
LIRC Python Package

Release 1.0.1

Apr 15, 2021

1	Installation	3
2	Usage Quick Start	5
2.1	Customizing the Client	5
2.2	Sending IR	6
2.3	Handling Errors	6
3	Further Documentation	7
4	Site Documentation	9
4.1	Installing the LIRC Python & System Package	9
4.2	Hardware Setup	10
4.3	Configuring System LIRC	10
4.4	Usage	13
4.5	API Specification	14
4.6	Changelog	17
4.7	Contributor Covenant Code of Conduct	18
4.8	MIT License	20
4.9	Contributing Guidelines	20
5	Indices and tables	23
	Python Module Index	25
	Index	27

This is a python package that allows you to interact with the daemon in the [Linux Infrared Remote Control](#) package. By interacting with the daemon, it allows you to programmatically send IR signals from a computer. This package is for emitting IR signals, but it does not support listening to IR codes.

More information on the lircd daemon, socket interface, reply packet format, etc. can be found at <https://www.lirc.org/html/lircd.html>

CHAPTER 1

Installation

This package is hosted on PyPI and can be installed through pip.

```
$ pip install lirc
```

However since this is a wrapper around the LIRC daemon, it is expected that LIRC is installed and setup on the given system as well.

More information on that can be found in the [installation](#) portion of the documentation.

2.1 Customizing the Client

```
import lirc

client = lirc.Client()

print(client.version())
>>> '0.10.1'
```

To use this package, we instantiate a `Client`. When we initialize the `Client` in the example above, we use the defaults by passing it no arguments.

The defaults use the `LircdConnection` class. These defaults depend on your operating system and can be looked up in the full documentation. However, we can customize these defaults if desired.

```
import socket
import lirc

client = lirc.Client(
    connection=lirc.LircdConnection(
        address="/var/run/lirc/lircd",
        socket=socket.socket(socket.AF_UNIX, socket.SOCK_STREAM),
        timeout = 5.0
    )
)
```

The address specifies how to reach the `lircd` daemon. On Windows, we pass a `(hostname, port)` tuple since we connect over TCP. However on Linux and macOS, we pass in the socket path as a string. For the client in the example above, we set it up using the defaults for a Linux machine. While it illustrates what is customizable, it is not a practical example since you could just call `Client()` if you're on Linux and achieve the same outcome.

2.2 Sending IR

```
import lirc

client = lirc.Client()
client.send_once("my-remote-name", "KEY_POWER")
client.send_once("my-remote-name", "KEY_3", repeat_count=2)
```

With sending IR, we can use the *send_once* method and optionally, send multiple by using the *repeat_count* keyword argument.

2.3 Handling Errors

```
import lirc

client = lirc.Client()

try:
    client.send_once('some-remote', 'key_power')
except lirc.LircdCommandFailureError as error:
    print('The command we sent failed! Check the error message')
    print(error)
```

If the command was not successful, a `LircdCommandFailureError` exception will be thrown. There are other errors that may be raised, which can be looked up in the full documentation, but this is the most likely when sending commands.

CHAPTER 3

Further Documentation

More information on how to setup the system installed LIRC, how to use this python library, and a full API specification can be found at <https://lirc.readthedocs.io/>

4.1 Installing the LIRC Python & System Package

Since this package is merely a wrapper around the LIRC daemon, it is expected that LIRC is installed and setup on the given system as well to be able to use the python package.

4.1.1 Python Package

This package is hosted on PyPI and can be installed through pip.

```
$ pip install lirc
```

4.1.2 System LIRC Package

While LIRC was originally created for Linux, there are ports of LIRC to macOS and Windows which this python package is compatible with.

Linux:

- It is highly likely that the package manager on your system already has LIRC packaged up and ready to be installed for you. e.g. `sudo apt install lirc` on Ubuntu.
- If not, you may have to [compile and install](#) it manually, but I would avoid that if possible.

