pyudev

Release 0.21.0

July 21, 2016

Contents

pyudev 0.21.0 (Changelog, installation)

pyudev is a LGPL licenced, pure Python 2/3 binding to libudev, the device and hardware management and information library of Linux.

Almost the complete libudev functionality is exposed. You can:

- Enumerate devices, filtered by specific criteria (*pyudev.Context*)
- Query device information, properties and attributes,
- Monitor devices, both synchronously and asynchronously with background threads, or within the event loops of Qt (`pyudev.pyqt4`, `pyudev.pyside`), glib (*pyudev.glib*) and wxPython (*pyudev.wx*).

CHAPTER 1

Documentation

Thanks to the power of libudev, usage of pyudev is very simple. Getting the labels of all partitions just takes a few lines:

```
>>> import pyudev
>>> context = pyudev.Context()
>>> for device in context.list_devices(subsystem='block', DEVTYPE='partition'):
...     print(device.get('ID_FS_LABEL', 'unlabeled partition'))
...
boot
swap
system
```

A user guide gives an introduction into common operations and concepts of pyudev, the API documentation provides a detailed reference:

1.1 Installation

1.1.1 Python versions and implementations

pyudev supports CPython from 2.6 up to the latest Python 3 release, and PyPy 1.5. Jython may work, too, but is not tested. Generally any Python implementation compatible with CPython 2.6 should work.

1.1.2 Dependencies

pyudev needs libudev 151 and newer, earlier versions of libudev as found on dated Linux systems may work, but are not tested and not officially supported. It is written in pure Python based on `ctypes`, so no compilers or headers are required for installation.

To use any of the toolkit integration modules. the corresponding toolkit must be available, but no toolkit is required during installation.

1.1.3 Installation from Cheeseshop

Install pyudev from the Cheeseshop with pip:

```
pip install pyudev
```

1.1.4 Installation from source code

Close the public repository:

```
git clone https://github.com/lunaryorn/pyudev.git
```

Or download tarball:

```
curl -OL https://github.com/lunaryorn/pyudev/tarball/master
```

Then install pyudev from the source code tree:

```
python setup.py install
```

1.2 User guide

This guide gives an introduction in how to use pyudev for common operations like device enumeration or monitoring:

Contents

A detailed reference is provided in the API documentation.

1.2.1 Getting started

Import pyudev and verify that you're using the latest version:

```
>>> import pyudev
>>> pyudev.__version__
u'0.16'
>>> pyudev.udev_version()
181
```

This prints the version of pyudev itself and of the underlying libudev.

1.2.2 A note on versioning

pyudev supports libudev 151 or newer, but still tries to cover the most recent libudev API completely. If you are using older libudev releases, some functionality of pyudev may be unavailable, simply because libudev is too old to support a specific feature. Whenever this is the case, the minimum required version of udev is noted in the

documentation (see *Device.is_initialized* for an example). If no version is specified for an attribute or a method, it is available on all supported libudev versions. You can check the version of the underlying libudev with *pyudev.udev_version()*.

1.2.3 Enumerating devices

A common use case is to enumerate available devices, or a subset thereof. But before you can do anything with pyudev, you need to establish a "connection" to the udev device database first. This connection is represented by a library *Context*:

```
>>> context = pyudev.Context()
```

The *Context* is the central object of pyudev and libudev. You will need a *Context* object for almost anything in pyudev. With the `context` you can now enumerate the available devices:

```
>>> for device in context.list_devices():
...     device
...
Device(u'/sys/devices/LNXSYSTM:00')
Device(u'/sys/devices/LNXSYSTM:00/LNXCPU:00')
Device(u'/sys/devices/LNXSYSTM:00/LNXCPU:01')
...
```

By default, `list_devices()` yields all devices available on the system as *Device* objects, but you can filter the list of devices with keyword arguments to enumerate all available partitions for example:

```
>>> for device in context.list_devices(subsystem='block', DEVTYPE='partition'):
...    print(device)
...
Device(u'/sys/devices/pci0000:00/0000:00:0d.0/host2/target2:0:0/2:0:0:0/block/sda/sda1')
Device(u'/sys/devices/pci0000:00/0000:00:0d.0/host2/target2:0:0/2:0:0:0/block/sda/sda2')
Device(u'/sys/devices/pci0000:00/0000:00:0d.0/host2/target2:0:0/2:0:0:0/block/sda/sda3')
```

The choice of the right filters depends on the use case and generally requires some knowledge about how udev classifies and categorizes devices. This is out of the scope of this guide. Poke around in `/sys/` to get a feeling for the udev-way of device handling, read the udev documentation or one of the tutorials in the net.

The keyword arguments of `list_devices()` provide the most common filter operations. You can apply other, less common filters by calling one of the `match_*` methods on the *Enumerator* returned by of `list_devices()`.

1.2.4 Accessing individual devices directly

If you just need a single specific *Device*, you don't need to enumerate all devices with a specific filter criterion. Instead, you can directly create *Device* objects from a device path (*Devices.from_path()*), by from a subsystem and device name (*Devices.from_name()*) or from a device file (*Devices.from_device_file()*). The following code gets the *Device* object for the first hard disc in three different ways:

```
>>> pyudev.Devices.from_path(context, '/sys/block/sda')
Device(u'/sys/devices/pci0000:00/0000:00:0d.0/host2/target2:0:0/2:0:0:0/block/sda')
>>> pyudev.Devices.from_name(context, 'block', 'sda')
Device(u'/sys/devices/pci0000:00/0000:00:0d.0/host2/target2:0:0/2:0:0:0/block/sda')
>>> pyudev.Devices.from_device_file(context, '/dev/sda')
Device(u'/sys/devices/pci0000:00/0000:00:0d.0/host2/target2:0:0/2:0:0:0/block/sda')
```

As you can see, you need to pass a *Context* to both methods as reference to the udev database from which to retrieve information about the device.

Note: The *Device* objects created in the above example refer to the same device. Consequently, they are considered equal:

```
>>> pyudev.Devices.from_path(context, '/sys/block/sda') == pyudev.Devices.from_name(context, 'block',
True
```

Whereas *Device* objects referring to different devices are unequal:

```
>>> pyudev.Devices.from_name(context, 'block', 'sda') == pyudev.Devices.from_name(context, 'block',
False
```

1.2.5 Querying device information

As you've seen, *Device* represents a device in the udev database. Each such device has a set of "device properties" (not to be confused with Python properties as created by `property()`!) that describe the capabilities and features of this device as well as its relationship to other devices.

Common device properties are also available as properties of a *Device* object. For instance, you can directly query the `device_node` and the `device_type` of block devices:

```
>>> for device in context.list_devices(subsystem='block'):
...     print('{0} ({1})'.format(device.device_node, device.device_type))
...
/dev/sr0 (disk)
/dev/sda (disk)
/dev/sda1 (partition)
/dev/sda2 (partition)
/dev/sda3 (partition)
```

For all other properties, *Device* provides a dictionary-like interface to directly access the device properties. You'll get the same information as with the generic properties:

```
>>> for device in context.list_devices(subsystem='block'):
...     print('{0} ({1})'.format(device['DEVNAME'], device['DEVTYPE']))
...
/dev/sr0 (disk)
/dev/sda (disk)
/dev/sda1 (partition)
/dev/sda2 (partition)
/dev/sda3 (partition)
```

Warning: When filtering devices, you have to use the device property names. The names of corresponding properties of *Device* will generally **not** work. Compare the following two statements:

```
>>> [device.device_node for device in context.list_devices(subsystem='block', DEVTYPE='partition')]
[u'/dev/sda1', u'/dev/sda2', u'/dev/sda3']
>>> [device.device_node for device in context.list_devices(subsystem='block', device_type='partitio
[]
```

But you can also query many device properties that are not available as Python properties on the *Device* object with a convenient mapping interface, like the filesystem type. *Device* provides a convenient mapping interface for this purpose:

```
>>> for device in context.list_devices(subsystem='block', DEVTYPE='partition'):
...     print('{0} ({1})'.format(device.device_node, device.get('ID_FS_TYPE')))
...
/dev/sda1 (ext3)
/dev/sda2 (swap)
/dev/sda3 (ext4)
```

Note: Such device specific properties may not be available on devices. Either use `get()` to specify default values for missing properties, or be prepared to catch `KeyError`.

Most device properties are computed by udev rules from the driver- and device-specific "device attributes". The *`Device.attributes`* mapping gives you access to these attributes, but generally you should not need these. Use the device properties whenever possible.

1.2.6 Examing the device hierarchy

A *`Device`* is part of a device hierarchy, and can have a *`parent`* device that more or less resembles the physical relationship between devices. For instance, the *`parent`* of partition devices is a *`Device`* object that represents the disc the partition is located on:

```
>>> for device in context.list_devices(subsystem='block', DEVTYPE='partition'):
...     print('{0} is located on {1}'.format(device.device_node, device.parent.device_node))
...
/dev/sda1 is located on /dev/sda
/dev/sda2 is located on /dev/sda
/dev/sda3 is located on /dev/sda
```

Generally, you should not rely on the direct parent-child relationship between two devices. Instead of accessing the parent directly, search for a parent within a specific subsystem, e.g. for the parent `block` device, with *`find_parent()`*:

```
>>> for device in context.list_devices(subsystem='block', DEVTYPE='partition'):
...     print('{0} is located on {1}'.format(device.device_node, device.find_parent('block').device_no
...
/dev/sda1 is located on /dev/sda
/dev/sda2 is located on /dev/sda
/dev/sda3 is located on /dev/sda
```

This also save you the tedious work of traversing the device tree manually, if you are interested in grand parents, like the name of the PCI slot of the SCSI or IDE controller of the disc that contains a partition:

```
>>> for device in context.list_devices(subsystem='block', DEVTYPE='partition'):
...     print('{0} attached to PCI slot {1}'.format(device.device_node, device.find_parent('pci')['PCI
...
/dev/sda1 attached to PCI slot 0000:00:0d.0
/dev/sda2 attached to PCI slot 0000:00:0d.0
/dev/sda3 attached to PCI slot 0000:00:0d.0
```

1.2.7 Monitoring devices

Synchronous monitoring

The Linux kernel emits events whenever devices are added, removed (e.g. a USB stick was plugged or unplugged) or have their attributes changed (e.g. the charge level of the battery changed). With *`pyudev.Monitor`* you can react

on such events, for example to react on added or removed mountable filesystems:

```
>>> monitor = pyudev.Monitor.from_netlink(context)
>>> monitor.filter_by('block')
>>> for device in iter(monitor.poll, None):
...     if 'ID_FS_TYPE' in device:
...         print('{0} partition {1}'.format(device.action, device.get('ID_FS_LABEL')))
...
add partition MULTIBOOT
remove partition MULTIBOOT
```

After construction of a monitor, you can install an event filter on the monitor using *filter_by()*. In the above example only events from the `block` subsystem are handled.

Note: Always prefer *filter_by()* and *filter_by_tag()* over manually filtering devices (e.g. by `device.subsystem == 'block'` or `tag in device.tags`). These methods install the filter on the *kernel side*. A process waiting for events is thus only woken up for events that match these filters. This is much nicer in terms of power consumption and system load than executing filters in the process itself.

