# pydle Documentation

***Release 0.8.5***

**Shiz**

**Aug 15, 2019**

# Contents

pydle is a compact, flexible and standards-abiding IRC library for Python 3, written out of frustration with existing solutions.

# CHAPTER 1

# Features

- **Well-organized, easily extensible**

  Thanks to the modular setup, pydle's functionality is seperated in modules according to the standard they were defined in. This makes specific functionality trivial to find and decreases unwanted coupling, as well as allowing users to pick-and-choose the functionality they need.

  No spaghetti code.

- **Compliant**

  pydle contains modules, or "features" in pydle terminology, for almost every relevant IRC standard:

  - RFC1459: The standard that defines the basic functionality of IRC - no client could live without.
  - TLS: Support for chatting securely using TLS encryption.
  - CTCP: The IRC Client-to-Client Protocol, allowing clients to query eachother for data.
  - ISUPPORT: A method for the server to indicate non-standard or extended functionality to a client, and for clients to activate said functionality if needed.
  - WHOX: Easily query status information for a lot of users at once.
  - IRCv3.1: An ongoing effort to bring the IRC protocol to the twenty-first century, featuring enhancements such as extended capability negotiation and SASL authentication.
  - IRCv3.2 *(in progress)*: The next, in-development iteration of IRCv3. Features among others advanced message tagging, a generalized metadata system, and online status monitoring.

  No half-assing functionality.

- **Asynchronous**

  IRC is an asychronous protocol; it only makes sense a clients that implements it is asynchronous as well. Built on top of the wonderful asyncio library, pydle relies on proven technologies to deliver proper high-performance asynchronous functionality and primitives. pydle allows using Futures to make asynchronous programming just as intuitive as doing regular blocking operations.

  No callback spaghetti.

- **Liberally licensed**

  The 3-clause BSD license ensures you can use pydle whenever, for what purpose you want.

  No arbitrary restrictions.

CHAPTER 2

Contents

## 2.1 Introduction to pydle

### 2.1.1 What is pydle?

pydle is an IRC library for Python 3.4 and up.

Although old and dated on some fronts, IRC is still used by a variety of communities as the real-time communication method of choice, and the most popular IRC networks can still count on tens of thousands of users at any point during the day.

pydle was created out of perceived lack of a good, Pythonic, IRC library solution that also worked with Python 3. It attempts to follow the standards properly, while also having functionality for the various extensions to the protocol that have been made over the many years.

### 2.1.2 What isn't pydle?

pydle is not an end-user IRC client. Although a simple client may be trivial to implement using pydle, pydle itself does not seek out to be a full-fledged client. It does, however, provide the building blocks to which you can delegate most, if not all, of your IRC protocol headaches.

pydle also isn't production-ready: while the maintainers try their utmost best to keep the API stable, pydle is still in heavy development, and APIs are prone to change or removal at least until version 1.0 has been reached.

### 2.1.3 Requirements

Most of pydle is written in pure, portable Python that only relies on the standard library. Optionally, if you plan to use pydle's SASL functionality for authentication, the excellent pure-sasl library is required.

All dependencies can be installed using the standard package manager for Python, pip, and the included requirements file:

```
pip install -r requirements.txt
```

### 2.1.4 Compatibility

pydle works in any interpreter that implements Python 3.2 or higher. Although mainly tested in CPython, the standard Python implementation, there is no reason why pydle itself should not work in alternative implementations like PyPy, as long as they support the Python 3.2 language requirements.

## 2.2 Using pydle

**Note:** This section covers basic use of pydle. To see the full spectrum of what pydle is capable of, refer to the *API reference*.

