# Dpowers

dp0s

Apr 25, 2024

# PREPERATION

Source code: https://github.com/dp0s/Dpowers

CHAPTER

ONE

# INTRODUCTION

The Dpowers are a collection of python tools for common automization tasks, such as:

- Send / receive events from keyboard, mouse and other devices.
- Wait for a certain key combination or sequence to trigger your own code.
- Observe and interact with windows on your screen.
- Display notifications, dialog boxes and tray icons.
- Access the clipboard content.
- Edit images.

The Dpowers package bundles existing open-source projects into a unified python interface. It provides a high level of flexibility due to the following characteristics:

- Adaptable. Each job can be performed by several backends of your choice. Switch between backends dynamically in one line of code.
- Modular. Each sub-package (a.k.a. sub-power) can be used independently.
- Easy to extend. You can add your own power and/or your own backend without touching existing files.
- Cross-platform by nature. (More backends need to be added and tested though to be fully cross-platform.)

Benefits:

- Save time by learning one command syntax to access several backends.
- Combine the advantages of two or more backends into a single tool.
- Future safe. If one backend becomes out-dated, replace it by a more recent one. No need to change your code.
- Short and intuitive commands are prefered to type less.
- Some higher level classes are included to enhance the backend's functionality. (Such as *Dpowers.Win* and *Dpowers.KeyWaiter*)

CHAPTER

TWO

# REQUIREMENTS

- python 3.6 or later
- Currently only tested on apt based Linux systems (Debian, Ubuntu, Linux Mint).

## 2.1 Installation and Dependencies

Install from PyPI:

```
$ pip install -U Dpowers
```

(This will automatically install the package 'Dhelpers' as a necessary dependency. All of Dhelpers' functions and classes can be used independently for your own projects, i.e. without the Dpowers package installed, via `pip install -U Dhelpers`.)

After first installation, you should try:

```
>>> import Dpowers
```

This should never raise an exception, otherwise please file a bug report.

---

**Note:** The pip install command does NOT install any of the backend dependencies. You need to manually install the dependencies for the backends you want to use.

---

The following prints a list of all dependencies for all backends on your system (given example is for Debian/Ubuntu/Linux Mint):

```
>>> import Dpowers
>>> print(Dpowers.Adaptor.install_instructions())
sudo apt install python3-tk xclip xdotool yad zenity xsel wmctrl
pip install -U pynput pillow eyed3 wand pystray
```

Execute the output lines in your shell and you should be able to use all backends.

## 2.2 Importing and Adapting

### 2.2.1 Step 1: Import

This is done as usual. Examples:

```
import Dpowers
from Dpowers import keyb, ntfy
```

By default the imported objects are unadapted, i.e. there's no backend chosen yet. If you try using them, you'll get an exception:

```
>>> from Dpowers import keyb
>>> keyb.tap("a")
AdaptionError: No backend chosen for following adaptor:
<Dpowers.events.sending.keybpower.KeyboardAdaptor object at 0x7fedf46e00b8 with creation_
↪name 'keyb', primary instance of group 'default', backend: None>
```

### 2.2.2 Step 2: Adapt

For each object, choose a backend by calling its *adapt()* method. If you call it without any arguments, the default backend for your platform will be chosen depending on your system:

```
>>> from Dpowers import keyb
>>> keyb.adapt()  # pynput is the default backend in this example
<module 'Dpowers.events.sending.keybpower.adapt_pynput'>
>>> keyb.adapt("pynput") # another way to select pynput
<module 'Dpowers.events.sending.keybpower.adapt_pynput'>
>>> keyb.adapt("evdev")  # manually chose another backend
<module 'Dpowers.events.sending.keybpower.adapt_evdev'>
>>> keyb.tap("a") # check if it works
>>> a
```

**Note:** Calling the adapt method will import the corresponding backend module (if it hasn't been imported before). It raises an exception if the backend is not supported on your system or the backend's dependencies could not be found.

### 2.2.3 Alternative: autoadapt

You can perform the two steps (import and adapt) in only one line:

```
import Dpowers.autoadapt
# which is equivalent to
import Dpowers
Dpowers.activate_autoadapt()
```

This will try to adapt ALL adaptable objects to their default backend if possible, and prints a warning for each exception encountered. The list of default backends is defined in Dpowers .default_backends.py

Alternatively, the wildcard import also activates autoadapt:

```
>>> from Dpowers import *
>>> keyb.tap("a") # check if it works
>>> a
```

A list of all names imported this way:

```
>>> Dpowers.__all__
['autoadapt', 'keyb', 'mouse', 'ntfy', 'dlg', 'hook', 'Icon', 'Win', 'sendwait',
↪'nfsendwait', 'clip', 'Dfuncs', 'events', 'KeyWaiter', 'TriggerManager', 'Image',
↪'mp3tag', 'sleep', 'sound', 'hotkeys', 'launch', 'Layout', 'Dpowers', 'Dhelpers']
```

## 2.3 Overview of adaptable objects

An adaptable object can be coupled to a specific backend by calling its *adapt()* method. There are two kinds of adaptable objects within Dpowers - Adaptors and AdaptiveClasses.

### 2.3.1 Adaptors

- *Dpowers.keyb*: Send key events.
- *Dpowers.mouse*: Send mouse events.
- *Dpowers.hook*: Receive events from keyboard, mouse and other devices.
- *Dpowers.clip*: Access and modify the clipboard content.
- *Dpowers.ntfy*: Post notifcations on the desktop.
- *Dpowers.dlg*: Show dialog boxes and wait for user confirmation.