Eventually, you can receive events from the monitor. As you can see, a *Monitor* is iterable and synchronously yields occurred events. If you iterate over a *Monitor*, you will synchronously receive events in an endless loop, until you raise an exception, or `break` the loop.

This is the quick and dirty way of monitoring, suitable for small scripts or quick experiments. In most cases however, simply iterating over the monitor is not sufficient, because it blocks the main thread, and can only be stopped if an event occurs (otherwise the loop is not entered and you have no chance to `break` it).

Asynchronous monitoring

For such use cases, pyudev provides asynchronous monitoring with *MonitorObserver*. You can use it to log added and removed mountable filesystems to a file, for example:

```
>>> monitor = pyudev.Monitor.from_netlink(context)
>>> monitor.filter_by('block')
>>> def log_event(action, device):
...     if 'ID_FS_TYPE' in device:
...         with open('filesystems.log', 'a+') as stream:
...             print('{0} - {1}'.format(action, device.get('ID_FS_LABEL')), file=stream)
...
>>> observer = pyudev.MonitorObserver(monitor, log_event)
>>> observer.start()
```

The `observer` gets an event handler (`log_event()` in this case) which is asynchronously invoked on every event emitted by the underlying `monitor` after the observer has been started using `start()`.

Warning: The callback is invoked from a *different* thread than the one in which the `observer` was created. Be sure to protect access to shared resource properly when you access them from the callback (e.g. by locking).

The `observer` can be stopped at any moment using `stop()`():

```
>>> observer.stop()
```

Warning: Do *not* call *stop()* from the event handler, neither directly nor indirectly. Use *send_stop()* if you need to stop monitoring from inside the event handler.

GUI toolkit integration

If you're using a GUI toolkit, you already have the event system of the GUI toolkit at hand. pyudev provides observer classes that seamlessly integration in the event system of the GUI toolkit and relieve you from caring with synchronisation issues that would occur with thread-based monitoring as implemented by *MonitorObserver*.

pyudev supports all major GUI toolkits available for Python:

- Qt 5 using `pyudev.pyqt5`
- Qt 4 using `pyudev.pyqt4` for the PyQt4 binding or `pyudev.pyside` for the PySide binding
- PyGtk 2 using *pyudev.glib*
- wxWidgets and wxPython using *pyudev.wx*

Each of these modules provides an observer class that observers the monitor asynchronously and emits proper signals upon device events.

For instance, the above example would look like this in a PySide application:

```
>>> from pyudev.pyside import QUDevMonitorObserver
>>> monitor = pyudev.Monitor.from_netlink(context)
>>> observer = QUDevMonitorObserver(monitor)
>>> observer.deviceEvent.connect(log_event)
>>> monitor.start()
```

Device objects as booleans

The use of a Device object in a boolean context as a shorthand for a comparison with None is an error.

The Device class inherits from the abstract Mapping class, as it maps udev property names to their values. Consequently, if a Device object has no udev properties, an unusual but not impossible occurance, the object is interpreted as False in a boolean context.

1.3 API documentation

This document provides API reference documentation for pyudev. Refer to the User guide for an introduction into pyudev.

pyudev	The `Context` provides the connection to the udev device database and enumerates devices.
`pyudev.pyqt4`	
`pyudev.pyqt5`	
`pyudev.pyside`	
pyudev.glib	`MonitorObserver` integrates device monitoring into the Glib
pyudev.wx	`MonitorObserver` integrates device monitoring into the wxPython_

1.3.1 `pyudev` - libudev binding

A binding to libudev.

The *Context* provides the connection to the udev device database and enumerates devices. Individual devices are represented by the *Device* class.

Device monitoring is provided by *Monitor* and *MonitorObserver*. With `pyudev.pyqt4`, `pyudev.pyside`,

pyudev.glib and *pyudev.wx* device monitoring can be integrated into the event loop of various GUI toolkits.

Context	A device database connection.
Device	A single device with attached attributes and properties.
Devices	Class for constructing *Device* objects from various kinds of data.
Monitor	A synchronous device event monitor.
MonitorObserver	An asynchronous observer for device events.

Version information

pyudev.**__version__**
The version of *pyudev* as string. This string contains a major and a minor version number, and optionally a revision in the form `major.minor.revision`. As said, the `revision` part is optional and may not be present.

This attribute is mainly intended for display purposes, use *__version_info__* to check the version of *pyudev* in source code.

pyudev.**__version_info__**
The version of *pyudev* as tuple of integers. This tuple contains a major and a minor number, and optionally a revision number in the form `(major, minor, revision)`. As said, the `revision` component is optional and may not be present.

New in version 0.10.

pyudev.**udev_version**()
Get the version of the underlying udev library.

udev doesn't use a standard major-minor versioning scheme, but instead labels releases with a single consecutive number. Consequently, the version number returned by this function is a single integer, and not a tuple (like for instance the interpreter version in `sys.version_info`).

As libudev itself does not provide a function to query the version number, this function calls the `udevadm` utility, so be prepared to catch `EnvironmentError` and `CalledProcessError` if you call this function.

Return the version number as single integer. Raise `ValueError`, if the version number retrieved from udev could not be converted to an integer. Raise `EnvironmentError`, if `udevadm` was not found, or could not be executed. Raise `subprocess.CalledProcessError`, if `udevadm` returned a non-zero exit code. On Python 2.7 or newer, the `output` attribute of this exception is correctly set.

New in version 0.8.

`Context` – UDev database context

class pyudev.**Context**
A device database connection.

This class represents a connection to the udev device database, and is really *the* central object to access udev. You need an instance of this class for almost anything else in pyudev.

This class itself gives access to various udev configuration data (e.g. *sys_path*, *device_path*), and provides device enumeration (*list_devices()*).

Instances of this class can directly be given as `udev *` to functions wrapped through `ctypes`.

__init__()
Create a new context.

sys_path
The `sysfs` mount point defaulting to `/sys'` as unicode string.

device_path
The device directory path defaulting to `/dev` as unicode string.

run_path
The run runtime directory path defaulting to `/run` as unicode string.

Required udev version: 167

New in version 0.10.

log_priority
The logging priority of the interal logging facitility of udev as integer with a standard `syslog` priority. Assign to this property to change the logging priority.

UDev uses the standard `syslog` priorities. Constants for these priorities are defined in the `syslog` module in the standard library:

```
>>> import syslog
>>> context = pyudev.Context()
>>> context.log_priority = syslog.LOG_DEBUG
```

New in version 0.9.

list_devices(***kwargs*)
List all available devices.

The arguments of this method are the same as for *Enumerator.match()*. In fact, the arguments are simply passed straight to method *match()*.

This function creates and returns an *Enumerator* object, that can be used to filter the list of devices, and eventually retrieve *Device* objects representing matching devices.

Changed in version 0.8: Accept keyword arguments now for easy matching.

`Enumerator` – device enumeration and filtering

class pyudev.**Enumerator**
A filtered iterable of devices.

To retrieve devices, simply iterate over an instance of this class. This operation yields *Device* objects representing the available devices.

Before iteration the device list can be filtered by subsystem or by property values using *match_subsystem()* and *match_property()*. Multiple subsystem (property) filters are combined using a logical OR, filters of different types are combined using a logical AND. The following filter for instance:

```
devices.match_subsystem('block').match_property(
    'ID_TYPE', 'disk').match_property('DEVTYPE', 'disk')
```

means the following:

```
subsystem == 'block' and (ID_TYPE == 'disk' or DEVTYPE == 'disk')
```

Once added, a filter cannot be removed anymore. Create a new object instead.

Instances of this class can directly be given as given `udev_enumerate *` to functions wrapped through `ctypes`.

match(***kwargs*)
Include devices according to the rules defined by the keyword arguments. These keyword arguments are interpreted as follows:

- The value for the keyword argument `subsystem` is forwarded to *match_subsystem()*.
- The value for the keyword argument `sys_name` is forwarded to *match_sys_name()*.
- The value for the keyword argument `tag` is forwarded to *match_tag()*.
- The value for the keyword argument `parent` is forwarded to *match_parent()*.
- All other keyword arguments are forwareded one by one to *match_property()*. The keyword argument itself is interpreted as property name, the value of the keyword argument as the property value.

All keyword arguments are optional, calling this method without no arguments at all is simply a noop.

Return the instance again.

New in version 0.8.

Changed in version 0.13: Add `parent` keyword.

match_subsystem(*subsystem*, *nomatch=False*)
Include all devices, which are part of the given `subsystem`.

`subsystem` is either a unicode string or a byte string, containing the name of the subsystem. If `nomatch` is `True` (default is `False`), the match is inverted: A device is only included if it is *not* part of the given `subsystem`.

Note that, if a device has no subsystem, it is not included either with value of `nomatch` True or with value of `nomatch` False.

Return the instance again.

match_sys_name(*sys_name*)
Include all devices with the given name.

`sys_name` is a byte or unicode string containing the device name.

Return the instance again.

New in version 0.8.

match_property(*prop*, *value*)
Include all devices, whose `prop` has the given `value`.

`prop` is either a unicode string or a byte string, containing the name of the property to match. `value` is a property value, being one of the following types:

- `int()`
- `bool()`
- A byte string
- Anything convertable to a unicode string (including a unicode string itself)

Return the instance again.

match_attribute(*attribute*, *value*, *nomatch=False*)
Include all devices, whose `attribute` has the given `value`.

`attribute` is either a unicode string or a byte string, containing the name of a sys attribute to match. `value` is an attribute value, being one of the following types:

- `int()`,
- `bool()`
- A byte string
- Anything convertable to a unicode string (including a unicode string itself)

If `nomatch` is `True` (default is `False`), the match is inverted: A device is include if the `attribute` does *not* match the given `value`.

Note: If `nomatch` is `True`, devices which do not have the given `attribute` at all are also included. In other words, with `nomatch=True` the given `attribute` is *not* guaranteed to exist on all returned devices.

Return the instance again.

match_tag(*tag*)
Include all devices, which have the given `tag` attached.

`tag` is a byte or unicode string containing the tag name.

Return the instance again.

Required udev version: 154

New in version 0.6.

match_parent(*parent*)
Include all devices on the subtree of the given `parent` device.

The `parent` device itself is also included.

`parent` is a *Device*.

Return the instance again.

Required udev version: 172

New in version 0.13.

match_is_initialized()
Include only devices, which are initialized.

Initialized devices have properly set device node permissions and context, and are (in case of network devices) fully renamed.

Currently this will not affect devices which do not have device nodes and are not network interfaces.

Return the instance again.

See also:

Device.is_initialized

Required udev version: 165

New in version 0.8.

__iter__()
Iterate over all matching devices.

Yield *Device* objects.

`Devices` – constructing *Device* objects

class `pyudev.`**`Devices`**
Class for constructing *`Device`* objects from various kinds of data.

Construction of device objects

classmethod `from_path`(*context*, *path*)
Create a device from a device `path`. The `path` may or may not start with the `sysfs` mount point:

```
>>> from pyudev import Context, Device
>>> context = Context()
>>> Devices.from_path(context, '/devices/platform')
Device(u'/sys/devices/platform')
>>> Devices.from_path(context, '/sys/devices/platform')
Device(u'/sys/devices/platform')
```

`context` is the *`Context`* in which to search the device. `path` is a device path as unicode or byte string.