### 2.2.1 A simple client

The most basic way to use pydle is instantiating a `pydle.Client` object, connecting it to a server and having it handle messages forever using `pydle.Client.handle_forever()`. pydle will automatically take care of ensuring that the connection persists, and will reconnect if for some reason disconnected unexpectedly.

```
import pydle

client = pydle.Client('MyBot')
# Client.connect() is a blocking function.
client.connect('irc.freenode.net', tls=True)
client.handle_forever()
```

### 2.2.2 Adding functionality

Of course, the above client doesn't really do much, except idling and error recovery. To truly start adding functionality to the client, subclass `pydle.Client` and override one or more of the IRC callbacks.

```
import pydle

class MyClient(pydle.Client):
    """ This is a simple bot that will greet people as they join the channel. """

    def on_connect(self):
        super().on_connect()
        # Can't greet many people without joining a channel.
        self.join('#kochira')

    def on_join(self, channel, user):
        super().on_join(channel, user)
        self.message(channel, 'Hey there, {user}!', user=user)

client = MyClient('MyBot')
client.connect('irc.freenode.net', tls=True)
client.handle_forever()
```

This trivial example shows a few things:

1. `pydle.Client.on_connect()` is a callback that gets invoked as soon as the client is fully connected to the server.
2. `pydle.Client.on_join()` is a callback that gets invoked whenever a user joins a channel.
3. Trivially enough, we can use `pydle.Client.join()` to instruct the client to join a channel.
4. Finally, we can use `pydle.Client.message()` to send a message to a channel or to a user; it will even format the message for us according to advanced string formatting.

**Hint:** It is recommended to call the callbacks of the parent class using `super()`, to make sure whatever functionality implemented by your parent classes gets called too: pydle will gracefully handle the call even if no functionality was implemented or no callbacks overridden.

### 2.2.3 Multiple servers, multiple clients

Any pydle client instance can only be connected to a single server. That doesn't mean that you are restricted to only being active on a single server at once, though. Using a *pydle.ClientPool*, you can instantiate multiple clients, connect them to different servers using *pydle.ClientPool.connect()*, and handle them within a single loop.

```
import pydle

class MyClient(pydle.Client):
    """ This is a simple bot that will greet people as they join the channel. """

    def on_connect(self):
        super().on_connect()
        # Can't greet many people without joining a channel.
        self.join('#kochira')

    def on_join(self, channel, user):
        super().on_join(channel, user)
        self.message(channel, 'Hey there, {user}!', user=user)

# Setup pool and connect clients.
pool = pydle.ClientPool()
servers = [ 'irc.freenode.net', 'irc.rizon.net', 'irc.esper.net' ]

for server in servers:
    client = MyClient('MyBot')
    pool.connect(client, server, tls=True)

# Handle all clients in the pool at once.
pool.handle_forever()
```

**Warning:** While multiple *pydle.ClientPool* instances can be created and ran, you should ensure a client is only active in a single *pydle.ClientPool* at once. Being active in multiple pools can lead to strange things like receiving messages twice, or interleaved outgoing messages.

### 2.2.4 Mixing and matching

Thanks to pydle's modular "feature" system, you don't have to support everything you want to support. You can choose just to select the options you think you need for your client by using *pydle.featurize()* to create a base class out of the featured you need.

```
import pydle

# Create a client that just supports the base RFC1459 spec, CTCP and an IRC services-
↪style account system.
MyBaseClient = pydle.featurize(pydle.features.RFC1459Support, pydle.features.
↪CTCPSupport, pydle.features.AccountSupport)

class MyClient(MyBaseClient):
    ...
```

A list of all available built-in features and their use can be found at the *API reference*.

In addition to this, you can of course also write your own features. Feature writing is discussed thoroughly in the *feature section*. Once you have written a feature, you can just featurize it on top of an existing client class.

```
import pydle
import vendor

# Add vendor feature on top of the base client.
MyBaseClient = pydle.featurize(pydle.Client, vendor.VendorExtensionSupport)

class MyClient(MyBaseClient):
    ...
```

### 2.2.5 Asynchronous functionality

Some actions inevitably require blocking and waiting for a result. Since pydle is an asynchronous library where a client runs in a single thread, doing this blindly could lead to issues like the operation blocking the handling of messages entirely.