An Adaptor is an instance of the common baseclass *Dpowers.Adaptor*. Each Adaptor provides a collection of methods which automatically call the corresponding backend's functions.

### 2.3.2 AdaptiveClasses

- *Dpowers.Win*: Find and manipulate windows on your screen.
- *Dpowers.Icon*: Create and display tray icons, including context menu.
- Dpowers.Image: Edit and assemble image files.
- *Dpowers.KeyWaiter*: Collect key events until a condition is fullfilled.
- *Dpowers.TriggerManager*: Define event patterns (such as key or button sequences) to trigger your own functions.

An AdaptiveClass is a subclass of the common baseclass *Dpowers.AdaptiveClass*. All instances created by this class will share the same backend.

## 2.4 Basic Examples

### 2.4.1 Display a tray icon with customized menu

Reference: *Dpowers.ntfy Dpowers.Icon*

```
from Dpowers import autoadapt, Icon, ntfy
myicon = Icon()

@myicon.additem
def my_custom_menu_item():
    ntfy("You clicked the custom menu item.")

myicon.start()
```

### 2.4.2 Define a key sequence to trigger a function

Reference: *Dpowers.TriggerManager*

```
from Dpowers import TriggerManager, sleep
TriggerManager.adapt()
MyTriggers = TriggerManager().hook_keys()

@MyTriggers.sequence("ctrl d d_rls")
def myfunction():
    print("Control + d was pressed")

MyTriggers.start()
sleep(30)
MyTriggers.stop()
```

### 2.4.3 Define a combined key / button sequence as trigger

Reference: *Dpowers.TriggerManager*

```
from Dpowers import TriggerManager
TriggerManager.adapt(keys="evdev", buttons="pynput")

CombinedTriggers = TriggerManager(timeout = None)
Keys = CombinedTriggers.hook_keys()
Buttons = CombinedTriggers.hook_buttons()

@Keys.sequence("ctrl d")
def myfunction():
    print("Control + d was pressed")

@Buttons.sequence("mleft")
def myfunction2():
    print("Left mouse button was pressed.")
```

(continues on next page)

(continued from previous page)

```
@CombinedTriggers.sequence("Ctrl mleft")
def myfunction3():
    print("Ctrl + left mouse button was pressed")

CombinedTriggers.start()
# this will run in background until CombinedTriggers.stop()
```

## 2.5 Advanced Examples

### 2.5.1 Click on a window to paste its properties to the clipboard

Reference: *Dpowers.ntfy* *Dpowers.dlg* *Dpowers.clip* *Dpowers.Win*

```
from Dpowers import autoadapt, ntfy, Win, dlg, clip


def display_win_info():
    ntfy("Click on a window", 3)

    x = Win(loc="SELECT").all_info()
    winprops = x[:3] + ((x[1], x[2]),) + x[3:]

    show = [str(winprops[0]) + " [ID]", str(winprops[1]) + " [TITLE]",
        str(winprops[2]) + " [CLASS]", str(winprops[3]),
        str(winprops[4]) + " [PID]",
        str(winprops[5]) + " [GEOMETRY] (x,y,width,height)"]

    ret = dlg.choose(show, default=3, title="Window information",
            text="Save to clipboard:", width=700)

    if ret is not None:
        for i in range(len(show)):
            if ret == show[i]:
                clip.fill(winprops[i], notify=True)
                break

display_win_info()
```

This function is pre-defined in the module Dpowers.Dfuncs.py:

```
from Dpowers import autoadapt, Dfuncs
Dfuncs.display_win_info()
```

### 2.5.2 Launch the browser and simultaneously redirect key presses

Reference: `Dpowers.launch` *Dpowers.Win* *Dpowers.Win.wait_num_change* *Dpowers.KeyWaiter*

```
from Dpowers import autoadapt, launch, Win, KeyWaiter, ntfy

def firefox_launch():

    with KeyWaiter(100, 15, endevents="Return", capture=True) as address:
        FirefoxWindows = Win("^Mozilla Firefox$")  # the ^ and $
        # mark that we want an exact title match  (regular expression)
        launch("firefox", "-P", "default", check=True, check_err=False)
        newWin = FirefoxWindows.wait_num_change(+1, timeout=10)

    if not newWin: return
    if newWin.num != 1: raise ValueError
    newWin.activate()

    code = address.exitcode
    if code not in  ("endevent", "__exit__"):
        raise ValueError(f"Wrong exitcode: {code}")
    address.reinject(delay=1)
```

## 2.6 Adaptors

**class** Dpowers.**Adaptor**(*main_info=None*, *, *group='default'*, *_primary_name=False*, ***method_infos*)

Abstract baseclass for all of Dpower's Adaption classes.

**__init__**(*main_info=None*, *, *group='default'*, *_primary_name=False*, ***method_infos*)

Usually, you do not need to create Adaptor instances yourself as a default instance is already available for each subclass (see below).

**Parameters**

- **main_info** (*str*) – Name of the backend to be used, passed on to *adapt()*.
- **group** (*str*) – Name of the instance group to which the new instance will belong.
- **_primary_name** – Used only for internal documentation purposes. It allows specifiying the primary/default instance for each instance group.
- **method_infos** – Passed on to *adapt()*.