Windows:

- WinLIRC at <http://winlirc.sourceforge.net/> is a port for Windows. It works a bit differently since it is just a collection of files in a folder that you run. More information on that can be found at [using LIRC on Windows](#).

macOS:

- There is a port on MacPorts at <https://ports.macports.org/port/lirc/summary> with it's source code on GitHub at <https://github.com/andyvand/LIRC>. However, it doesn't appear to be maintained any longer and is not the

latest LIRC version. You can then run `port install lirc` or build the package from source using the instructions on the README of the GitHub repository.

4.2 Hardware Setup

In order to use this package, you'll need to have an IR emitter or transceiver hooked up to your computer.

4.3 Configuring System LIRC

Once you have your IR emitter or transceiver hooked up to your computer, you'll want to configure the system installed LIRC to ensure it works for emitting IR.

You'll also want to ensure you have a configuration file for the remote control that you want to emulate when emitting IR since whatever you're sending IR to will likely only understand IR codes from certain remotes.

This process will be different depending on the operating system you are using. Below are instructions for Linux, Windows, and macOS. See the [LIRC configuration guide](#) for more information. However, WinLIRC will be a bit different and you should read their own resources if you are on Windows as well.

4.3.1 Linux

LIRC configuration is typically in `/etc/lirc/`. The two things you'll have to figure out on your own is the `lirc_options.conf` file and adding your remote configuration file as these are dependent on the hardware you use for your setup. However, I can give general recommendations or what I typically do.

For `lirc_options.conf`, the only change I make is to change the driver from `devinput` to `default`. `Devinput` works fine for receiving IR, but it will not allow you to emit IR. This driver is dependent on your hardware, but LIRC just works with most devices on this driver nowadays.

For the remote configuration file, if you're using a common remote control, you may be able to find it in the [LIRC remote control database](#). Otherwise, you'll have to create it yourself. This can be done with [LIRC's IR record utility](#). However, I've had much better luck using a [RedRat3-II](#) and RedRat's [IR Signal Database](#) for creating the remote configuration file. The RedRat3-II is now discontinued, although [its driver's are still available](#), but you could look into the [RedRatX](#) or see if you can find a [RedRat3-II used](#). Place this generated remote configuration file in your `lircd.conf.d` folder.

If you're using an Iguanaworks IR Transceiver, you may find the discussion below useful. Basically, the device should just work on the default driver.

- <https://github.com/iguanaworks/iguanair/issues/39>

4.3.2 Windows

You'll want to make sure you install WinLIRC at <http://winlirc.sourceforge.net/>. This is the LIRC port for Windows which corresponds with version 0.9.0 of LIRC. Past that, you can run the WinLIRC executable file and select the "Input Plugin" for your device. Then, you can select the remote configuration and click OK. You should now be able to select your remote and send key codes. As long as the program is running in the background (it minimizes to the tray), this package will be able to connect to it.

4.3.3 macOS

On macOS, the paths are almost the same as the Linux ones, just prefixed with `/opt/local/`. Therefore, the LIRC configuration is typically at `/opt/local/etc/lirc/` and the lircd socket is at `/opt/local/var/run/lirc/lircd`

Refer to the Linux section for the rest of the configuration as they are almost the same besides the `/opt/local/` prefix. However, on macOS, there is also no default driver like there is on Linux. You'll have to figure out what devices will work and what driver it needs so you can input that into `lirc_options.conf`.

