ChetBot Automation API Reference

Version 0.4.2

The ChetBot API provides a simple Python interface to interact with your Android or iOS app. It makes use of lazily-evaulated view selectors to allow you to efficient queries to navigate your UI. (See #View)

Device-level operations

`home()`

Simulate a press of the device's "Home" key

`back()`

Simulate a press of the device's "Back" key

(Android only)

`launch_app(name)`

Starts the app with the given name or package name. If no name is specified, launches the app under test.

`airplane_mode(duration)`

Switch to airplane more for `duration` seconds.

`action(name)`

Get a custom action by looking up its name.

`rotate(orientation)`

Rotate the device to the given orientation.

Allowed `orientation` values are:

- `PORTRAIT`
- `LANDSCAPE` (left)
- `LANDSCAPE_RIGHT`
- `UPSIDE_DOWN`

`screenshot()`

Take a screenshot and show it in the test report.

View

A `View` object is selector for views on the screen. Selectors are lazily evaluated and can be chained. `__` is an alias for the `View` constructor.

e.g.

```
__('Confirm').closest_to('Enable notifications') \
    .tap()
```

This example taps the view with the "Enable notifications" label that's nearest to the "Confirm" label. Views are only searched for and tapped when `.tap()` is executed.

`__(label[, id, class])`

Selects a view on the screen. Passing `label` selects any views with the text specified. Passing `id` selects the view with ID specified. Passing `class` You should try to use textual labels where possible (this is the default) to create clear reproducible tests.

```
# Find the "Sign up" button
__('Sign up')

# Find the UISwitch that contains the label "Notifications"
__(class='UISwitch').containing('Notifications')
```

`.topmost()`

Select the view that's the closest to the top.

`.bottommost()`

Select the view that's the closest to the bottom.

`.leftmost()`

Select the view that's the closest to the left.

`.rightmost()`

Select the view that's the closest to the right.

`.centermost()`

Select the view that's the closest to the center.

`.first()`

Selects the `.top()` and then the `.leftmost()` view from those selected.

`.last()`

Selects the `.bottom()` and then the `.rightmost()` view from those selected.

`.location()`

Returns a `[x, y]` pair of the location of the view in the window.

`.closest_to(label[, id, class, distance_function])`

Select the view closest to the first view in the view selector given.

`distance_function` defaults to `EUCLIDIAN`.

Valid `distance_functions` are:

- `EUCLIDIAN`
- `HORIZONTALLY`
- `VERTICALLY`

Or you can specify a custom distance function, e.g.

```
lambda (x1,y1),(x2,y2): abs(x2-x1)
```

Example usage:

```
__('More info').closest_to(__('username'))
```

`.view(label[, id, class])`

Selects descendent views matching the given view selector from the first selected view.

`.find(label[, id, class])`

Same as `.view(...)` but scolls the selected view if scrollable.

`.containing(label[, id, class])`

Filters the selected view leaving any that contain or are a view matching the view selector provided.

`.parent()`

Gets the parent view of the selected view.

`.ancestor(...)`

Gets the first ancestor of the selected view matching the selector given.

`.tap()`

Tap the center of the selected view.

`.long_press([duration])`

Long presses the center of the selected view for the specified number of seconds. 'duration' defaults to 1 second.

`.dragFrom(to[, duration, delay])`

Tap and hold the center of the selected view and drag to the center of the selected view specified by `to`. `duration` defaults to `1.0`.

`.dragTo(from[, duration, delay])`

The reverse of `.dragFrom(...)`.

`.swipe(direction[, distance, distance_units, duration, timing_function])`

Swipe from the center of the selected view in the direction given for over the given distance for the given duration. `distance` defaults to 3cm. `duration` defaults to 0.25s. `timing_function` defaults to `LINEAR`.

Note: To scroll you should prefer `.scroll(...)` over `.swipe(...)`

`distance`s can be specified using the following `distance_unit`s:

- `PX`
- `CM`
- `IN`
- `VW` (proportion of screen width, between 0.0 and 1.0)
- `VH` (proportion of screen height, between 0.0 and 1.0)
- `VMIN` (proportion of small screen dimension, between 0.0 and 1.0)
- `VMAX` (proportion of largest screen dimension, between 0.0 and 1.0)
- `"%"` (percentage of the screen extremities in line with the swipe)

The `direction`s available are:

- `LEFT`
- `RIGHT`

- `UP`
- `DOWN`

`timing_function`s available are - `LINEAR` - `ACCELERATE` - `DECELERATE` - `ACCELERATE_DECELERATE`

`.fling(...)`

Same as `.swipe(...)` but with a default `timing_function` of `ACCELERATE`.

`.scroll(direction[, distance, duration])`

Scroll the view. If the selected view is not scrollable, selects the first scrollable descendent. Throws an error if no scrollable views are found. `distance` defaults to half of the view size in the corresponding dimension. `duration` defaults to 0.5s.

The `direction`s available are:

- `LEFT`
- `RIGHT`
- `UP`
- `DOWN`
- `TO_BEGINNING`
- `TO_END`

`BEGINNING` and `END` intelligently guess the scroll direction and scroll to the left/top or right/bottom appropriately.

`.select()`

Taps if not `.is_selected()`.

`.unselect()`

Taps if `.is_selected()`.

`.exists()`

Returns true if any view have been selected.

`.count()`

Returns the number of selected views.

`.is_selected()`

Returns wheter the first selected view is enabled/selected. For use with checkboxes, radio buttons, etc.

`.text()`

Returns the textual content of the selected view.

`.type()`

Returns the class of the selected view.

`.id()`

Returns the ID of the selected view.

`.size()`

Returns a `(width, height)` tuple of the size of the selected view in pixels.

`.width()`

Returns the width of the selected view in pixels.

`.height()`

Returns the height of the selected view in pixels.

`.flash([color])`

Highlight all selected views with the color given. `color` defaults to `MAGENTA`.

Available `color`s are:

- `RED`
- `GREEN`
- `BLUE`
- `YELLOW`
- `CYAN`
- `MAGENTA`
- `WHITE`
- `BLACK`

wait

`wait(duration)`

Wait for the given number of seconds.

`wait_for(view)`

Wait for the view seletor given to match one or more views. Polls the view hierarchy periodically.

expect

Expect a condition to be `True`. If the condition is `False` flags an error and stops the test. If passed a `__(...)` selector, asserts that `.count() > 0`. If two arguments are passed, expects that the seconds evaluates to the first. e.g.

```
expect(keyboard.is_open())
expect(__('Sign in'))
expect('On', __(id='toggle').text())
```

keyboard

`.is_open()`

Returns True if the keyboard is visible.

`.is_hidden()`

Returns False if the keyboard is visible.

`.type(text)`

Types text on the keyboard. Fails the test if the keyboard is not visible.

`.press_return()`

Press the return key on the keyboard. Note: Different apps handle this in different ways. Equilvalent to `keyboard.type('\n')`

`.dismiss()`

Press the "Hide keyboard" (back button) key to dismiss the keyboard. Fails if the keyboard is not showing.

(Android only)

Lower-lever APIs under consideration

Multi-touch

Fine-grained touch control allowing you to specify the motion of each digit.

Accelerometer

Common motions such as turning the device face-down, picking the device up and shaking.

Camera

Connect a real camera or supply mock images.

System alerts

Detect and respond to system-level popups on iOS.

Localisation

Switch the device to a different locale.

Screenshot comparison

Automatically compare screenshots to previous builds. Notifies you of any changes.

Location

Set mock lat/long/accuracy.

Tethering

Allow simulating tethering on iOS. iOS reduces the amount of screen real-estate available when tethering.