# python-ldap-faker

***Release 1.1.0***

**Caltech IMSS ADS**

**Nov 22, 2022**

# CONTENTS

Current version is 1.1.0.

This package provides a fake `python-ldap` interface that can be used for automated testing of code that uses `python-ldap`. With `python-ldap-faker` you will be able to test your LDAP code without having to stand up an actual LDAP server, and also without having to use complicated `unittest.mock.patch` and `unittest.mock.Mock` setups.

When writing tests for code that talks to an LDAP server with `python-ldap`, we want to be able to control `python-ldap` interactions in our tests to ensure that our own code works properly. This may include populating the LDAP server with fixture data, monitoring if, when and how `python-ldap` calls are made by our code, and ensuring our code handles `python-ldap` exceptions properly.

Managing an actual LDAP server during our tests is usually out of the question, so typically we revert to patching the `python-ldap` code to use mock objects instead, but this is very verbose and can lead to test code errors in practice.

This package provides replacement `ldap.initialize`, `ldap.set_option` and `ldap.get_option` functions, as well as a test-instrumented `ldap.ldap.ldapobject.LDAPObject` replacement.

CHAPTER

ONE

# INSTALLATION

To install from PyPI:

```
pip install python-ldap-faker
```

If you want, you can run the tests:

```
python -m unittest discover
```

CHAPTER

TWO

# FEATURES:

- These `python-ldap` global functions are faked:
  - `ldap.initialize`
  - `ldap.set_option`
  - `ldap.get_option`
- These `ldap.ldapobject.LDAPObject` methods are faked:
  - `set_option`
  - `get_option`
  - `start_tls_s`
  - `simple_bind_s`
  - `unbind_s`
  - `search_s`
  - `search_ext`
  - `result3`
  - `compare_s`
  - `add_s`
  - `modify_s`
  - `rename_s`
  - `delete_s`
- For `search_ext` and `search_s`, your filter string will be validated as a valid LDAP filter, and your filter will be applied directly to your objects in our fake "server" to generate the result list. No canned searches!
- Inspect your call history for all calls (name, arguments), and test the order in which they were made
- Simulate multiple fake LDAP "servers" with different sets of objects that correspond to different LDAP URIs.
- Ease your test setup with *`LDAPFakerMixin`*, a mixin for `unittest.TestCase`
  - Automatically manages patching `python-ldap` for the code under test
  - Populate objects into one or more LDAP "servers" with fixture files
  - Provides the following test instrumentation for inspecting state after the test:
    - Access to the full object store for each LDAP uri accessed
    - All connections made

        * All `python-ldap` API calls made
        * All `python-ldap` LDAP options set
    - Provides test isolation: object store changes, connections, call history, option changes are all reset between tests
    - Use handy LDAP specific asserts to ease your testing
- Define your own hooks to change the behavior of your fake "servers"
- Support behavior for specific LDAP implementations:
    - Redhat Directory Server/389 implementation support: have your test believe it's talking to an RHDS/389 server.

CHAPTER

THREE

# QUICKSTART

The easiest way to use `python-ldap-faker` in your `unittest` based tests is to use the *`LDAPFakerMixin`* mixin for `unittest.TestCase`.

This will patch `ldap.initialize`, `ldap.set_option` and `ldap.get_option` to use our *`FakeLDAP`* interface, and load fixtures in from JSON files to use as test data.

Let's say we have a class `App` in our `myapp` module that does LDAP work that we want to test.

First, prepare a file named `data.json` with the objects you want loaded into your fake LDAP server. Let's say you want your data to consist of some `posixAccount` objects. If we make `data.json` look like this:

```
[
   [
      "uid=foo,ou=bar,o=baz,c=country",
      {
            "uid": ["foo"],
            "cn": ["Foo Bar"],
            "uidNumber": ["123"],
            "gidNumber": ["123"],
            "homeDirectory": ["/home/foo"],
            "userPassword": ["the password"],
            "objectclass": [
               "posixAccount",
               "top"
            ]
      }
   ],
   [
      "uid=fred,ou=bar,o=baz,c=country",
      {
            "uid": ["fred"],
            "cn": ["Fred Flintstone"],
            "uidNumber": ["124"],
            "gidNumber": ["124"],
            "homeDirectory": ["/home/fred"],
            "userPassword": ["the fredpassword"],
            "objectclass": [
               "posixAccount",
               "top"
            ]
      }
   ],
```

(continues on next page)

(continued from previous page)

```
   [
      "uid=barney,ou=bar,o=baz,c=country",
      {
            "uid": ["barney"],
            "cn": ["Barney Rubble"],
            "uidNumber": ["125"],
            "gidNumber": ["125"],
            "homeDirectory": ["/home/barney"],
            "userPassword": ["the barneypassword"],
            "objectclass": [
               "posixAccount",
               "top"
            ]
      }
   ]
]
```

We can write our `TestCase` like so:

```
import unittest

import ldap
from ldap_faker import LDAPFakerMixin

from myapp import App

class YourTestCase(LDAPFakerMixin, unittest.TestCase):

    ldap_modules = ['myapp']
    ldap_fixtures = 'data.json'

    def test_auth_works(self):
        app = App()
        # A method that does a `simple_bind_s`
        app.auth('fred', 'the fredpassword')
        conn = self.get_connections()[0]
        self.assertLDAPConnectionMethodCalled(
            conn, 'simple_bind_s',
            {'who': 'uid=fred,ou=bar,o=baz,c=country', 'cred': 'the fredpassword'}
        )

    def test_correct_connection_options_were_set(self):
        app = App()
        app.auth('fred', 'the fredpassword')
        conn = self.get_connections()[0]
        self.assertLDAPConnectionOptionSet(conn, ldap.OPT_X_TLX_NEWCTX, 0)

    def test_tls_was_used_before_auth(self):
        app = App()
        app.auth('fred', 'the fredpassword')
        conn = self.get_connections()[0]
        self.assertLDAPConnectiontMethodCalled(conn, 'start_tls_s')
```

(continues on next page)

(continued from previous page)

```
        self.assertLDAPConnectionMethodCalledAfter(conn, 'simple_bind_s', 'start_tls_s')
```

## 3.1 Faking LDAP servers

`python-ldap-faker` stores all LDAP objects in a fake LDAP "server" class: *ObjectStore*, and all our fake `python-ldap` methods operate on the LDAP objects in that object store via the exposed methods on *ObjectStore*.

You won't typically use *ObjectStore* directly, but instead you'll use *LDAPServerFactory* to register *ObjectStore* objects to correspond to specific LDAP URIs (e.g. `ldap://server.example.com`). Our main fake `python-ldap` interface class *FakeLDAP* uses the *LDAPServerFactory* to assign the correct *ObjectStore* when *FakeLDAP.initialize* is called by our code under test.

### 3.1.1 Structure of LDAP records

`python-ldap-faker` tries to pretend it is `python-ldap` as much as possible. Important to this is to mimic how `python-ldap` and LDAP servers represent LDAP objects.