If the *main_info* and/or *method_infos* parameter is given, the new instance is immediately adapted using its *adapt()* method.

If neither *main_info* nor *method_infos* is specified (default), and `Adpator.autoadapt_active` is `True`, choose the default backend according to the instance group and Dpowers.default_backends.py.

Otherwise an unadapted instance is created.

**adapt**(*main_info=None*, *, *raise_error=True*, *require_backend=True*, ***method_infos*)

Choose the backend for this Adaptor instance. If neither *main_info* nor *method_infos* is specified, the default backend for this instance group is selected, as defined in the module Dpowers.default_backends.py.

**Parameters**

- **main_info** (*str*) – Name of the backend to be used.
- **raise_error** (*bool*) – If set to `False`, exceptions caused during backend import are suppressed. A warning is printed instead.
- **require_backend** (*bool*) – If set to `True` (default), a `ValueError` will be raised if no default backend could be found.
- **method_infos** – Explanation to be added .

**Returns**

- The module object of the backend's adapt_*.py file.
- `False` if an exception occurs but is suppressed (see parameters *raise_error* and *require_backend*).

**Raises**

- `Dpowers.AdaptionError` if there is an exception during importing the backend.
- `ValueError` if used without arguments and no default backend could be found (you can disable this via *require_backend=False*).

### 2.6.1 keyb (KeyboardAdaptor)

`Dpowers.`**keyb**

Default instance of KeyboardAdaptor class.

How to import:

```
from Dpowers import keyb

# choose the default backend for your system:
keyb.adapt()

# alternatively, choose one of the following backends:
keyb.adapt('evdev')
keyb.adapt('pynput')
keyb.adapt('xdotool_bash')
```

How to install dependencies for available backends:

```
>>> print(keyb.install_instructions())
pip install -U pynput
sudo apt install xdotool
```

**class** `Dpowers.`**KeyboardAdaptor**(*main_info=None*, *, *group='default'*, *_primary_name=False*, ***method_infos*)

Send key events to the system via the chosen backend.

See Dpowers.events.keybutton_names.py for a list of allowed key names and key groups. Most of the defined keys can be called by more than one name equivalently. Simply pass one of the key's names as a string to the methods defined below.

For some backends (such as *evdev*) it might be neccessary to manually specify the layout of the keyboard currently used via the *set_layout()* method.

Subclass of *Dpowers.Adaptor*.

**adapt**(*main_info=None*, *, *raise_error=True*, *require_backend=True*, ***method_infos*)

Choose the backend for this instance. See *Adaptor.adapt()*.

**default_delay = 0**

*Class attribute.* Default time (in milliseconds) to wait between sending several keys in a row. You can set a custom value to this attribute for the whole class or for a single instance.

**default_duration = 1**

*Class attribute.* Default time (in milliseconds) to wait between sending press and release event of the same key. You can set a custom value to this attribute for the whole class or for a single instance.

**send**(*string*, *auto_rls=True*, *delay=None*)

Convenience method to send several kind of key patterns in one command. It integrates the functionality of *tap()*, *comb()* and *text()*.

**Parameters**

- **string** (*str*) – Content to be sent, format see below.
- **auto_rls** (*bool*) – If `True`, send press and release events for all keys specified inside < >. If `False`, normal key names are interpreted as press events and you must attach `_rls` to a key name to send the corresponding release event.
- **delay** (*int*) – Time (in milliseconds) to wait between tapping keys. Defaults to *default_delay*.

The **string** parameter will be parsed from left to right according to the following rules:

- There are two special characters: < and >. All content between them will be interpreted as key names (as if called by the *tap()* method):

```
keyb.send("<return>") # is equivalent to
keyb.tap("return")
```

- Everything outside of a < > block will be sent literally using the *text()* method:

```
keyb.send("normal text input..") # is equivalent to
keyb.text("normal text input..")
```

- Use <key1+key2> to send key combinations:

```
keyb.send("<ctrl+s>") # is equivalent to
keyb.comb("ctrl","s") # is equivalent to
keyb.send("<ctrl s s_rls ctrl_rls>", auto_rls=False)
```

- Chain keys and texts as in the following examples:

```
keyb.send("<home>hello world<return end>") # equivalent to
keyb.tap("home","h","e","l","l","o"," ","w","o","r","l","d",
"return", "end")

keyb.send("<key1 key2 key3>") # equivalent to
keyb.send("<key1><key2><key3>")

keyb.send("<shift+home backspace>this line was deleted")
# is equivalent to
keyb.comb("shift","home")
```

(continues on next page)

(continued from previous page)

```
keyb.tap("backspace")
keyb.text("this line was deleted")
```

- If not paired, < and > are interpreted literally.
- Use <><something> to send literal <*something*> as text output.

**tap**(*key*, **keys*, *delay=None*, *duration=None*, *repeat=1*)

Simulate a key tap (i.e. send a press event followed by a release event of the same key), or multiple key taps in a row.

**Parameters**

- **key** (*str*) – Name of the (first) key to be sent.
- **keys** (*str*) – Further names of keys to be sent in sequence.
- **delay** (*int*) – Time (in milliseconds) to wait after tapping a key. Defaults to *default_delay*.
- **duration** (*int*) – Time (in milliseconds) to wait between each press and release event. Defaults to *default_duration*.
- **repeat** – Number of times this sequence should be repeated.