4.3.4 Example Remote Configuration File

The following is an example of a remote configuration file that would be placed inside of the `lircd.conf.d/` folder. This is for a [KENMORE_253-79081](#), remote taken from the [LIRC remote database](#).

```
# Please make this file available to others
# by sending it to <lirc@bartelmus.de>
#
# this config file was automatically generated
# using lirc-0.9.0-pre1(default) on Sun Sep  7 00:53:46 2014
#
# contributed by Steven Shamlian
#
# brand: Kenmore
# model no. of remote control: Unknown
# devices being controlled by this remote: Kenmore 253.79081
#
# Kernel revision: 3.12.26+
# Driver/device option: --driver default --device /dev/lirc0
# Capture device: Vishay TSOP6238 to Raspberry Pi GPIO pin 23
# Kernel modules: lirc_rpi
# Type of device controlled: Air Conditioner
# Devices controlled: Kenmore 253.79081
#
# Remote Layout:
#
# /-----\
# |KEY_POWER      KEY_TIME|
# |               |
# |KEY_VOLUMEUP   KEY_UP  |
# |       KEY_PLAY       |
# |KEY_VOLUMEDOWN KEY_DOWN|
# |       KEY_SAVE       |
# |KEY_SHUFFLE     KEY_SLEEP|
# |       KEY_PAUSE      |
# \-----/
# VOLUME keys are for fan speed
# PLAY starts air conditioner
# PAUSE makes unit fan-only
# SAVE is Energy Saver mode
# SHUFFLE is for Automatic Fan

begin remote

    name    KENMORE_253-79081
    bits    16
```

(continues on next page)

(continued from previous page)

```

flags SPACE_ENC|CONST_LENGTH
eps      30
aeps     100

header    9159  4455
one       639  1615
zero      639   486
ptrail    637
repeat    9103  2199
pre_data_bits  16
pre_data   0x10AF
gap       108066
toggle_bit_mask 0x0

begin codes
    KEY_POWER      0x8877
    KEY_TIME       0x609F
    KEY_VOLUMEUP   0x807F
    KEY_VOLUMEDOWN 0x20DF
    KEY_PLAY       0x906F
    KEY_UP         0x708F
    KEY_DOWN       0xB04F
    KEY_SAVE       0x40BF
    KEY_SHUFFLE    0xF00F
    KEY_SLEEP      0x00FF
    KEY_PAUSE      0xE01F
end codes

end remote

```

4.3.5 Example LIRC Options Configuration File

This is a `lirc_options.conf` file, taken from `/etc/lirc/lirc_options.conf` on a Linux machine, to get a feel for the configuration options offered.

```

# These are the default options to lircd, if installed as
# /etc/lirc/lirc_options.conf. See the lircd(8) and lircmd(8)
# manpages for info on the different options.
#
# Some tools including mode2 and irw uses values such as
# driver, device, plugindir and loglevel as fallback values
# in not defined elsewhere.

[lircd]
nodaemon      = False
driver        = default
device        = auto
output        = /var/run/lirc/lircd
pidfile       = /var/run/lirc/lircd.pid
plugindir     = /usr/lib/lirc/plugins
permission    = 666
allow-simulate = No
repeat-max    = 600
#effective-user =
#listen       = [address:]port

```

(continues on next page)

(continued from previous page)

```
#connect          = host[:port]
#loglevel         = 6
#release         = true
#release_suffix  = _EVUP
#logfile         = ...
#driver-options  = ...

[lircmd]
uinput          = False
nodaemon        = False

# [modinit]
# code = /usr/sbin/modprobe lirc_serial
# code1 = /usr/bin/setfacl -m g:lirc:rw /dev/uinput
# code2 = ...

# [lircd-uinput]
# add-release-events = False
# release-timeout    = 200
# release-suffix     = _EVUP
```

4.4 Usage

Once you've installed the `lirc` python package, there will be a number of things you can now import from it to get started.

```
from lirc import Client, LircdConnection
```

The most relevant of these is `Client`, since this is the main class you will be using. `LircdConnection` is the object that is used to configure the connection to LIRC when you initialize the `Client`.

4.4.1 Initializing Lirc

The `Client` class takes in one optional keyword argument: `connection`. This connection must be a `LircdConnection`. This connection, if not specified manually, will have default values that depend on the operating system you are on. So the simplest way to construct `Client` is with no arguments at all.

```
from lirc import Client

lirc_client = Client()
```

Overriding LIRC Defaults on Initialization

However, if the defaults don't work for us? Let's say we're on Windows and we want to connect over TCP to a remote LIRC server on another Windows machine. So we've passed in an address to override the default so it doesn't look for the daemon on the localhost. `socket` and `timeout` are passed in just to show that we can, these are already the defaults on Windows.

```
import socket
from lirc import Client, LircdConnection

client = Client(
    connection=LircdConnection(
        address=("10.16.30.2", 8765),
        socket=socket.socket(socket.AF_INET, socket.SOCK_STREAM),
        timeout=5.0
    )
)
```

LIRC Initialization Defaults per Operating System

From the options we may pass into the `LircdConnection`, `address` and `socket` will change depending on the operating system you are using. The `timeout` always defaults to 5.0 (seconds).