LDAP objects have these characteristics:

- The primary key for an LDAP object is the `dn`. The `dn` is case-insensitive in all `python-ldap` methods. For example, these two statements should operate on the same object:

```
ldap_obj.simple_bind_s("uid=foo,ou=bar,o=baz,c=country", "the password")
ldap_obj.simple_bind_s("UID=FOO,OU=BAR,O=BAZ,C=COUNTRY", "the password")
```

- Simliarly, `basedn`, wherever required, is case-insensitive.
- When doing searches (`search_s`, `search_ext`), LDAP object attributes and values are compared case-insensitively. These searches should all return the same set of objects:

```
ldap_obj.search_s("ou=bar,o=baz,c=country", ldap.SCOPE_SUBTREE, '(uid=bar)')
ldap_obj.search_s("ou=bar,o=baz,c=country", ldap.SCOPE_SUBTREE, '(UID=bar)')
ldap_obj.search_s("ou=bar,o=baz,c=country", ldap.SCOPE_SUBTREE, '(uid=bAr)')
```

- LDAP objects returned by `ldap.search_s` have this type: `Tuple[str, Dict[str, List[bytes]]`. and this structure:

```
('the dn', {'attribute1': [b'value1', b'value2'], ...})
```

### 3.1.2 LDAPServerFactory

*LDAPServerFactory* objects allow you to register *ObjectStore* bound to particular LDAP URIs so that when someone uses our *FakeLDAP.initialize* method, it gets properly instrumented with a **copy** of the `ObjectStore` from the `LDAPServerFactory`. *FakeLDAP* takes a fully loaded *LDAPServerFactory* object as a constructor object.

---

**Note:** Note that we said a **copy** of the `ObjectStore`. Since the primary use of `python-ldap-faker` is in testing, and we want to ensure good test isolation, we should start each test with a fresh copy of original `ObjectStore` data for our LDAP URI so that we can ensure that any modifications to that data came only from our code under test.

---

### 3.1.3 ObjectStore

The core of `python-ldap-faker` is the *ObjectStore* class. This behaves as the LDAP "server" with which our fake `python-ldap` interface interacts. In order to do meaningful work with it, it needs to be loaded with LDAP objects. There are three methods on *ObjectStore* that you can use to load your objects:

- *ObjectStore.register_object*: load a single object into the object store
- *ObjectStore.register_objects*: load a list of objects into the object store
- *ObjectStore.load_objects*: load a list of objects from a JSON file into the object store

Once loaded into *ObjectStore*, we make a fully case-insensitive internal-only copy of the object (stored in *ObjectStore.objects* for use in executing searches, but the data returned will be the case-sensitive versions of those objects (the case-sensitive versions are stored in *ObjectStore.raw_objects*).

#### Data Types for ObjectStore.register_object(s)

Each object loaded into *ObjectStore.register_object* or *ObjectStore.register_objects* must be of this type:

ldap_faker.types.**LDAPRecord**

The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of `Tuple[str, Dict[str, List[bytes]]]`

Example:

```
(
    'uid=user,ou=mydept,o=myorg,c=country',
    {
        'cn': [b'Firstname User1'],
        'uid': [b'user'],
        'uidNumber': [b'123'],
        'gidNumber': [b'456'],
        'homeDirectory': [b'/home/user'],
        'loginShell': [b'/bin/bash'],
        'userPassword': [b'the password'],
        'objectclass': [b'posixAccount', b'top']
    }
)
```

Thus:

- `dn` is a `str`
- Attribute names are `str`
- Attribute values are `List[bytes]`

#### File format for ObjectStore.load_objects

Unfortunately, JSON has neither a `Tuple` type nor a `bytes` type, so we need to use lists and strings instead, and convert them to the appropriate types after reading the JSON file. Thus in our JSON files, we must provide our data as `List[List[str, Dict[str, List[str]]]]` instead. Example:

```
[
  [
    'uid=foo,ou=bar,o=baz,c=country',
    {
      "uid": ["foo"],
      "cn": ["Foo Bar"],
      "uidNumer": ["123"],
      "gidNumer": ["123"],
      "homeDirectory": ["/home/foo"],
      "userPassword": ["the password"],
      "ojectclass": [
        "posixAccount",
        "top"
      ]
    }
  ]
]
```

If you structure your file of LDAP objects like that, and pass in the filename to *`ObjectStore`*, we'll load the data from the file and convert that struct to `List[Tuple[str, List[bytes]]]` before using the result with *`ObjectStore.register_objects`*.`

## 3.2 Specific LDAP implementations supported

Out of the box, our "server" class `ObjectStore` supports searching, adding, updating and deleting objects like a regular LDAP server.

Real LDAP implementations (Redhat Directory Server, 389, openldap, Active Directory) can have special behavior and side-effects that you may need to support in order to run your tests properly.

Currently, we support some special behavior for one implementation: Redhat Directory Server/389.

### 3.2.1 Redhat Directory Server/389

To get these behaviors, add the `389` tag to your `ObjectStore`:

```
>>> store = ObjectStore(tags=['389'])
```

In `LDAPFakerMixin`, apply the tags with like this for a single, default server:

```
import unittest
from ldap_faker import LDAPFakerMixin


class TestDefaultTaggedServer(LDAPFakerMixin, unittest.TestCase):

```

(continues on next page)

(continued from previous page)

```
    ldap_modules = ['myapp']
    ldap_fixtures = ('data.json', ['389'])
```

Or like this for a named server:

```
import unittest
from ldap_faker import LDAPFakerMixin


class TestDefaultTaggedServer(LDAPFakerMixin, unittest.TestCase):

    ldap_modules = ['myapp']
    ldap_fixtures = [
        ('server1.json', 'ldap://server1', ['389']),
    ]
```

### Features supported

Operational attributes

- `entryid`
- `nsUniqueId`
- `entrydn`
- `createTimestamp`
- `modifyTimestamp`
- `creatorName`
- `modifierName`

These work like they should in RHDS/389. They are not returned unless specifically asked for during searches, and they are read-only. The timestamps and names will be updated automatically.

`nsrole` and `nsroledn`

User objects support the `nsroledn` (writeable) and `nsrole` (read-only) attributes. Adding a DN to `nsroledn` makes it appear automatically in `nsrole`, and any objects with `` `objectClass `` of `ldapsubentry` will affect `nsrole` as it does in RHDS/389.

`nsrole` and `nsroledn` are operational attributes; they must be specifically requested during searches.

---

**Important:** In RHDS/389, users do not seem to be identified by objectclass. We're simulating this by assuming that any object with a `userPassword` attribute on it is a user.

---

ldapsubentries

The three `ldapsubentry` objectclasses are supported and behave as they do in RHDS/389:

- `nsManagedRoleDefinition`: does nothing when added or removed
- `nsNestedRoleDefinition`: user objects will gain the proper DN if they match one of this object's `nsroledn` entries.

- `nsFilteredRoleDefinition`: user objects will gain the proper DN if they match this object's `nsRoleFilter`.

## 3.3 Authentication and Authorization

Just like with real LDAP, you'll need to bind to the fake LDAP "server" before you can do certain LDAP operations.