Examples:

```
keyb.tap("a")
keyb.tap(1)
keyb.tap("esc")
keyb.tap("f9")
```

**press**(*key*, **keys*, *delay=None*, *translate_names=True*)

Send key press event(s) to the system. You need to manually send the corresponding release event(s) afterwards.

**Parameters**

See *tap()*.

**rls**(*key*, **keys*, *delay=None*)

Send key release event(s) to the system.

**Parameters**

See *tap()*.

**comb**(*key1*, *key2*, **keys*, *delay=None*, *duration=None*)

Simulates a key combination, such as *Control+S* or *Shift+Alt+F5*. Sends all the press events first and then the release events in reverse order.

**Parameters**

- **key1** (*str*) – The is the name of the first key to be pressed (and hence the last to be released.)
- **key2** (*str*) – The name of the second key in the combination.
- **keys** (*str*) – Further key names.
- **delay** (*int*) – Time (in milliseconds) to wait between pressing/releasing the sequence of keys. Defaults to *default_delay*.
- **duration** (*int*) – Time (in milliseconds) to wait between sending the press and release event of the final key. Defaults to *default_duration*.

Examples:

```
keyb.comb("ctrl","s")
keyb.comb("shift","alt","f5")
```

**custom_text_method = True**

Default value for parameter **custom** of method *text()*.

**text**(*text*, *delay=None*, *custom=None*, ***kwargs*)

Send a plain text string to the system as if the user would type it.

**Parameters**

- **text** (*str*) – Any text you want to send. Depending on the backend, not all symbols might be supported.
- **delay** (*int*) – Time (in milliseconds) to wait between sending single characters. Defaults to *default_delay*.
- **custom** (*bool*) – If set to True, prefer the backend's special text method if implemented. Otherwise use *tap()* for sending single characters via the backend's press/rls methods.
- **kwargs** – Custom keyword arguments to pass to the backend's text method.

**property layout**

Return the currently used keyboard layout for this instance. Use *set_layout()* to change it manually.

**set_layout**(*name*, *make_default=False*, *backend_layout=None*, *use_shifted=None*)

Manually set the effective keyboard layout and translate the key codes from the backend accordingly. This is e.g. necessary for the *evdev* backend to make sure that the correct keys are sent to the system.

**Parameters**

- **name** (*str*) – Name of the keyboard layout you use. This is usually a two character abbreviation, such as *'de'* or *'us'*. A list of available layouts can be found in the folder Dhelpers.KeyboardLayouts.layouts_imported_from_xkb.
- **make_default** (*bool*) – If set to True, all other instances of *KeyboardAdaptor* with the same backend as this instance will use this layout by default. Raises an exception if the backend for this instance has not been chosen yet.
- **backend_layout** (*str*) – Manually set the assumed backend layout. Normally this is set by the backend, so don't use this parameter unless you know what you are doing.
- **use_shifted** (*bool*) – Whether to use manualy key shifting when sending keys. Normally this is set by the backend, so don't use this parameter unless you know what you are doing.

Example for a German keyboard:

```
>>> from Dpowers import keyb
>>> keyb.adapt("evdev")
<module 'Dpowers.events.sending.keybpower.adapt_evdev'>
>>> keyb.layout # returns default layout for evdev backend
us
>>> keyb.tap("[") # sents the wrong key on a German keyboard:
>>> ü
>>> # change to German layout & set as default for evdev backend:
>>> keyb.set_layout("de", make_default=True)
>>> keyb.tap("[") # now the correct key is sent:
>>> [
```

**property key**

This object allows accessing Dpowers' internal key objects and key groups as defined in the module Dpowers.events.keybutton_names.py. Find a key object by calling it's name:

```
>>> keyb.key("a")
<Dpowers.events.keybutton_classes.NamedKey object at 0x7f8bd3fea450 with␣
↪standard name 'a'>
>>> keyb.key("ctrl")
<Dpowers.events.keybutton_classes.NamedKey object at 0x7f8bd3ff2ba0 with␣
↪standard name 'CtrlL'>
>>> keyb.key(1)
<Dpowers.events.keybutton_classes.NamedKey object at 0x7f8bd3ff0080 with␣
↪standard name '1'>
>>> # check if two names belong to the same key object:
>>> keyb.key("ctrl") is keyb.key("leftcontrol")
True
```

You can find key groups via the **.group** attribute and check if a key belongs to it:

```
>>> keyb.key.group.arrow_keys
[<Dpowers.events.keybutton_classes.NamedKey object at 0x7f8bd3ff3650 with␣
↪standard name 'up'>, <Dpowers.events.keybutton_classes.NamedKey object at␣
↪0x7f8bd3ff3620 with standard name 'down'>, <Dpowers.events.keybutton_classes.
↪NamedKey object at 0x7f8bd3ff3740 with standard name 'left'>, <Dpowers.events.
↪keybutton_classes.NamedKey object at 0x7f8bd3ff37a0 with standard name 'right
↪'>]
>>> "up" in keyb.key.group.arrow_keys
True
>>> 0 in keyb.key.group.arrow_keys
False
>>> 0 in keyb.key.group.digits
True
```

**class** Dpowers.KeyboardAdaptor.**AdaptiveClass**

A baseclass to create your own AdaptiveClasses. See *Dpowers.AdaptiveClass*.

### 2.6.2 mouse (MouseAdaptor)

Dpowers.**mouse**

Default instance of MouseAdaptor class.