Return a *`Device`* object for the device. Raise *`DeviceNotFoundAtPathError`*, if no device was found for `path`.

New in version 0.18.

classmethod `from_sys_path`(*context*, *sys_path*)
Create a new device from a given `sys_path`:

```
>>> from pyudev import Context, Device
>>> context = Context()
>>> Devices.from_sys_path(context, '/sys/devices/platform')
Device(u'/sys/devices/platform')
```

`context` is the *`Context`* in which to search the device. `sys_path` is a unicode or byte string containing the path of the device inside `sysfs` with the mount point included.

Return a *`Device`* object for the device. Raise *`DeviceNotFoundAtPathError`*, if no device was found for `sys_path`.

New in version 0.18.

classmethod `from_name`(*context*, *subsystem*, *sys_name*)
Create a new device from a given `subsystem` and a given `sys_name`:

```
>>> from pyudev import Context, Device
>>> context = Context()
>>> sda = Devices.from_name(context, 'block', 'sda')
>>> sda
Device(u'/sys/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda')
>>> sda == Devices.from_path(context, '/block/sda')
```

`context` is the *`Context`* in which to search the device. `subsystem` and `sys_name` are byte or unicode strings, which denote the subsystem and the name of the device to create.

Return a *`Device`* object for the device. Raise *`DeviceNotFoundByNameError`*, if no device was found with the given name.

New in version 0.18.

classmethod `from_device_number`(*context*, *typ*, *number*)
Create a new device from a device `number` with the given device `type`:

```
>>> import os
>>> from pyudev import Context, Device
>>> ctx = Context()
>>> major, minor = 8, 0
>>> device = Devices.from_device_number(context, 'block',
...     os.makedev(major, minor))
>>> device
Device(u'/sys/devices/pci0000:00/0000:00:11.0/host0/target0:0:0/0:0:0:0/block/sda')
>>> os.major(device.device_number), os.minor(device.device_number)
(8, 0)
```

Use `os.makedev()` to construct a device number from a major and a minor device number, as shown in the example above.

> **Warning:** Device numbers are not unique across different device types. Passing a correct number with a wrong type may silently yield a wrong device object, so make sure to pass the correct device type.

`context` is the *Context*, in which to search the device. `type` is either `'char'` or `'block'`, according to whether the device is a character or block device. `number` is the device number as integer.

Return a *Device* object for the device with the given device `number`. Raise *DeviceNotFoundByNumberError*, if no device was found with the given device type and number.

New in version 0.18.

classmethod `from_device_file`(*context*, *filename*)

Create a new device from the given device file:

```
>>> from pyudev import Context, Device
>>> context = Context()
>>> device = Devices.from_device_file(context, '/dev/sda')
>>> device
Device(u'/sys/devices/pci0000:00/0000:00:0d.0/host2/target2:0:0/2:0:0:0/block/sda')
>>> device.device_node
u'/dev/sda'
```

> **Warning:** Though the example seems to suggest that `device.device_node == filename` holds with `device = Devices.from_device_file(context, filename)`, this is only true in a majority of cases. There *can* be devices, for which this relation is actually false! Thus, do *not* expect *device_node* to be equal to the given `filename` for the returned *Device*. Especially, use *device_node* if you need the device file of a *Device* created with this method afterwards.

`context` is the *Context* in which to search the device. `filename` is a string containing the path of a device file.

Return a *Device* representing the given device file. Raise `DeviceNotFoundByFileError` if `filename` is no device file at all or if `filename` does not exist or if its metadata was inaccessible.

New in version 0.18.

classmethod `from_environment`(*context*)

Create a new device from the process environment (as in `os.environ`).

This only works reliable, if the current process is called from an udev rule, and is usually used for tools executed from `IMPORT=` rules. Use this method to create device objects in Python scripts called from udev rules.

context is the library *Context*.

Return a *Device* object constructed from the environment. Raise *DeviceNotFoundInEnvironmentError*, if no device could be created from the environment.

Required udev version: 152

New in version 0.18.

classmethod METHODS()
Return methods that obtain a *Device* from a variety of different data.

Returns a list of from_* methods.

Return type list of class methods

New in version 0.18.

Device – accessing device information

class pyudev.**Device**
A single device with attached attributes and properties.

This class subclasses the Mapping ABC, providing a read-only dictionary mapping property names to the corresponding values. Therefore all well-known dicitionary methods and operators (e.g. .keys(), .items(), in) are available to access device properties.

Aside of the properties, a device also has a set of udev-specific attributes like the path inside sysfs.

Device objects compare equal and unequal to other devices and to strings (based on *device_path*). However, there is no ordering on *Device* objects, and the corresponding operators >, <, <= and >= raise TypeError.

Warning: **Never** use object identity (is operator) to compare *Device* objects. *pyudev* may create multiple *Device* objects for the same device. Instead compare devices by value using == or !=.

Device objects are hashable and can therefore be used as keys in dictionaries and sets.

They can also be given directly as udev_device * to functions wrapped through ctypes.

Construction of device objects

classmethod from_path(*context*, *path*)
New in version 0.4.

Deprecated since version 0.18: Use *Devices.from_path* instead.

classmethod from_sys_path(*context*, *sys_path*)
Changed in version 0.4: Raise NoSuchDeviceError instead of returning None, if no device was found for sys_path.

Changed in version 0.5: Raise *DeviceNotFoundAtPathError* instead of NoSuchDeviceError.

Deprecated since version 0.18: Use *Devices.from_sys_path* instead.

classmethod from_name(*context*, *subsystem*, *sys_name*)
New in version 0.5.

Deprecated since version 0.18: Use *Devices.from_name* instead.

classmethod from_device_number(*context*, *typ*, *number*)
New in version 0.11.

Deprecated since version 0.18: Use *Devices.from_device_number* instead.

classmethod from_device_file(*context*, *filename*)
New in version 0.15.

Deprecated since version 0.18: Use *Devices.from_device_file* instead.

classmethod from_environment(*context*)
New in version 0.6.

Deprecated since version 0.18: Use *Devices.from_environment* instead.

General attributes

context
The *Context* to which this device is bound.

New in version 0.5.

sys_path
Absolute path of this device in `sysfs` including the `sysfs` mount point as unicode string.

sys_name
Device file name inside `sysfs` as unicode string.

sys_number
The trailing number of the *sys_name* as unicode string, or `None`, if the device has no trailing number in its name.

Note: The number is returned as unicode string to preserve the exact format of the number, especially any leading zeros:

```
>>> from pyudev import Context, Device
>>> context = Context()
>>> device = Devices.from_path(context, '/sys/devices/LNXSYSTM:00')
>>> device.sys_number
u'00'
```

To work with numbers, explicitly convert them to ints:

```
>>> int(device.sys_number)
0
```

New in version 0.11.

device_path
Kernel device path as unicode string. This path uniquely identifies a single device.

Unlike *sys_path*, this path does not contain the `sysfs` mount point. However, the path is absolute and starts with a slash `'/'`.

tags
A *Tags* object representing the tags attached to this device.

The *Tags* object supports a test for a single tag as well as iteration over all tags:

```
>>> from pyudev import Context
>>> context = Context()
>>> device = next(iter(context.list_devices(tag='systemd')))
>>> 'systemd' in device.tags
True
>>> list(device.tags)
[u'seat', u'systemd', u'uaccess']
```

Tags are arbitrary classifiers that can be attached to devices by udev scripts and daemons. For instance, systemd uses tags for multi-seat support.

Required udev version: 154

New in version 0.6.

Changed in version 0.13: Return a *Tags* object now.

Device driver and subsystem

subsystem
Name of the subsystem this device is part of as unicode string.

Returns name of subsystem if found, else None

Return type unicode string or NoneType

driver
The driver name as unicode string, or `None`, if there is no driver for this device.

New in version 0.5.

device_type
Device type as unicode string, or `None`, if the device type is unknown.

```
>>> from pyudev import Context
>>> context = Context()
>>> for device in context.list_devices(subsystem='net'):
...     '{0} - {1}'.format(device.sys_name, device.device_type or 'ethernet')
...
u'eth0 - ethernet'
u'wlan0 - wlan'
u'lo - ethernet'
u'vboxnet0 - ethernet'
```

New in version 0.10.

Device nodes

device_node
Absolute path to the device node of this device as unicode string or `None`, if this device doesn't have a device node. The path includes the device directory (see *Context.device_path*).

This path always points to the actual device node associated with this device, and never to any symbolic links to this device node. See *device_links* to get a list of symbolic links to this device node.

Warning: For devices created with *from_device_file()*, the value of this property is not necessary equal to the `filename` given to *from_device_file()*.

device_number
The device number of the associated device as integer, or `0`, if no device number is associated.

Use `os.major()` and `os.minor()` to decompose the device number into its major and minor number:

```
>>> import os
>>> from pyudev import Context, Device
>>> context = Context()
>>> sda = Devices.from_name(context, 'block', 'sda')
>>> sda.device_number
2048L
>>> (os.major(sda.device_number), os.minor(sda.device_number))
(8, 0)
```

For devices with an associated *device_node*, this is the same as the `st_rdev` field of the stat result of the *device_node*:

```
>>> os.stat(sda.device_node).st_rdev
2048
```

New in version 0.11.

device_links
An iterator, which yields the absolute paths (including the device directory, see *Context.device_path*) of all symbolic links pointing to the *device_node* of this device. The paths are unicode strings.

UDev can create symlinks to the original device node (see *device_node*) inside the device directory. This is often used to assign a constant, fixed device node to devices like removeable media, which technically do not have a constant device node, or to map a single device into multiple device hierarchies. The property provides access to all such symbolic links, which were created by UDev for this device.

> **Warning:** Links are not necessarily resolved by *Devices.from_device_file()*. Hence do *not* rely on `Devices.from_device_file(context, link).device_path == device.device_path` from any `link` in `device.device_links`.

Device initialization time

is_initialized
`True`, if the device is initialized, `False` otherwise.

A device is initialized, if udev has already handled this device and has set up device node permissions and context, or renamed a network device.

Consequently, this property is only implemented for devices with a device node or for network devices. On all other devices this property is always `True`.

It is *not* recommended, that you use uninitialized devices.

See also:

time_since_initialized

Required udev version: 165

New in version 0.8.

time_since_initialized
The time elapsed since initialization as `timedelta`.

This property is only implemented on devices, which need to store properties in the udev database. On all other devices this property is simply zero `timedelta`.

See also:

`is_initialized`

Required udev version: 165

New in version 0.8.

Device hierarchy

`parent`
The parent *`Device`* or `None`, if there is no parent device.

`ancestors`
Yield all ancestors of this device from bottom to top.

Return an iterator yielding a *`Device`* object for each ancestor of this device from bottom to top.

New in version 0.16.

`children`
Yield all direct children of this device.

Note: In udev, parent-child relationships are generally ambiguous, i.e. a parent can have multiple children, *and* a child can have multiple parents. Hence, *child.parent* == *parent* does generally *not* hold for all *child* objects in *parent.children*. In other words, the *`parent`* of a device in this property can be different from this device!

