform a click with the right and middle mouse buttons, respectively.

Dragging the Mouse

Dragging means moving the mouse while holding down one of the mouse buttons. For example, you can move files between folders by dragging the folder icons, or you can move appointments around in a calendar app.

PyAutoGUI provides the pyautogui.dragTo() and pyautogui.drag() functions to drag the mouse cursor to a new location or a location relative to its current one. The arguments for dragTo() and drag() are the same as moveTo() and move(): the x-coordinate/horizontal movement, the y-coordinate/vertical movement, and an optional duration of time. (macOS does not drag correctly when the mouse moves too quickly, so passing a duration keyword argument is recommended.)

To try these functions, open a graphics-drawing application such as MS Paint on Windows, Paintbrush on macOS, or GNU Paint on Linux. (If you don't have a drawing application, you can use the online one at *https://sumopaint.com/*.) I will use PyAutoGUI to draw in these applications.

With the mouse cursor over the drawing application's canvas and the Pencil or Brush tool selected, enter the following into a new file editor window and save it as *spiralDraw.py*:

```
   import pyautogui, time
 ? time.sleep(5)
 ? pyautogui.click()    # Click to make the window active.
   distance = 300
   change = 20
   while distance > 0:
    ? pyautogui.drag(distance, 0, duration=0.2)   # Move right.
```

```
? distance = distance - change

? pyautogui.drag(0, distance, duration=0.2)    # Move down.

? pyautogui.drag(-distance, 0, duration=0.2)  # Move left.

  distance = distance - change

  pyautogui.drag(0, -distance, duration=0.2)  # Move up.
```

When you run this program, there will be a five-second delay ? for you to move the mouse cursor over the drawing program's window with the Pencil or Brush tool selected. Then *spiralDraw.py* will take control of the mouse and click to make the drawing program's window active ?. The *active window* is the window that currently accepts keyboard input, and the actions you take—like typing or, in this case, dragging the mouse—will affect that window. The active window is also known as the *focused* or *foreground window*. Once the drawing program is active, *spiralDraw.py* draws a square spiral pattern like the one on the left of Figure 20-2. While you can also create a square spiral image by using the Pillow module discussed in Chapter 19, creating the image by controlling the mouse to draw it in MS Paint lets you make use of this program's various brush styles, like in Figure 20-2 on the right, as well as other advanced features, like gradients or the fill bucket. You can preselect the brush settings yourself (or have your Python code select these settings) and then run the spiral-drawing program.
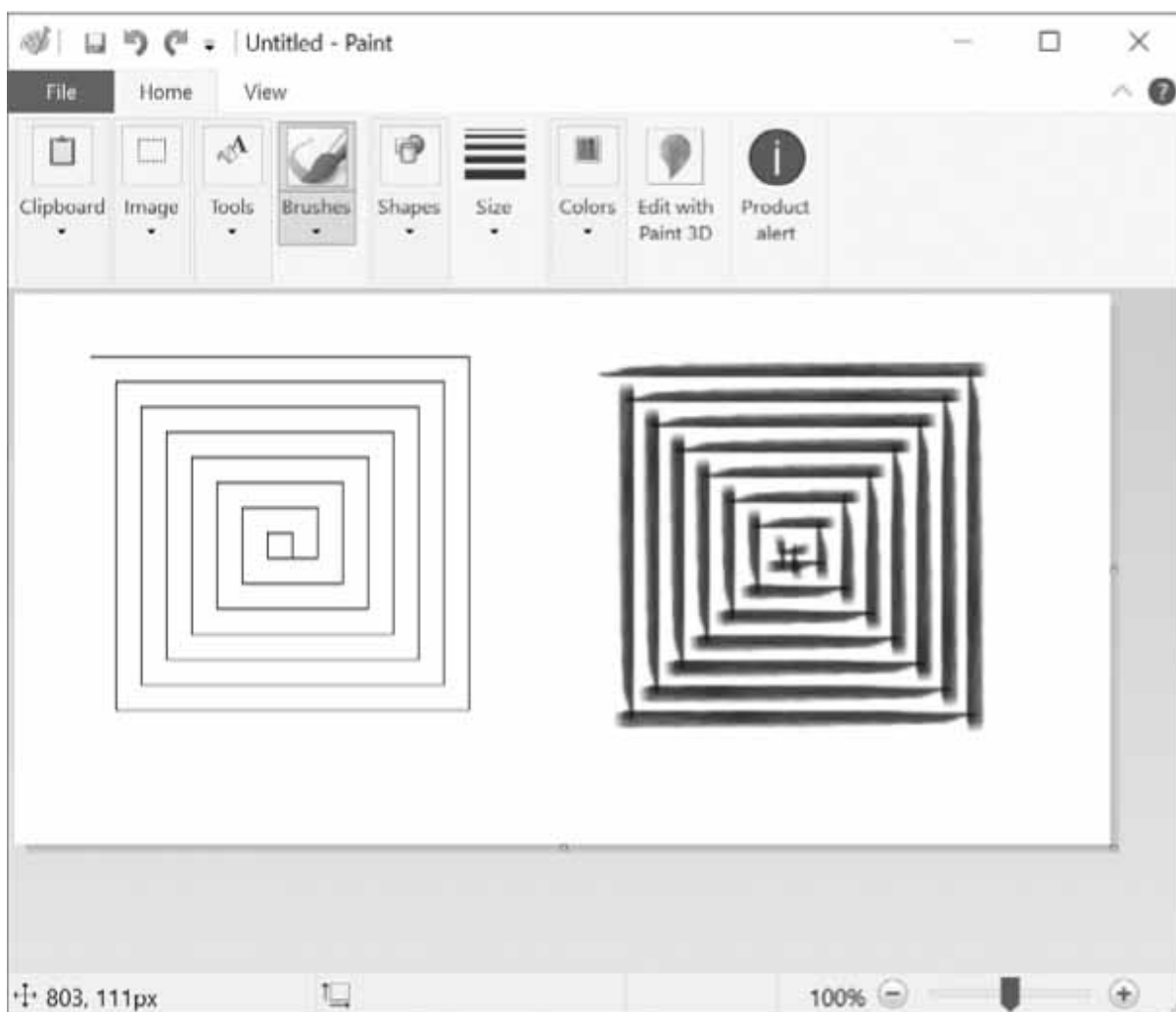

Figure 20-2: The results from the pyautogui.drag() *example, drawn with MS Paint's different brushes*

The distance variable starts at 200, so on the first iteration of the while loop, the first drag() call drags the cursor 200 pixels to the right, taking 0.2 seconds ?. distance is then decreased to 195 ?, and the second drag() call drags the cursor 195 pixels down ?. The third drag() call drags the cursor –195 horizontally (195 to the left) ?, distance is decreased to 190, and the last drag() call drags the cursor 190 pixels up. On each iteration, the mouse is dragged right, down, left, and up, and distance is slightly smaller than it was in the previous iteration. By looping over this code, you can move the mouse cursor to draw a square spiral.

You could draw this spiral by hand (or rather, by mouse), but you'd have to work slowly to be so precise. PyAutoGUI can do it in a few seconds!

> **NOTE**
>
> *At the time of this writing, PyAutoGUI can't send mouse clicks or keystrokes to certain programs, such as antivirus software (to prevent viruses from disabling the software) or video games on Windows (which use a different method of receiving mouse and keyboard input). You can check the online documentation at* https://pyautogui.readthedocs.io/ *to see if these features have been added.*

Scrolling the Mouse

The final PyAutoGUI mouse function is scroll(), which you pass an integer argument for how many units you want to scroll the mouse up or down. The size of a unit varies for each operating system and application, so you'll have to experiment to see exactly how far it scrolls in your particular situation. The scrolling takes place at the mouse cursor's current position. Passing a positive integer scrolls up, and passing a negative integer scrolls down. Run the following in Mu Editor's interactive shell while the mouse cursor is over the Mu Editor window:

```
>>> pyautogui.scroll(200)
```

You'll see Mu scroll upward if the mouse cursor is over a text field that can be scrolled up.

PLANNING YOUR MOUSE MOVEMENTS

One of the difficulties of writing a program that will automate clicking the screen is finding the x- and y-coordinates of the things you'd like to click. The pyautogui.mouseInfo() function can help you with this.