How to import:

```
from Dpowers import mouse

# choose the default backend for your system:
mouse.adapt()

# alternatively, choose one of the following backends:
mouse.adapt('evdev')
mouse.adapt('pynput')
```

How to install dependencies for available backends:

```
>>> print(mouse.install_instructions())
pip install -U pynput
```

**class** Dpowers.**MouseAdaptor**(*main_info=None*, ***, *group='default'*, *_primary_name=False*, ***method_infos*)

Subclass of *Dpowers.Adaptor*.

**adapt**(*main_info=None*, ***, *raise_error=True*, *require_backend=True*, ***method_infos*)

Choose the backend for this instance. See *Adaptor.adapt()*.

**default_delay = 0**

*Class attribute.* Default time (in milliseconds) to wait between sending several buttons in a row. You can set a custom value to this attribute for the whole class or for a single instance.

**default_duration = 1**

*Class attribute.* Default time (in milliseconds) to wait between sending press and release event of the same button. You can set a custom value to this attribute for the whole class or for a single instance.

**class** Dpowers.MouseAdaptor.**AdaptiveClass**

A baseclass to create your own AdaptiveClasses. See *Dpowers.AdaptiveClass*.

### 2.6.3 clip (ClipboardAdaptor)

**Examples**

- *Click on a window to paste its properties to the clipboard*

Dpowers.**clip**

Default instance of ClipboardAdaptor class.

How to import:

```
from Dpowers import clip

# choose the default backend for your system:
clip.adapt()

# alternatively, choose one of the following backends:
clip.adapt('xclip_bash')
clip.adapt('xsel_bash')
```

How to install dependencies for available backends:

```
>>> print(clip.install_instructions())
sudo apt install xsel xclip
```

**class** Dpowers.**ClipboardAdaptor**(*main_info=None*, ***, *group='default'*, *_primary_name=False*, ***method_infos*)

Subclass of *Dpowers.Adaptor*.

**adapt**(*main_info=None*, ***, *raise_error=True*, *require_backend=True*, ***method_infos*)

Choose the backend for this instance. See *Adaptor.adapt()*.

**get**(*selection=0*) → str

Retrieves the content

**Parameters**
**selection** – The selection to be retrieved. Defaults to 0, i.e. the standard clipboard.

**Returns str**
retrieved content

**class** Dpowers.ClipboardAdaptor.**AdaptiveClass**

A baseclass to create your own AdaptiveClasses. See *Dpowers.AdaptiveClass*.

### 2.6.4 ntfy (NotificationAdaptor)

**Examples**

- *Display a tray icon with customized menu*
- *Click on a window to paste its properties to the clipboard*

Dpowers.**ntfy**

Default instance of NotificationAdaptor class.

How to import:

```
from Dpowers import ntfy

# choose the default backend for your system:
ntfy.adapt()

# alternatively, choose one of the following backends:
ntfy.adapt('notify_send_bash')
```

**class** Dpowers.**NotificationAdaptor**(*main_info=None*, ***, *group='default'*, *_primary_name=False*, ***method_infos*)

Subclass of *Dpowers.Adaptor*.

**adapt**(*main_info=None*, ***, *raise_error=True*, *require_backend=True*, ***method_infos*)

Choose the backend for this instance. See *Adaptor.adapt()*.

**class** Dpowers.NotificationAdaptor.**AdaptiveClass**

A baseclass to create your own AdaptiveClasses. See *Dpowers.AdaptiveClass*.

### 2.6.5 dlg (DialogAdaptor)

**Examples**

- *Click on a window to paste its properties to the clipboard*

Dpowers.**dlg**

Default instance of DialogAdaptor class.

How to import:

```
from Dpowers import dlg

# choose the default backend for your system:
dlg.adapt()

# alternatively, choose one of the following backends:
dlg.adapt('tkinter')
dlg.adapt('zenity_bash')
```

How to install dependencies for available backends:

```
>>> print(dlg.install_instructions())
sudo apt install python3-tk zenity
```

**class** Dpowers.**DialogAdaptor**(*main_info=None*, *, *group='default'*, *_primary_name=False*, ***method_infos*)

Subclass of *Dpowers.Adaptor*.

**adapt**(*main_info=None*, *, *raise_error=True*, *require_backend=True*, ***method_infos*)

Choose the backend for this instance. See *Adaptor.adapt()*.

**date**(*selected=None*, ***kwargs*)

selected_date must be of form (dd,mm,yyyy). returns the same format.

**class** Dpowers.DialogAdaptor.**AdaptiveClass**

A baseclass to create your own AdaptiveClasses. See *Dpowers.AdaptiveClass*.

### 2.6.6 hook (HookAdaptor)

Dpowers.**hook**

Default instance of HookAdaptor class.

How to import:

```
from Dpowers import hook

# choose the default backend for your system:
hook.adapt()

# alternatively, choose one of the following backends:
hook.adapt('evdev')
hook.adapt('pynput')
```

How to install dependencies for available backends:

```
>>> print(hook.install_instructions())
pip install -U pynput
```

**class** Dpowers.**HookAdaptor**(*main_info=None*, *, *group='default'*, *_primary_name=False*, ***method_infos*)

Subclass of *Dpowers.Adaptor*.

**adapt**(*main_info=None*, *, *raise_error=True*, *require_backend=True*, ***method_infos*)

Choose the backend for this instance. See *Adaptor.adapt()*.

**class** Dpowers.HookAdaptor.**AdaptiveClass**

A baseclass to create your own AdaptiveClasses. See *Dpowers.AdaptiveClass*.