Note: As the underlying library does not provide any means to directly query the children of a device, this property performs a linear search through all devices.

Return an iterable yielding a *`Device`* object for each direct child of this device.

Required udev version: 172

Changed in version 0.13: Requires udev version 172 now.

`find_parent`(*subsystem*, *device_type=None*)
Find the parent device with the given `subsystem` and `device_type`.

`subsystem` is a byte or unicode string containing the name of the subsystem, in which to search for the parent. `device_type` is a byte or unicode string holding the expected device type of the parent. It can be `None` (the default), which means, that no specific device type is expected.

Return a parent *`Device`* within the given `subsystem` and, if `device_type` is not `None`, with the given `device_type`, or `None`, if this device has no parent device matching these constraints.

New in version 0.9.

Device events

`action`
The device event action as string, or `None`, if this device was not received from a *`Monitor`*.

Usual actions are:

'add' A device has been added (e.g. a USB device was plugged in)

'remove' A device has been removed (e.g. a USB device was unplugged)

'change' Something about the device changed (e.g. a device property)

'online' The device is online now

'offline' The device is offline now

> **Warning:** Though the actions listed above are the most common, this property *may* return other values, too, so be prepared to handle unknown actions!

New in version 0.16.

sequence_number
The device event sequence number as integer, or `0` if this device has no sequence number, i.e. was not received from a *Monitor*.

New in version 0.16.

Device properties

__iter__()
Iterate over the names of all properties defined for this device.

Return a generator yielding the names of all properties of this device as unicode strings.

Deprecated since version 0.21: Will be removed in 1.0. Access properties with Device.properties.

__len__()
Return the amount of properties defined for this device as integer.

Deprecated since version 0.21: Will be removed in 1.0. Access properties with Device.properties.

__getitem__(*prop*)
Get the given property from this device.

`prop` is a unicode or byte string containing the name of the property.

Return the property value as unicode string, or raise a `KeyError`, if the given property is not defined for this device.

Deprecated since version 0.21: Will be removed in 1.0. Access properties with Device.properties.

asint(*prop*)
Get the given property from this device as integer.

`prop` is a unicode or byte string containing the name of the property.

Return the property value as integer. Raise a `KeyError`, if the given property is not defined for this device, or a `ValueError`, if the property value cannot be converted to an integer.

Deprecated since version 0.21: Will be removed in 1.0. Use Device.properties.asint() instead.

asbool(*prop*)
Get the given property from this device as boolean.

A boolean property has either a value of `'1'` or of `'0'`, where `'1'` stands for `True`, and `'0'` for `False`. Any other value causes a `ValueError` to be raised.

`prop` is a unicode or byte string containing the name of the property.

Return `True`, if the property value is `'1'` and `False`, if the property value is `'0'`. Any other value raises a `ValueError`. Raise a `KeyError`, if the given property is not defined for this device.

Deprecated since version 0.21: Will be removed in 1.0. Use Device.properties.asbool() instead.

Sysfs attributes

attributes
The system attributes of this device as read-only *Attributes* mapping.

System attributes are basically normal files inside the the device directory. These files contain all sorts of information about the device, which may not be reflected by properties. These attributes are commonly used for matching in udev rules, and can be printed using `udevadm info --attribute-walk`.

The values of these attributes are not always proper strings, and can contain arbitrary bytes.

New in version 0.5.

Deprecated members

traverse()
Traverse all parent devices of this device from bottom to top.

Return an iterable yielding all parent devices as *Device* objects, *not* including the current device. The last yielded *Device* is the top of the device hierarchy.

Deprecated since version 0.16: Will be removed in 1.0. Use *ancestors* instead.

class pyudev.**Attributes**
udev attributes for *Device* objects.

New in version 0.5.

device
The *Device* to which these attributes belong.

asstring(*attribute*)
Get the given `attribute` for the device as unicode string.

Parameters **attribute** (*unicode or byte string*) – the key for an attribute value

Returns the value corresponding to `attribute`, as unicode

Return type unicode

Raises

- **KeyError** – if no value found for `attribute`
- **UnicodeDecodeError** – if value is not convertible

asint(*attribute*)
Get the given `attribute` as an int.

Parameters **attribute** (*unicode or byte string*) – the key for an attribute value

Returns the value corresponding to `attribute`, as an int

Return type int

Raises

- **KeyError** – if no value found for `attribute`

- **UnicodeDecodeError** – if value is not convertible to unicode
- **ValueError** – if unicode value can not be converted to an int

asbool(*attribute*)
Get the given `attribute` from this device as a bool.

Parameters **attribute** (*unicode or byte string*) – the key for an attribute value

Returns the value corresponding to `attribute`, as bool

Return type bool

Raises

- **KeyError** – if no value found for `attribute`
- **UnicodeDecodeError** – if value is not convertible to unicode
- **ValueError** – if unicode value can not be converted to a bool

A boolean attribute has either a value of `'1'` or of `'0'`, where `'1'` stands for `True`, and `'0'` for `False`. Any other value causes a `ValueError` to be raised.

class pyudev.**Tags**
A iterable over *Device* tags.

Subclasses the `Container` and the `Iterable` ABC.

__iter__()
Iterate over all tags.

Yield each tag as unicode string.

__contains__(*tag*)
Check for existence of `tag`.

`tag` is a tag as unicode string.

Return `True`, if `tag` is attached to the device, `False` otherwise.

Device exceptions

class pyudev.**DeviceNotFoundError**
An exception indicating that no *Device* was found.

Changed in version 0.5: Rename from `NoSuchDeviceError` to its current name.

class pyudev.**DeviceNotFoundAtPathError**(*sys_path*)
A *DeviceNotFoundError* indicating that no *Device* was found at a given path.

sys_path
The path that caused this error as string.

class pyudev.**DeviceNotFoundByNameError**(*subsystem*, *sys_name*)
A *DeviceNotFoundError* indicating that no *Device* was found with a given name.

subsystem
The subsystem that caused this error as string.

sys_name
The sys name that caused this error as string.

class pyudev.**DeviceNotFoundByNumberError**(*typ*, *number*)
A *DeviceNotFoundError* indicating, that no *Device* was found for a given device number.

device_number
The device number causing this error as integer.

device_type
The device type causing this error as string. Either `'char'` or `'block'`.

class pyudev.**DeviceNotFoundInEnvironmentError**
A *DeviceNotFoundError* indicating, that no *Device* could be constructed from the process environment.

Monitor – device monitoring

class pyudev.**Monitor**
A synchronous device event monitor.

A *Monitor* objects connects to the udev daemon and listens for changes to the device list. A monitor is created by connecting to the kernel daemon through netlink (see *from_netlink()*):

```
>>> from pyudev import Context, Monitor
>>> context = Context()
>>> monitor = Monitor.from_netlink(context)
```

Once the monitor is created, you can add a filter using *filter_by()* or *filter_by_tag()* to drop incoming events in subsystems, which are not of interest to the application:

```
>>> monitor.filter_by('input')
```

When the monitor is eventually set up, you can either poll for events synchronously:

```
>>> device = monitor.poll(timeout=3)
>>> if device:
...     print('{0.action}: {0}'.format(device))
...
```

Or you can monitor events asynchronously with *MonitorObserver*.

To integrate into various event processing frameworks, the monitor provides a `selectable` file description by *fileno()*. However, do *not* read or write directly on this file descriptor.

Instances of this class can directly be given as `udev_monitor *` to functions wrapped through `ctypes`.

Changed in version 0.16: Remove `from_socket()` which is deprecated, and even removed in recent udev versions.

classmethod from_netlink(*context*, *source=u'udev'*)
Create a monitor by connecting to the kernel daemon through netlink.

`context` is the *Context* to use. `source` is a string, describing the event source. Two sources are available:

`'udev'` (the default) Events emitted after udev as registered and configured the device. This is the absolutely recommended source for applications.

`'kernel'` Events emitted directly after the kernel has seen the device. The device has not yet been configured by udev and might not be usable at all. **Never** use this, unless you know what you are doing.

Return a new *Monitor* object, which is connected to the given source. Raise `ValueError`, if an invalid source has been specified. Raise `EnvironmentError`, if the creation of the monitor failed.

context
The *Context* to which this monitor is bound.

New in version 0.5.

started
`True`, if this monitor was started, `False` otherwise. Readonly.

See also:

start()

New in version 0.16.

fileno()
Return the file description associated with this monitor as integer.

This is really a real file descriptor ;), which can be watched and `select.select()`ed.

filter_by(*subsystem*, *device_type=None*)
Filter incoming events.

`subsystem` is a byte or unicode string with the name of a subsystem (e.g. `'input'`). Only events originating from the given subsystem pass the filter and are handed to the caller.

If given, `device_type` is a byte or unicode string specifying the device type. Only devices with the given device type are propagated to the caller. If `device_type` is not given, no additional filter for a specific device type is installed.

These filters are executed inside the kernel, and client processes will usually not be woken up for device, that do not match these filters.

Changed in version 0.15: This method can also be after *start()* now.

filter_by_tag(*tag*)
Filter incoming events by the given `tag`.

`tag` is a byte or unicode string with the name of a tag. Only events for devices which have this tag attached pass the filter and are handed to the caller.

Like with *filter_by()* this filter is also executed inside the kernel, so that client processes are usually not woken up for devices without the given `tag`.

Required udev version: 154

New in version 0.9.

Changed in version 0.15: This method can also be after *start()* now.

remove_filter()
Remove any filters installed with *filter_by()* or *filter_by_tag()* from this monitor.

> **Warning:** Up to udev 181 (and possibly even later versions) the underlying `udev_monitor_filter_remove()` seems to be broken. If used with affected versions this method always raises `ValueError`.

Raise `EnvironmentError` if removal of installed filters failed.

New in version 0.15.

start()
Start this monitor.

The monitor will not receive events until this method is called. This method does nothing if called on an already started *Monitor*.

Note: Typically you don't need to call this method. It is implicitly called by *poll()* and *__iter__()*.

See also:

started

Changed in version 0.16: This method does nothing if the *Monitor* was already started.

set_receive_buffer_size(*size*)
Set the receive buffer `size`.

`size` is the requested buffer size in bytes, as integer.

Note: The CAP_NET_ADMIN capability must be contained in the effective capability set of the caller for this method to succeed. Otherwise `EnvironmentError` will be raised, with `errno` set to `EPERM`. Unprivileged processes typically lack this capability. You can check the capabilities of the current process with the python-prctl module:

```
>>> import prctl
>>> prctl.cap_effective.net_admin
```

Raise `EnvironmentError`, if the buffer size could not bet set.

New in version 0.13.

poll(*timeout=None*)
Poll for a device event.

You can use this method together with `iter()` to synchronously monitor events in the current thread:

```
for device in iter(monitor.poll, None):
    print('{0.action} on {0.device_path}'.format(device))
```

Since this method will never return `None` if no `timeout` is specified, this is effectively an endless loop. With `functools.partial()` you can also create a loop that only waits for a specified time:

```
for device in iter(partial(monitor.poll, 3), None):
    print('{0.action} on {0.device_path}'.format(device))
```

This loop will only wait three seconds for a new device event. If no device event occurred after three seconds, the loop will exit.