### 3.3.1 Authorization within python-ldap-faker

Like a real LDAP server, these write operations require you to successfully do a non-anonymous bind:

- `add_s`
- `delete_s`
- `modify_s`
- `rename_s`

### 3.3.2 Anonymous binds

You don't need to do anything special to allow anonymous binds. This should work:

```
ldap_obj = fake_ldap.initialize('ldap://server')
ldap_obj.simple_bind_s()
```

So does this:

```
ldap_obj = fake_ldap.initialize('ldap://server')
ldap_obj.search_s('ou=bar,o=baz,c=country', ldap.SCOPE_SUBTREE, '(uid=user)')
```

### 3.3.3 Authenticated binds

To do an authenticated bind, you'll need to load an appropriately configured user object into the `ObjectStore` for your connection.

When you do an authenticated bind via `FakeLDAPObject.simple_bind_s`, `python-ldap-faker` will look in its `ObjectStore` for an object with the `dn` of `who`, and it will compare `cred` with the first value of that object's `userPassword` attribute specifically.

If, for example, your code wants to bind as `uid=foo,ou=bar,o=baz,c=country` with password `the password`, then `python-ldap-faker` will expect an object in the `ObjectStore` that minimally looks like this:

```
(
  'uid=foo,ou=bar,o=baz,c=country',
  {
      "userPassword": [b"the password"],
  }
)
```

## 3.4 Using ldap_faker with unittest

Most of the purpose of `python-ldap-faker` is to make automated testing of code that uses `python-ldap` easier.

To this end, `python-ldap-faker` provides *LDAPFakerMixin*, a mixin class for `unittest.TestCase` which handles all the hard work of patching and instrumenting the appropriate `python-ldap` functions, objects and methods.

*LDAPFakerMixin* will do the following things for you:

- Read data from JSON fixture files to populate one or more *ObjectStore* objects (our fake LDAP server class)
- Associate those *ObjectStore* objects with particular LDAP URIs
- Patch `ldap.initialize` to return *FakeLDAPObject* objects configured with the appropriate *ObjectStore* for the LDAP URI passed into *FakeLDAP.initialize*

### 3.4.1 Configuring your LDAPFakerMixin TestCase

We need to set two class attributes on *LDAPFakerMixin* in order for it to properly set up your tests:

- *LDAPFakerMixin.ldap_modules*: The list of your code's modules in which to patch `ldap.initialize`, `ldap.set_option` and `ldap.get_option`
- *LDAPFakerMixin.ldap_fixtures*: A list of JSON fixture files with which to create the *ObjectStore* objects

#### LDAPFakerMixin.ldap_modules

*LDAPFakerMixin* uses `unittest.mock.patch` to patch your code so that it uses our fake versions of `ldap.initialize`, `ldap.set_option` and `ldap.get_option` instead of the real one. The way `patch` works is that it must apply the patch within the context of your module that does `import ldap`, not within the `ldap` module itself. Thus, to make *LDAPFakerMixin* work for you, you must list all the modules for code under test in which you do `import ldap`.

To list all the modules in which the code under test does `import ldap`, use the *LDAPFakerMixin.ldap_modules* class attribute.

For example, if you have a class `MyLDAPUsingClass` in the module `myapp.myldapstuff`, and you do `import ldap` in `myapp.myldapstuff`, for instance:

```
import ldap

class MyLDAPUsingClass:

    def connect(self, uid: str, password: str):
        self.conn = ldap.initialize('ldap://server')
        self.conn.set_option(ldap.OPT_X_TLS_NEWCTX, 0)
        self.conn.start_tls_s()
        self.conn.simple_bind_s(
          f'uid={uid},ou=bar,o=baz,c=country',
          'the password'
        )
```

To test this code, you would use this for `ldap_modules`:

```
import unittest
from ldap_faker import LDAPFakerMixin

from myapp.myldapstuff import MyLDAPUsingClass

class TestMyLDAPUsingCLass(LDAPFakerMixin, unittest.TestCase):

    ldap_modules = ['myapp.myldapstuff']
```

### LDAPFakerMixin.ldap_fixtures

In order to effectively test your `python-ldap` using code, you'll need to populate an *LDAPServerFactory* one or more *ObjectStore* objects bound to LDAP URIs. We use *LDAPFakerMixin.ldap_fixtures* to declare file paths to fixture files to use to populate those `ObjectClass` objects.

- Fixture files are JSON files in the format described in *File format for ObjectStore.load_objects*.
- File paths are either absolute paths or are treated as relative to the folder in which your `TestCase` resides.
- Fixtures are loaded into the *LDAPServerFactory* **once** per `unittest.TestCase` via the `unittest.TestCase.setUpClass` classmethod.

You can configure your *LDAPFakerMixin* to use fixtures one of two ways:

- Use a single default fixture that will be used no matter which LDAP URI is passed to *FakeLDAP.initialize*
- Bind each fixture to specific a LDAP URI. This allows you simulate talking to several different LDAP servers.

---

**Note:** When binding fixtures to particular LDAP URIs, if your tries to use *FakeLDAP.initialize* with an LDAP URI that was not explicitly configured, `python-ldap-faker` will raise `ldap.SERVER_DOWN`

---

This form sets up one default fixture:

```
import unittest
from ldap_faker import LDAPFakerMixin

from myapp.myldapstuff import MyLDAPUsingClass

class TestMyLDAPUsingCLass(LDAPFakerMixin, unittest.TestCase):

    ldap_fixtures = 'objects.json'
```

This form binds fixtures to LDAP URIs:

```
import unittest
from ldap_faker import LDAPFakerMixin

from myapp.myldapstuff import MyLDAPUsingClass

class TestMyLDAPUsingCLass(LDAPFakerMixin, unittest.TestCase):

    ldap_fixtures = [
      ('server1.json', 'ldap://server1.example.com'),
```

(continues on next page)

(continued from previous page)

```
    ('server2.json', 'ldap://server2.example.com')
]
```

### 3.4.2 Test isolation

Each test method on your `unittest.TestCase` will get a fresh, unaltered **copy** of the fixture data, and connections, call histories, options set from previous test methods will be cleared.

### 3.4.3 Test support offered by LDAPFakerMixin

For each test you run, your test will have access to the *FakeLDAP* instance used for that test through the *LDAPFakerMixin.fake_ldap* instance attribute. Each test gets a fresh *FakeLDAP* instance.

