Handling the mouse

The package **`pynput.mouse`** contains classes for controlling and monitoring the mouse.

Controlling the mouse

Use `pynput.mouse.Controller` like this:

```
from pynput.mouse import Button, Controller

mouse = Controller()

# Read pointer position
print('The current pointer position is {0}'.format(
    mouse.position))

# Set pointer position
mouse.position = (10, 20)
print('Now we have moved it to {0}'.format(
    mouse.position))

# Move pointer relative to current position
mouse.move(5, -5)

# Press and release
mouse.press(Button.left)
mouse.release(Button.left)

# Double click; this is different from pressing and releasing
# twice on macOS
mouse.click(Button.left, 2)

# Scroll two steps down
mouse.scroll(0, 2)
```

Monitoring the mouse

Use `pynput.mouse.Listener` like this:

```
from pynput import mouse

def on_move(x, y):
    print('Pointer moved to {0}'.format(
        (x, y)))

def on_click(x, y, button, pressed):
    print('{0} at {1}'.format(
        'Pressed' if pressed else 'Released',
        (x, y)))
    if not pressed:
```

```
        # Stop listener
        return False

def on_scroll(x, y, dx, dy):
    print('Scrolled {0} at {1}'.format(
        'down' if dy < 0 else 'up',
        (x, y)))

# Collect events until released
with mouse.Listener(
        on_move=on_move,
        on_click=on_click,
        on_scroll=on_scroll) as listener:
    listener.join()

# ...or, in a non-blocking fashion:
listener = mouse.Listener(
    on_move=on_move,
    on_click=on_click,
    on_scroll=on_scroll)
listener.start()
```

A mouse listener is a `threading.Thread`, and all callbacks will be invoked from the thread.

Call `pynput.mouse.Listener.stop` from anywhere, raise `StopException` or return `False` from a callback to stop the listener.

When using the non-blocking version above, the current thread will continue executing. This might be necessary when integrating with other GUI frameworks that incorporate a main-loop, but when run from a script, this will cause the program to terminate immediately.

The mouse listener thread

The listener callbacks are invoked directly from an operating thread on some platforms, notably *Windows*.

This means that long running procedures and blocking operations should not be invoked from the callback, as this risks freezing input for all processes.

A possible workaround is to just dispatch incoming messages to a queue, and let a separate thread handle them.

Handling mouse listener errors

If a callback handler raises an exception, the listener will be stopped. Since callbacks run in a dedicated thread, the exceptions will not automatically be reraised.

To be notified about callback errors, call `Thread.join` on the listener instance:

```
from pynput import mouse
```

```
class MyException(Exception): pass

def on_click(x, y, button, pressed):
    if button == mouse.Button.left:
        raise MyException(button)

# Collect events until released
with mouse.Listener(
        on_click=on_click) as listener:
    try:
        listener.join()
    except MyException as e:
        print('{0} was clicked'.format(e.args[0]))
```

Toggling event listening for the mouse listener

Once `pynput.mouse.Listener.stop` has been called, the listener cannot be restarted, since listeners are instances of `threading.Thread`.

If your application requires toggling listening events, you must either add an internal flag to ignore events when not required, or create a new listener when resuming listening.

Synchronous event listening for the mouse listener

To simplify scripting, synchronous event listening is supported through the utility class `pynput.mouse.Events`. This class supports reading single events in a non-blocking fashion, as well as iterating over all events.

To read a single event, use the following code:

```
from pynput import mouse

# The event listener will be running in this block
with mouse.Events() as events:
    # Block at most one second
    event = events.get(1.0)
    if event is None:
        print('You did not interact with the mouse within one second')
    else:
        print('Received event {}'.format(event))
```

To iterate over mouse events, use the following code:

```
from pynput import mouse

# The event listener will be running in this block
with mouse.Events() as events:
    for event in events:
        if event.button == mouse.Button.right:
            break
```

```
            else:
                print('Received event {}'.format(event))
```

Please note that the iterator method does not support non-blocking operation, so it will wait for at least one mouse event.

The events will be instances of the inner classes found in `pynput.mouse.Events`.

Ensuring consistent coordinates between listener and controller on Windows

Recent versions of _Windows_ support running legacy applications scaled when the system scaling has been increased beyond 100%. This allows old applications to scale, albeit with a blurry look, and avoids tiny, unusable user interfaces.