The `pyautogui.mouseInfo()` function is meant to be called from the interactive shell, rather than as part of your program. It launches a small application named MouseInfo that's included with PyAutoGUI. The window for the application looks like Figure 20-3.
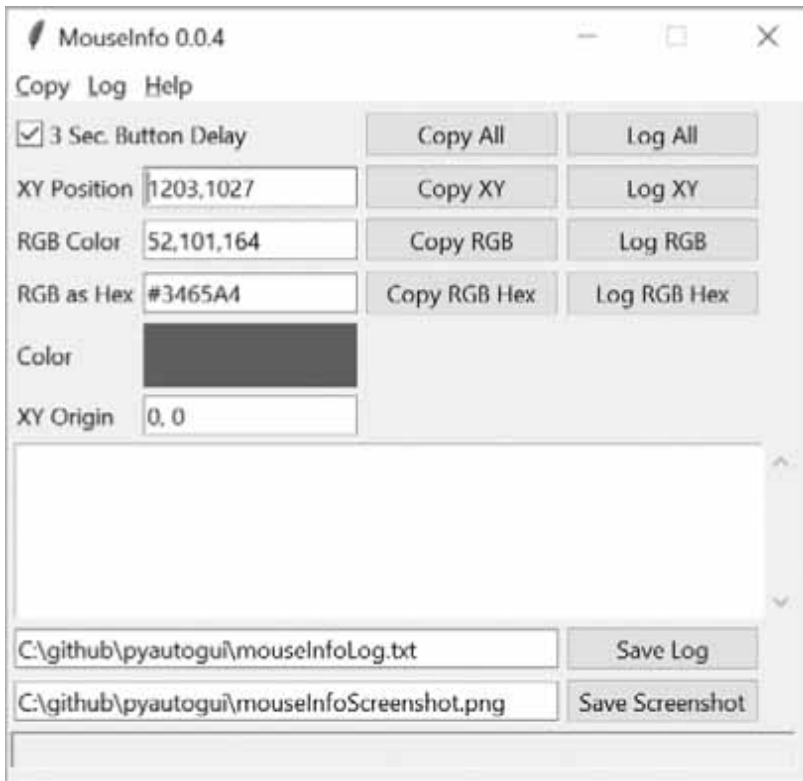
Figure 20-3: The MouseInfo application's window

Enter the following into the interactive shell:

```
>>> import pyautogui
>>> pyautogui.mouseInfo()
```

This makes the MouseInfo window appear. This window gives you information about the mouse's cursor current position, as well the color of the pixel underneath the mouse cursor, as a three-integer RGB tuple and as a hex value. The color itself appears in the color box in the window.

To help you record this coordinate or pixel information, you can click one of the eight Copy or Log buttons. The Copy All, Copy XY, Copy RGB, and Copy RGB Hex buttons will copy their respective information to the clipboard. The Log All, Log XY, Log RGB, and Log RGB Hex buttons will write their respective information to the large text field in the window. You can save the text in this log text field by clicking the Save Log button.

By default, the 3 Sec. Button Delay checkbox is checked, causing a three-second delay between clicking a Copy or Log button and the copying or logging taking place. This gives you a short amount of time in which to click the button and then move the mouse into your desired position. It may be easier to uncheck this box, move the mouse into position, and press the F1 to F8 keys to copy or log the mouse position. You can

look at the Copy and Log menus at the top of the MouseInfo window to find out which key maps to which buttons.

For example, uncheck the 3 Sec. Button Delay, then move the mouse around the screen while pressing the F6 button, and notice how the x- and y-coordinates of the mouse are recorded in the large text field in the middle of the window. You can later use these coordinates in your PyAutoGUI scripts.

For more information on MouseInfo, review the complete documentation at *https://mouseinfo.readthedocs.io/*.

WORKING WITH THE SCREEN

Your GUI automation programs don't have to click and type blindly. PyAutoGUI has screenshot features that can create an image file based on the current contents of the screen. These functions can also return a Pillow `Image` object of the current screen's appearance. If you've been skipping around in this book, you'll want to read Chapter 17 and install the `pillow` module before continuing with this section.

On Linux computers, the `scrot` program needs to be installed to use the screenshot functions in PyAutoGUI. In a Terminal window, run **sudo apt-get install scrot** to install this program. If you're on Windows or macOS, skip this step and continue with the section.

Getting a Screenshot

To take screenshots in Python, call the `pyautogui.screenshot()` function. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> im = pyautogui.screenshot()
```

The `im` variable will contain the `Image` object of the screenshot. You can now call methods on the `Image` object in the `im` variable, just like any other `Image` object. Chapter 19 has more information about `Image` objects.

Analyzing the Screenshot

Say that one of the steps in your GUI automation program is to click a gray button. Before calling the `click()` method, you could take a screenshot and look at the pixel where the script is about to click. If it's not the same gray as the gray button, then your program knows something is wrong. Maybe the window moved unexpectedly, or maybe a pop-up dialog has blocked the button. At this point, instead of continuing—and

possibly wreaking havoc by clicking the wrong thing—your program can "see" that it isn't clicking the right thing and stop itself.

You can obtain the RGB color value of a particular pixel on the screen with the pixel() function. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> pyautogui.pixel((0, 0))
(176, 176, 175)
>>> pyautogui.pixel((50, 200))
(130, 135, 144)
```

Pass pixel() a tuple of coordinates, like (0, 0) or (50, 200), and it'll tell you the color of the pixel at those coordinates in your image. The return value from pixel() is an RGB tuple of three integers for the amount of red, green, and blue in the pixel. (There is no fourth value for alpha, because screenshot images are fully opaque.)

PyAutoGUI's pixelMatchesColor() function will return True if the pixel at the given x- and y-coordinates on the screen matches the given color. The first and second arguments are integers for the x- and y-coordinates, and the third argument is a tuple of three integers for the RGB color the screen pixel must match. Enter the following into the interactive shell:

```
   >>> import pyautogui
? >>> pyautogui.pixel((50, 200))
   (130, 135, 144)
? >>> pyautogui.pixelMatchesColor(50, 200, (130, 135, 144))
   True
? >>> pyautogui.pixelMatchesColor(50, 200, (255, 135, 144))
   False
```

After using pixel() to get an RGB tuple for the color of a pixel at specific coordinates ?, pass the same coordinates and RGB tuple to pixelMatchesColor() ?, which should return True. Then change a value in the RGB tuple and call pixelMatchesColor() again for the same coordinates ?. This should return false. This method can be useful to call whenever your GUI automation programs are about to call click(). Note that the color at the given coordinates must *exactly* match. If it is even slightly different—for example, (255, 255, 254) instead of (255, 255, 255)—then pixelMatchesColor() will return False.

IMAGE RECOGNITION

But what if you do not know beforehand where PyAutoGUI should click? You can use image recognition instead. Give PyAutoGUI an image of what you want to click, and let it figure out the coordinates.

For example, if you have previously taken a screenshot to capture the image of a Submit button in *submit.png*, the `locateOnScreen()` function will return the coordinates where that image is found. To see how `locateOnScreen()` works, try taking a screenshot of a small area on your screen; then save the image and enter the following into the interactive shell, replacing `'submit.png'` with the filename of your screenshot:

```
>>> import pyautogui
>>> b = pyautogui.locateOnScreen('submit.png')
>>> b
Box(left=643, top=745, width=70, height=29)
>>> b[0]
643
>>> b.left
643
```

The `Box` object is a named tuple that `locateOnScreen()` returns and has the x-coordinate of the left edge, the y-coordinate of the top edge, the width, and the height for the first place on the screen the image was found. If you're trying this on your computer with your own screenshot, your return value will be different from the one shown here.