On Linux, this will attempt to connect to the lircd socket at `/var/run/lirc/lircd` and create a socket using `AF_UNIX` and `SOCK_STREAM`.

On macOS, it will be almost identical to Linux except that all the paths will be prefixed by `/opt/local/` so the connection to the lircd socket will instead be at `/opt/local/var/run/lirc/lircd`. The socket that is created will be the same.

However if we are on Windows, we can't use unix domain sockets. Instead, WinLIRC uses TCP to communicate with the lirc daemon. So instead of a string for the address, it defaults to a tuple of ("localhost", 8765), which is the default on WinLIRC. The first part contains the address whereas the second is the port. Furthermore, the socket that is created uses `AF_INET` and `SOCK_STREAM` instead so we can connect over TCP.

4.4.2 Sending IR Codes

In order to send IR signals with our remote, we can use the `send_once` method on the `lirc.Client`.

```
import lirc

client = lirc.Client()
client.send_once('our-remote-name', 'key-in-the-remote-file')
```

Using the `send_once()` method is quite simple. For any method, such as this one, that takes in a remote and a key, the parameters are always in that order with the remote name first and then the key name. Because the `send_once` method does not get any meaningful data back from lircd, there is no return value from it. Instead, as is the case for most methods here that don't have a meaningful return value, a `lirc.LircdCommandFailureError` is raised if the command we sent failed.

Furthermore, we can also send the key in rapid succession. This is useful if we, say, want to go to channel 33.

```
client.send_once('our-remote-name', 'key_3', repeat_count=2)
```

4.5 API Specification

```
class lirc.Client(connection: lirc.connection.lircd_connection.LircdConnection = None)
```

Bases: object

Communicate with the lircd daemon.

close() → None

Close the connection to the socket.

driver_option(*key: str, value: str*) → None

Set driver-specific option named *key* to given value.

Parameters

- **key** – The key to set for the driver.
- **value** – The value for the key to set.

Raises `LircdCommandFailure` – If the command fails.

list_remote_keys(*remote: str*) → List[str]

List all the keys for a specific remote.

Parameters **remote** – The remote to list the keys of.

Raises `LircdCommandFailure` – If the command fails.

Returns The list of keys from the remote.

list_remotes() → List[str]

List all the remotes that lirc has in its *lircd.conf.d* folder.

Raises `LircdCommandFailure` – If the command fails.

Returns The list of all remotes.

send_once(*remote: str, key: str, repeat_count: int = 1*) → None

Send an lircd SEND_ONCE command.

Parameters

- **key** – The name of the key to send.
- **remote** – The remote to use keys from.
- **repeat_count** – The number of times to press this key.

Raises `LircdCommandFailure` – If the command fails.

send_start(*remote: str, key: str*) → None

Send an lircd SEND_START command.

This will repeat the given key until `send_stop` is called.

Parameters

- **remote** – The remote to use keys from.
- **key** – The name of the key to start sending.

Raises `LircdCommandFailure` – If the command fails.

send_stop(*remote: str = "", key: str = ""*) → None

Send an lircd SEND_STOP command.

The remote and key default to the remote and key last used with `send_start` if they are not specified, since the most likely use case is sending a `send_start` and then a `send_stop`.

Parameters

- **remote** – The remote to stop.
- **key** – The key to stop sending.

Raises `LircdCommandFailure` – If the command fails.

set_transmitters (*transmitters: Union[int, List[int]]*) → None
Set the active transmitters.