---

**Note:** For detailed information on any of the below, see the *Developer Interface*.

---

Some things to know about your *FakeLDAP* instance:

- *FakeLDAP.connections* lists all the *FakeLDAPObject* connections created during your test method, in the order they were made. One such object is created each time *FakeLDAP.initialize* is called by your code.
- *FakeLDAP.options* is a *OptionStore* object that records all the global LDAP options set during your test
- *FakeLDAP.calls* is a *CallHistory* object that records calls (with arguments) to *FakeLDAP.initialize*, *FakeLDAP.set_option*, *FakeLDAP.get_option*

Some things to know about the *FakeLDAPObject* objects in *FakeLDAP.connections*:

- *FakeLDAPObject.uri* is the LDAP URI requested
- *FakeLDAPObject.store* is our *ObjectStore* copy
- *FakeLDAP.options* is a *OptionStore* object that records all the LDAP options set on this connection during your test method
- *FakeLDAPObject.calls* is a *CallHistory* that records all `python-ldap` api calls (with arguments) that your code made to this `FakeLDAPObject`
- *FakeLDAPObject.bound_dn* is the `dn` of the user bound via `simple_bind_s`, if any. If this is `None`, we did anonymous binding.
- *FakeLDAPObject.tls_enabled* will be set to `True` if `start_tls_s` was used on this connection

## 3.5 Hooks: modifying ObjectStore behavior

`python-ldap-faker` provides a hook system to allow you to arbitrarily modify behavior of *ObjectStore*. Primarily this is provided so that you can emulate the behavior of the various LDAP implementations (Redhat Directory Server, Active Directory, openldap, etc.).

You can also use hooks in your test code to produce behavior that may not be available out of the box from `python-ldap-faker`.

Rules about hooks:

- Hooks are run in the order they are registered

- Each hook needs a callable with a particular signature
- Hooks are global – they apply to all *ObjectStore* instances and instances instantiated (unless they are tagged hooks)

### 3.5.1 Registering hooks

Hooks have a name and a callable signature. Here is an example of registering a hook to the `pre_set` hook, which will be run in *ObjectStore.set* before the object is saved to the internal storage, and requires the callable signature `Callable[[ObjectStore, LDAPRecord, Optional[str]], None]`:

```
from ldap_faker import hooks, ObjectStore, LDAPRecord


def pre_set_do_something_special(store: ObjectStore, record: LDAPRecord, bind_dn: str =␣
↪None) -> None:
    ...


hooks.register('pre_set', pre_set_do_something_special)
```

Thereafter, whenever any code calls *ObjectStore.set*, this function will be called with the store as the first argument, the record to be written as the second argument and the `bind_dn` of the binding user as the third argument.

### 3.5.2 Tagged hooks

Using tags, you can register a hook that will only apply to *ObjectStore* instances which are themselves tagged with one of those tags:

```
from ldap_faker import hooks, ObjectStore, LDAPRecord


def pre_set_do_something_special(store: ObjectStore, record: LDAPRecord, bind_dn: str =␣
↪None) -> None:
    print(f'{bind_dn} ran pre_set_do_something_sepcial')


hooks.register('pre_set', pre_set_do_something_special, tags=['special'])
```

This hook will only be executed for *ObjectStore* instances whose tags include `special`:

```
>>> store = ObjectStore(tags=['special'])
>>> obj = ('mydn', {'objectclass': [b'top']})
>>> store.set(obj, bind_dn='auser')
auser ran pre_set_do_something_special
```

It will not be executed for *ObjectStore* instances whose tags do not include special:

```
>>> store = ObjectStore(tags=['other'])
>>> obj = ('mydn', {'objectclass': [b'top']})
>>> store.set(obj, bind_dn='auser')
```

#### Tagging ObjectClass instances in LDAPFakerMixin

When using *LDAPFakerMixin*, you can tag `ldap_fixtures` with particular tags.

To tag the default "server", specify the fixture as a 2-tuple, where the first element is the filename of the fixture file, and the second element is a list of tags:

```
import unittest
from ldap_faker import LDAPFakerMixin


class TestDefaultTaggedServer(LDAPFakerMixin, unittest.TestCase):

    ldap_modules = ['myapp']
    ldap_fixtures = ('data.json', ['special'])
```

To tag named "servers", you can tag individual servers by providing a 3-tuple instad of a 2-tuple, where the third element is the list of tags:

```
import unittest
from ldap_faker import LDAPFakerMixin


class TestDefaultTaggedServer(LDAPFakerMixin, unittest.TestCase):

    ldap_modules = ['myapp']
    ldap_fixtures = [
        ('server1.json', 'ldap://server1', ['special']),
        ('server2.json', 'ldap://server2')
    ]
```

Above, `ldap://server1` will use all hooks tagged with `special` in addition to any untagged hooks, while `ldap://server2` will use only the untagged hooks.

### 3.5.3 Available hooks

**pre_objectstore_init**

Signature: `Callable[[store: ObjectStore], None]`

Where `store` is the *ObjectStore* object.

This will be at the end of `ObjectStore.__init__`.

You can use this to set up any state you might need for later hooks by adding keys to *ObjectStore.controls*, or to add attributes to *ObjectStore.operational_attributes*.

**pre_set**

Signature: `Callable[[store: ObjectStore, record: LDAPRecord, bind_dn: Optional[str] = None], None]`

Where `store` is the *ObjectStore* object, `record` is the record to be `set` and `bind_dn` is the dn of the user doing the `set` (possibly `None`)

This will be executed on *ObjectStore.set* before the object actually gets saved.

*ObjectStore.set* is called for every write operation:

- *ObjectStore.load_objects*

- *ObjectStore.register_objects*
- *ObjectStore.register_object*
- *FakeLDAPObject.add_s*
- *FakeLDAPObject.modify_s*
- *FakeLDAPObject.delete_s*
- *FakeLDAPObject.rename_s*

**post_set**
Signature: `Callable[[store: ObjectStore, record: LDAPRecord, bind_dn: Optional[str] = None], None]`

Where `store` is the *ObjectStore* object, `record` is the record to be `set` and `bind_dn` is the dn of the user doing the `set` (possibly `None`).

This will be executed on *ObjectStore.set* after the object gets saved.

**pre_copy**
Signature: `Callable[[store: ObjectStore, dn: str], None]`

Where `store` is the *ObjectStore* object, and `dn` is the DN of the object to copy.

This will be executed on *ObjectStore.copy* before the object actually gets retrieved from the store to be copied.

**post_copy**
Signature: `Callable[[store: ObjectStore, data: LDAPData], LDAPData]`

Where `store` is the *ObjectStore* object, and `dn` is the DN of the object to copy. It should return the modified `LDAPData` dict.

This will be executed on *ObjectStore.copy* after the object is retrieved from the store and :py:func:`copy.deepcopy` has run, but before returning the data to the caller.

**pre_create**
Signature: `Callable[[store: ObjectStore, dn: str, modlist: AddModlist, bind_dn: str = None], None]`

Where `store` is the *ObjectStore* object, `dn` is the record to be created, `modlist` is modlist to be used for creating the record, and `bind_dn` is the dn of the user doing the `create` (possibly `None`).

This will be executed on *ObjectStore.create* before the modlist gets processed.

*ObjectStore.create* is what actually does the work when *FakeLDAPObject.add_s* is called.

**post_create**
Signature: `Callable[[store: ObjectStore, record: LDAPRecord, bind_dn: Optional[str] = None], None]`

Where `store` is the *ObjectStore* object, `record` is the record to be created, and `bind_dn` is the dn of the user doing the `create` (possibly `None`).

This will be executed on *ObjectStore.create* after the modlist has processed to build the object, but before it has been writen to the data store.