If the image cannot be found on the screen, `locateOnScreen()` returns `None`. Note that the image on the screen must match the provided image perfectly in order to be recognized. If the image is even a pixel off, `locateOnScreen()` raises an `ImageNotFoundException` exception. If you've changed your screen resolution, images from previous screenshots might not match the images on your current screen. You can change the scaling in the display settings of your operating system, as shown in Figure 20-4.
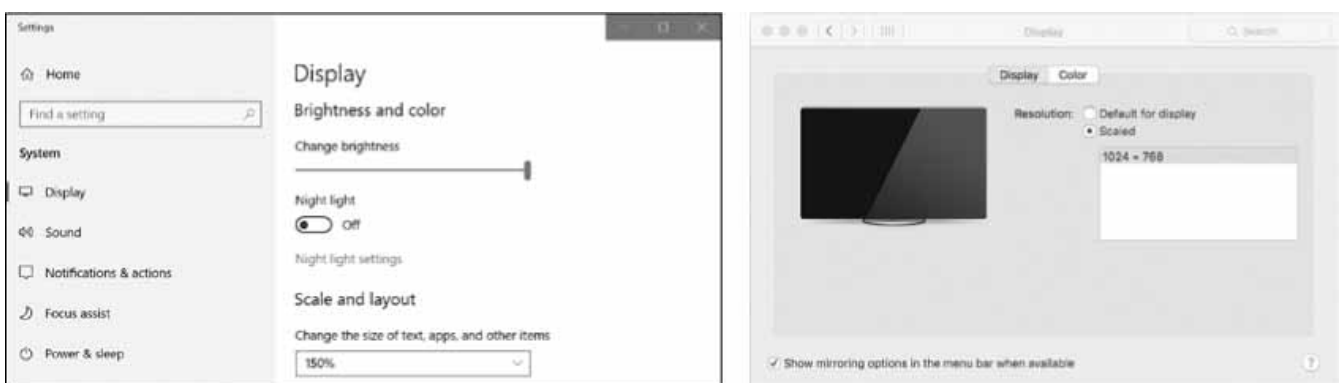

Figure 20-4: The scale display settings in Windows 10 (left) and macOS (right)

If the image can be found in several places on the screen, `locateAllOnScreen()` will return a `Generator` object. Generators are beyond the scope of this book, but you can pass them to `list()` to return a list of four-integer tuples. There will be one four-integer tuple for each location where the image is found on the screen. Continue the interactive shell example by entering the following (and replacing `'submit.png'` with your own image filename):

```
>>> list(pyautogui.locateAllOnScreen('submit.png'))
[(643, 745, 70, 29), (1007, 801, 70, 29)]
```

Each of the four-integer tuples represents an area on the screen. In the example above, the image appears in two locations. If your image is only found in one area, then using `list()` and `locateAllOnScreen()` returns a list containing just one tuple.

Once you have the four-integer tuple for the specific image you want to select, you can click the center of this area by passing the tuple to `click()`. Enter the following into the interactive shell:

```
>>> pyautogui.click((643, 745, 70, 29))
```

As a shortcut, you can also pass the image filename directly to the `click()` function:

```
>>> pyautogui.click('submit.png')
```

The `moveTo()` and `dragTo()` functions also accept image filename arguments. Remember `locateOnScreen()` raises an exception if it can't find the image on the screen, so you should call it from inside a `try` statement:

```
try:
    location = pyautogui.locateOnScreen('submit.png')
except:
    print('Image could not be found.')
```

Without the `try` and `except` statements, the uncaught exception would crash your program. Since you can't be sure that your program will always find the image, it's a good idea to use the `try` and `except` statements when calling `locateOnScreen()`.

GETTING WINDOW INFORMATION

Image recognition is a fragile way to find things on the screen; if a single pixel is a different color, then `pyautogui.locateOnScreen()` won't find the image. If you need to find

where a particular window is on the screen, it's faster and more reliable to use PyAutoGUI's window features.

NOTE

As of version 0.9.46, PyAutoGUI's window features work only on Windows, not on macOS or Linux. These features come from PyAutoGUI's inclusion of the PyGetWindow module.

Obtaining the Active Window

The active window on your screen is the window currently in the foreground and accepting keyboard input. If you're currently writing code in the Mu Editor, the Mu Editor's window is the active window. Of all the windows on your screen, only one will be active at a time.

In the interactive shell, call the `pyautogui.getActiveWindow()` function to get a `Window` object (technically a `Win32Window` object when run on Windows).

Once you have that `Window` object, you can retrieve any of the object's attributes, which describe its size, position, and title:

`left, right, top, bottom` A single integer for the x- or y-coordinate of the window's side

`topleft, topright, bottomleft, bottomright` A named tuple of two integers for the (x, y) coordinates of the window's corner

`midleft, midright, midleft, midright` A named tuple of two integers for the (x, y) coordinate of the middle of the window's side

`width, height` A single integer for one of the window's dimensions, in pixels

`size` A named tuple of two integers for the (width, height) of the window

`area` A single integer representing the area of the window, in pixels

`center` A named tuple of two integers for the (x, y) coordinate of the window's center

`centerx, centery` A single integer for the x- or y-coordinate of the window's center

`box` A named tuple of four integers for the (left, top, width, height) measurements of the window

`title` A string of the text in the title bar at the top of the window

To get the window's position, size, and title information from the `window` object, for example, enter the following into the interactive shell:

```
>>> import pyautogui
>>> fw = pyautogui.getActiveWindow()
>>> fw
Win32Window(hWnd=2034368)
>>> str(fw)
'<Win32Window left="500", top="300", width="2070", height="1208", title="Mu 1.0.1 – test1.py">'
>>> fw.title
'Mu 1.0.1 – test1.py'
>>> fw.size
(2070, 1208)
>>> fw.left, fw.top, fw.right, fw.bottom
(500, 300, 2070, 1208)
>>> fw.topleft
(256, 144)
>>> fw.area
2500560
>>> pyautogui.click(fw.left + 10, fw.top + 20)
```

You can now use these attributes to calculate precise coordinates within a window. If you know that a button you want to click is always 10 pixels to the right of and 20 pixels down from the window's top-left corner, and the window's top-left corner is at screen coordinates (300, 500), then calling `pyautogui.click(310, 520)` (or `pyautogui.click(fw.left + 10, fw.top + 20)` if `fw` contains the `Window` object for the window) will click the button. This way, you won't have to rely on the slower, less reliable `locateOnScreen()` function to find the button for you.

Other Ways of Obtaining Windows

While `getActiveWindow()` is useful for obtaining the window that is active at the time of the function call, you'll need to use some other function to obtain `Window` objects for the other windows on the screen.

The following four functions return a list of `Window` objects. If they're unable to find any windows, they return an empty list:

pyautogui.getAllWindows() Returns a list of `Window` objects for every visible window on the screen.

pyautogui.getWindowsAt(x, y) Returns a list of `Window` objects for every visible window that includes the point (x, y).

pyautogui.getWindowsWithTitle(title) Returns a list of `Window` objects for every visible window that includes the string `title` in its title bar.

pyautogui.getActiveWindow() Returns the `Window` object for the window that is currently receiving keyboard focus.

PyAutoGUI also has a `pyautogui.getAllTitles()` function, which returns a list of strings of every visible window.

Manipulating Windows

Windows attributes can do more than just tell you the size and position of the window. You can also set their values in order to resize or move the window. For example, enter the following into the interactive shell:

```
>>> import pyautogui
>>> fw = pyautogui.getActiveWindow()
? >>> fw.width # Gets the current width of the window.
1669
? >>> fw.topleft # Gets the current position of the window.
(174, 153)
? >>> fw.width = 1000 # Resizes the width.
? >>> fw.topleft = (800, 400) # Moves the window.
```