Example

```
import lirc
c = lirc.Client()
c.set_transmitters(1)
c.set_transmitters([1,3,5])
```

Parameters **transmitters** – The transmitters to set active.

Raises `LircdCommandFailure` – If the command fails.

simulate (*remote: str, key: str, repeat_count: int = 1, keycode: int = 0*) → None
Simulate an IR event.

The `--allow-simulate` command line option to `lircd` must be active for this command not to fail.

Example Format: `<code> <repeat count> <button name> <remote control name>`

Such as: `0000000000f40bf0 00 KEY_UP ANIMAX`

Parameters

- **remote** – The remote to simulate key presses from.
- **key** – The key on the remote to simulate.
- **repeat_count** – The number of times to simulate the key press.
- **keycode** – `lircd(8)` describes this option as a 16 hexadecimal digit number encoding of the IR signal. However, it says it is depreciated and should be ignored.

Raises `LircdCommandFailure` – If the command fails.

start_logging (*path: Union[str, pathlib.Path]*) → None
Send a `lircd SET_INPUTLOG` command which sets the path to log all `lircd` received data to.

Parameters **path** – The path to start logging `lircd` recieved data to.

Raises `LircdCommandFailure` – If the command fails.

stop_logging () → None
Stop logging to the `inputlog` path from `start_logging`.

Raises `LircdCommandFailure` – If the command fails.

version () → str
Retrieve the version of LIRC

Raises `LircdCommandFailure` – If the command fails.

Returns The version of LIRC being used.

class `lirc.LircdConnection` (*address: Union[str, tuple] = None, socket: socket.socket = None, timeout: float = 5.0*)
Bases: `lirc.connection.abstract_connection.AbstractConnection`

address

Retrieve the address that this `lircd` connection is connected to.

Returns The current address being used.

close()

Closes the socket connection.

connect()

Connect to the socket at the address both specified on init.

Raises `LircdConnectionError` – If the address is invalid or lircd is not running.

readline() → str

Read a line of data from the lircd socket.

We read 4096 bytes at a time as the buffer size. Therefore after data is read from the socket, all the lines are stored in a buffer if there is more than 1 and subsequent calls grab a line that stored in that buffer until it is empty. Then, another call to the socket would be made.

Raises

- `TimeoutError` – If we are not able to grab data from the socket in a specified amount of time (the initial timeout time on initialization).
- `LircdSocketError` – If some other error happened when trying to read from the socket.

Returns A line from the lircd socket.

send(data: str)

Send a command to the lircd socket connection.

Parameters **data** – The data to send to the lircd socket.

Raises `TypeError` – if data is not a string.

4.6 Changelog

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

4.6.1 1.0.1 - 2020-12-26

Fixed

- PyPI is complaining that v1.0.0 is already taken, since it was a release that was deleted from a previous mistake.

4.6.2 1.0.0 - 2020-12-26

Added

- `DefaultConnection.address` and `DefaultConnection.socket` may raises an `UnsupportedOperatingSystemError` if the operating system you're on is not MacOS, Linux, or Windows.

Changed

- `lirc.Client` raises a `TypeError` instead of a `ValueError` now if a connection is passed in that is not an instance of `LircdConnection`.
- `send` on `lirc.Client` is now called `send_once`.

- `start_repeat` on `lirc.Client` is now called `send_start`.
- `stop_repeat` on `lirc.Client` is now called `send_stop`.

Removed

- `socket` property from `LircdConnection`.

Fixed

- The `remote` and `key` optional arguments to the `lirc.Client`'s `stop_repeat` method were not overriding the last sent `remote` and `key`.

4.6.3 0.2.0 - 2020-12-13

Added

- `LircdConnection` to handle configuring the connection on `Client`.

Changed

- `Lirc` is now named `Client`.
- `Client` now takes in a `connection` as the optional argument to configure its connection. That connection must be a `LircdConnection` class if you would like to customize the connection. The `LircdConnection` takes in an `address`, `socket`, and `timeout` with optional keyword arguments. Anything not specified with use the defaults for that operating system.