## 2.7 Adaptive Classes

**class** Dpowers.**AdaptiveClass**

**adaptor**

*Class attribute.* This is an instance of a subclass of *Dpowers.Adaptor*. It determines the backends available for this AdaptiveClass and is shared among all created instances. This Adaptor instance is used internally to access all backend specific functions. Usually you shouldn't use this attribute, but prefer the AdaptiveClass' methods.

**classmethod adapt**(*main_info=None*, *, *raise_error=True*, *require_backend=True*, ***method_infos*)

Changes the backend for all instances of this AdaptiveClass, including already created instances (unless *adapt_instance()* was used).

For parameters see *Dpowers.Adaptor.adapt()*.

**adapt_instance**(*main_info=None*, *, *raise_error=True*, *require_backend=True*, ***method_infos*)

Changes the backend for this instance only.

For parameters see *Dpowers.Adaptor.adapt()*.

### 2.7.1 Win / Win.Search

**Examples**

- *Click on a window to paste its properties to the clipboard*
- *Launch the browser and simultaneously redirect key presses*

**class** Dpowers.**Win**(*title_or_ID=None*, *wcls=None*, *, *pid=None*, *loc=None*, *at_least_one=False*, *limit=None*, *visible=None*)

Subclass of *Dpowers.AdaptiveClass*.

Associated Adaptor class: Dpowers.WindowAdaptor

How to import:

```
from Dpowers import Win

# choose the default backend for your system:
Win.adapt()

# alternatively, choose one of the following backends:
Win.adapt('xtools_bash')
```

How to install dependencies for available backends:

```
>>> print(Win.install_instructions())
sudo apt install wmctrl xdotool
```

An object of this class represents one or more graphical windows of your operating system.

When a new instance of this class is created, a search is performed to find all existing windows matching the given parameters. Each window is identified by a unique system-wide identification number ( ID). The matching IDs are saved in ascending order as a static list. Use the *IDs()* method to see them. All methods below will operate on this list of initially found windows.

Two *Win()* objects are considered identical if their list of *IDs()* are identical.

**classmethod adapt**(*main_info=None*, *, *raise_error=True*, *require_backend=True*, ***method_infos*)

Choose the backend for this class. See *AdaptiveClass.adapt()*. For parameters see *Dpowers.Adaptor.adapt()*.

**__init__**(*title_or_ID=None*, *wcls=None*, *, *pid=None*, *loc=None*, *at_least_one=False*, *limit=None*, *visible=None*)

**Parameters**

- **title_or_ID** (*str, int*) –
  - A string is interpreted as the title to search for. The string can be contained anywhere in the window's title to be considered a match.
  - An integer is interpreted as a window ID. Specifying this directly will skip the search and all other parameters are hence ignored.
- **wcls** (*str*) – Window class to search for. This must be an exact match.
- **pid** (*int*) – Process ID of the window to search for.
- **loc** (*tuple*) – Specify screen coordinates (*x*,*y*) for this parameter to retrieve the top-most window at that location. Specifying this will ignore all other parameters.
- **at_least_one** (*bool*) – If set to `True`, check that at least one matching window has been found.
- **limit** (*int*) – Maximum number of windows to include into the window object. If more matching windows exist, only the ones with the lowest IDs will be included.
- **visible** (*bool*) – If set to True, only visible windows will be considered.

**Raises**

**WindowNotFoundError** – If *at_least_one* is `True` and no matching window was found.

In order to find the currently active (i.e. topmost) window, simply call this class without parameters:

```
active_window = Win()
```

Several different parameters can be passed at once to narrow down the search. A window must fullfill all specified parameters to be a match:

```
dpowers_doc_win = Win('Dpowers documentation', 'Navigator.firefox')
# searches for windows with
# 'Dpowers documentation' contained in title
# AND
# window class is equal to 'Navigator.firefox' (Firefox browser)
```

In place of passing a single value for the parameters *ID*, *title*, *wcls* or *pid*, it is possible to pass a list (or tuple or set) of allowed values instead, which will search for a wider scope of windows:

```
dpowers_browser_win = Win('Dpowers documentation',
    ('Navigator.firefox','chromium-browser.Chromium-browser'))
# searches for windows with
# 'Dpowers documentation' contained in title
#   AND
# (window class is equal to 'Navigator.firefox'
#  OR window class is equal to 'chromium-browser.Chromium-browser')
```

In many cases it is desirable to make sure that exactly one matching window is found for each instance to avoid confusion. In other cases it might be useful to operate on a group of windows simultaneously.

---

**Note:** If a window was initially found and closed in the meantime, its ID will still be in the internal list of the *Win()* object. This can result in an Exception when trying to interact with a non-existent window ID. The *update()* or *remove_non_existing()* methods can be used to avoid this problem.

---

**update**()

Re-perform the initial search for matching existing windows and update this window object's internal ID list. This removes non-existing windows and searches for new matches.

**remove_non_existing**()

Cleanse this window object's internal ID list by removing those that do not exist anymore.

**property num**

Number (*int*) of windows found to match the given criteria. Equals zero if there was no match at all.

**ID**()

Returns the identification number (*int*) of this window if exactly one match was found. Returns `None` if no match was found. Raises `ValueError` if there was more than one matching window.

**IDs**()

Returns a tuple of length *num* containing the IDs of all found windows.