First, we use the `Window` object's attributes to find out information about the window's size ? and position ?. After calling these functions in Mu Editor, the window should move ? and become narrower ?, as in Figure 20-5.
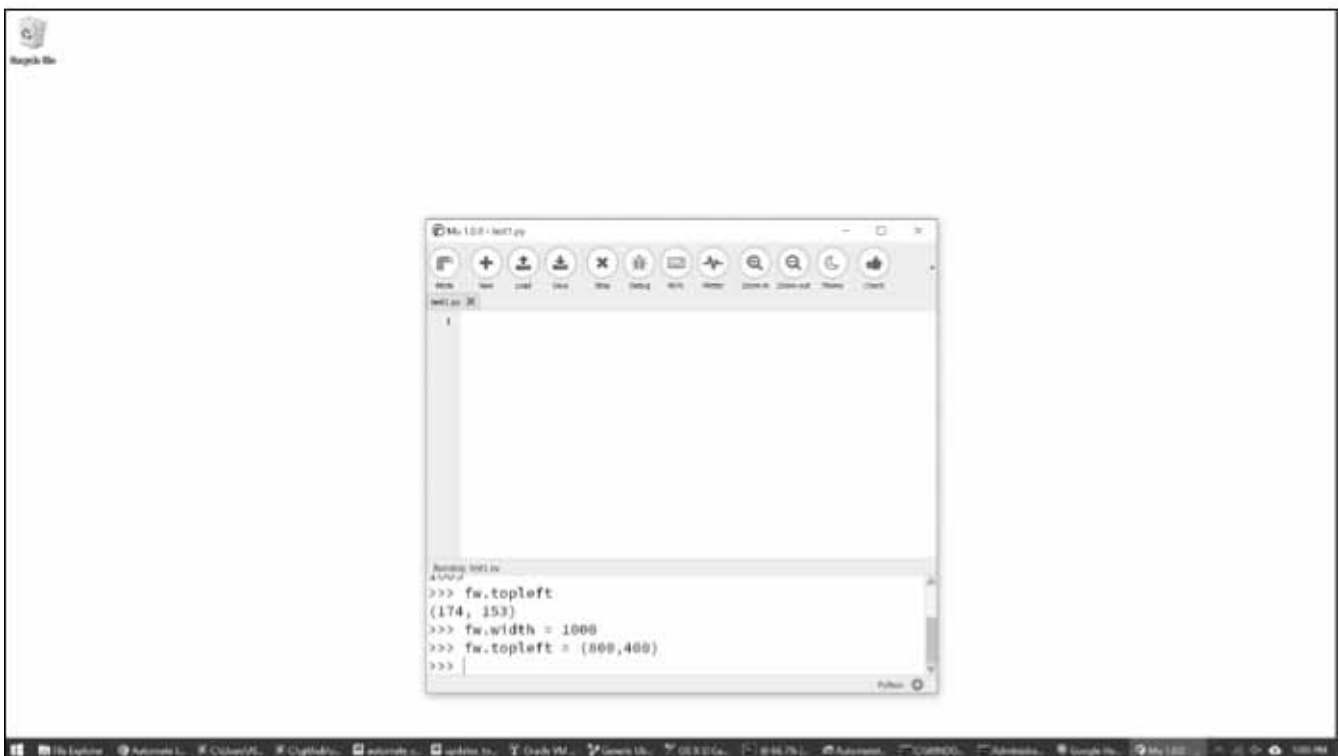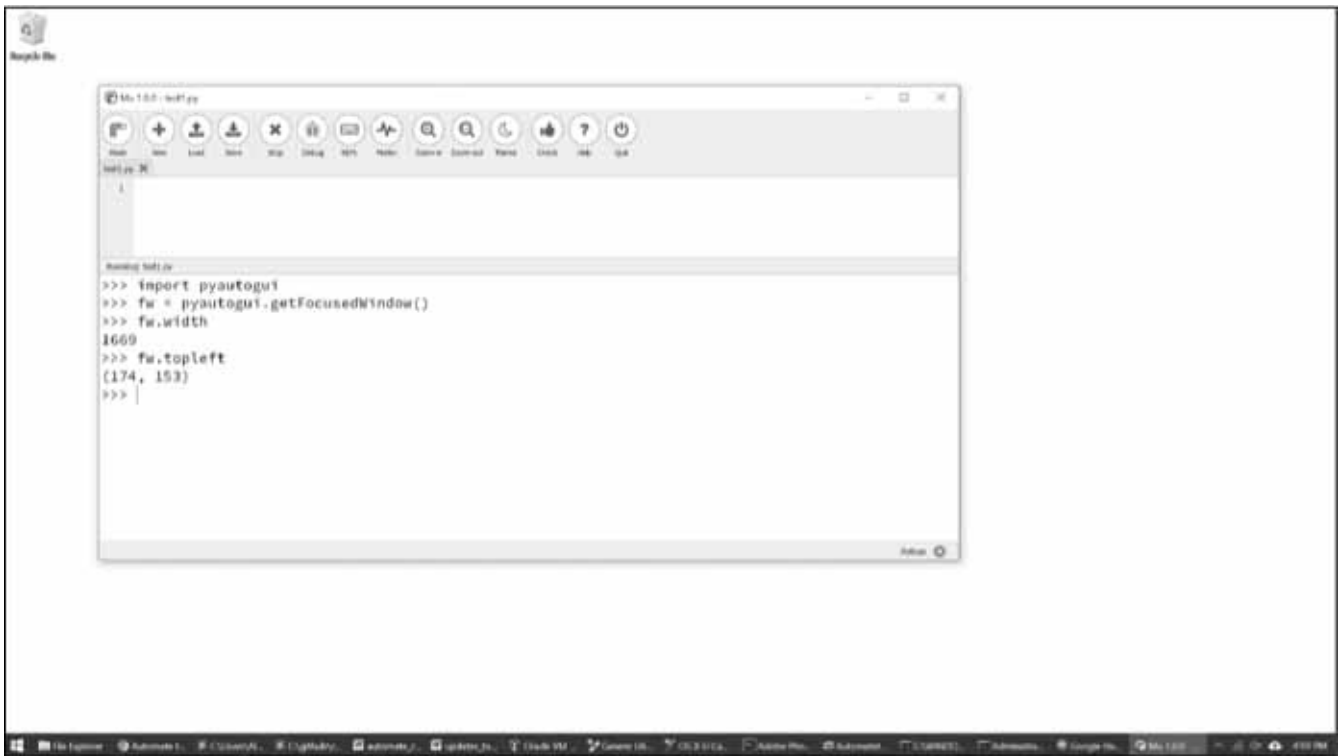
Figure 20-5: The Mu Editor window before (top) and after (bottom) using the Window *object attributes to move and resize it*

You can also find out and change the window's minimized, maximized, and activated states. Try entering the following into the interactive shell:

```
>>> import pyautogui
>>> fw = pyautogui.getActiveWindow()
? >>> fw.isMaximized # Returns True if window is maximized.
   False
? >>> fw.isMinimized # Returns True if window is minimized.
   False
```

```
? >>> fw.isActive # Returns True if window is the active window.

   True

? >>> fw.maximize() # Maximizes the window.

   >>> fw.isMaximized

   True

? >>> fw.restore() # Undoes a minimize/maximize action.

? >>> fw.minimize() # Minimizes the window.

   >>> import time

   >>> # Wait 5 seconds while you activate a different window:

? >>> time.sleep(5); fw.activate()

? >>> fw.close() # This will close the window you're typing in.
```

The `isMaximized` ?, `isMinimized` ?, and `isActive` ? attributes contain Boolean values that indicate whether the window is currently in that state. The `maximize()` ?, `minimize()` ?, `activate()` ?, and `restore()` ? methods change the window's state. After you maximize or minimize the window with `maximize()` or `minimize()`, the `restore()` method will restore the window to its former size and position.

The `close()` method ? will close a window. Be careful with this method, as it may bypass any message dialogs asking you to save your work before quitting the application.

The complete documentation for PyAutoGUI's window-controlling feature can be found at *https://pyautogui.readthedocs.io/*. You can also use these features separately from PyAutoGUI with the PyGetWindow module, documented at *https://pygetwindow.readthedocs.io/*.

CONTROLLING THE KEYBOARD

PyAutoGUI also has functions for sending virtual keypresses to your computer, which enables you to fill out forms or enter text into applications.

Sending a String from the Keyboard

The `pyautogui.write()` function sends virtual keypresses to the computer. What these keypresses do depends on what window is active and what text field has focus. You may want to first send a mouse click to the text field you want in order to ensure that it has focus.

As a simple example, let's use Python to automatically type the words *Hello, world!* into a file editor window. First, open a new file editor window and position it in the

upper-left corner of your screen so that PyAutoGUI will click in the right place to bring it into focus. Next, enter the following into the interactive shell:

```
>>> pyautogui.click(100, 200); pyautogui.write('Hello, world!')
```

Notice how placing two commands on the same line, separated by a semicolon, keeps the interactive shell from prompting you for input between running the two instructions. This prevents you from accidentally bringing a new window into focus between the `click()` and `write()` calls, which would mess up the example.