Removed

- `DEFAULT_SOCKET_PATH` constant on `Client`. It no longer makes sense with cross-platform support.
- `ENCODING` constant on `Client`.
- `socket_path` and `socket_timeout` on the `Lirc` constructor.

4.6.4 0.1.0 - 2020-07-13

- Initial Release

4.7 Contributor Covenant Code of Conduct

4.7.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

4.7.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences

- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

4.7.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

4.7.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

4.7.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at eugenetriguba@gmail.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

4.7.6 Attribution

This Code of Conduct is adapted from the [Contributor Covenant](https://www.contributor-covenant.org/version/1/4/code-of-conduct.html), version 1.4, available at <https://www.contributor-covenant.org/version/1/4/code-of-conduct.html>

For answers to common questions about this code of conduct, see <https://www.contributor-covenant.org/faq>

4.8 MIT License

Copyright (c) 2020 Eugene Triguba

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

4.9 Contributing Guidelines

Thank you for your interest! If you find a bug, want to suggest an improvement, or any other change, please open a issue first. This ensures your time is not wasted if you were planning on creating a pull request, but the changes suggested do not work for this project.

4.9.1 General Guidelines

Commit history:

- Try to keep a clean commit history so it is easier to see your changes. Keep functional changes and refactorings in separate commits.

Commit messages:

- Have a short one line summary of your change followed by as many paragraphs of explanation as you need. This is the place to clarify any subtleties you have in your implementation, document other approaches you tried that didn’t end up working, any limitations on your implementation, etc. The most important part here is to describe why you made the change you did, not simply what the change you made is.

Changelog:

- Please ensure to update the changelog by adding a new bullet under an Added, Changed, Deprecated, Removed, Fixed, or Security section headers under the Unreleased version. If any of those sections are not present, feel free to add the one you need. See [Keep a Changelog](#) if you need guidance on what makes a good entry since this project follows those principles.

Tests:

- Ensure the tests pass: `poetry run task test` to run all tests.
- For any significant code changes, there must be tests to accompany them. All unit tests are written with `pytest`.

Code Format:

- There is a pre-commit pipeline to ensure a standard code format. Make sure to install the pre-commit hooks before making any commits with `pre-commit install`.

CI Pipeline:

- There is a CI pipeline that is run using Github Actions on commits to master, dev, and on pull requests. This pipeline must pass for your changes to be accepted.

4.9.2 Getting Up & Running

This project uses [Poetry](#) for the build system and dependency management. To get started, you will want that installed on your system.

Once you've installed Poetry, you can install the dependencies, this package, and go into the virtual environment.

```
$ poetry install
$ poetry shell
```

From inside the virtual environment, we can work with the package and easily run the tasks for this project such as `task test` and `task lint` that are in the `pyproject.toml` file.

CHAPTER 5

Indices and tables

- `genindex`
- `search`

I

`lirc`, [14](#)

A

`address` (*lirc.LircdConnection attribute*), 16

C

`Client` (*class in lirc*), 14

`close()` (*lirc.Client method*), 14

`close()` (*lirc.LircdConnection method*), 17

`connect()` (*lirc.LircdConnection method*), 17

D

`driver_option()` (*lirc.Client method*), 15

L

`lirc` (*module*), 14

`LircdConnection` (*class in lirc*), 16

`list_remote_keys()` (*lirc.Client method*), 15

`list_remotes()` (*lirc.Client method*), 15

R

`readline()` (*lirc.LircdConnection method*), 17

S

`send()` (*lirc.LircdConnection method*), 17

`send_once()` (*lirc.Client method*), 15

`send_start()` (*lirc.Client method*), 15

`send_stop()` (*lirc.Client method*), 15

`set_transmitters()` (*lirc.Client method*), 15

`simulate()` (*lirc.Client method*), 16

`start_logging()` (*lirc.Client method*), 16

`stop_logging()` (*lirc.Client method*), 16

V

`version()` (*lirc.Client method*), 16