Fortunately, pydle implements coroutines which allow you to handle a blocking operation almost as if it were a regular operation, while still retaining the benefits of asynchronous program flow. Coroutines allow pydle to be notified when a blocking operation is done, and then resume execution of the calling function appropriately. That way, blocking operations do not block the entire program flow,

In order for a function to be declared as a coroutine, it has to be decorated using the *pydle.coroutine()* decorator. It can then call functions that would normally block using Python's `yield from` operator. Since a function that calls a blocking function is itself blocking too, it has to be declared a coroutine as well.

---

**Hint:** As with a lot of things, documentation is key. Documenting that your function does blocking operations lets the caller know how to call the function, and to include the fact that it calls blocking operations in its own documentation for its own callers.

---

For example, if you are implementing an administrative system that works based off nicknames, you might want to check if the users are identified to `NickServ`. However, WHOISing a user using `pydle.Client.whois()` would be a blocking operation. Thanks to coroutines and `pydle.Client.whois()` being a blocking operation compatible with coroutines, the act of WHOISing will not block the entire program flow of the client.

```
import pydle
ADMIN_NICKNAMES = [ 'Shiz', 'rfw' ]

class MyClient(pydle.Client):
    """
    This is a simple bot that will tell you if you're an administrator or not.
    A real bot with administrative-like capabilities would probably be better off␣
↪maintaining a cache
    that would be invalidated upon parting, quitting or changing nicknames.
    """

    def on_connect(self):
        super().on_connect()
        self.join('#kochira')

    @pydle.coroutine
    def is_admin(self, nickname):
        """
        Check whether or not a user has administrative rights for this bot.
        This is a blocking function: use a coroutine to call it.
        See pydle's documentation on blocking functionality for details.
        """
        admin = False

        # Check the WHOIS info to see if the source has identified with NickServ.
        # This is a blocking operation, so use yield.
        if source in ADMIN_NICKNAMES:
            info = yield from self.whois(source)
            admin = info['identified']

        return admin

    @pydle.coroutine
    def on_message(self, target, source, message):
        super().on_message(target, source, message)

        # Tell a user if they are an administrator for this bot.
        if message.startswith('!adminstatus'):
            admin = yield from self.is_admin(source)

            if admin:
                self.message(target, '{source}: You are an administrator.',␣
↪source=source)
            else:
                self.message(target, '{source}: You are not an administrator.',␣
↪source=source)
```

Writing your own blocking operation that can work with coroutines is trivial: just make your blocking method return a *pydle.Future* instance (without the act of creating and returning the Future itself being blocking), and any coroutine yielding it will halt execution until the returned future is resolved, using either *pydle.Future.set_result()* or *pydle.Future.set_exception()*, while pydle can still handle everything else.

You can create a *pydle.Future* instance that belongs to the client by calling `pydle.async.EventLoop.create_future()`.

## 2.3 Features

pydle's main IRC functionality is divided into separate modular components called "features". These features allow you to mix and match your client to fit exactly to your requirements, as well as provide an easy way to extend pydle yourself, without having to dive into the source code.

### 2.3.1 Built-in features

The following features are packaged with pydle and live in the *pydle.features* namespace.

#### RFC1459

*API:* *pydle.features.RFC1459Support*

RFC1459 is the bread and butter of IRC: it is the standard that defines the very base concepts of the IRC protocol, ranging from what a channel is to the basic commands to channel limits. If you want your client to have actually any useful IRC functionality, it is recommend to include this feature.

#### Transport Layer Security (TLS)

*API:* *pydle.features.TLSSupport*

Support for secure connections to the IRC server using Transport Layer Security.

This allows, if the server supports it, for encrypted connections between the server and the client, to prevent snooping and provide two-way authentication: both for the server to ensure its identity to the client, and for the client to ensure its identity to the server. The latter can also be used in certain service packages to automatically identify to the user account.