Python will first send a virtual mouse click to the coordinates (100, 200), which should click the file editor window and put it in focus. The `write()` call will send the text *Hello, world!* to the window, making it look like Figure 20-6. You now have code that can type for you!
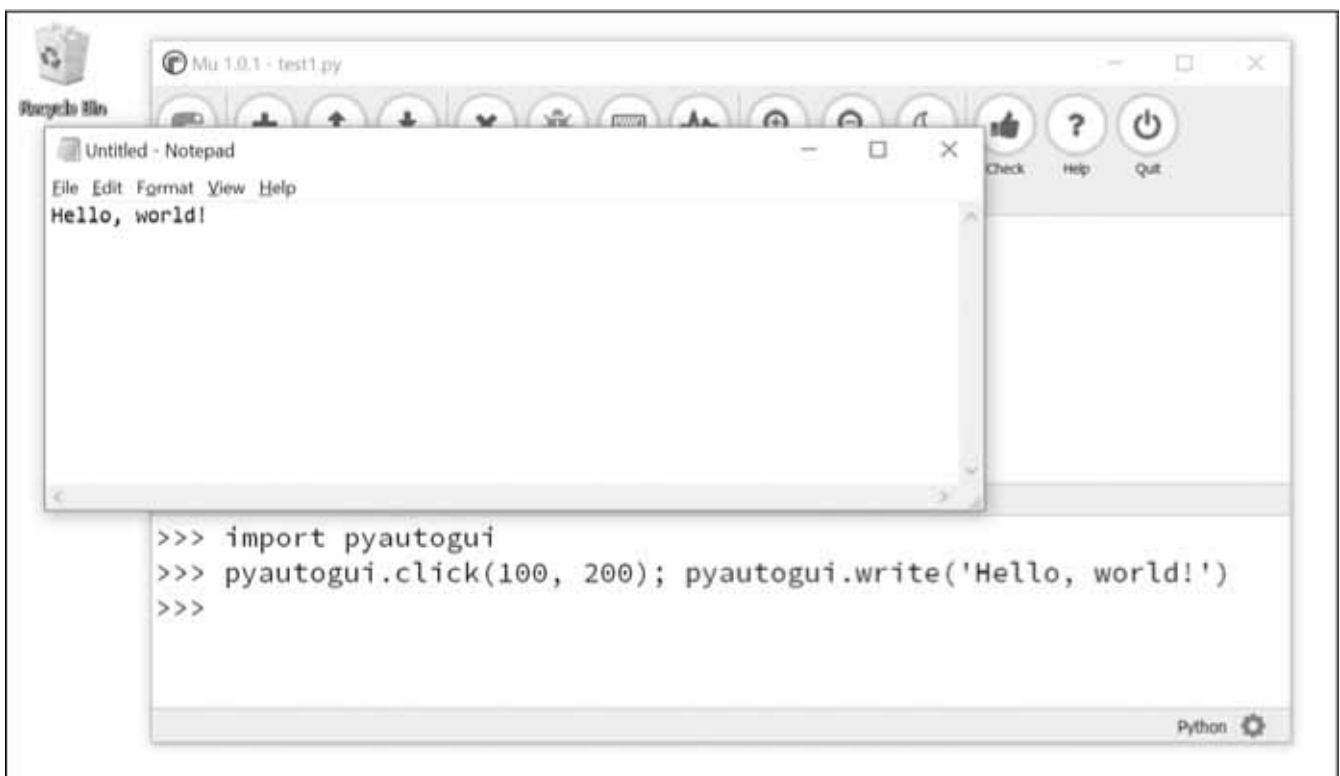

Figure 20-6: Using PyAutogGUI to click the file editor window and type Hello, world! *into it*

By default, the `write()` function will type the full string instantly. However, you can pass an optional second argument to add a short pause between each character. This second argument is an integer or float value of the number of seconds to pause. For example, `pyautogui.write('Hello, world!', 0.25)` will wait a quarter-second after typing *H*, another quarter-second after *e*, and so on. This gradual typewriter effect may be useful for slower applications that can't process keystrokes fast enough to keep up with PyAutoGUI.

For characters such as *A* or *!*, PyAutoGUI will automatically simulate holding down the SHIFT key as well.

Key Names

Not all keys are easy to represent with single text characters. For example, how do you represent SHIFT or the left arrow key as a single character? In PyAutoGUI, these keyboard keys are represented by short string values instead: `'esc'` for the ESC key or `'enter'` for the ENTER key.

Instead of a single string argument, a list of these keyboard key strings can be passed to `write()`. For example, the following call presses the A key, then the B key, then the left arrow key twice, and finally the X and Y keys:

```
>>> pyautogui.write(['a', 'b', 'left', 'left', 'X', 'Y'])
```

Because pressing the left arrow key moves the keyboard cursor, this will output *XYab*. Table 20-1 lists the PyAutoGUI keyboard key strings that you can pass to `write()` to simulate pressing any combination of keys.

You can also examine the `pyautogui.KEYBOARD_KEYS` list to see all possible keyboard key strings that PyAutoGUI will accept. The `'shift'` string refers to the left SHIFT key and is equivalent to `'shiftleft'`. The same applies for `'ctrl'`, `'alt'`, and `'win'` strings; they all refer to the left-side key.

Table 20-1: `PyKeyboard` Attributes

Keyboard key string	Meaning
`'a'`, `'b'`, `'c'`, `'A'`, `'B'`, `'C'`, `'1'`, `'2'`, `'3'`, `'!'`, `'@'`, `'#'`, and so on	The keys for single characters
`'enter'` (or `'return'` or `'\n'`)	The ENTER key
`'esc'`	The ESC key
`'shiftleft'`, `'shiftright'`	The left and right SHIFT keys
`'altleft'`, `'altright'`	The left and right ALT keys
`'ctrlleft'`, `'ctrlright'`	The left and right CTRL keys
`'tab'` (or `'\t'`)	The TAB key
`'backspace'`, `'delete'`	The BACKSPACE and DELETE keys
`'pageup'`, `'pagedown'`	The PAGE UP and PAGE DOWN keys
`'home'`, `'end'`	The HOME and END keys
`'up'`, `'down'`, `'left'`, `'right'`	The up, down, left, and right arrow keys
`'f1'`, `'f2'`, `'f3'`, and so on	The F1 to F12 keys

Keyboard key string	Meaning
`'volumemute'`, `'volumedown'`, `'volumeup'`	The mute, volume down, and volume up keys (some keyboards do not have these keys, but your operating system will still be able to understand these simulated keypresses)
`'pause'`	The PAUSE key
`'capslock'`, `'numlock'`, `'scrolllock'`	The CAPS LOCK, NUM LOCK, and SCROLL LOCK keys
`'insert'`	The INS or INSERT key
`'printscreen'`	The PRTSC or PRINT SCREEN key
`'winleft'`, `'winright'`	The left and right WIN keys (on Windows)
`'command'`	The Command (⌘) key (on macOS)
`'option'`	The OPTION key (on macOS)

Pressing and Releasing the Keyboard

Much like the `mouseDown()` and `mouseUp()` functions, `pyautogui.keyDown()` and `pyautogui.keyUp()` will send virtual keypresses and releases to the computer. They are passed a keyboard key string (see Table 20-1) for their argument. For convenience, PyAutoGUI provides the `pyautogui.press()` function, which calls both of these functions to simulate a complete keypress.

Run the following code, which will type a dollar sign character (obtained by holding the SHIFT key and pressing 4):

```
>>> pyautogui.keyDown('shift'); pyautogui.press('4'); pyautogui.keyUp('shift')
```

This line presses down SHIFT, presses (and releases) 4, and then releases SHIFT. If you need to type a string into a text field, the `write()` function is more suitable. But for applications that take single-key commands, the `press()` function is the simpler approach.

Hotkey Combinations

A *hotkey* or *shortcut* is a combination of keypresses to invoke some application function. The common hotkey for copying a selection is CTRL-C (on Windows and Linux) or ⌘-C (on macOS). The user presses and holds the CTRL key, then presses the C key, and then releases the C and CTRL keys. To do this with PyAutoGUI's `keyDown()` and `keyUp()`

functions, you would have to enter the following:

```
pyautogui.keyDown('ctrl')
pyautogui.keyDown('c')
pyautogui.keyUp('c')
pyautogui.keyUp('ctrl')
```

This is rather complicated. Instead, use the `pyautogui.hotkey()` function, which takes multiple keyboard key string arguments, presses them in order, and releases them in the reverse order. For the CTRL-C example, the code would simply be as follows:

```
pyautogui.hotkey('ctrl', 'c')
```

This function is especially useful for larger hotkey combinations. In Word, the CTRL-ALT-SHIFT-S hotkey combination displays the Style pane. Instead of making eight different function calls (four `keyDown()` calls and four `keyUp()` calls), you can just call `hotkey('ctrl', 'alt', 'shift', 's')`.