`timeout` is a floating point number that specifies a time-out in seconds. If omitted or `None`, this method blocks until a device event is available. If `0`, this method just polls and will never block.

Note: This method implicitly calls *start()*.

Return the received *Device*, or `None` if a timeout occurred. Raise `EnvironmentError` if event retrieval failed.

See also:

Device.action The action that created this event.

Device.sequence_number The sequence number of this event.

New in version 0.16.

Deprecated members

enable_receiving()
Switch the monitor into listing mode.

Connect to the event source and receive incoming events. Only after calling this method, the monitor listens for incoming events.

Note: This method is implicitly called by *__iter__()*. You don't need to call it explicitly, if you are iterating over the monitor.

Deprecated since version 0.16: Will be removed in 1.0. Use *start()* instead.

receive_device()
Receive a single device from the monitor.

> **Warning:** You *must* call *start()* before calling this method.

The caller must make sure, that there are events available in the event queue. The call blocks, until a device is available.

If a device was available, return `(action, device)`. `device` is the *Device* object describing the device. `action` is a string describing the action. Usual actions are:

'add' A device has been added (e.g. a USB device was plugged in)

'remove' A device has been removed (e.g. a USB device was unplugged)

'change' Something about the device changed (e.g. a device property)

'online' The device is online now

'offline' The device is offline now

Raise `EnvironmentError`, if no device could be read.

Deprecated since version 0.16: Will be removed in 1.0. Use *Monitor.poll()* instead.

__iter__()
Wait for incoming events and receive them upon arrival.

This methods implicitly calls *start()*, and starts polling the *fileno()* of this monitor. If a event comes in, it receives the corresponding device and yields it to the caller.

The returned iterator is endless, and continues receiving devices without ever stopping.

Yields `(action, device)` (see *receive_device()* for a description).

Deprecated since version 0.16: Will be removed in 1.0. Use an explicit loop over *poll()* instead, or monitor asynchronously with *MonitorObserver*.

`MonitorObserver` – asynchronous device monitoring

class pyudev.**MonitorObserver**(*monitor*, *event_handler=None*, *callback=None*, **args*, ***kwargs*)
An asynchronous observer for device events.

This class subclasses `Thread` class to asynchronously observe a *Monitor* in a background thread:

```
>>> from pyudev import Context, Monitor, MonitorObserver
>>> context = Context()
>>> monitor = Monitor.from_netlink(context)
>>> monitor.filter_by(subsystem='input')
>>> def print_device_event(device):
...     print('background event {0.action}: {0.device_path}'.format(device))
>>> observer = MonitorObserver(monitor, callback=print_device_event, name='monitor-observer')
>>> observer.daemon
True
>>> observer.start()
```

In the above example, input device events will be printed in background, until *stop()* is called on `observer`.

Note: Instances of this class are always created as daemon thread. If you do not want to use daemon threads for monitoring, you need explicitly set `daemon` to `False` before invoking `start()`.

See also:

Device.action The action that created this event.

Device.sequence_number The sequence number of this event.

New in version 0.14.

Changed in version 0.15: *Monitor.start()* is implicitly called when the thread is started.

monitor
Get the *Monitor* observer by this object.

__init__(*monitor*, *event_handler=None*, *callback=None*, **args*, ***kwargs*)
Create a new observer for the given `monitor`.

`monitor` is the *Monitor* to observe. `callback` is the callable to invoke on events, with the signature `callback(device)` where `device` is the *Device* that caused the event.

> **Warning:** `callback` is invoked in the observer thread, hence the observer is blocked while callback executes.

`args` and `kwargs` are passed unchanged to the constructor of `Thread`.

Deprecated since version 0.16: The `event_handler` argument will be removed in 1.0. Use the `callback` argument instead.

Changed in version 0.16: Add `callback` argument.

send_stop()
Send a stop signal to the background thread.

The background thread will eventually exit, but it may still be running when this method returns. This method is essentially the asynchronous equivalent to *stop()*.

Note: The underlying *monitor* is *not* stopped.

---

stop()
Synchronously stop the background thread.

Note: This method can safely be called from the observer thread. In this case it is equivalent to *send_stop()*.

Send a stop signal to the backgroud (see *send_stop()*), and waits for the background thread to exit (see join()) if the current thread is *not* the observer thread.

After this method returns in a thread *that is not the observer thread*, the callback is guaranteed to not be invoked again anymore.

Note: The underlying *monitor* is *not* stopped.

Changed in version 0.16: This method can be called from the observer thread.

1.3.2 pyudev.pyqt4 – PyQt4_ integration

Deprecated API

1.3.3 pyudev.pyqt5 – PyQt5 integration

1.3.4 pyudev.pyside – PySide_ integration

Deprecated API

1.3.5 pyudev.glib – Glib/Gtk 2 integration

Glib integration.

MonitorObserver integrates device monitoring into the Glib mainloop by turing device events into Glib signals.

glib and gobject from PyGObject must be available when importing this module. PyGtk is not required.

New in version 0.7.

class pyudev.glib.**MonitorObserver**(*monitor*)

An observer for device events integrating into the glib mainloop.

This class inherits GObject to turn device events into glib signals.

```
>>> from pyudev import Context, Monitor
>>> from pyudev.glib import MonitorObserver
>>> context = Context()
>>> monitor = Monitor.from_netlink(context)
>>> monitor.filter_by(subsystem='input')
>>> observer = MonitorObserver(monitor)
>>> def device_event(observer, device):
...     print('event {0} on device {1}'.format(device.action, device))
>>> observer.connect('device-event', device_event)
>>> monitor.start()
```

This class is a child of gobject.GObject.

monitor

The *Monitor* observed by this object.

event_source
The event source, which represents the watch on the *monitor* (as returned by `glib.io_add_watch()`), or `None`, if *enabled* is `False`.

enabled
Whether this observer is enabled or not.

If `True` (the default), this observer is enabled, and emits events. Otherwise it is disabled and does not emit any events.

New in version 0.14.

Signals

This class emits the following GObject signal:

device-event(observer, action, device)
Emitted upon any device event.

`observer` is the *MonitorObserver*, which emitted the signal. `device` is the *Device*, which caused this event.

Use *action* to get the type of event.

Deprecated API

class `pyudev.glib.`**GUDevMonitorObserver**(*monitor*)
An observer for device events integrating into the `glib` mainloop.

Deprecated since version 0.17: Will be removed in 1.0. Use *MonitorObserver* instead.

monitor
The *Monitor* observed by this object.

event_source
The event source, which represents the watch on the *monitor* (as returned by `glib.io_add_watch()`), or `None`, if *enabled* is `False`.

enabled
Whether this observer is enabled or not.

If `True` (the default), this observer is enabled, and emits events. Otherwise it is disabled and does not emit any events.

New in version 0.14.

Signals

This class emits the following GObject signals:

device-event(observer, action, device)
Emitted upon any device event. `observer` is the *GUDevMonitorObserver*, which emitted the signal. `action` is a unicode string containing the action name, and `device` is the *Device*, which caused this event.

Basically the last two arguments of this signal are simply the return value of *receive_device()*

device-added(observer, device)
Emitted if a *Device* is added (e.g a USB device was plugged).

device-removed(observer, device)
Emitted if a *Device* is removed (e.g. a USB device was unplugged).

device-changed(observer, device)
Emitted if a *Device* was somehow changed (e.g. a change of a property)

device-moved(observer, device)
Emitted if a *Device* was renamed, moved or re-parented.

1.3.6 pyudev.wx – wxPython integration

Wx integration.

MonitorObserver integrates device monitoring into the wxPython_ mainloop by turing device events into wx events.

`wx` from wxPython_ must be available when importing this module.

New in version 0.14.

class pyudev.wx.**MonitorObserver**(*monitor*)
An observer for device events integrating into the `wx` mainloop.

This class inherits `EvtHandler` to turn device events into wx events:

```
>>> from pyudev import Context, Monitor
>>> from pyudev.wx import MonitorObserver
>>> context = Context()
>>> monitor = Monitor.from_netlink(context)
>>> monitor.filter_by(subsystem='input')
>>> observer = MonitorObserver(monitor)
>>> def device_event(event):
...     print('action {0} on device {1}'.format(event.device.action, event.device))
>>> observer.Bind(EVT_DEVICE_EVENT, device_event)
>>> monitor.start()
```

This class is a child of `wx.EvtHandler`.

New in version 0.17.

monitor
The *Monitor* observed by this object.

enabled
Whether this observer is enabled or not.

If `True` (the default), this observer is enabled, and emits events. Otherwise it is disabled and does not emit any events.

Events

MonitorObserver posts the following event:

pyudev.wx.**EVT_DEVICE_EVENT**
Emitted upon any device event. Receivers get a *DeviceEvent* object as argument.

class pyudev.wx.**DeviceEvent**
Argument object for *EVT_DEVICE_EVENT*.

device
The *Device* object that caused this event.

Use *action* to get the type of event.

Deprecated members

action
A unicode string containing the action name.

Deprecated since version 0.17: Will be removed in 1.0. Use *action* instead.

Deprecated API

class pyudev.wx.**WxUDevMonitorObserver**(*monitor*)
An observer for device events integrating into the wx mainloop.

Deprecated since version 0.17: Will be removed in 1.0. Use *MonitorObserver* instead.

monitor
The *Monitor* observed by this object.

enabled
Whether this observer is enabled or not.

If True (the default), this observer is enabled, and emits events. Otherwise it is disabled and does not emit any events.

Events

WxUDevMonitorObserver posts the following events in addition to *EVT_DEVICE_EVENT*:

pyudev.wx.**EVT_DEVICE_ADDED**
Emitted if a *Device* is added (e.g a USB device was plugged). Receivers get a *DeviceAddedEvent* object as argument.

Deprecated since version 0.17: Will be removed in 1.0.

pyudev.wx.**EVT_DEVICE_REMOVED**
Emitted if a *Device* is removed (e.g. a USB device was unplugged). Receivers get a *DeviceRemovedEvent* object as argument.

Deprecated since version 0.17: Will be removed in 1.0.

pyudev.wx.**EVT_DEVICE_CHANGED**
Emitted if a *Device* was somehow changed (e.g. a change of a property). Receivers get a *DeviceChangedEvent* object as argument.

Deprecated since version 0.17: Will be removed in 1.0.

pyudev.wx.**EVT_DEVICE_MOVED**
Emitted if a *Device* was renamed, moved or re-parented. Receivers get a *DeviceMovedEvent* object as argument.

class pyudev.wx.**DeviceAddedEvent**
class pyudev.wx.**DeviceRemovedEvent**
class pyudev.wx.**DeviceChangedEvent**

class pyudev.wx.**DeviceMovedEvent**

Argument objects for *EVT_DEVICE_ADDED*, *EVT_DEVICE_REMOVED*, *EVT_DEVICE_CHANGED* and *EVT_DEVICE_MOVED*.

Deprecated since version 0.17: Will be removed in 1.0.

device

The *Device* object that caused this event.