In order to connect to a TLS-enabled server, supply `tls=True` to *pydle.features.TLSSupport.connect()*.

---

**Hint:** pydle does not verify server-side TLS certificates by default; to enable certificate verification, supply `tls_verify=True` to *pydle.features.TLSSupport.connect()* as well.

---

In order to supply a client certificate, *pydle.features.TLSSupport* takes 3 additional constructor parameters:

- `tls_client_cert`: A path to the TLS client certificate.
- `tls_client_cert_key`: A path to the TLS client certificate key.
- `tls_client_cert_password`: The optional password for the certificate key.

#### Client-to-Client Protocol (CTCP)

*API:* *pydle.features.CTCPSupport*

Support for encapsulation of out-of-band features into standard IRC messages using the Client-to-Client Protocol.

This allows you to send meta-messages to other users, requesting e.g. their local time, client version, and more, and respond to such requests. It adds *pydle.Client.ctcp(target, query, contents=None)*, which allows you to send a CTCP request to a target, and *pydle.Client.ctcp_reply(target, query, contents=None)*, which allows you to respond to CTCP requests.

In addition, it registers the *pydle.Client.on_ctcp(from, query, contents)* hook, which allows you to act upon *any* CTCP request, and a per-type hook in the form of *pydle.Client.on_ctcp_<type>(from, contents)*, which allows you to act upon CTCP requests of type *type*. *type* will always be lowercased. A few examples of *type* can be: *action*, *time*, *version*.

Finally, it registers the *pydle.Client.on_ctcp_reply(from, queyr, contents)* hook, which acts similar to the above hook, except it is triggered when the client receives a CTCP response. It also registers *pydle.Client.on_ctcp_<type>_reply*, which works similar to the per-type hook described above.

### Server-side Extension Support (ISUPPORT)

*API:* `pydle.features.ISUPPORTSupport`

Support for IRC protocol extensions using the ISUPPORT message.

This feature allows pydle to support protocol extensions which are defined using the non-standard *ISUPPORT* (005) message. It includes built-in support for a number of popular *ISUPPORT*-based extensions, like *CASEMAPPING*, *CHANMODES*, *NETWORK* and *PREFIX*.

It also provides the generic *pydle.Client.on_isupport_type(value)* hook, where *type* is the type of *ISUPPORT*-based extension that the server indicated support for, and *value* is the optional value of said extension, or *None* if no value was present.

### Account System

*API:* `pydle.features.AccountSupport`

Support for a generic IRC account system.

Most IRC networks have some kind of account system that allows users to register and manage their nicknames and personas. This feature provides additional support in pydle for this idea and its integration into the networks.

Currently, all it does is set the *identified* and *account* fields when doing a *WHOIS* query (*pydle.Client.whois(user)*) on someone, which indicate if the target user has identified to their account, and if such, their account name, if available.

### Extended User Tracking

*API:* `pydle.features.WHOXSupport`

Support for better user tracking using WHOX.

This feature allows pydle to perform more accurate tracking of usernames, idents and account names, using the WHOX IRC extension. This allows pydle's internal user database to be more accurate and up-to-date.

### IRCv3

*API:* `pydle.features.IRCv3Support`

A shortcut for IRCv3.1 and IRCv3.2 support; see below.

### IRCv3.1

*API:* `pydle.features.IRCv3_1Support`

IRCv3.1 support.

The **'IRCv3 Working Group'_** is a working group organized by several network, server author, and client author representatives with the intention to standardize current non-standard IRC practices better, and modernize certain parts of the IRC protocol. The IRCv3 standards are specified as a bunch of extension specifications on top of the last widely-used IRC version, IRC v2.7, also known as RFC1459.

The IRCv3.1 specification adds useful features to IRC from a client perspective, including SASL authentication, support for indicating when a user identified to their account, and indicating when a user went away from their PC.