**pre_update**
Signature: `Callable[[store: ObjectStore, dn: str, modlist: Modlist, bind_dn: str = None], None]`

Where `store` is the *ObjectStore* object, `dn` is the record to be modified`, `modlist` is modlist to be applied to the record, and `bind_dn` is the dn of the user doing the `update` (possibly `None`).

This will be executed on *ObjectStore.update* before the object actually gets saved.

*ObjectStore.update* is what actually does the work when *FakeLDAPObject.modify_s* is called.

**post_update**
Signature: `Callable[[store: ObjectStore, record: LDAPRecord, bind_dn: Optional[str] = None], None]`

Where `store` is the *ObjectStore* object, `record` is the updated record and `bind_dn` is the dn of the user doing the `update` (possibly `None`)

This will be executed on *ObjectStore.update* after the modlist has been applied to the object, but before it has been writen to the data store.

**pre_delete**
Signature: `Callable[[store: ObjectStore, record: LDAPRecord, bind_dn: Optional[str] = None], None]`

Where `store` is the *ObjectStore* object, `record` is the record to deleted, and `bind_dn` is the dn of the user doing the `set` (possibly `None`).

This will be executed on *ObjectStore.delete* before the object actually gets deleted from the data store.

*ObjectStore.delete* is what actually does the work when *FakeLDAPObject.delete_s* is called, and is also called during *FakeLDAPObject.rename_s* to delete the old object.

**post_delete**
Signature: `Callable[[store: ObjectStore, record: LDAPRecord, bind_dn: Optional[str] = None], None]`

Where `store` is the *ObjectStore* object, `record` is the record deleted, and `bind_dn` is the dn of the user doing the `set` (possibly `None`).

This will be executed on *ObjectStore.delete* after the object actually gets deleted from the data store.

**pre_register_object**
Signature: `Callable[[store: ObjectStore, record: LDAPRecord], None]`

Where `store` is the *ObjectStore* object and `record` is the record to be registered.

This will be executed on *ObjectStore.register_object* before the object actually gets saved.

**post_register_object**
Signature: `Callable[[store: ObjectStore, record: LDAPRecord], None]`

Where `store` is the *ObjectStore* object and `record` is the record that was registered.

This will be executed on *ObjectStore.register_object* after the object gets saved.

**pre_register_objects**
Signature: `Callable[[store: ObjectStore, records: List[LDAPRecord]], None]`

Where `store` is the *ObjectStore* object and `records` is the list of records to be registered.

This will be executed on *ObjectStore.register_objects* before the objects actually get saved.

**post_register_objects**
Signature: `Callable[[store: ObjectStore, records: List[LDAPRecord]], None]`

Where `store` is the *ObjectStore* object and `records` are the records that were registered.

This will be executed on *ObjectStore.register_objects* after the objects get saved.

**pre_load_objects**
Signature: `Callable[[store: ObjectStore, filename: str], None]`

Where `store` is the *ObjectStore* object and `filename` is the name of the data file to load.

This will be executed on *ObjectStore.load_objects* before the file gets loaded.

**post_load_objects**
Signature: `Callable[[store: ObjectStore, records: List[LDAPRecord]], None]`

Where `store` is the *ObjectStore* object and `records` are the records that were loaded from the file.

This will be executed on *ObjectStore.load_objects* after the objects loaded from the file get saved.

## 3.6 Developer Interface

This part of the documentation covers all the classes and functions that make up `python-ldap-faker`.

### 3.6.1 Unittest Support

**class** ldap_faker.**LDAPFakerMixin**(**args*, ***kwargs*)

This is a mixin for use with `unittest.TestCase`. Properly configured, it will patch `ldap.initialize` to use our *FakeLDAP.initialize* fake function instead, which will return *FakeLDAPObject* objects instead of `ldap.ldapobject.LDAPObject` objects.

*ldap_modules* is a list of python module paths in which we should patch `ldap.initialize` with our *FakeLDAP.initialize* method. For example:

```
class TestMyStuff(LDAPFakerMixin, unittest.TestCase):

    ldap_modules = ['myapp.module']
```

will cause *LDAPFakerMixin* to patch `myapp.module.ldap.initialize`.

*ldap_fixtures* names one or more JSON files containing LDAP records to load into a *ObjectStore* via *ObjectStore.load_objects*. *ldap_fixtures* can be either a single string, a `Tuple[str, List[str]]`, or a list of `Tuple[str, str, List[str]]`.

If we define our test class like so:

```
class TestMyStuff(LDAPFakerMixin, unittest.TestCase):

    ldap_fixtures = 'myfixture.json'
```

We will build our `LDAPServerFactory` with a single default `ObjectStore` with the contents of `myfixture.json` loaded in.

If we define our test class like so:

```
class TestMyStuff(LDAPFakerMixin, unittest.TestCase):

    ldap_fixtures = ('myfixture.json', ['389'])
```

We will build our `LDAPServerFactory` with a single default `ObjectStore` with the contents of `myfixture.json` loaded in, with the tag *389* applied to it.

If we define our test class like this instead:

```
class TestMyStuff(LDAPFakerMixin, unittest.TestCase):

    ldap_fixtures = [
        ('server1.json', 'ldap://server1', []),
        ('server2.json', 'ldap://read-server2', ['389']),
    ]
```

we will build our *LDAPServerFactory* with two *ObjectStore* objects. The first will have the data from `server1.json` and will be used with uri `ldap://server1`. The second will be used with uri `ldap://server2` and have the data from with the contents of `server2.json` loaded in, and will have the tag `389` applied to it.

---

**Note:** The tags are used when configuring behavior for our `ObjectStore`. The `389` tag tells the *ObjectStore* to emulate a 389 type LDAP server (Redhat Directory Server).

---

**ldap_modules: List[str] = []**

The list of python paths to modules that import `ldap`

**ldap_fixtures: Optional[ldap_faker.types.LDAPFixtureList] = None**

The filenames of fixtures to load into our fake LDAP servers

**server_factory: *LDAPServerFactory***

The *LDAPServerFactory* configured by our *setUpClass*

**fake_ldap: *FakeLDAP***

the *FakeLDAP* instance created by *setUp*

**classmethod resolve_file**(*filename: str*) → str

Given `filename`, if that filename is a non-absolute path, resolve that filename to an absolute path under the folder in which our subclass' file resides. If `filename` is an absoute path, don't change it.

**Parameters**
**filename** – the non-absolute file path to a fixture file

**Raises**
**FileNotFoundError** – the fixture file did not exist

**Returns**
The absolute path to the fixture file.

**classmethod load_servers**(*server_factory:* LDAPServerFactory) → None

Configure `server_factory` with one or more `ObjectStore` objects by looking at *ldap_fixtures*, a dict where the key is a uri and the value is the name of a JSON file to use as the objects for the associated `ObjectStore`

---

**Note:** If you want to populate your *LDAPServerFactory* in a different way than loading directly from the JSON files listed in *ldap_fixtures*, this is the classmethod you want to override.

---

**Parameters**
**server_factory** – the `LDAPServerFactory` object to populate

**classmethod setUpClass**()

Build the `LDAPServerFactory` we'll use and save it as a class attribute.