This scaling is unfortunately inconsistently applied to a mouse listener and a controller: the listener will receive physical coordinates, but the controller has to work with scaled coordinates.

This can be worked around by telling Windows that your application is DPI aware. This is a process global setting, so _pynput_ cannot do it automatically. Do enable DPI awareness, run the following code:

```
import ctypes


PROCESS_PER_MONITOR_DPI_AWARE = 2

ctypes.windll.shcore.SetProcessDpiAwareness(PROCESS_PER_MONITOR_DPI_AWARE)
```

Reference

class `pynput.mouse.Controller` [source]

A controller for sending virtual mouse events to the system.

`click`(*button*, *count=1*) [source]

Emits a button click event at the current position.

The default implementation sends a series of press and release events.

Parameters:	• **button** (*Button*) – The button to click. • **count** (*int*) – The number of clicks to send.

`move`(*dx*, *dy*) [source]

Moves the mouse pointer a number of pixels from its current position.

Parameters:	• **x** (*int*) – The horizontal offset. • **dy** (*int*) – The vertical offset.
Raises:	**ValueError** – if the values are invalid, for example out of bounds

`position`

The current position of the mouse pointer.

This is the tuple `(x, y)`, and setting it will move the pointer.

`press`(*button*) [source]

Emits a button press event at the current position.

Parameters:	**button** (*Button*) – The button to press.

`release`(*button*) [source]

Emits a button release event at the current position.

Parameters:	**button** (*Button*) – The button to release.

`scroll`(*dx*, *dy*) [source]

Sends scroll events.

Parameters:	• **dx** (*int*) – The horizontal scroll. The units of scrolling is undefined. • **dy** (*int*) – The vertical scroll. The units of scrolling is undefined.
Raises:	**ValueError** – if the values are invalid, for example out of bounds

class `pynput.mouse.Listener`(*on_move=None*, *on_click=None*, *on_scroll=None*, *suppress=Fa* [source]

A listener for mouse events.

Instances of this class can be used as context managers. This is equivalent to the following code:

```
listener.start()
try:
    listener.wait()
    with_statements()
finally:
    listener.stop()
```

This class inherits from **`threading.Thread`** and supports all its methods. It will set **`daemon`** to `True` when created.

Parameters:	• **on_move** (*callable*) – The callback to call when mouse move events occur. It will be called with the arguments `(x, y)`, which is the new pointer position. If this callback raises **`StopException`** or returns `False`, the listener is stopped. • **on_click** (*callable*) – The callback to call when a mouse button is clicked.

It will be called with the arguments `(x, y, button, pressed)`, where `(x, y)` is the new pointer position, `button` is one of the `Button` values and `pressed` is whether the button was pressed.

If this callback raises `StopException` or returns `False`, the listener is stopped.

- **on_scroll** (*callable*) –
The callback to call when mouse scroll events occur.

  It will be called with the arguments `(x, y, dx, dy)`, where `(x, y)` is the new pointer position, and `(dx, dy)` is the scroll vector.

  If this callback raises `StopException` or returns `False`, the listener is stopped.

- **suppress** (*bool*) – Whether to suppress events. Setting this to `True` will prevent the input events from being passed to the rest of the system.
- **kwargs** –
Any non-standard platform dependent options. These should be prefixed with the platform name thus: `darwin_`, `xorg_` or `win32_`.

  Supported values are:

  `darwin_intercept`

  A callable taking the arguments `(event_type, event)`, where `event_type` is any mouse related event type constant, and `event` is a `CGEventRef`.
  This callable can freely modify the event using functions like`Quartz.CGEventSetIntegerValueField`. If this callable does not return the event, the event is suppressed system wide.

  `win32_event_filter`

  A callable taking the arguments `(msg, data)`, where `msg` is the current message, and `data` associated data as a MSLLHOOKSTRUCT.
  If this callback returns `False`, the event will not be propagated to the listener callback.

  If `self.suppress_event()` is called, the event is suppressed system wide.

[source]

__init__(*on_move=None*, *on_click=None*, *on_scroll=None*, *suppress=False*, ***kwargs*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

`running`
Whether the listener is currently running.

`start`()
Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

`stop`() ¶
Stops listening for events.

When this method returns, no more events will be delivered. Once this method has been called, the listener instance cannot be used any more, since a listener is a **`threading.Thread`**, and once stopped it cannot be restarted.

To resume listening for event, a new listener must be created.

`wait`()
Waits for this listener to become ready.