Including this feature entirely will activate all IRCv3.1 functionality for pydle. You can also opt-in to only select the two major features of IRCv3.1, the capability negotiation framework and SASL authentication support, as described below, by only including their features.

### Capability Negotiation Support

*API:* `pydle.features.ircv3.CapabilityNegotiationSupport`

Support for *capability negotiation* for IRC protocol extensions.

This feature enables support for a generic framework for negotiating IRC protocol extension support between the client and the server. It was quickly found that *ISUPPORT* alone wasn't sufficient, as it only advertises support from the server side instead of allowing the server and client to negotiate. This is a generic base feature: enabling it on its own won't do much, instead other features like the IRCv3.1 support feature, or the SASL authentication feature will rely on it to work.

This feature adds three generic hooks for feature authors whose features makes use of capability negotiation:

- **`pydle.Client.on_capability_<cap>_available(value)`: Called when the server indicates capability *cap* is a**
  Is passed a value as given by the IRC server, or *None* if no value was given Should return either a boolean indicating whether or not to request the capability, or a string indicating to request the capability with the returned value.
- **`pydle.Client.on_capability_<cap>_enabled()`: Called when the server has acknowledged the request of capa**
  has been enabled. Should return one of three values: *pydle.CAPABILITY_NEGOTIATING* when the capability will be further negotiated, *pydle.CAPABILITY_NEGOTIATED* when the capability has been negotiated successfully, or *pydle.CAPABILITY_FAILED* when negotiation of the capability has failed. If the function returned *pydle.CAPABILITY_NEGOTIATING*, it has to call *pydle.Client.capability_negotiated(cap, success=True)* when negotiating is finished.
- `pydle.Client.on_capability_<cap>_disabled()`: Called when a previously-enabled capability *cap* has been disabled.

### User Authentication Support (SASL)

*API:* `pydle.features.ircv3.SASLSupport`

Support for user authentication using SASL.

This feature enables users to identify to their network account using the SASL protocol and practices. Three extra arguments are added to the *pydle.Client* constructor:

- `sasl_username`: The SASL username.
- `sasl_password`: The SASL password.
- `sasl_identity`: The identity to use. Default, and most common, is `''`.
- `sasl_mechanism`: The SASL mechanism to force. Default involves auto-selection from server-supported mechanism, or a *PLAIN*' fallback.

These arguments are also set as attributes.

Currently, pydle's SASL support requires on the Python pure-sasl package and is thus limited to the mechanisms it supports. The `EXTERNAL` mechanism is also supported without, however.

### IRCv3.2

*API:* `pydle.features.IRCv3_2Support`

Support for the IRCv3.2 specification.

The IRCv3.2 specification is the second iteration of specifications from the **'IRCv3 Working Group'_**. This set of specification is still under development, and may change at any time. pydle's support is conservative, likely incomplete and to-be considered experimental.

pydle currently supports the following IRCv3.2 extensions:

- IRCv3.2 improved capability negotiation.
- Indication of changed ident/host using CHGHOST.
- Indication of *ident and host* in RFC1459's /NAMES command response.
- Monitoring of a user's online status using MONITOR.
- Message tags to add metadata to messages.
- Arbitrary key/value storage using METADATA.

As with the IRCv3.1 features, using this feature enables all of pydle's IRCv3.2 support. A user can also opt to only use individual large IRCv3.2 features by using the features below.

### Online Status Monitoring

*API:* `pydle.features.ircv3.MonitoringSupport`

Support for monitoring a user's online status.

This feature allows a client to monitor the online status of certain nicknames. It adds the *pydle.Client.monitor(nickname)* and *pydle.Client.unmonitor(nickname)* APIs to add and remove nicknames from the monitor list.

If a monitored user comes online, *pydle.Client.on_user_online(nickname)* will be called. Similarly, if a user disappears offline, *pydle.Client.on_user_offline(nickname)* will be called.

### Tagged Messages

*API:* `pydle.features.ircv3.TaggedMessageSupport`

Support for message metadata using tags.

