pynput

Release 1.7.2

Dec 21, 2020

Contents

This library allows you to control and monitor input devices.

It contains subpackages for each type of input device supported:

pynput.mouse Contains classes for controlling and monitoring a mouse or trackpad.

pynput.keyboard Contains classes for controlling and monitoring the keyboard.

All modules mentioned above are automatically imported into the `pynput` package. To use any of them, import them from the main package:

```
from pynput import mouse, keyboard
```

CHAPTER 1

Forcing a specific backend

pynput attempts to use the backend suitable for the current platform, but this automatic choice is possible to override.

If the environment variables `$PYNPUT_BACKEND_KEYBOARD` or `$PYNPUT_BACKEND` are set, their value will be used as backend name for the keyboard classes, and if `$PYNPUT_BACKEND_MOUSE` or `$PYNPUT_BACKEND` are set, their value will be used as backend name for the mouse classes.

Available backends are:

- `darwin`, the default for *macOS*.
- `win32`, the default for *Windows*.
- `uinput`, an optional backend for *Linux* requiring *root* privileges and supporting only keyboards.
- `xorg`, the default for other operating systems.
- `dummy`, a non-functional, but importable, backend. This is useful as mouse backend when using the `uinput` backend.

CHAPTER 2

Table of contents

2.1 Handling the mouse

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

2.1.1 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)
```

(continues on next page)

(continued from previous page)

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

2.1.2 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)
```

2.1.3 Reference

class pynput.mouse.**Controller**

A controller for sending virtual mouse events to the system.

click(*button*, *count=1*)

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*)

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*)
Emits a button press event at the current position.

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

release(*button*)
Emits a button release event at the current position.

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

scroll(*dx*, *dy*)
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=False*, ***kwargs*)
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.

__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.

2.2 Handling the keyboard

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

2.2.1 Controlling the keyboard

Use `pynput.keyboard.Controller` like this:

```
from pynput.keyboard import Key, Controller

keyboard = Controller()

# Press and release space
keyboard.press(Key.space)
keyboard.release(Key.space)

# Type a lower case A; this will work even if no key on the
# physical keyboard is labelled 'A'
keyboard.press('a')
keyboard.release('a')

# Type two upper case As
keyboard.press('A')
keyboard.release('A')
with keyboard.pressed(Key.shift):
    keyboard.press('a')
    keyboard.release('a')

# Type 'Hello World' using the shortcut type method
keyboard.type('Hello World')
```

2.2.2 Monitoring the keyboard

Use `pynput.keyboard.Listener` like this:

```
from pynput import keyboard

def on_press(key):
    try:
        print('alphanumeric key {0} pressed'.format(
            key.char))
    except AttributeError:
        print('special key {0} pressed'.format(
            key))

def on_release(key):
    print('{0} released'.format(
        key))
    if key == keyboard.Key.esc:
        # Stop listener
        return False

# Collect events until released
with keyboard.Listener(
```

(continues on next page)

(continued from previous page)

```
        on_press=on_press,
        on_release=on_release) as listener:
    listener.join()

# ...or, in a non-blocking fashion:
listener = keyboard.Listener(
    on_press=on_press,
    on_release=on_release)
listener.start()
```

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

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

The `key` parameter passed to callbacks is a `pynput.keyboard.Key`, for special keys, a `pynput.keyboard.KeyCode` for normal alphanumeric keys, or just `None` for unknown keys.

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 keyboard 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 keyboard 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 keyboard

class MyException(Exception): pass

def on_press(key):
    if key == keyboard.Key.esc:
        raise MyException(key)

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

Toggling event listening for the keyboard listener

Once `pynput.keyboard.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 keyboard listener

To simplify scripting, synchronous event listening is supported through the utility class `pynput.keyboard.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 keyboard

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

To iterate over keyboard events, use the following code:

```
from pynput import keyboard

# The event listener will be running in this block
with keyboard.Events() as events:
    for event in events:
        if event.key == keyboard.Key.esc:
            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 keyboard event.

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

Global hotkeys

A common use case for keyboard monitors is reacting to global hotkeys. Since a listener does not maintain any state, hotkeys involving multiple keys must store this state somewhere.

pynput provides the class `pynput.keyboard.HotKey` for this purpose. It contains two methods to update the state, designed to be easily interoperable with a keyboard listener: `pynput.keyboard.HotKey.press` and `pynput.keyboard.HotKey.release` which can be directly passed as listener callbacks.

The intended usage is as follows:

