Handling the keyboard

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

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')
```

Monitoring the keyboard

Use `pynput.keyboard.Listener` like this:

```
from pynput.keyboard import Key, Listener

def on_press(key):
    print('{0} pressed'.format(
        key))

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

# Collect events until released
with Listener(
```

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

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

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

Starting a keyboard listener may be subject to some restrictions on your platform.

On *Mac OSX*, 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 *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.

Reference

class `pynput.keyboard.Controller` [source]

A controller for sending virtual keyboard events to the system.

exception **`InvalidCharacterException`** [source]

The exception raised when and 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 `Controller.`**`InvalidKeyException`** [source]

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

Its first argument is the `key` parameter.

`Controller.`**`alt_gr_pressed`**

Whether *altgr* is pressed.

`Controller.`**`alt_pressed`**

Whether any *alt* key is pressed.

`Controller.`**`ctrl_pressed`**

Whether any *ctrl* key is pressed.

`Controller.`**`modifiers`**

The currently pressed modifier keys.

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.

`Controller.press`(*key*) [source]

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`

`Controller.pressed`(**args*, ***kwds*) [source]

Executes a block with some keys pressed.

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

`Controller.release`(*key*) [source]

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`

`Controller.shift_pressed`

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

`Controller.touch`(*key*, *is_press*) [source]

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.

`Controller.type`(*string*) [source]

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*) [source]

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

`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 mouse events.

When this method returns, no more events will be delivered.

`wait`()

Waits for this listener to become ready.

class `pynput.keyboard.Key` [source]

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 = <*Key.f1*: 0>

A generic Alt key. This is a modifier.

alt_gr = <*Key.f1*: 0>

The AltGr key. This is a modifier.

alt_l = <*Key.f1*: 0>

The left Alt key. This is a modifier.

alt_r = <*Key.f1*: 0>

The right Alt key. This is a modifier.

backspace = <*Key.f1*: 0>

The Backspace key.

caps_lock = <*Key.f1*: 0>

The CapsLock key.

cmd = <*Key.f1*: 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 = <*Key.f1*: 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 = <*Key.f1*: 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 = <*Key.f1*: 0>

A generic Ctrl key. This is a modifier.

ctrl_l = <*Key.f1*: 0>

The left Ctrl key. This is a modifier.

ctrl_r = <*Key.f1*: 0>

The right Ctrl key. This is a modifier.

delete = <*Key.f1*: 0>

The Delete key.

down = <*Key.f1*: 0>

A down arrow key.

end = <*Key.f1: 0*>

The End key.

enter = <*Key.f1: 0*>

The Enter or Return key.

esc = <*Key.f1: 0*>

The Esc key.

f1 = <*Key.f1: 0*>

The function keys. F1 to F20 are defined.

home = <*Key.f1: 0*>

The Home key.

insert = <*Key.f1: 0*>

The Insert key. This may be undefined for some platforms.

left = <*Key.f1: 0*>

A left arrow key.

menu = <*Key.f1: 0*>

The Menu key. This may be undefined for some platforms.

num_lock = <*Key.f1: 0*>

The NumLock key. This may be undefined for some platforms.

page_down = <*Key.f1: 0*>

The PageDown key.

page_up = <*Key.f1: 0*>

The PageUp key.

pause = <*Key.f1: 0*>

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

print_screen = <*Key.f1: 0*>

The PrintScreen key. This may be undefined for some platforms.

right = <*Key.f1: 0*>

A right arrow key.

scroll_lock = <*Key.f1: 0*>

The ScrollLock key. This may be undefined for some platforms.

shift = <*Key.f1: 0*>

A generic Shift key. This is a modifier.

shift_l = <*Key.f1*: 0>

The left Shift key. This is a modifier.

shift_r = <*Key.f1*: 0>

The right Shift key. This is a modifier.

space = <*Key.f1*: 0>

The Space key.

tab = <*Key.f1*: 0>

The Tab key.

up = <*Key.f1*: 0>

An up arrow key.

class pynput.keyboard.**KeyCode**(*vk=0*, *char=None*, *is_dead=False*) [source]

classmethod **from_char**(*char*) [source]

Creates a key from a character.

Parameters:	**char** (*str*) – The character.
Returns:	a key code

classmethod **from_dead**(*char*) [source]

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*) [source]

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*) [source]

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_cl`

Parameters:	**key** (*KeyCode*) – The key to join with this key.
Returns:	a key code
Raises ValueError:	
	if the keys cannot be joined