CHAPTER 2

Support

Please report issues, bugs and questions to the issue tracker, but respect the following guidelines:

- Check that the issue has not already been reported.
- Check that the issue is not already fixed in the `master` branch.
- Open issues with clear title and a detailed description in grammatically correct, complete sentences.
- Include the Python version and the udev version (see `udevadm --version`) in the description of your issue.

CHAPTER 3

Development

The source code is hosted on GitHub:

```
git clone https://github.com/lunaryorn/pyudev.git
```

If you want to contribute to pyudev, please read the guidelines for contributions and the testsuite documentation.

3.1 Contribute

Please fork the repository, and send pull requests with new features or bug fixes, but respect the following guidelines:

- Read how to properly contribute to open source projects on GitHub.
- Understand the branching model.
- Use a topic branch based on the `develop` branch to easily amend a pull request later, if necessary.
- Write good commit messages.
- Squash commits on the topic branch before opening a pull request.
- Respect **PEP 8** (use pep8 to check your coding style compliance).
- Add unit tests if possible (refer to the testsuite documentation).
- Add API documentation in docstrings.
- Open a pull request. that relates to but one subject with a clear title and description in grammatically correct, complete sentences.

Complying to these guidelines greatly increase the change of getting your pull request merged. You will be asked to improve your changeset if your pull request breaks any of the above guidelines.

If you intend to make larger changes, especially if these changes break the ABI, please ask on the mailing list first.

3.2 Testsuite documentation

This document explains the pyudev test suite and how to add new tests to this suite.

The pyudev testsuite uses the powerful pytest unittest framework, accompied by the nice mock library for mocking native functions and heavily extended with plugins to support the tests.

3.2.1 Test running

Direct testing using tox_

If you are on a Linux system run all tests with **tox_**. This tool automatically creates virtualenvs (see **virtualenv_**), installs all packages required by the test suite, and runs the tests.

Run all pyudev tests against Python 2.7, Python 3.2 and PyPy:

```
tox -e py27,py32,pypy
```

Pass any arguments you want to **py.test** after two dashes `--`:

```
tox -e py27,py32,pypy -- --enable-privileged
```

Notes

Device samples

Many pyudev tests run against the real device database of the system the tests are executed on. As testing against the whole database takes a long time, tests are run against a random sample by default. With the command line options provided by `udev_database` you can configure the size of this sample, or run the tests against a single device or the whole database.

Privileged tests

Some tests need to execute privileged operations like loading or unloading of kernel modules to trigger real udev events. These tests are disabled by default. Refer to `privileged` for more information on how to enable these tests and configure them properly.

3.2.2 `plugins` – Testsuite plugins

Plugins to support the pyudev testsuite.

The following plugins are provided and enabled:

`privileged` – Privileged operations

Support privileged operations to trigger real udev events.

This plugin adds *`load_dummy()`* and *`unload_dummy()`* to the `pytest` namespace.

Command line options

The plugin adds the following command line options to **py.test**:

`--enable-privileged`
Enable privileged tests. You'll need to have **sudo** configured correctly in order to run tests with this option.

Configuration

In order to execute these tests without failure, you need to configure **sudo** to allow the user that executes the test to run the following commands:

- `modprobe dummy`
- `modprobe -r dummy`

To do so, create a file `/etc/sudoers.d/20pyudev-tests` with the following content:

```
me ALL = (root) NOPASSWD: /sbin/modprobe dummy, /sbin/modprobe -r dummy
```

Replace `me` with your actual user name. `NOPASSWD:` tells **sudo** not to ask for a password when executing these commands. This is simply for the sake of convenience and to allow unattended test execution. Remove this word if you want to be asked for a password.

Make sure to change the owner and group to `root:root` and the permissions of this file to `440` afterwards, other **sudo** will refuse to load the file. Also check the file with **visudo** to prevent syntactic errors:

```
$ chown root:root /etc/sudoers.d/20pyudev-tests
$ chmod 440 /etc/sudoers.d/20pyudev-tests
$ visudo -c -f /etc/sudoers.d/20pyudev-tests
```

`pytest` namespace

The plugin adds the following functions to the `pytest` namespace:

`plugins.privileged.`**`load_dummy`**`()`
Load the `dummy` module.

If privileged tests are disabled, the current test is skipped.

`plugins.privileged.`**`unload_dummy`**`()`
Unload the `dummy` module.

If privileged tests are disabled, the current test is skipped.

`fake_monitor` – A fake `Monitor`

Provide a fake *`Monitor`*.

This fake monitor allows to trigger arbitrary events. Use this class to test class building upon monitor without the need to rely on real events generated by privileged operations as provided by the *`privileged`* plugin.

class `plugins.fake_monitor.`**`FakeMonitor`**(*device_to_emit*)
A fake *`Monitor`* which allows you to trigger arbitrary events.

This fake monitor implements the complete *`Monitor`* interface and works on real file descriptors so that you can `select()` the monitor.

`close``()`
Close sockets acquired by this monitor.

`trigger_event``()`
Trigger an event on clients of this monitor.

Funcargs

The plugin provides the following funcargs:

plugins.fake_monitor.**fake_monitor**(*request*)
Return a FakeMonitor, which emits the platform device as returned by the fake_monitor_device funcarg on all triggered actions.

> **Warning:** To use this funcarg, you have to provide the fake_monitor_device funcarg!

mock_libudev – Mock calls to libudev

Plugin to mock calls to libudev.

This plugin adds *libudev_list()* to the pytest namespace.

plugins.mock_libudev.**libudev_list**(*function*, *items*)
Mock a libudev linked list:

```
with pytest.libudev_list(device._libudev, 'udev_device_get_tag_list_entry', ['foo', 'bar']):
    assert list(device.tags) == ['foo', 'bar']
```

function is a string containing the name of the libudev function that returns the list. items is an iterable yielding items which shall be returned by the mocked list function. An item in items can either be a tuple with two components, where the first component is the item name, and the second the item value, or a single element, which is the item name. The item value is None in this case.

travis – Support for Travis CI

Support for Travis CI.

Test markers

pytest.mark.**not_on_travis**
Do not run the decorated test on Travis CI:

```
@pytest.mark.not_on_travis
def test_foo():
    assert True
```

test_foo will not be run on Travis CI.

pytest namespace

The plugin adds the following functions to the pytest namespace:

plugins.travis.**is_on_travis_ci**()
Determine whether the tests run on Travis CI.

Return True, if so, or False otherwise.

CHAPTER 4

Endorsements

If you're using pyudev and want to say something about it please add yourself to the endorsements page.

4.1 pyudev Users

If you are using pyudev and would like the world to know how and why, here is the place. Just submit a PR with an addition to the documentation, something like:

4.1.1 Choice of information about yourself.

What you are doing with pyudev and why it beats the alternatives.

CHAPTER 5

Other reading

5.1 Changelog

5.1.1 0.21.0 (July 20, 2016)

- Deprecate use of Device object as mapping from udev property names to values.
- Add a Properties class and a Device.properties() method for udev properties.
- Fix places where Device object was incorrectly used in a boolean context.
- Return an empty string, not None, if the property value is an empty string.
- Exceptions re-raised from libudev calls now have a message string.
- Insert a warning about using a Device in a boolean context in docs.
- Infrastructure for vagrant tests is removed.
- Various internal refactorings.
- Extensive test improvements.
- Numerous documentation fixes.

5.1.2 0.20.0 (April 29, 2016)

- Remove parsing code added in previous release.
- No longer do CI for Python 2.6.
- Eliminate all wildcard imports and __all__ statements.
- No longer use deprecated Device.from_sys_path() method.
- Minor pylint induced changes.
- Documentation fixes.

5.1.3 0.19.0 (Feb 3, 2016)

- Restore raising KeyError by Attributes.as* methods when attribute not found.
- Explicitly require six module.

- Never raise a DeviceNotFoundError when iterating over a device enumeration.
- Device.subsystem() now returns None if device has no subsystem.
- Add DeprecationWarnings to deprecated Device methods.
- Replace "/" with "!" in Device.from_name() sys_name parameter.
- Add some unstable classes for parsing some kinds of values.
- Make version info more like Python's including micro numbers and levels.
- Refactor some internal modules into subdirectories.
- Work on tests and reproducers.

5.1.4 0.18 (Dec 1, 2015)

- DeviceNotFoundError is no longer a subtype of LookupError
- Added support for pyqt5 monitor observer
- Added discover module, which looks up a device on limited information
- Attributes class no longer extends Mapping, extends object instead
- Attributes class no longer inherits [] operator, Mapping methods
- Attributes class objects are no longer iterable
- Attributes.available_attributes property added
- Attributes.get() method, with usual semantics, defined
- Device.from_* methods are deprecated, uses Devices.from_* methods instead
- Device.from_device_file() now raises DeviceNotFoundByFileError
- Device.from_device_number() now raises DeviceNotFoundByNumberError
- Devices.from_interface_index() method added
- Devices.from_kernel_device() method added
- Numerous testing infrastructure changes

5.1.5 0.17 (Aug 26, 2015)

- #52: Remove global libudev object
- #57: Really start the monitor on *pyudev.Monitor.poll()*
- #60: Do not use `select.select()` to avoid hitting its file descriptor limit
- #58: Force non-blocking IO in *pyudev.Monitor* to avoid blocking on receiving the device
- #63: Set proper flags on pipe fds.
- #65: Handle irregular polling events properly.
- #50: Add *pyudev.wx.MonitorObserver* and deprecate *pyudev.wx.WxUDevMonitorObserver*
- #50: Add *pyudev.glib.MonitorObserver* and deprecate *pyudev.glib.GUDevMonitorObserver*
- #50: Add `pyudev.pyqt4.MonitorObserver` and deprecate `pyudev.pyqt4.QUDevMonitorObserver`
- #50: Add `pyudev.pyside.MonitorObserver` and deprecate `pyudev.pyside.QUDevMonitorObserver`

- Add a wrapper function to retry interruptible system calls.

5.1.6 0.16.1 (Aug 02, 2012)

- #53: Fix source distribution
- #54: Fix typo in test

5.1.7 0.16 (Jul 25, 2012)

- Remove `pyudev.Monitor.from_socket()`.
- Deprecate *`pyudev.Device.traverse()`* in favor of *`pyudev.Device.ancestors`*.
- #47: Deprecate *`pyudev.Monitor.receive_device()`* in favor of *`pyudev.Monitor.poll`*.
- #47: Deprecate *`pyudev.Monitor.enable_receiving`* in favor of *`pyudev.Monitor.start`*.
- #47: Deprecate *`pyudev.Monitor.__iter__`* in favor of explicit looping or *`pyudev.MonitorObserver`*.
- #49: Deprecate `event_handler` to *`pyudev.MonitorObserver`* in favour of `callback` argument.
- #46: Continuously test pyudev on Travis-CI.
- Add *`pyudev.Device.ancestors`*.
- Add *`pyudev.Device.action`*.
- #10: Add *`pyudev.Device.sequence_number`*.
- #47: Add *`pyudev.Monitor.poll()`*.
- #47: Add *`pyudev.Monitor.started`*.
- #49: Add `callback` argument to *`pyudev.Monitor`*.
- *`pyudev.Monitor.start()`* can be called repeatedly.
- #45: Get rid of 2to3
- #43: Fix test failures on Python 2.6
- Fix signature in declaration of `udev_monitor_set_receive_buffer_size`.
- #44: Test wrapped signatures with help of `gccxml`.
- Fix compatibility with udev 183 and newer in *`pyudev.Context`*.
- *`pyudev.MonitorObserver.stop()`* can be called from the observer thread.