This feature allows pydle to parse message metadata that is transmitted using 'tags'. Currently, this has no impact on any APIs or hooks for client developers.

#### Metadata

*API:* `pydle.features.ircv3.MetadataSupport`

Support for user and channel metadata.

This allows you to set and unset arbitrary key-value information on yourself and on channels, as well as retrieve such values from other users and channels.

### 2.3.2 Writing features

## 2.4 API reference

### 2.4.1 Client API

**class** pydle.**Client**
*pydle.Client* implements the featureset of *pydle.BasicClient* with all the features in the *pydle.features* namespace added. For the full reference, check the *pydle.BasicClient* documentation and the *Feature API reference*.

**class** pydle.**MinimalClient**
*pydle.MinimalClient* implements the featureset of *pydle.BasicClient* with some vital features in the *pydle.features* namespace added, namely:

- *pydle.features.RFC1459Support*
- *pydle.features.TLSSupport*
- *pydle.features.CTCPSupport*
- *pydle.features.ISUPPORTSupport*
- *pydle.features.WHOXSupport*

For the full reference, check the *pydle.BasicClient* documentation and the *Feature API reference*.

---

**class** pydle.**ClientPool**(*clients=None*, *eventloop=None*)
A pool of clients that are ran and handled in parallel.

**connect**(*client: pydle.client.BasicClient*, **args*, ***kwargs*)
Add client to pool.

**disconnect**(*client*)
Remove client from pool.

**handle_forever**()
Main loop of the pool: handle clients forever, until the event loop is stopped.

---

pydle.**featurize**(**features*)
Put features into proper MRO order.

**class** pydle.**BasicClient**(*nickname*, *fallback_nicknames=[]*, *username=None*, *realname=None*, *eventloop=None*, ***kwargs*)
Base IRC client class. This class on its own is not complete: in order to be able to run properly, _has_message, _parse_message and _create_message have to be overloaded.

users

A `dict` mapping a username to a `dict` with general information about that user. Available keys in the information dict:

- `nickname`: The user's nickname.
- `username`: The user's reported username on their source device.
- `realname`: The user's reported real name (GECOS).
- `hostname`: The hostname where the user is connecting from.

`channels`

A `dict` mapping a joined channel name to a `dict` with information about that channel. Available keys in the information dict:

- `users`: A `set` of all users currently in the channel.

**connect**(*hostname=None*, *port=None*, *reconnect=False*, ***kwargs*)
Connect to IRC server.

**connected**
Whether or not we are connected.

**disconnect**(*expected=True*)
Disconnect from server.

**handle_forever**()
Handle data forever.

**in_channel**(*channel*)
Check if we are currently in the given channel.

**is_channel**(*chan*)
Check if given argument is a channel name or not.

**is_same_channel**(*left*, *right*)
Check if given channel names are equal.

**is_same_nick**(*left*, *right*)
Check if given nicknames are equal.

**on_connect**()
Callback called when the client has connected successfully.

**on_raw**(*message*)
Handle a single message.

**on_unknown**(*message*)
Unknown command.

**raw**(*message*)
Send raw command.

**rawmsg**(*command*, **args*, ***kwargs*)
Send raw message.

**run**(**args*, ***kwargs*)
Connect and run bot in event loop.

### 2.4.2 Asynchronous API

pydle.**coroutine**(*func*)
Decorator to mark coroutines.

If the coroutine is not yielded from before it is destroyed, an error message is logged.

**class** pydle.**Future**
This class is *almost* compatible with concurrent.futures.Future.

Differences:

- result() and exception() do not take a timeout argument and raise an exception when the future isn't done yet.
- Callbacks registered with add_done_callback() are always called via the event loop's call_soon_threadsafe().
- This class is not compatible with the wait() and as_completed() methods in the concurrent.futures package.

**done**()
Return True if the future is done.

Done means either that a result / exception are available, or that the future was cancelled.