**title**()

Returns the title (*str*) of this window if exactly one match was found. Returns `None` if no match was found. Raises `ValueError` if there was more than one matching window.

**titles**()

Returns a tuple of length *num* containing the titles of all found windows. Use `titles_gen()` to create a generator instead of a tuple.

**wcls**()

Returns the window class (*str*) of this window if exactly one match was found. Returns `None` if no match was found. Raises `ValueError` if there was more than one matching window.

**wclasses**()

Returns a tuple of length *num* containing the window classes of all found windows. Use `wclasses_gen()` to create a generator instead of a tuple.

**pid**()

Returns the process ID (*int*) of this window if exactly one match was found. Returns `None` if no match was found. Raises `ValueError` if there was more than one matching window.

**pids**()

Returns a tuple of length *num* containing the process IDs of all found windows. Use `pids_gen()` to create a generator instead of a tuple.

**classmethod screen_res**()

Returns the screen resolution in pixels as a tuple (*screen_width, screen_height*).

**width**()

Returns the width in pixels (*int*) of this window if exactly one match was found. Returns `None` if no match was found. Raises `ValueError` if there was more than one matching window.

**widths**()

Returns a tuple of length *num* containing the widths of all found windows. Use `widths_gen()` to create a generator instead of a tuple.

**height**()

Returns the height in pixels (*int*) of this window if exactly one match was found. Returns `None` if no match was found. Raises `ValueError` if there was more than one matching window.

**heights**()

Returns a tuple of length *num* containing the heights of all found windows. Use `heights_gen()` to create a generator instead of a tuple.

**geometry**()

Returns the 4 border coordinates of this window if exactly one match was found. Returns `None` if no match was found. Raises `ValueError` if there was more than one matching window.

The return value is a 4-element tuple containing the window's border coordinates as *int* in pixels as follows: (left, top, right, bottom) = (Xmin, Ymin, Xmax, Ymax).

**geometries**()

Returns a tuple of length *num* containing the geometries of all found windows. Use `geometries_gen()` to create a generator instead of a tuple.

**info**()

Returns the 2-element tuple (*title()*, *wcls()*) of this window if exactly one match was found. Returns `None` if no match was found. Raises `ValueError` if there was more than one matching window.

**infos**()

Returns a tuple of length *num* containing the *info()* tuples of all found windows. Use `infos_gen()` to create a generator instead of a tuple.

**all_info**()

Returns the 5-element tuple (*ID()*, *title()*, *wcls()*, *pid()*, *geometry()*) of this window if exactly one match was found. Returns `None` if no match was found. Raises `ValueError` if there was more than one matching window.

**all_infos**()

Returns a tuple of length *num* containing the *all_info()* tuples of all found windows. Use `all_infos_gen()` to create a generator instead of a tuple.

**close**(*all=False*)

Closes this window.

**Parameters**

**all** – If set `True`, apply this command to all found windows.

**Raises**

**ValueError** – If *all* is `False` (default) and more than one window matches.

**kill**(*all=False*)

Kill (force close) this window.

**Parameters**
**all** – If set True, apply this command to all found windows.

**Raises**
**ValueError** – If *all* is False (default) and more than one window matches.

**minimize**(*all=False*)

Minimize this window.

**Parameters**
**all** – If set True, apply this command to all found windows.

**Raises**
**ValueError** – If *all* is False (default) and more than one window matches.

**maximize**(*all=False*)

Maximize this window.

**Parameters**
**all** – If set True, apply this command to all found windows.

**Raises**
**ValueError** – If *all* is False (default) and more than one window matches.

**max_left**(*all=False*)

Maximize this window to the left side of the screen.

**Parameters**
**all** – If set True, apply this command to all found windows.

**Raises**
**ValueError** – If *all* is False (default) and more than one window matches.

**max_right**(*all=False*)

Maximize this window to the right side of the screen.

**Parameters**
**all** – If set True, apply this command to all found windows.

**Raises**
**ValueError** – If *all* is False (default) and more than one window matches.

**move**(*x=-1*, *y=-1*, *width=-1*, *height=-1*, *all=False*)

Move and resize this window. Specifying -1 for one of the coordinates will keep the current value (default).

**Parameters**

- **x** (*int*) – New coordinate of left border.
- **y** (*int*) – New cordinate of top border
- **width** (*int*) – New width of window.
- **height** (*int*) – New hight of window.
- **all** – If set True, apply this command to all found windows.

**Raises**
**ValueError** – If *all* is False (default) and more than one window matches.

**activate**(*all=False*)

Activates this window.

**Parameters**
**all** – If set `True`, apply this command to all found windows.

**Raises**
**ValueError** – If *all* is `False` (default) and more than one window matches.

**wait_active**(*timeout=5*, *pause_when_found=0.05*, *timestep=0.2*)

Wait until the specified window is active. If several windows are matching, wait until any of them is active.

**Parameters**

- **timeout** (*float*) – Seconds to wait before returning anyway.
- **pause_when_found** (*float*) – Time to sleep after success.
- **timestep** (*float*) – Interval how often the condition is checked.

**Returns**

- The *Win()* object for the active window in case of success.
- `False` in case of a timeout.

**wait_not_active**(*timeout=5*, *timestep=0.2*)

Same as *wait_active()*, but reversed. It waits until the specified window(s) is (are) not active anymore.

**Returns**

- `True` in case of success.
- `False` in case of a timeout.