We do this as a classmethod because constructing our *ObjectStore* objects is time consuming and we don't want to have to do it for each of our tests.

**classmethod tearDownClass**()

Delete our *server_factory* so we con't corrupt future tests or leak memory.

**setUp**()

Create a *FakeLDAP* instance, make it use the *server_factory* that our *setUpClass* created, and patch `ldap.initialize` in each of the modules named in *ldap_modules*. Save the *FakeLDAP* instance to our *fake_ldap* attribute for later use in our test code.

**tearDown**()

Undo the patches we made in *setUp*

**last_connection**() → Optional[*FakeLDAPObject*]

Return the *FakeLDAPObject* for the last connection made during ourtest. Hopefully a useful shortcut for when we only make one connection.

**Returns**
The last connection made

**get_connections**(*uri: Optional[str] = None*) → List[*FakeLDAPObject*]

Return a the list of *FakeLDAPObject* objects generated during our test, optionally filtered by LDAP URI.

**Keyword Arguments**
**uri** – the LDAP URI by which to filter our connections

**assertGlobalOptionSet**(*option: int*, *value: ldap_faker.types.LDAPOptionValue*) → None

Assert that a global LDAP option was set.

**Parameters**

- **option** – an LDAP option (e.g. ldap.OPT_DEBUG_LEVEL)
- **value** – the value we expect the option to be set to

**assertGlobalFunctionCalled**(*api_name: str*) → None

Assert that a global LDAP function was called.

**Parameters**
**api_name** – the name of the function to look for (e.g. `initialize`)

**assertLDAPConnectionOptionSet**(*conn:* FakeLDAPObject, *option: str*, *value: ldap_faker.types.LDAPOptionValue*) → None

Assert that a specific *FakeLDAPObject* option was set with a specific value.

**Parameters**

- **conn** – the connection object to examine
- **option** – the code for the option (e.g. `ldap.OPT_X_TLS_NEWCTX`)
- **value** – the value we expect the option to be set to

**assertLDAPConnectionMethodCalled**(*conn:* FakeLDAPObject, *api_name: str*, *arguments: Optional[Dict[str, Any]] = None*) → None

Assert that a specific *FakeLDAPObject* method was called, possibly specifying the specific arguments it should have been called with.

**Parameters**

- **conn** – the connection object to examine

- **api_name** – the name of the function to look for (e.g. `simple_bind_s`)

**Keyword Arguments**

**arguments** – if given, assert that the call exists AND was called this set of arguments. See *LDAPCallRecord* for how the `arguments` dict should be constructed.

**assertLDAPConnectionMethodCalledAfter**(*conn:* FakeLDAPObject, *api_name: str*, *target_api_name: str*) → None

Assert that a specific *FakeLDAPObject* method was called after another specific *FakeLDAPObject* method.

**Parameters**

- **conn** – the connection object to examine
- **api_name** – the name of the function to look for (e.g. `simple_bind_s`)
- **target_api_name** – the name of the function which should appear before `api_name` in the call history

**class** ldap_faker.**LDAPCallRecord**(*api_name: str*, *args: Dict[str, Any]*)

This is a single LDAP call record, used by *CallHistory* to store information about calls to LDAP api functions.

*api_name* is the name of the LDAP api call made (e.g. `simple_bind_s`, `search_s`).

*args* is the argument list of the call, including defaults for keyword arguments not passed. This is a dict where the key is the name of the positional or keyword argument, and the value is the passed in (or default) value for that argument.

### Example

If we make this call to a patched *FakeLDAPObject*:

```
ldap_obj.search_s('ou=bar,o=baz,c=country', ldap.SCOPE_SUBTREE, '(uid=foo)')
```

This will be recorded as:

```
LDAPCallRecord(
    api_name='search_s',
    args={
        'base': 'ou=bar,o=baz,c=country',
        'scope': 2,
        'filterstr': '(uid=foo)',
        'attrlist': None,
        'attrsonly': 0
    }
)
```

**api_name:** `str`

the name LDAP api call

**args:** `Dict[str, Any]`

the args and kwargs dict

**class** ldap_faker.**CallHistory**(*calls: Optional[List[*LDAPCallRecord*]] = None*)

This class records the `python-ldap` call history for a particular *FakeLDAPObject* as *LDAPCallRecord* objects. It works in conjunction with the `@record_call` decorator. An *CallHistory* object will be configured on each *FakeLDAPObject* and on each *FakeLDAP* object capture their call history.

We use this in our tests with appropriate asserts to ensure that our code called the `python-ldap` methods we expected, in the order we expected, with the arguments we expected.

**filter_calls**(*api_name: str*) → List[*LDAPCallRecord*]

Filter our call history by function name.

**Parameters**
**api_name** – look through our history for calls to this function

**Returns**
A list of (`api_name`, `arguments`) tuples in the order in which the calls were made. Arguments is a `Dict[str, Any]`.

**property calls: List[*LDAPCallRecord*]**

This property returns the list of all calls made against the parent object.

### Example

To test that your code did a `ldap.simple_bind_s` call with the usernam and password you expected, you could do:

```
from unittest import TestCase
import ldap
from ldap_faker import LDAPFakerMixin

from my_code import App

class MyTest(LDAPFakerMixin, TestCase):

    ldap_modules = ['my_code']
    ldap_fixtures = 'myfixture.json'

    def test_option_was_set(self):
        app = MyApp()
        app.do_the_thing()
        conn = self.ldap_faker.connections[0]
        self.assertEqual(
            conn.calls,
            [('simple_bind_s', {'who': 'uid=foo,ou=dept,o=org,c=country', 'cred
↪': 'pass'})]
        )
```

**Returns**
Returns a list of 2-tuples, one for each method call made since the last reset. Each tuple contains the name of the API and a dictionary of arguments. Argument defaults are included.

**property names: List[str]**

Returns the list names of `python-ldap` functions or methods called, in the order they were called. You can use this to test whether an particulary

#### Example

To test that your code did at least one `ldap.add_s` call, you could do:

```
from unittest import TestCase
import ldap
from ldap_faker import LDAPFakerMixin

from my_code import App

class MyTest(LDAPFakerMixin, TestCase):

    ldap_modules = ['my_code']
    ldap_fixtures = 'myfixture.json'

    def test_option_was_set(self):
        app = MyApp()
        app.do_the_thing()
        conn = self.ldap_faker.connections[0]
        self.assertEqual('add_s" in conn.calls.names)
```

**Returns**
A list of method names, in the order they were called.

### 3.6.2 python-ldap replacements

**class** ldap_faker.**FakeLDAP**(*server_factory:* LDAPServerFactory)

We use this class to house our replacement code for these three prime `python-ldap` functions:

- `ldap.initialize`
- `ldap.set_option`
- `ldap.get_option`

The class takes a fully configured *LDAPServerFactory* as an argument, and will use that factory's collection of *OptionStore* objects to construct new *FakeLDAPObject* objects.

As a test runs, *FakeLDAP* keeps track of each LDAP connection made and each global LDAP call made so that they can be inspected after your code has run.