**result**()
Return the result this future represents.

If the future has been cancelled, raises CancelledError. If the future's result isn't yet available, raises InvalidStateError. If the future is done and has an exception set, this exception is raised.

**set_result**()
Mark the future done and set its result.

If the future is already done when this method is called, raises InvalidStateError.

**exception**()
Return the exception that was set on this future.

The exception (or None if no exception was set) is returned only if the future is done. If the future has been cancelled, raises CancelledError. If the future isn't done yet, raises InvalidStateError.

**set_exception**()
Mark the future done and set an exception.

If the future is already done when this method is called, raises InvalidStateError.

### 2.4.3 Features API

#### RFC1459

**class** pydle.features.**RFC1459Support**(*nickname*, *fallback_nicknames=[]*, *username=None*, *realname=None*, *eventloop=None*, ***kwargs*)
Basic RFC1459 client.

**away**(*message*)
Mark self as away.

**back**()
Mark self as not away.

**ban**(*channel*, *target*, *range=0*)
Ban user from channel. Target can be either a user or a host. This command will not kick: use kickban() for that. range indicates the IP/host range to ban: 0 means ban only the IP/host, 1+ means ban that many 'degrees' (up to 3 for IP addresses) of the host for range bans.

**connect**(*hostname=None*, *port=None*, *password=None*, ***kwargs*)
Connect to IRC server.

**cycle**(*channel*)
Rejoin channel.

**is_channel**(*chan*)
Check if given argument is a channel name or not.

**is_same_channel**(*left*, *right*)
Check if given nicknames are equal in the server's case mapping.

**is_same_nick**(*left*, *right*)
Check if given nicknames are equal in the server's case mapping.

**join**(*channel*, *password=None*)
Join channel, optionally with password.

**kick**(*channel*, *target*, *reason=None*)
Kick user from channel.

**kickban**(*channel*, *target*, *reason=None*, *range=0*)
Kick and ban user from channel.

**message**(*target*, *message*)
Message channel or user.

**notice**(*target*, *message*)
Notice channel or user.

**on_channel_message**(*target*, *by*, *message*)
Callback received when the client received a message in a channel.

**on_channel_notice**(*target*, *by*, *message*)
Callback called when the client received a notice in a channel.

**on_connect**()
Callback called when the client has connected successfully.

**on_invite**(*channel*, *by*)
Callback called when the client was invited into a channel by someone.

**on_join**(*channel*, *user*)
Callback called when a user, possibly the client, has joined the channel.

**on_kick**(*channel*, *target*, *by*, *reason=None*)
Callback called when a user, possibly the client, was kicked from a channel.

**on_kill**(*target*, *by*, *reason*)
Callback called when a user, possibly the client, was killed from the server.

**on_message**(*target*, *by*, *message*)
Callback called when the client received a message.

**on_mode_change**(*channel*, *modes*, *by*)
Callback called when the mode on a channel was changed.

**on_nick_change**(*old*, *new*)
Callback called when a user, possibly the client, changed their nickname.

**on_notice**(*target*, *by*, *message*)
Callback called when the client received a notice.

**on_part**(*channel*, *user*, *message=None*)
Callback called when a user, possibly the client, left a channel.

**on_private_message** (*target*, *by*, *message*)
Callback called when the client received a message in private.

**on_private_notice** (*target*, *by*, *message*)
Callback called when the client received a notice in private.

**on_quit** (*user*, *message=None*)
Callback called when a user, possibly the client, left the network.

**on_topic_change** (*channel*, *message*, *by*)
Callback called when the topic for a channel was changed.

**on_user_invite** (*target*, *channel*, *by*)
Callback called when another user was invited into a channel by someone.

**on_user_mode_change** (*modes*)
Callback called when a user mode change occurred for the client.

**part** (*channel*, *message=None*)
Leave channel, optionally with message.

**quit** (*message=None*)
Quit network.

**set_mode** (*target*, **modes*)
Set mode on target. Users should only rely on the mode actually being changed when receiving an on_{channel,user}_mode_change callback.