**wait_exist**(*timeout=5*, *pause_when_found=0.05*, *timestep=0.2*, *min_wincount=1*, *max_wincount=None*)

Waits until the number of matching windows is between *min_wincount* and *max_wincount*. By default, this means at least one matching window must exist.

**Parameters**

- **timeout** (*float*) – Seconds to wait before returning anyway.
- **pause_when_found** (*float*) – Time to sleep after success.
- **timestep** (*float*) – Interval how often the condition is checked.
- **min_wincount** (*int*) – Minimum number of matching windows necessary to continue.
- **max_wincount** – Maximum number of matching windows to continue. If set to `None` (default), there is no maximum.

**Returns**

- The *Win()* object containing all matching windows in case of success.
- `False` in case of a timeout.

**wait_not_exist**(*timeout=5*, *timestep=0.2*)

Wrapper for *wait_exist()* with *min_wincount* = *max_wincount* = 0.

**Returns**

- `True` in case of success.
- `False` in case of a timeout.

**wait_num_change**(*num_change*, *timeout=5*, *pause_when_found=0.05*, *timestep=0.2*)

Wrapper for *wait_exist()*. Waits until the number of currently matching windows has changed by *num_change* (relative to the initial found number *num*).

**Parameters**

**num_change** (*int*) – A positive or negative integer.

**Returns**

- The *Win()* object of the newly found window(s) if *num_change* is positive.
- The *Win()* object of the closed window(s) if *num_change* is negative. (These do not exist anymore.)
- `False` in case of a timeout.

Useful when opening a new window if other windows of the same class are already existing, see for example: *Launch the browser and simultaneously redirect key presses*.

**wait_exist_activate**(*timeout=5*, *timestep=0.2*)

Wait until at least one matching window exists and activate it immediately.

**Returns**

- The *Win()* object for the found window.
- `False` in case of timeout or if activate failed.

**class** Dpowers.Win.**Search**

Subclass of `Dpowers.windowpower.windowobjects.WindowSearch`, using the same adaptor instance (and thus the same backend) as *Dpowers.Win*.

### 2.7.2 Icon

**Examples**

- *Display a tray icon with customized menu*

**class** Dpowers.**Icon**

Subclass of *Dpowers.AdaptiveClass*.

Associated Adaptor class: `Dpowers.IconAdaptor`

How to import:

```
from Dpowers import Icon

# choose the default backend for your system:
Icon.adapt()

# alternatively, choose one of the following backends:
Icon.adapt('pystray')
Icon.adapt('yad_bash')
```

How to install dependencies for available backends:

```
>>> print(Icon.install_instructions())
pip install -U pillow pystray
sudo apt install yad
```

**classmethod adapt**(*main_info=None*, ***, *raise_error=True*, *require_backend=True*, ***method_infos*)

Choose the backend for this class. See *AdaptiveClass.adapt()*. For parameters see *Dpowers.Adaptor.adapt()*.

**menuitem**(*text=None*, *func=None*)

Adds a new entry to the icon's context menu.

**Parameters**

- **text** (*str*) – Text of the menu entry. If omitted `func.__name__` will be used as text (underscores are replaced by space).
- **func** – Function to execute when clicking this menu entry.

**additem**(*func*)

A decorator to apply *menuitem()* directly onto a function.

### 2.7.3 TriggerManager

**Examples**

- *Define a key sequence to trigger a function*
- *Define a combined key / button sequence as trigger*

**class** Dpowers.**TriggerManager**(*timeout=60*, *buffer=4*)

Subclass of *Dpowers.AdaptiveClass*.

Associated Adaptor class: *Dpowers.HookAdaptor*

How to import:

```
from Dpowers import TriggerManager

# choose the default backend for your system:
TriggerManager.adapt()

# alternatively, choose one of the following backends:
TriggerManager.adapt('evdev')
TriggerManager.adapt('pynput')
```

How to install dependencies for available backends:

```
>>> print(TriggerManager.install_instructions())
pip install -U pynput
```

**classmethod adapt**(*main_info=None*, ***, *raise_error=True*, *require_backend=True*, ***method_infos*)

Choose the backend for this class. See *AdaptiveClass.adapt()*. For parameters see *Dpowers.Adaptor.adapt()*.

### 2.7.4 KeyWaiter

**Examples**

- *Launch the browser and simultaneously redirect key presses*

**class** Dpowers.**KeyWaiter**(**args*, *eventmap=None*, *keys=True*, *buttons=False*, *press=True*, *release=False*, *write_rls=True*, *join_events=False*, ***kwargs*)

Subclass of *Dpowers.AdaptiveClass*.

Associated Adaptor class: *Dpowers.HookAdaptor*

How to import:

```
from Dpowers import KeyWaiter

# choose the default backend for your system:
KeyWaiter.adapt()

# alternatively, choose one of the following backends:
KeyWaiter.adapt('evdev')
KeyWaiter.adapt('pynput')
```

How to install dependencies for available backends:

```
>>> print(KeyWaiter.install_instructions())
pip install -U pynput
```

**classmethod adapt**(*main_info=None*, ***, *raise_error=True*, *require_backend=True*, ***method_infos*)

Choose the backend for this class. See *AdaptiveClass.adapt()*. For parameters see *Dpowers.Adaptor.adapt()*.

CHAPTER

THREE

# INDICES AND TABLES

- genindex

# INDEX

## Symbols



## A



## C



## D



## G



## H



## I



## K

## L



## M



## N



## P



## R



## S



## T



## U



## W