---

**Note:** This is meant to be a disposable object, recreated for each test method. When used properly, all internal state (connections made, calls made, options set) will be empty at the start of every test.

---

**Parameters**
**server_factory** – a fully configured *LDAPServerFactory*

**connections:** List[*FakeLDAPObject*]

list of *FakeLDAPObject* connections created in the order in which they were requested

**calls:** *CallHistory*

The call history for global ldap function calls

**options:** *OptionStore*

A dictionary of LDAP options set

**initialize**(*uri: str*, *trace_level: int = 0*, *trace_file: ~typing.TextIO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *trace_stack_limit: int = None*, *fileno: ~typing.Any = None*) → *FakeLDAPObject*

This is the method we use to patch `ldap.initialize` when we are testing our LDAP code. When it is called, we will ask our `FakeLDAP.server_factory` factory for the *ObjectStore* most appropriate for the LDAP uri `uri`, create a *FakeLDAPObject* with a `copy.deepcopy` of that *ObjectStore*, and return the *FakeLDAPObject*.

---

**Note:** Of all the arguments in our signature, we only actually use `uri`. The other arguments are recorded in our *FakeLDAP.calls* call history, but are otherwise ignored.

---

**Parameters**

- **uri** – an LDAP URI
- **trace_level** – logging level (ignored)
- **trace_file** – file descriptor to which to write traces (ignored)
- **trace_stack_limit** – stack limit of tracebacks in the debug log (ignored)
- **fileno** – a socket or file descriptor (ignored)

**Raises**

`ldap.SERVER_DOWN` – could not find an appropriate *ObjectStore* for `uri`

**Returns**

A properly configured *FakeLDAPObject*

**set_option**(*option: int*, *invalue: ldap_faker.types.LDAPOptionValue*) → None

Set a global `python-ldap option`. This will create a key `option` in our *FakeLDAP.options* dictionary and set its value to `value`.

### Example

In your test code, you can thus test whether your code set the proper global LDAP option like so:

```
from unittest import TestCase
import ldap
from ldap_faker import LDAPFakerMixin

from my_code import App

class MyTest(LDAPFakerMixin, TestCase):

    ldap_modules = ['my_code']
    ldap_fixtures = 'myfixture.json'

    def test_option_was_set(self):
        app = MyApp()
        app.set_the_option(ldap.OPT_DEBUG_LEVEL, 1)
        self.assertEqual(self.ldap_faker.options[ldap.OPT_DEBUG_LEVEL], 1)
```

**Parameters**

- **option** – an option from python-ldap
- **invalue** – the value to set for the option

`get_option`(*option: int*) → ldap_faker.types.LDAPOptionValue

Get a global python-ldap option. If our code hasn't set an `option` yet, return the default from `python-ldap` for that option.

**Parameters**
**option** – an option from python-ldap

**Returns**
The value currently set for the option.

`has_connection`(*uri: str*) → bool

Test to see whether an `ldap.initialize` call was made with LDAP URI of `uri`.

**Parameters**
**uri** – The LDAP URI to look for in our connection history

**Returns**
`True` if at least one connection to `uri` was made, `False` otherwise.

`get_connections`(*uri: str*) → List[*FakeLDAPObject*]

Return a list of *`FakeLDAPObject`* connections to LDAP URI `uri`.

**Parameters**
**uri** – The LDAP URI to look for in our connection history

**Returns**
A list of *`FakeLDAPObject`* objects associated with LDAP URI `uri`.

`connection_calls`(*api_name: Optional[str] = None*, *uri: Optional[str] = None*) → *CallHistory*

Filter our the call history for our connections by function name and optionally LDAP URI.

Args:

**Keyword Arguments**

- **api_name** – restrict through our history for calls to this function
- **uri** – restrict our search to only calls to this URI

**Returns**
A *`CallHistory`* with combined calls from the filtered connections.

`class ldap_faker.FakeLDAPObject`(*uri: str*, *store: Optional[*ObjectStore*] = None*)

This class simulates most of the interface of `ldap.ldapobject.LDAPObject` which is the object that gets returned when you call `ldap.initialize()`.

---

**Note:** This is a disposable object that should be recreated for each test, mostly because changes to our `ObjectStore` can't be undone without re-copying from its source in `Servers`.

---

**Parameters**
**uri** – the LDAP URI of the connection

**Keyword Arguments**
**directory** – a populated *`ObjectStore`*

**uri:** *str*

the LDAP URI for this connection

**hostname**

port for this connection

**Type**
the host

**options:** *OptionStore*

we store data from *set_option* calls here

**store:** *ObjectStore*

our copy of our ObjectStore for this connection

**calls:** *CallHistory*

The method call history

**tls_enabled:** *bool*

Set to True if *start_tls_s* was called

**bound_dn:** *Optional[str]*

Set by *simple_bind_s* to the dn of the user after success

**deref:** *int*

Controls whether aliases are automatically dereferenced

**protocol_version:** *int*

Version of LDAP in use (always `ldap.VERSION3`)

**sizelimit:** *int*

Limit on size of message to receive from server

**network_timeout:** *int*

Limit on waiting for a network response, in seconds.

**timelimit:** *int*

Limit on waiting for any response, in seconds.

**timeout:** *int*

Limit on waiting for any response, in seconds.

**set_option**(*option: int*, *invalue: ldap_faker.types.LDAPOptionValue*) → None

This method sets the value of the `ldap.ldap.ldapobject.LDAPObject` option specified by `option` to `invalue`.

**Parameters**

- **option** – the option
- **value** – the value to set the option to

**Raises**
**ValueError** – `option` is not a valid `python-ldap` option

**get_option**(*option: int*) → ldap_faker.types.LDAPOptionValue

This method returns the value of the `ldap.ldap.ldapobject.LDAPObject` option specified by `option`.

**Note:** If your code did not call `FakeLDAPOption.set_option` for this option, we'll get `KeyError`

**Parameters**
**option** – the option

**Raises**

- **ValueError** – `option` is not a valid `python-ldap` option
- **KeyError** – `option` is not a valid `python-ldap` option

**Returns**
The value of the option

**simple_bind_s**(*who: str = None*, *cred: str = None*, *serverctrls: List[LDAPControl] = None*, *clientctrls: List[LDAPControl] = None*) → Optional[Tuple[Union[int, str], List[Tuple[str, Dict[str, List[bytes]]]]], int, List[LDAPControl]]]

Perform a bind. This will look in the object store for an object with dn of `who` and compare `cred` to the `userPassword` attribute for that object.

**Keyword Arguments**

- **who** – the dn of the user with which to bind
- **cred** – the password for that user

**Raises**
**ldap.INVALID_CREDENTIALS** – the `who` did not match the `cred`

**whoami_s**() → str

This synchronous method implements the LDAP "Who Am I?" extended operation.

It is useful for finding out to find out which identity is assumed by the LDAP server after a bind.