5.1.8 0.15 (Mar 1, 2012)

- #20: Add *`remove_filter()`*.
- #40: Add user guide to the documentation.
- #39: Add *`pyudev.Device.from_device_file()`*.
- `errno.EINVAL` from underlying libudev functions causes `ValueError` instead of `EnvironmentError`.
- *`pyudev.MonitorObserver`* calls *`pyudev.Monitor.enable_receiving()`* when started.

- #20: *pyudev.Monitor.filter_by()* and *pyudev.Monitor.filter_by_tag()* can be called after *pyudev.Monitor.enable_receiving()*.

5.1.9 0.14 (Feb 10, 2012)

- Host documentation at http://pyudev.readthedocs.org (thanks to the readthedocs.org team for this service)
- #37: Add *pyudev.wx.WxUDevMonitorObserver* for wxPython (thanks to Tobias Eberle).
- Add *pyudev.MonitorObserver*.
- Add *pyudev.glib.GUDevMonitorObserver.enabled*, `pyudev.pyqt4.QUDevMonitorObserver.enabled` and `pyudev.pyside.QUDevMonitorObserver.enabled`.

5.1.10 0.13 (Nov 4, 2011)

- #36: Add *pyudev.Monitor.set_receive_buffer_size()* (thanks to Rémi Rérolle).
- Add *pyudev.Enumerator.match_parent()*.
- Add `parent` keyword argument to *pyudev.Enumerator.match()*.
- #31: Add *pyudev.Enumerator.match_attribute()*.
- Add `nomatch` argument to *pyudev.Enumerator.match_subsystem()* and *pyudev.Enumerator.match_attribute()*.
- Remove `pyudev.Enumerator.match_children()` in favour of *pyudev.Enumerator.match_parent()*.
- #34: *pyudev.Device.tags* returns a *pyudev.Tags* object.
- *pyudev.Device.children* requires udev version 172 now

5.1.11 0.12 (Aug 31, 2011)

- #32: Fix memory leak.
- #33: Fix Python 3 support for *pyudev.glib*.
- Fix license header in `pyudev._compat`.

5.1.12 0.11 (Jun 26, 2011)

- #30: Add *pyudev.Device.sys_number*.
- #29: Add *pyudev.Device.from_device_number()*
- #29: Add *pyudev.Device.device_number*.
- Support PyPy.

5.1.13 0.10 (Apr 20, 2011)

- Add *pyudev.__version_info__*
- Add *pyudev.Device.device_type*
- *pyudev.Context*, *pyudev.Enumerator*, *pyudev.Device* and *pyudev.Monitor* can directly be passed to `ctypes`-wrapped functions.
- #24: Add *pyudev.Context.run_path*.

5.1.14 0.9 (Mar 09, 2011)

- #21: Add *pyudev.Device.find_parent()*.
- #22: Add *pyudev.Monitor.filter_by_tag()*.
- Add *pyudev.Context.log_priority*.
- Improve error reporting, if libudev is missing.

5.1.15 0.8 (Jan 08, 2011)

- #16: Add *pyudev.Enumerator.match()*.
- Add keyword arguments to *pyudev.Context.list_devices()*.
- #19: Add *pyudev.Enumerator.match_sys_name()*.
- #18: Add *pyudev.udev_version()*.
- #17: Add *pyudev.Device.is_initialized*.
- #17: Add *pyudev.Device.time_since_initialized*.
- #17: Add *pyudev.Enumerator.match_is_initialized()*
- Fix support for earlier releases of udev.
- Document minimum udev version for all affected attributes.

5.1.16 0.7 (Nov 15, 2010)

- #15: Add *pyudev.glib.GUDevMonitorObserver*.

5.1.17 0.6 (Oct 03, 2010)

- #8: Add *pyudev.Device.tags*.
- #8: Add *pyudev.Enumerator.match_tag()*.
- #11: Add *pyudev.Device.from_environment()*
- #5: Add `pyudev.pyside`
- #14: Remove apipkg dependency.
- #14: Require explicit import of `pyudev.pyqt4`.
- Fix licence headers in source files.

5.1.18 0.5 (Sep 06, 2010)

- Support Python 3.
- #6: Add *pyudev.Device.attributes* (thanks to Daniel Lazzari).
- #6: Add *pyudev.Attributes* (thanks to Daniel Lazzari).
- #7: *pyudev.Device.context* and *pyudev.Monitor.context* are part of the public API.
- #9: Add *pyudev.Device.driver*.
- #12: Add *pyudev.Device.from_name()*.
- Rename `pyudev.NoSuchDeviceError` to *pyudev.DeviceNotFoundError*.
- *pyudev.Device.from_sys_path()* raises *pyudev.DeviceNotFoundAtPathError*.
- #13: Fix `AttributeError` in *pyudev.Device.device_node*.
- Improve and extend documentation.
- Add more tests.

5.1.19 0.4 (Aug 23, 2010)

API changes

- #3: Rename `udev` to *pyudev*.
- #3: Rename `qudev` to `pyudev.pyqt4`.
- Add *pyudev.Device.from_path()*.
- *pyudev.Device.from_sys_path()* raises `pyudev.NoSuchDeviceError`.
- *pyudev.Monitor.receive_device()* raises `EnvironmentError`.
- `errno`, `strerror` and `filename` attributes of `EnvironmentError` exceptions have meaningful content.
- Fix `NameError` in `pyudev.Monitor.from_socket()`
- `subsystem` argument to *pyudev.Monitor.filter_by()* is mandatory.
- Delete underlying C objects if *pyudev.Device* is garbage-collected.
- Fix broken signal emitting in `pyudev.pyqt4.QUDevMonitorObserver`.

5.1.20 0.3 (Jul 28, 2010)

- #1: Fix documentation to reflect the actual behaviour of the underlying API
- Raise `TypeError` if `udev.Device` are compared with >, >=, < or <=.
- Add `udev.Enumerator.match_children()`.
- Add `udev.Device.children`.
- Add `qudev.QUDevMonitorObserver.deviceChanged()`.
- Add `qudev.QUDevMonitorObserver.deviceMoved()`.

5.1.21 0.2 (Jun 28, 2010)

- Add `udev.Monitor`.
- Add `udev.Device.asbool()`.
- Add `udev.Device.asint()`.
- Remove type magic in `udev.Device.__getitem__()`.
- Add `qudev`.

5.1.22 0.1 (May 03, 2010)

- Initial release.
- Add `udev.Context`.
- Add `udev.Device`.
- Add `udev.Enumerator`.