SETTING UP YOUR GUI AUTOMATION SCRIPTS

GUI automation scripts are a great way to automate the boring stuff, but your scripts can also be finicky. If a window is in the wrong place on a desktop or some pop-up appears unexpectedly, your script could be clicking on the wrong things on the screen. Here are some tips for setting up your GUI automation scripts:

- Use the same screen resolution each time you run the script so that the position of windows doesn't change.
- The application window that your script clicks should be maximized so that its buttons and menus are in the same place each time you run the script.
- Add generous pauses while waiting for content to load; you don't want your script to begin clicking before the application is ready.
- Use `locateOnScreen()` to find buttons and menus to click, rather than relying on XY coordinates. If your script can't find the thing it needs to click, stop the program rather than let it continue blindly clicking.
- Use `getWindowsWithTitle()` to ensure that the application window you think your script is clicking on exists, and use the `activate()` method to put that window in the foreground.
- Use the `logging` module from Chapter 11 to keep a log file of what your script has done. This way, if you have to stop your script halfway through a process, you can

- change it to pick up from where it left off.

- Add as many checks as you can to your script. Think about how it could fail if an unexpected pop-up window appears or if your computer loses its internet connection.

- You may want to supervise the script when it first begins to ensure that it's working correctly.

You might also want to put a pause at the start of your script so the user can set up the window the script will click on. PyAutoGUI has a `sleep()` function that acts identically to `time.sleep()` (it just frees you from having to also add `import time` to your scripts). There is also a `countdown()` function that prints numbers counting down to give the user a visual indication that the script will continue soon. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> pyautogui.sleep(3) # Pauses the program for 3 seconds.
>>> pyautogui.countdown(10) # Counts down over 10 seconds.
10 9 8 7 6 5 4 3 2 1
>>> print('Starting in ', end=''); pyautogui.countdown(3)
Starting in 3 2 1
```

These tips can help make your GUI automation scripts easier to use and more able to recover from unforeseen circumstances.

REVIEW OF THE PYAUTOGUI FUNCTIONS

Since this chapter covered many different functions, here is a quick summary reference:

`moveTo(x, y)` Moves the mouse cursor to the given x and y coordinates.

`move(xOffset, yOffset)` Moves the mouse cursor relative to its current position.

`dragTo(x, y)` Moves the mouse cursor while the left button is held down.

`drag(xOffset, yOffset)` Moves the mouse cursor relative to its current position while the left button is held down.

`click(x, y, button)` Simulates a click (left button by default).

`rightClick()` Simulates a right-button click.

`middleClick()` Simulates a middle-button click.

`doubleClick()` Simulates a double left-button click.

`mouseDown(x, y, button)` Simulates pressing down the given button at the position x, y.

mouseUp(x, y, button) Simulates releasing the given button at the position *x, y*.

scroll(units) Simulates the scroll wheel. A positive argument scrolls up; a negative argument scrolls down.

write(message) Types the characters in the given message string.

write([key1, key2, key3]) Types the given keyboard key strings.

press(key) Presses the given keyboard key string.

keyDown(key) Simulates pressing down the given keyboard key.

keyUp(key) Simulates releasing the given keyboard key.

hotkey([key1, key2, key3]) Simulates pressing the given keyboard key strings down in order and then releasing them in reverse order.

screenshot() Returns a screenshot as an **Image** object. (See Chapter 19 for information on **Image** objects.)

getActiveWindow(), getAllWindows(), getWindowsAt(), and getWindowsWithTitle() These functions return Window objects that can resize and reposition application windows on the desktop.

getAllTitles() Returns a list of strings of the title bar text of every window on the desktop.

CAPTCHAS AND COMPUTER ETHICS

"Completely Automated Public Turing test to tell Computers and Humans Apart" or "captchas" are those small tests that ask you to type the letters in a distorted picture or click on photos of fire hydrants. These are tests that are easy, if annoying, for humans to pass but nearly impossible for software to solve. After reading this chapter, you can see how easy it is to write a script that could, say, sign up for billions of free email accounts or flood users with harassing messages. Captchas mitigate this by requiring a step that only a human can pass.

However not all websites implement captchas, and these can be vulnerable to abuse by unethical programmers. Learning to code is a powerful and exciting skill, and you may be tempted to misuse this power for personal gain or even just to show off. But just as an unlocked door isn't justification for trespass, the responsibility for your programs falls upon you, the programmer. There is nothing clever about circumventing systems to cause harm, invade privacy, or gain unfair advantage. I hope that my efforts in writing this book enable you to become your most productive self, rather than a mercenary one.

PROJECT: AUTOMATIC FORM FILLER

Of all the boring tasks, filling out forms is the most dreaded of chores. It's only fitting that now, in the final chapter project, you will slay it. Say you have a huge amount of data in a spreadsheet, and you have to tediously retype it into some other application's form interface—with no intern to do it for you. Although some applications will have an Import feature that will allow you to upload a spreadsheet with the information, sometimes it seems that there is no other way than mindlessly clicking and typing for hours on end. You've come this far in this book; you know that *of course* must be a way to automate this boring task.

The form for this project is a Google Docs form that you can find at *https://autbor.com/form*. It looks like Figure 20-7.

Figure 20-7: The form used for this project

At a high level, here's what your program should do:

1. Click the first text field of the form.
2. Move through the form, typing information into each field.
3. Click the Submit button.
4. Repeat the process with the next set of data.

This means your code will need to do the following:

1. Call `pyautogui.click()` to click the form and Submit button.
2. Call `pyautogui.write()` to enter text into the fields.
3. Handle the `KeyboardInterrupt` exception so the user can press CTRL-C to quit.

Open a new file editor window and save it as *formFiller.py*.

Step 1: Figure Out the Steps

Before writing code, you need to figure out the exact keystrokes and mouse clicks that will fill out the form once. The application launched by calling `pyautogui.mouseInfo()` can help you figure out specific mouse coordinates. You need to know only the coordinates of the first text field. After clicking the first field, you can just press TAB to move focus to the next field. This will save you from having to figure out the x- and y-coordinates to click for every field.

Here are the steps for entering data into the form:

1. Put the keyboard focus on the Name field so that pressing keys types text into the field.
2. Type a name and then press TAB.
3. Type a greatest fear and then press TAB.
4. Press the down arrow key the correct number of times to select the wizard power source: once for *wand*, twice for *amulet*, three times for *crystal ball*, and four times for *money*. Then press TAB. (Note that on macOS, you will have to press the down arrow key one more time for each option. For some browsers, you may need to press ENTER as well.)
5. Press the right arrow key to select the answer to the RoboCop question. Press it once for *2*, twice for *3*, three times for *4*, or four times for *5* or just press the spacebar to select *1* (which is highlighted by default). Then press TAB.
6. Type an additional comment and then press TAB.
7. Press ENTER to "click" the Submit button.

8. After submitting the form, the browser will take you to a page where you will need to follow a link to return to the form page.

Different browsers on different operating systems might work slightly differently from the steps given here, so check that these keystroke combinations work for your computer before running your program.

Step 2: Set Up Coordinates

Load the example form you downloaded (Figure 20-7) in a browser by going to *https://autbor.com/form*.

Make your source code look like the following:

```python
#! python3
# formFiller.py - Automatically fills in the form.

import pyautogui, time

# TODO: Give the user a chance to kill the script.

# TODO: Wait until the form page has loaded.

# TODO: Fill out the Name Field.

# TODO: Fill out the Greatest Fear(s) field.

# TODO: Fill out the Source of Wizard Powers field.

# TODO: Fill out the RoboCop field.

# TODO: Fill out the Additional Comments field.

# TODO: Click Submit.

# TODO: Wait until form page has loaded.

# TODO: Click the Submit another response link.
```

Now you need the data you actually want to enter into this form. In the real world, this data might come from a spreadsheet, a plaintext file, or a website, and it would

require additional code to load into the program. But for this project, you'll just hardcode all this data in a variable. Add the following to your program:

```python3
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

formData = [{'name': 'Alice', 'fear': 'eavesdroppers', 'source': 'wand',
             'robocop': 4, 'comments': 'Tell Bob I said hi.'},
            {'name': 'Bob', 'fear': 'bees', 'source': 'amulet', 'robocop': 4,
             'comments': 'n/a'},
            {'name': 'Carol', 'fear': 'puppets', 'source': 'crystal ball',
             'robocop': 1, 'comments': 'Please take the puppets out of the
             break room.'},
            {'name': 'Alex Murphy', 'fear': 'ED-209', 'source': 'money',
             'robocop': 5, 'comments': 'Protect the innocent. Serve the public
             trust. Uphold the law.'},
            ]

--snip--
```

The `formData` list contains four dictionaries for four different names. Each dictionary has names of text fields as keys and responses as values. The last bit of setup is to set PyAutoGUI's `PAUSE` variable to wait half a second after each function call. Also, remind the user to click on the browser to make it the active window. Add the following to your program after the `formData` assignment statement:

```python
pyautogui.PAUSE = 0.5
print('Ensure that the browser window is active and the form is loaded!')
```

Step 3: Start Typing Data

A `for` loop will iterate over each of the dictionaries in the `formData` list, passing the values in the dictionary to the PyAutoGUI functions that will virtually type in the text fields.

Add the following code to your program:

```python3
#! python3
# formFiller.py - Automatically fills in the form.
```

```
--snip--

for person in formData:
    # Give the user a chance to kill the script.
    print('>>> 5-SECOND PAUSE TO LET USER PRESS CTRL-C <<<')
  ❶ time.sleep(5)

--snip--
```

As a small safety feature, the script has a five-second pause ❶ that gives the user a chance to hit CTRL-C (or move the mouse cursor to the upper-left corner of the screen to raise the `FailSafeException` exception) to shut the program down in case it's doing something unexpected. After the code that waits to give the page time to load, add the following:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

  ❶ print('Entering %s info...' % (person['name']))
  ❷ pyautogui.write(['\t', '\t'])

    # Fill out the Name field.
  ❸ pyautogui.write(person['name'] + '\t')

    # Fill out the Greatest Fear(s) field.
  ❹ pyautogui.write(person['fear'] + '\t')

--snip--
```

We add an occasional `print()` call to display the program's status in its Terminal window to let the user know what's going on ❶.

Since the form has had time to load, call `pyautogui.write(['\t', '\t'])` to press TAB twice and put the Name field into focus ❷. Then call `write()` again to enter the string in `person['name']` ❸. The `'\t'` character is added to the end of the string passed to `write()` to simulate pressing TAB, which moves the keyboard focus to the next field, Greatest Fear(s). Another call to `write()` will type the string in `person['fear']` into this field and then tab to the next field in the form ❹.

Step 4: Handle Select Lists and Radio Buttons

The drop-down menu for the "wizard powers" question and the radio buttons for the RoboCop field are trickier to handle than the text fields. To click these options with the mouse, you would have to figure out the x- and y-coordinates of each possible option. It's easier to use the keyboard arrow keys to make a selection instead.

Add the following to your program:

```python3
#! python3
# formFiller.py - Automatically fills in the form.

--snip--


    # Fill out the Source of Wizard Powers field.
  ? if person['source'] == 'wand':
     ? pyautogui.write(['down', '\t'] , 0.5)

    elif person['source'] == 'amulet':
        pyautogui.write(['down', 'down', '\t'] , 0.5)

    elif person['source'] == 'crystal ball':
        pyautogui.write(['down', 'down', 'down', '\t'] , 0.5)

    elif person['source'] == 'money':
        pyautogui.write(['down', 'down', 'down', 'down', '\t'] , 0.5)


    # Fill out the RoboCop field.
  ? if person['robocop'] == 1:
     ? pyautogui.write([' ', '\t'] , 0.5)

    elif person['robocop'] == 2:
        pyautogui.write(['right', '\t'] , 0.5)

    elif person['robocop'] == 3:
        pyautogui.write(['right', 'right', '\t'] , 0.5)

    elif person['robocop'] == 4:
        pyautogui.write(['right', 'right', 'right', '\t'] , 0.5)

    elif person['robocop'] == 5:
        pyautogui.write(['right', 'right', 'right', 'right', '\t'] , 0.5)


--snip--
```

Once the drop-down menu has focus (remember that you wrote code to simulate pressing TAB after filling out the Greatest Fear(s) field), pressing the down arrow key will move to the next item in the selection list. Depending on the value in person['source'],

your program should send a number of down arrow keypresses before tabbing to the next field. If the value at the `'source'` key in this user's dictionary is `'wand'` ?, we simulate pressing the down arrow key once (to select *Wand*) and pressing TAB ?. If the value at the `'source'` key is `'amulet'`, we simulate pressing the down arrow key twice and pressing TAB, and so on for the other possible answers. The `0.5` argument in these `write()` calls add a half-second pause in between each key so that our program doesn't move too fast for the form.

The radio buttons for the RoboCop question can be selected with the right arrow keys —or, if you want to select the first choice ?, by just pressing the spacebar ?.

Step 5: Submit the Form and Wait

You can fill out the Additional Comments field with the `write()` function by passing `person['comments']` as an argument. You can type an additional `'\t'` to move the keyboard focus to the next field or the Submit button. Once the Submit button is in focus, calling `pyautogui.press('enter')` will simulate pressing the ENTER key and submit the form. After submitting the form, your program will wait five seconds for the next page to load.

Once the new page has loaded, it will have a *Submit another response* link that will direct the browser to a new, empty form page. You stored the coordinates of this link as a tuple in `submitAnotherLink` in step 2, so pass these coordinates to `pyautogui.click()` to click this link.

With the new form ready to go, the script's outer `for` loop can continue to the next iteration and enter the next person's information into the form.

Complete your program by adding the following code:

```python3
#! python3
# formFiller.py - Automatically fills in the form.


--snip--


    # Fill out the Additional Comments field.
    pyautogui.write(person['comments'] + '\t')


    # "Click" Submit button by pressing Enter.
    time.sleep(0.5) # Wait for the button to activate.
    pyautogui.press('enter')


    # Wait until form page has loaded.
    print('Submitted form.')
```

```
    time.sleep(5)


    # Click the Submit another response link.
    pyautogui.click(submitAnotherLink[0], submitAnotherLink[1])
```

Once the main `for` loop has finished, the program will have plugged in the information for each person. In this example, there are only four people to enter. But if you had *4,000* people, then writing a program to do this would save you a lot of time and typing!

DISPLAYING MESSAGE BOXES

The programs you've been writing so far all tend to use plaintext output (with the `print()` function) and input (with the `input()` function). However, PyAutoGUI programs will use your entire desktop as its playground. The text-based window that your program runs in, whether it's Mu or a Terminal window, will probably be lost as your PyAutoGUI program clicks and interacts with other windows. This can make getting input and output from the user hard if the Mu or Terminal windows get hidden under other windows.

To solve this, PyAutoGUI offers pop-up message boxes to provide notifications to the user and receive input from them. There are four message box functions:

pyautogui.alert(text) Displays `text` and has a single OK button.

pyautogui.confirm(text) Displays `text` and has OK and Cancel buttons, returning either `'OK'` or `'Cancel'` depending on the button clicked.

pyautogui.prompt(text) Displays `text` and has a text field for the user to type in, which it returns as a string.

pyautogui.password(text) Is the same as `prompt()`, but displays asterisks so the user can enter sensitive information such as a password.