```
from pynput import keyboard

def on_activate():
```

(continues on next page)

(continued from previous page)

```
    print('Global hotkey activated!')

def for_canonical(f):
    return lambda k: f(l.canonical(k))

hotkey = keyboard.HotKey(
    keyboard.HotKey.parse('<ctrl>+<alt>+h'),
    on_activate)
with keyboard.Listener(
        on_press=for_canonical(hotkey.press),
        on_release=for_canonical(hotkey.release)) as l:
    l.join()
```

This will create a hotkey, and then use a listener to update its state. Once all the specified keys are pressed simultaneously, `on_activate` will be invoked.

Note that keys are passed through `pynput.keyboard.Listener.canonical` before being passed to the `HotKey` instance. This is to remove any modifier state from the key events, and to normalise modifiers with more than one physical button.

The method `pynput.keyboard.HotKey.parse` is a convenience function to transform shortcut strings to key collections. Please see its documentation for more information.

To register a number of global hotkeys, use the convenience class `pynput.keyboard.GlobalHotKeys`:

```
from pynput import keyboard

def on_activate_h():
    print('<ctrl>+<alt>+h pressed')

def on_activate_i():
    print('<ctrl>+<alt>+i pressed')

with keyboard.GlobalHotKeys({
        '<ctrl>+<alt>+h': on_activate_h,
        '<ctrl>+<alt>+i': on_activate_i}) as h:
    h.join()
```

2.2.3 Reference

class pynput.keyboard.**Controller**

A controller for sending virtual keyboard events to the system.

exception InvalidCharacterException

The exception raised when an invalid character is encountered in the string passed to *Controller.type()*.

Its first argument is the index of the character in the string, and the second the character.

exception InvalidKeyException

The exception raised when an invalid `key` parameter is passed to either *Controller.press()* or *Controller.release()*.

Its first argument is the `key` parameter.

alt_gr_pressed

Whether *altgr* is pressed.

Please note that this reflects only the internal state of this controller. See *modifiers* for more information.

alt_pressed
Whether any *alt* key is pressed.

Please note that this reflects only the internal state of this controller. See *modifiers* for more information.

ctrl_pressed
Whether any *ctrl* key is pressed.

Please note that this reflects only the internal state of this controller. See *modifiers* for more information.

modifiers
The currently pressed modifier keys.

Please note that this reflects only the internal state of this controller, and not the state of the operating system keyboard buffer. This property cannot be used to determine whether a key is physically pressed.

Only the generic modifiers will be set; when pressing either *Key.shift_l*, *Key.shift_r* or *Key.shift*, only *Key.shift* will be present.

Use this property within a context block thus:

```
with controller.modifiers as modifiers:
    with_block()
```

This ensures that the modifiers cannot be modified by another thread.

press(*key*)
Presses a key.

A key may be either a string of length 1, one of the *Key* members or a *KeyCode*.

Strings will be transformed to *KeyCode* using `KeyCode.char()`. Members of *Key* will be translated to their `value()`.

Parameters **key** – The key to press.

Raises

- ***InvalidKeyException*** – if the key is invalid
- **ValueError** – if `key` is a string, but its length is not `1`

pressed(**args*)
Executes a block with some keys pressed.

Parameters **keys** – The keys to keep pressed.

release(*key*)
Releases a key.

A key may be either a string of length 1, one of the *Key* members or a *KeyCode*.

Strings will be transformed to *KeyCode* using `KeyCode.char()`. Members of *Key* will be translated to their `value()`.

Parameters **key** – The key to release. If this is a string, it is passed to `touches()` and the returned releases are used.

Raises

- ***InvalidKeyException*** – if the key is invalid

- **ValueError** – if `key` is a string, but its length is not `1`

shift_pressed
Whether any *shift* key is pressed, or *caps lock* is toggled.

Please note that this reflects only the internal state of this controller. See *modifiers* for more information.

tap(*key*)
Presses and releases a key.

This is equivalent to the following code:

```
controller.press(key)
controller.release(key)
```

Parameters **key** – The key to press.

Raises

- ***InvalidKeyException*** – if the key is invalid
- **ValueError** – if `key` is a string, but its length is not `1`

touch(*key*, *is_press*)
Calls either *press()* or *release()* depending on the value of `is_press`.

Parameters

- **key** – The key to press or release.
- **is_press** (*bool*) – Whether to press the key.

Raises ***InvalidKeyException*** – if the key is invalid

type(*string*)
Types a string.

This method will send all key presses and releases necessary to type all characters in the string.

Parameters **string** (*str*) – The string to type.

Raises ***InvalidCharacterException*** – if an untypable character is encountered

class pynput.keyboard.**Listener**(*on_press=None*, *on_release=None*, *suppress=False*, ***kwargs*)
A listener for keyboard 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_press** (*callable*) – The callback to call when a button is pressed.

  It will be called with the argument `(key)`, where `key` is a *KeyCode*, a *Key* or `None` if the key is unknown.

- **on_release** (*callable*) – The callback to call when a button is released.

  It will be called with the argument `(key)`, where `key` is a *KeyCode*, a *Key* or `None` if the key is unknown.

- **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 `Quartz.kCGEventKeyDown` or `Quartz.kCGEventKeyDown`, 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 KBDLLHOOKSTRUCT.

  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.

__init__(*on_press=None*, *on_release=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.

class pynput.keyboard.**Key**
A class representing various buttons that may not correspond to letters. This includes modifier keys and function keys.

The actual values for these items differ between platforms. Some platforms may have additional buttons, but these are guaranteed to be present everywhere.

alt = 0
A generic Alt key. This is a modifier.

alt_gr = 0
The AltGr key. This is a modifier.

alt_l = 0
The left Alt key. This is a modifier.

alt_r = 0
The right Alt key. This is a modifier.

backspace = 0
The Backspace key.

caps_lock = 0
The CapsLock key.

cmd = 0
A generic command button. On *PC* platforms, this corresponds to the Super key or Windows key, and on *Mac* it corresponds to the Command key. This may be a modifier.

cmd_l = 0
The left command button. On *PC* platforms, this corresponds to the Super key or Windows key, and on *Mac* it corresponds to the Command key. This may be a modifier.

cmd_r = 0
The right command button. On *PC* platforms, this corresponds to the Super key or Windows key, and on *Mac* it corresponds to the Command key. This may be a modifier.

ctrl = 0
A generic Ctrl key. This is a modifier.

ctrl_l = 0
The left Ctrl key. This is a modifier.

ctrl_r = 0
The right Ctrl key. This is a modifier.

delete = 0
The Delete key.

down = 0
A down arrow key.

end = 0
The End key.

enter = 0
The Enter or Return key.

esc = 0
The Esc key.

f1 = 0
The function keys. F1 to F20 are defined.

home = 0
The Home key.

insert = 0
The Insert key. This may be undefined for some platforms.

left = 0
A left arrow key.

media_next = 0
The next track button.

media_play_pause = 0
The play/pause toggle.

media_previous = 0
The previous track button.

media_volume_down = 0
The volume down button.

media_volume_mute = 0
The volume mute button.

media_volume_up = 0
The volume up button.

menu = 0
The Menu key. This may be undefined for some platforms.

num_lock = 0
The NumLock key. This may be undefined for some platforms.

page_down = 0
The PageDown key.

page_up = 0
The PageUp key.

pause = 0
The Pause/Break key. This may be undefined for some platforms.

print_screen = 0
The PrintScreen key. This may be undefined for some platforms.

right = 0
A right arrow key.

scroll_lock = 0
The ScrollLock key. This may be undefined for some platforms.

shift = 0
A generic Shift key. This is a modifier.

shift_l = 0
The left Shift key. This is a modifier.

shift_r = 0
The right Shift key. This is a modifier.

space = 0
The Space key.

tab = 0
The Tab key.

up = 0
An up arrow key.

class pynput.keyboard.**KeyCode**(*vk=None*, *char=None*, *is_dead=False*, ***kwargs*)
A *KeyCode* represents the description of a key code used by the operating system.

classmethod from_char(*char*, ***kwargs*)
Creates a key from a character.

Parameters **char** (*str*) – The character.

Returns a key code

classmethod from_dead(*char*, ***kwargs*)
Creates a dead key.

Parameters **char** – The dead key. This should be the unicode character representing the stand alone character, such as `'~'` for *COMBINING TILDE*.

Returns a key code

classmethod from_vk(*vk*, ***kwargs*)
Creates a key from a virtual key code.

Parameters

- **vk** – The virtual key code.
- **kwargs** – Any other parameters to pass.

Returns a key code

join(*key*)
Applies this dead key to another key and returns the result.

Joining a dead key with space (`' '`) or itself yields the non-dead version of this key, if one exists; for example, `KeyCode.from_dead('~').join(KeyCode.from_char(' '))` equals `KeyCode.from_char('~')` and `KeyCode.from_dead('~').join(KeyCode.from_dead('~'))`.

Parameters **key** (*KeyCode*) – The key to join with this key.

Returns a key code

Raises **ValueError** – if the keys cannot be joined

2.3 Frequently asked question

2.3.1 How do I suppress specific events only?

Passing the `suppress=True` flag to listeners will suppress all events system-wide. If this is not what you want, you will have to employ different solutions for different backends.

If your backend of choice is not listed below, it does not support suppression of specific events.

macOS

For *macOS*, pass the named argument `darwin_intercept` to the listener constructor. This argument should be a callable taking the arguments `(event_type, event)`, where `event_type` is any mouse related event type constant, and `event` is a CGEventRef. The `event` argument can be manipulated by the functions found on the Apple documentation.

If the interceptor function determines that the event should be suppressed, return `None`, otherwise return the `event`, which you may modify.

Here is a keyboard example:

```
def darwin_intercept(event_type, event):
    import Quartz
    length, chars = Quartz.CGEventKeyboardGetUnicodeString(
        event, 100, None, None)
    if length > 0 and chars == 'x':
        # Suppress x
        return None
    elif length > 0 and chars == 'a':
        # Transform a to b
        Quartz.CGEventKeyboardSetUnicodeString(event, 1, 'b')
    else:
        return event
```

Windows

For *Windows*, pass the argument named `win32_event_filter` to the listener constructor. This argument should be a callable taking the arguments `(msg, data)`, where `msg` is the current message, and `data` associated data as a MSLLHOOKSTRUCT or a *KBDLLHOOKSTRUCT*, depending on whether you are creating a mouse or keyboard listener.

If the filter function determines that the event should be suppressed, call `suppress_event` on the listener. If you return `False`, the event will be hidden from other listener callbacks.

Here is a keyboard example:

```
# Values for MSLLHOOKSTRUCT.vkCode can be found here:
# https://docs.microsoft.com/en-us/windows/win32/inputdev/virtual-key-codes
def win32_event_filter(msg, data):
    if data.vkCode == 0x58:
        # Suppress x
        listener.suppress_event()
```

2.3.2 When using a packager I get an `ImportError` on startup

This happens when using a packager, such as *PyInstaller*, to package your application.

The reason for the error is that the packager attempts to build a dependency tree of the modules used by inspecting `import` statements, and *pynput* finds the platform dependent backend modules at runtime using `importlib`.

To solve this problem, please consult the documentation of your tool to find how to explicitly add modules.

Which modules to add depends on your distribution platform. The backend modules are those starting with an underscore (`'_'`) in the `pynput.keyboard` and `pynput.mouse` packages. Additionally, you will need modules with corresponding names from the `pynput._util` package.

2.4 Platform limitations

pynput aims at providing a unified *API* for all supported platforms. In some cases, however, that is not entirely possible.

2.4.1 Linux

On *Linux*, *pynput* uses *X*, so the following must be true:

- An *X server* must be running.
- The environment variable `$DISPLAY` must be set.

The latter requirement means that running *pynput* over *SSH* generally will not work. To work around that, make sure to set `$DISPLAY`:

```
$ DISPLAY=:0 python -c 'import pynput'
```

Please note that the value `DISPLAY=:0` is just an example. To find the actual value, please launch a terminal application from your desktop environment and issue the command `echo $DISPLAY`.

2.4.2 macOS

Recent versions of *macOS* restrict monitoring of the keyboard for security reasons. For that reason, one of the following must be true:

- The process must run as root.
- Your application must be white listed under *Enable access for assistive devices*. Note that this might require that you package your application, since otherwise the entire *Python* installation must be white listed.
- On versions after *Mojave*, you may also need to whitelist your terminal application if running your script from a terminal.

Please note that this does not apply to monitoring of the mouse or trackpad.

2.4.3 Windows

On *Windows*, virtual events sent by *other* processes may not be received. This library takes precautions, however, to dispatch any virtual events generated to all currently running listeners of the current process.

Furthermore, sending key press events will properly propagate to the rest of the system, but the operating system does not consider the buttons to be truly *pressed*. This means that key press events will not be continuously emitted as when holding down a physical key, and certain key sequences, such as *shift* pressed while pressing arrow keys, do not work as expected.

- genindex

Index

N

P

R

S

T

U

W