5.2 Licencing

```
                    GNU LESSER GENERAL PUBLIC LICENSE
                         Version 2.1, February 1999

 Copyright (C) 1991, 1999 Free Software Foundation, Inc.
     51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL.  It also counts
 as the successor of the GNU Library Public License, version 2, hence
 the version number 2.1.]

                             Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
Licenses are intended to guarantee your freedom to share and change
free software--to make sure the software is free for all its users.

  This license, the Lesser General Public License, applies to some
specially designated software packages--typically libraries--of the
Free Software Foundation and other authors who decide to use it.  You
can use it too, but we suggest you first think carefully about whether
this license or the ordinary General Public License is the better
strategy to use in any particular case, based on the explanations below.

  When we speak of free software, we are referring to freedom of use,
not price.  Our General Public Licenses are designed to make sure that
you have the freedom to distribute copies of free software (and charge
for this service if you wish); that you receive source code or can get
it if you want it; that you can change the software and use pieces of
it in new free programs; and that you are informed that you can do
these things.
```

```
  To protect your rights, we need to make restrictions that forbid
distributors to deny you these rights or to ask you to surrender these
rights.  These restrictions translate to certain responsibilities for
you if you distribute copies of the library or if you modify it.

  For example, if you distribute copies of the library, whether gratis
or for a fee, you must give the recipients all the rights that we gave
you.  You must make sure that they, too, receive or can get the source
code.  If you link other code with the library, you must provide
complete object files to the recipients, so that they can relink them
with the library after making changes to the library and recompiling
it.  And you must show them these terms so they know their rights.

  We protect your rights with a two-step method: (1) we copyright the
library, and (2) we offer you this license, which gives you legal
permission to copy, distribute and/or modify the library.

  To protect each distributor, we want to make it very clear that
there is no warranty for the free library.  Also, if the library is
modified by someone else and passed on, the recipients should know
that what they have is not the original version, so that the original
author's reputation will not be affected by problems that might be
  Finally, software patents pose a constant threat to the existence of
any free program.  We wish to make sure that a company cannot
effectively restrict the users of a free program by obtaining a
restrictive license from a patent holder.  Therefore, we insist that
any patent license obtained for a version of the library must be
consistent with the full freedom of use specified in this license.
  Most GNU software, including some libraries, is covered by the
ordinary GNU General Public License.  This license, the GNU Lesser
General Public License, applies to certain designated libraries, and
is quite different from the ordinary General Public License.  We use
this license for certain libraries in order to permit linking those
libraries into non-free programs.
  When a program is linked with a library, whether statically or using
a shared library, the combination of the two is legally speaking a
combined work, a derivative of the original library.  The ordinary
General Public License therefore permits such linking only if the
entire combination fits its criteria of freedom.  The Lesser General
Public License permits more lax criteria for linking other code with
the library.
  We call this license the "Lesser" General Public License because it
does Less to protect the user's freedom than the ordinary General
Public License.  It also provides other free software developers Less
of an advantage over competing non-free programs.  These disadvantages
are the reason we use the ordinary General Public License for many
libraries.  However, the Lesser license provides advantages in certain
special circumstances.
  For example, on rare occasions, there may be a special need to
encourage the widest possible use of a certain library, so that it becomes
a de-facto standard.  To achieve this, non-free programs must be
allowed to use the library.  A more frequent case is that a free
library does the same job as widely used non-free libraries.  In this
case, there is little to gain by limiting the free library to free
software only, so we use the Lesser General Public License.
  In other cases, permission to use a particular library in non-free
programs enables a greater number of people to use a large body of
free software.  For example, permission to use the GNU C Library in
```

```
non-free programs enables many more people to use the whole GNU
operating system, as well as its variant, the GNU/Linux operating
system.
  Although the Lesser General Public License is Less protective of the
users' freedom, it does ensure that the user of a program that is
linked with the Library has the freedom and the wherewithal to run
that program using a modified version of the Library.
  The precise terms and conditions for copying, distribution and
modification follow.  Pay close attention to the difference between a
"work based on the library" and a "work that uses the library".  The
former contains code derived from the library, whereas the latter must
be combined with the library in order to run.


  GNU LESSER GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION
  0. This License Agreement applies to any software library or other
program which contains a notice placed by the copyright holder or
other authorized party saying it may be distributed under the terms of
this Lesser General Public License (also called "this License").
Each licensee is addressed as "you".
  A "library" means a collection of software functions and/or data
prepared so as to be conveniently linked with application programs
(which use some of those functions and data) to form executables.
  The "Library", below, refers to any such software library or work
which has been distributed under these terms.  A "work based on the
Library" means either the Library or any derivative work under
copyright law: that is to say, a work containing the Library or a
portion of it, either verbatim or with modifications and/or translated
straightforwardly into another language.  (Hereinafter, translation is
included without limitation in the term "modification".)
  "Source code" for a work means the preferred form of the work for
making modifications to it.  For a library, complete source code means
all the source code for all modules it contains, plus any associated
interface definition files, plus the scripts used to control compilation
and installation of the library.
  Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running a program using the Library is not restricted, and output from
such a program is covered only if its contents constitute a work based
on the Library (independent of the use of the Library in a tool for
writing it).  Whether that is true depends on what the Library does
and what the program that uses the Library does.
  1. You may copy and distribute verbatim copies of the Library's
complete source code as you receive it, in any medium, provided that
you conspicuously and appropriately publish on each copy an
appropriate copyright notice and disclaimer of warranty; keep intact
all the notices that refer to this License and to the absence of any
warranty; and distribute a copy of this License along with the
Library.
  You may charge a fee for the physical act of transferring a copy,
and you may at your option offer warranty protection in exchange for a
fee.
  2. You may modify your copy or copies of the Library or any portion
of it, thus forming a work based on the Library, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:
    a) The modified work must itself be a software library.
```

```
    b) You must cause the files modified to carry prominent notices
    stating that you changed the files and the date of any change.
    c) You must cause the whole of the work to be licensed at no
    charge to all third parties under the terms of this License.
    d) If a facility in the modified Library refers to a function or a
    table of data to be supplied by an application program that uses
    the facility, other than as an argument passed when the facility
    is invoked, then you must make a good faith effort to ensure that,
    in the event an application does not supply such function or
    table, the facility still operates, and performs whatever part of
    its purpose remains meaningful.
    (For example, a function in a library to compute square roots has
    a purpose that is entirely well-defined independent of the
    application.  Therefore, Subsection 2d requires that any
    application-supplied function or table used by this function must
    be optional: if the application does not supply it, the square
    root function must still compute square roots.)
These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Library,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Library, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote
it.
Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Library.
In addition, mere aggregation of another work not based on the Library
with the Library (or with a work based on the Library) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.
  3. You may opt to apply the terms of the ordinary GNU General Public
License instead of this License to a given copy of the Library.  To do
this, you must alter all the notices that refer to this License, so
that they refer to the ordinary GNU General Public License, version 2,
instead of to this License.  (If a newer version than version 2 of the
ordinary GNU General Public License has appeared, then you can specify
that version instead if you wish.)  Do not make any other change in
these notices.
  Once this change is made in a given copy, it is irreversible for
that copy, so the ordinary GNU General Public License applies to all
subsequent copies and derivative works made from that copy.
  This option is useful when you wish to copy part of the code of
the Library into a program that is not a library.
  4. You may copy and distribute the Library (or a portion or
derivative of it, under Section 2) in object code or executable form
under the terms of Sections 1 and 2 above provided that you accompany
it with the complete corresponding machine-readable source code, which
must be distributed under the terms of Sections 1 and 2 above on a
medium customarily used for software interchange.
  If distribution of object code is made by offering access to copy
from a designated place, then offering equivalent access to copy the
source code from the same place satisfies the requirement to
distribute the source code, even though third parties are not
```

```
compelled to copy the source along with the object code.
  5. A program that contains no derivative of any portion of the
Library, but is designed to work with the Library by being compiled or
linked with it, is called a "work that uses the Library".  Such a
work, in isolation, is not a derivative work of the Library, and
therefore falls outside the scope of this License.
  However, linking a "work that uses the Library" with the Library
creates an executable that is a derivative of the Library (because it
contains portions of the Library), rather than a "work that uses the
library".  The executable is therefore covered by this License.
Section 6 states terms for distribution of such executables.
  When a "work that uses the Library" uses material from a header file
that is part of the Library, the object code for the work may be a
derivative work of the Library even though the source code is not.
Whether this is true is especially significant if the work can be
linked without the Library, or if the work is itself a library.  The
threshold for this to be true is not precisely defined by law.
  If such an object file uses only numerical parameters, data
structure layouts and accessors, and small macros and small inline
functions (ten lines or less in length), then the use of the object
file is unrestricted, regardless of whether it is legally a derivative
work.  (Executables containing this object code plus portions of the
Library will still fall under Section 6.)
  Otherwise, if the work is a derivative of the Library, you may
distribute the object code for the work under the terms of Section 6.
Any executables containing that work also fall under Section 6,
whether or not they are linked directly with the Library itself.
  6. As an exception to the Sections above, you may also combine or
link a "work that uses the Library" with the Library to produce a
work containing portions of the Library, and distribute that work
under terms of your choice, provided that the terms permit
modification of the work for the customer's own use and reverse
engineering for debugging such modifications.
  You must give prominent notice with each copy of the work that the
Library is used in it and that the Library and its use are covered by
this License.  You must supply a copy of this License.  If the work
during execution displays copyright notices, you must include the
copyright notice for the Library among them, as well as a reference
directing the user to the copy of this License.  Also, you must do one
of these things:
    a) Accompany the work with the complete corresponding
    machine-readable source code for the Library including whatever
    changes were used in the work (which must be distributed under
    Sections 1 and 2 above); and, if the work is an executable linked
    with the Library, with the complete machine-readable "work that
    uses the Library", as object code and/or source code, so that the
    user can modify the Library and then relink to produce a modified
    executable containing the modified Library.  (It is understood
    that the user who changes the contents of definitions files in the
    Library will not necessarily be able to recompile the application
    to use the modified definitions.)
    b) Use a suitable shared library mechanism for linking with the
    Library.  A suitable mechanism is one that (1) uses at run time a
    copy of the library already present on the user's computer system,
    rather than copying library functions into the executable, and (2)
    will operate properly with a modified version of the library, if
    the user installs one, as long as the modified version is
    interface-compatible with the version that the work was made with.
```

```
    c) Accompany the work with a written offer, valid for at
    least three years, to give the same user the materials
    specified in Subsection 6a, above, for a charge no more
    than the cost of performing this distribution.
    d) If distribution of the work is made by offering access to copy
    from a designated place, offer equivalent access to copy the above
    specified materials from the same place.
    e) Verify that the user has already received a copy of these
    materials or that you have already sent this user a copy.
  For an executable, the required form of the "work that uses the
Library" must include any data and utility programs needed for
reproducing the executable from it.  However, as a special exception,
the materials to be distributed need not include anything that is
normally distributed (in either source or binary form) with the major
components (compiler, kernel, and so on) of the operating system on
which the executable runs, unless that component itself accompanies
the executable.
  It may happen that this requirement contradicts the license
restrictions of other proprietary libraries that do not normally
accompany the operating system.  Such a contradiction means you cannot
use both them and the Library together in an executable that you
distribute.
  7. You may place library facilities that are a work based on the
Library side-by-side in a single library together with other library
facilities not covered by this License, and distribute such a combined
library, provided that the separate distribution of the work based on
the Library and of the other library facilities is otherwise
permitted, and provided that you do these two things:
    a) Accompany the combined library with a copy of the same work
    based on the Library, uncombined with any other library
    facilities.  This must be distributed under the terms of the
    Sections above.
    b) Give prominent notice with the combined library of the fact
    that part of it is a work based on the Library, and explaining
    where to find the accompanying uncombined form of the same work.
  8. You may not copy, modify, sublicense, link with, or distribute
the Library except as expressly provided under this License.  Any
attempt otherwise to copy, modify, sublicense, link with, or
distribute the Library is void, and will automatically terminate your
rights under this License.  However, parties who have received copies,
or rights, from you under this License will not have their licenses
terminated so long as such parties remain in full compliance.
  9. You are not required to accept this License, since you have not
signed it.  However, nothing else grants you permission to modify or
distribute the Library or its derivative works.  These actions are
prohibited by law if you do not accept this License.  Therefore, by
modifying or distributing the Library (or any work based on the
Library), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Library or works based on it.
  10. Each time you redistribute the Library (or any work based on the
Library), the recipient automatically receives a license from the
original licensor to copy, distribute, link with or modify the Library
subject to these terms and conditions.  You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties with
this License.
  11. If, as a consequence of a court judgment or allegation of patent
```

```
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Library at all.  For example, if a patent
license would not permit royalty-free redistribution of the Library by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Library.
If any portion of this section is held invalid or unenforceable under any
particular circumstance, the balance of the section is intended to apply,
and the section as a whole is intended to apply in other circumstances.
It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system which is
implemented by public license practices.  Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.
This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.
  12. If the distribution and/or use of the Library is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Library under this License may add
an explicit geographical distribution limitation excluding those countries,
so that distribution is permitted only in or among countries not thus
excluded.  In such case, this License incorporates the limitation as if
written in the body of this License.
  13. The Free Software Foundation may publish revised and/or new
versions of the Lesser General Public License from time to time.
Such new versions will be similar in spirit to the present version,
but may differ in detail to address new problems or concerns.
Each version is given a distinguishing version number.  If the Library
specifies a version number of this License which applies to it and
"any later version", you have the option of following the terms and
conditions either of that version or of any later version published by
the Free Software Foundation.  If the Library does not specify a
license version number, you may choose any version ever published by
the Free Software Foundation.
  14. If you wish to incorporate parts of the Library into other free
programs whose distribution conditions are incompatible with these,
write to the author to ask for permission.  For software which is
copyrighted by the Free Software Foundation, write to the Free
Software Foundation; we sometimes make exceptions for this.  Our
decision will be guided by the two goals of preserving the free status
of all derivatives of our free software and of promoting the sharing
and reuse of software generally.



    NO WARRANTY
  15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO
WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW.
```

```
EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR
OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY
KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE
LIBRARY IS WITH YOU.  SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME
THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
  16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN
WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY
AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU
FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR
CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE
LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING
RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A
FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF
SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGES.


     END OF TERMS AND CONDITIONS
           How to Apply These Terms to Your New Libraries
  If you develop a new library, and you want it to be of the greatest
possible use to the public, we recommend making it free software that
everyone can redistribute and change.  You can do so by permitting
redistribution under these terms (or, alternatively, under the terms of the
ordinary General Public License).
  To apply these terms, attach the following notices to the library.  It is
safest to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least the
"copyright" line and a pointer to where the full notice is found.
    <one line to give the library's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>
    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.
    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.
    You should have received a copy of the GNU Lesser General Public
    License along with this library; if not, write to the Free Software
    Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
Also add information on how to contact you by electronic and paper mail.
You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the library, if
necessary.  Here is a sample; alter the names:
  Yoyodyne, Inc., hereby disclaims all copyright interest in the
  library `Frob' (a library for tweaking knobs) written by James Random Hacker.
  <signature of Ty Coon>, 1 April 1990
  Ty Coon, President of Vice
That's all there is to it!
introduced by others.
```

p

Symbols

A

C

D

E

F

G

I

L

M

N

P

R

S

T

U

W