**set_nickname** (*nickname*)
Set nickname to given nickname. Users should only rely on the nickname actually being changed when receiving an on_nick_change callback.

**set_topic** (*channel*, *topic*)
Set topic on channel. Users should only rely on the topic actually being changed when receiving an on_topic_change callback.

**unban** (*channel*, *target*, *range=0*)
Unban user from channel. Target can be either a user or a host. See ban documentation for the range parameter.

**whois** (*nickname*)
Return information about user. This is an blocking asynchronous method: it has to be called from a coroutine, as follows:

info = await self.whois('Nick')

**whowas** (*nickname*)
Return information about offline user. This is an blocking asynchronous method: it has to be called from a coroutine, as follows:

info = await self.whowas('Nick')

---

### Transport Layer Security

**class** pydle.features.**TLSSupport** (**args*, *tls_client_cert=None*, *tls_client_cert_key=None*, *tls_client_cert_password=None*, ***kwargs*)
TLS support.

Pass tls_client_cert, tls_client_cert_key and optionally tls_client_cert_password to have pydle send a client certificate upon TLS connections.

**connect** (*hostname=None*, *port=None*, *tls=False*, ***kwargs*)
Connect to a server, optionally over TLS. See pydle.features.RFC1459Support.connect for misc parameters.

**whois** (*nickname*)
Return information about user. This is an blocking asynchronous method: it has to be called from a coroutine, as follows:

info = await self.whois('Nick')

---

### Client-to-Client Protocol

**class** pydle.features.**CTCPSupport** (*nickname*, *fallback_nicknames=[]*, *username=None*, *realname=None*, *eventloop=None*, ***kwargs*)
Support for CTCP messages.

**ctcp** (*target*, *query*, *contents=None*)
Send a CTCP request to a target.

**ctcp_reply** (*target*, *query*, *response*)
Send a CTCP reply to a target.

**on_ctcp** (*by*, *target*, *what*, *contents*)
Callback called when the user received a CTCP message. Client subclasses can override on_ctcp_<type> to be called when receiving a message of that specific CTCP type, in addition to this callback.

**on_ctcp_reply** (*by*, *target*, *what*, *response*)
Callback called when the user received a CTCP response. Client subclasses can override on_ctcp_<type>_reply to be called when receiving a reply of that specific CTCP type, in addition to this callback.

---

### Account

**class** pydle.features.**AccountSupport** (*nickname*, *fallback_nicknames=[]*, *username=None*, *realname=None*, *eventloop=None*, ***kwargs*)

**whois** (*nickname*)
Return information about user. This is an blocking asynchronous method: it has to be called from a coroutine, as follows:

info = await self.whois('Nick')

---

### ISUPPORT

**class** pydle.features.**ISUPPORTSupport** (*nickname*, *fallback_nicknames=[]*, *username=None*, *realname=None*, *eventloop=None*, ***kwargs*)
ISUPPORT support.

---

#### Extended WHO

**class** pydle.features.**WHOXSupport**(*nickname*, *fallback_nicknames=[]*, *username=None*, *realname=None*, *eventloop=None*, ***kwargs*)

## 2.5 Licensing

### 2.5.1 pydle license

```
Copyright (c) 2014-2016, Shiz
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of the <organization> nor the
      names of its contributors may be used to endorse or promote products
      derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL SHIZ BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

pydle optionally relies on pure-sasl to provide SASL authentication methods; its license is printed in verbatim below.

### 2.5.2 pure-sasl license

```
http://www.opensource.org/licenses/mit-license.php

Copyright 2007-2011 David Alan Cridland
Copyright 2011 Lance Stout
Copyright 2012 Tyler L Hobbs

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:
```

(continues on next page)

(continued from previous page)

```
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.
```

# Python Module Index

## p

# Index

## P



## Q



## R



## S



## T



## U



## W