These functions also have an optional second parameter that accepts a string value to use as the title in the title bar of the message box. The functions won't return until the user has clicked a button on them, so they can also be used to introduce pauses into your PyAutoGUI programs. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> pyautogui.alert('This is a message.', 'Important')
'OK'
>>> pyautogui.confirm('Do you want to continue?') # Click Cancel
'Cancel'
>>> pyautogui.prompt("What is your cat's name?")
```

```
'Zophie'
>>> pyautogui.password('What is the password?')
'hunter2'
```

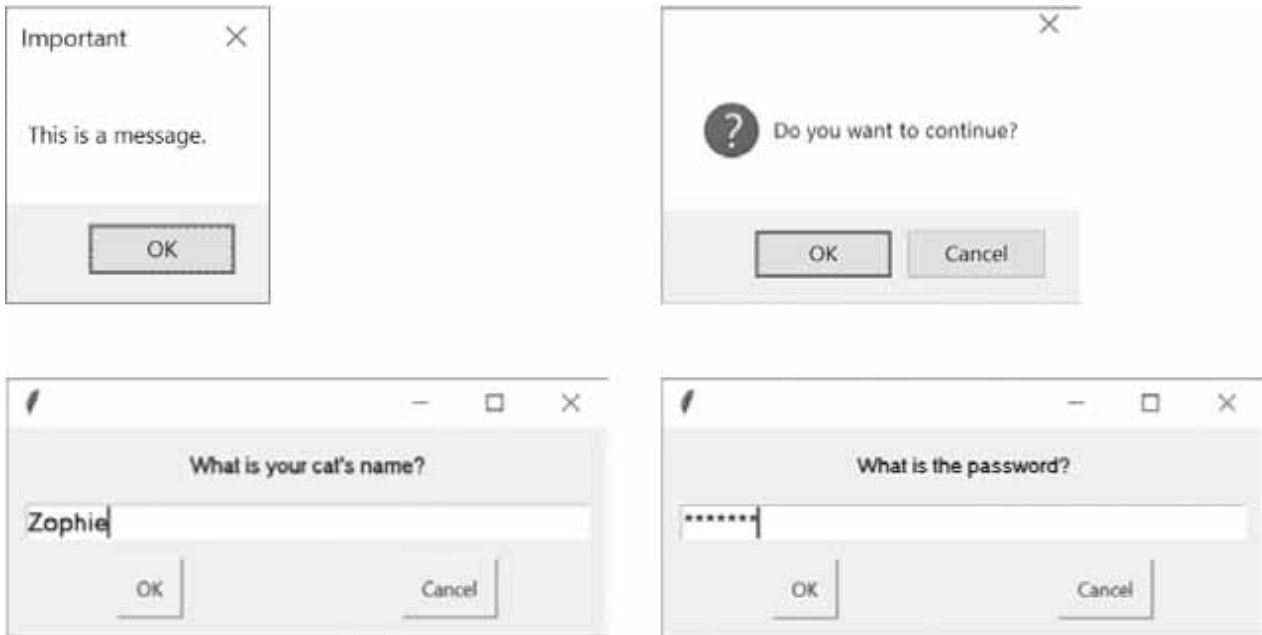The pop-up message boxes that these lines produce look like Figure 20-8.

Figure 20-8: From top left to bottom right, the windows created by alert(), confirm(), prompt(), *and* password()

These functions can be used to provide notifications or ask the user questions while the rest of the program interacts with the computer through the mouse and keyboard. The full online documentation can be found at *https://pymsgbox.readthedocs.io*.

SUMMARY

GUI automation with the pyautogui module allows you to interact with applications on your computer by controlling the mouse and keyboard. While this approach is flexible enough to do anything that a human user can do, the downside is that these programs are fairly blind to what they are clicking or typing. When writing GUI automation programs, try to ensure that they will crash quickly if they're given bad instructions. Crashing is annoying, but it's much better than the program continuing in error.

You can move the mouse cursor around the screen and simulate mouse clicks, keystrokes, and keyboard shortcuts with PyAutoGUI. The pyautogui module can also check the colors on the screen, which can provide your GUI automation program with enough of an idea of the screen contents to know whether it has gotten offtrack. You can even give PyAutoGUI a screenshot and let it figure out the coordinates of the area you want to click.

You can combine all of these PyAutoGUI features to automate any mindlessly repetitive task on your computer. In fact, it can be downright hypnotic to watch the mouse cursor move on its own and to see text appear on the screen automatically. Why not spend the time you saved by sitting back and watching your program do all your work for you? There's a certain satisfaction that comes from seeing how your cleverness has saved you from the boring stuff.

PRACTICE QUESTIONS

1. How can you trigger PyAutoGUI's fail-safe to stop a program?

2. What function returns the current `resolution()`?

3. What function returns the coordinates for the mouse cursor's current position?

4. What is the difference between `pyautogui.moveTo()` and `pyautogui.move()`?

5. What functions can be used to drag the mouse?

6. What function call will type out the characters of `"Hello, world!"`?

7. How can you do keypresses for special keys such as the keyboard's left arrow key?

8. How can you save the current contents of the screen to an image file named *screenshot.png*?

9. What code would set a two-second pause after every PyAutoGUI function call?

10. If you want to automate clicks and keystrokes inside a web browser, should you use PyAutoGUI or Selenium?

11. What makes PyAutoGUI error-prone?

12. How can you find the size of every window on the screen that includes the text `Notepad` in its title?

13. How can you make, say, the Firefox browser active and in front of every other window on the screen?

PRACTICE PROJECTS

For practice, write programs that do the following.

Looking Busy

Many instant messaging programs determine whether you are idle, or away from your computer, by detecting a lack of mouse movement over some period of time—say, 10 minutes. Maybe you're away from your computer but don't want others to see your

instant messenger status go into idle mode. Write a script to nudge your mouse cursor slightly every 10 seconds. The nudge should be small and infrequent enough so that it won't get in the way if you do happen to need to use your computer while the script is running.

Using the Clipboard to Read a Text Field

While you can send keystrokes to an application's text fields with `pyautogui.write()`, you can't use PyAutoGUI alone to read the text already inside a text field. This is where the Pyperclip module can help. You can use PyAutoGUI to obtain the window for a text editor such as Mu or Notepad, bring it to the front of the screen by clicking on it, click inside the text field, and then send the CTRL-A or ⌘-A hotkey to "select all" and CTRL-C or ⌘-C hotkey to "copy to clipboard." Your Python script can then read the clipboard text by running `import pyperclip` and `pyperclip.paste()`.

Write a program that follows this procedure for copying the text from a window's text fields. Use `pyautogui.getWindowsWithTitle('Notepad')` (or whichever text editor you choose) to obtain a Window object. The `top` and `left` attributes of this Window object can tell you where this window is, while the `activate()` method will ensure it is at the front of the screen. You can then click the main text field of the text editor by adding, say, `100` or `200` pixels to the `top` and `left` attribute values with `pyautogui.click()` to put the keyboard focus there. Call `pyautogui.hotkey('ctrl', 'a')` and `pyautogui.hotkey('ctrl', 'c')` to select all the text and copy it to the clipboard. Finally, call `pyperclip.paste()` to retrieve the text from the clipboard and paste it into your Python program. From there, you can use this string however you want, but just pass it to `print()` for now.

Note that the window functions of PyAutoGUI only work on Windows as of PyAutoGUI version 1.0.0, and not on macOS or Linux.

Instant Messenger Bot

Google Talk, Skype, Yahoo Messenger, AIM, and other instant messaging applications often use proprietary protocols that make it difficult for others to write Python modules that can interact with these programs. But even these proprietary protocols can't stop you from writing a GUI automation tool.

The Google Talk application has a search bar that lets you enter a username on your friend list and open a messaging window when you press ENTER. The keyboard focus automatically moves to the new window. Other instant messenger applications have similar ways to open new message windows. Write a program that will automatically send out a notification message to a select group of people on your friend list. Your program may have to deal with exceptional cases, such as friends being offline, the chat

window appearing at different coordinates on the screen, or confirmation boxes that interrupt your messaging. Your program will have to take screenshots to guide its GUI interaction and adopt ways of detecting when its virtual keystrokes aren't being sent.

> **NOTE**
>
> *You may want to set up some fake test accounts so that you don't accidentally spam your real friends while writing this program.*

Game-Playing Bot Tutorial

There is a great tutorial titled "How to Build a Python Bot That Can Play Web Games" that you can find a link to at *https://nostarch.com/automatestuff2/*. This tutorial explains how to create a GUI automation program in Python that plays a Flash game called Sushi Go Round. The game involves clicking the correct ingredient buttons to fill customers' sushi orders. The faster you fill orders without mistakes, the more points you get. This is a perfectly suited task for a GUI automation program—and a way to cheat to a high score! The tutorial covers many of the same topics that this chapter covers but also includes descriptions of PyAutoGUI's basic image recognition features. The source code for this bot is at *https://github.com/asweigart/sushigoroundbot/* and a video of the bot playing the game is at *https://youtu.be/lfk_T6VKhTE*.