**Returns**
{the dn}"

**Return type**
Empty string if we haven't bound as an identity, otherwise "dn

**search_ext**(*base: str*, *scope: int*, *filterstr: str = '(objectClass=*)'*, *attrlist: List[str] = None*, *attrsonly: int = 0*, *serverctrls: List[LDAPControl] = None*, *clientctrls: List[LDAPControl] = None*, *timeout: int = -1*, *sizelimit: int = 0*) → int

**result3**(*msgid: int = -1*, *all: int = 1*, *timeout: int = None*) → Tuple[Union[int, str], List[Tuple[str, Dict[str, List[bytes]]]], int, List[LDAPControl]]

Retrieve the results of our *FakeLDAPObject.search_ext* call.

---

**Note:** The `all` and `timeout` keyword arguments are ignored here.

---

**Keyword Arguments**

- **msgid** – the `msgid` returned by the *FakeLDAPObject.search_ext* call
- **all** – if 1, return all results at once; if 0, return them one at a time (ignored)

**Returns**
A `ldap.result3` 4-tuple.

**search_s**(*base: str*, *scope: int*, *filterstr: str = '(objectClass=*)'*, *attrlist: List[str] = None*, *attrsonly: int = 0*) → List[ldap_faker.types.LDAPRecord]

**start_tls_s**() → None

Negotiate TLS with server.

This sets our *tls_enabled* attribute to True.

**Raises**
**ldap.LOCAL_ERROR** – *start_tls_s* was done twice on the same connection

**compare_s**(*dn: str*, *attr: str*, *value: bytes*) → bool

Perform an LDAP comparison between the attribute named attr of entry dn, and the value value. For multi-valued attributes, the test is whether any of the values match value.

**Parameters**

- **dn** – the dn of the object to look at
- **attr** – the name of the attribute on our object to compare
- **value** – the value to which to compare the object value

**Raises**
**ldap.NO_SUCH_OBJECT** – no object with dn of dn exists in our object store

**Returns**
True if the values are equal, False otherwise.

**modify_s**(*dn*, *modlist: ldap_faker.types.ModList*) → Tuple[Union[int, str], List[Tuple[str, Dict[str, List[bytes]]]]], int, List[LDAPControl]]

Modify the object with dn of dn using the modlist modlist.

Each element in the list modlist should be a tuple of the form (mod_op: int, mod_type: str, mod_vals: Union[bytes, List[bytes]]), where mod_op indicates the operation (one of ldap.MOD_ADD, ldap.MOD_DELETE, or ldap.MOD_REPLACE, mod_type is a string indicating the attribute type name, and mod_vals is either a bytes value or a list of bytes values to add, delete or replace respectively. For the delete operation, mod_vals may be None indicating that all attributes are to be deleted.

---

**Note:** ldap.modlist.modifyModlist MAY be your friend here for generating modlists. Do read the note in those docs about ldap.MOD_DELETE / ldap.MOD_ADD vs. ldap.MOD_REPLACE to see whether that will affect you poorly.

---

### Example

Here is an example of constructing a modlist for modify_s:

```
>>> import ldap
>>> modlist = [
    (ldap.MOD_ADD, 'mail', [b'user@example.com', b'user+foo@example.com']),
    (ldap.MOD_REPLACE, 'cn', [b'My Name']),
    (ldap.MOD_DELETE, 'gecos', None)
]
```

**Parameters**

- **dn** – the dn of the object to delete
- **modlist** – a modlist suitable for modify_s

**Raises**

- **ldap.NO_SUCH_OBJECT** – no object with dn of **dn** exists in our object store
- **ldap.TYPE_OR_VALUE_EXISTS** – you tried to add an value to an attribute, but it was already in the value list
- **ldap.INSUFFICIENT_ACCESS** – you need to do a non-anonymous bind before doing this

**Returns**
A `ldap.result3` type 4-tuple.

**delete_s**(*dn: str*) → None

Delete the object with dn of **dn** from our object store.

Each element in the list modlist should be a tuple of the form `(mod_type: str, mod_vals: List[bytes])`, where `mod_type` is a string indicating the attribute type name, and `mod_vals` is either a string value or a list of string values to add, delete or replace respectively. For the delete operation, mod_vals may be `None` indicating that all attributes are to be deleted.

**Parameters**
**dn** – the dn of the object to delete

**Raises**

- **ldap.NO_SUCH_OBJECT** – no object with dn of **dn** exists in our object store
- **ldap.INSUFFICIENT_ACCESS** – you need to do a non-anonymous bind before doing this

**add_s**(*dn: str*, *modlist: ldap_faker.types.AddModList*) → None

Add an object with dn of **dn**.

`modlist` is similar the one passed to *modify_s*, except that the operation integer is omitted from the tuples in `modlist`. You might want to look into sub-module refmodule{ldap.modlist} for generating the modlist.

### Example

Here is an example of constructing a modlist for `add_s`:

```
>>> modlist = [
    ('uid', [b'user']),
    ('gidNumber', [b'1000']),
    ('uidNumber', [b'1000']),
    ('loginShell', [b'/bin/bash']),
    ('homeDirectory', [b'/home/user']),
    ('userPassword', [b'the password']),
    ('cn', [b'My Name']),
    ('objectClass', [b'top', b'posixAccount']),
]
```

**Parameters**

- **dn** – the dn of the object to add
- **modlist** – the add modlist

**Raises**

- **ldap.ALREADY_EXISTS** – an object with dn of **dn** already exists in our object store
- **ldap.INSUFFICIENT_ACCESS** – you need to do a non-anonymous bind before doing this

**rename_s**(*dn: str*, *newrdn: str*, *newsuperior: str = None*, *delold: int = 1*, *serverctrls: List[LDAPControl] = None*, *clientctrls: List[LDAPControl] = None*) → None

Take dn (the DN of the entry whose RDN is to be changed, and newrdn, the new RDN to give to the entry. The optional parameter newsuperior is used to specify a new parent DN for moving an entry in the tree (not all LDAP servers support this).

**Parameters**

- **dn** – the dn of the object to rename
- **newrdn** – the new RDN

**Keyword Arguments**

- **newsuperior** – the new basedn
- **delold** – if 1, delete the old entry after renaming, if 0, don't.

**Raises**

- **ldap.NO_SUCH_OBJECT** – no object with dn of dn exists in our object store
- **ldap.INSUFFICIENT_ACCESS** – you need to do a non-anonymous bind before doing this

**unbind_s**() → None

Unbind from the server.

This sets our *bound_dn* to None.

### 3.6.3 LDAP Server like objects

**class** ldap_faker.**LDAPServerFactory**

This class registers *ObjectStore* objects to be used by FakeLDAP.initialize() in constructing *FakeLDAPObject* objects. *ObjectStore* objects are named registered here by LDAP uri (in reality, any string).

You may do one of two things, but not both:

- Configure a default *ObjectStore* that will be used for all ldap.initialize calls regardless of uri
- Assign a specific *ObjectStore* for each uri you will be using in your code.

#### Example

To register a default *ObjectStore* that will be used for every uri passed to *FakeLDAP.initialize*:

```
>>> from ldap_faker import ObjectStore, LDAPServerFactory, FakeLDAP
>>> data = [ ... ]   # some LDAP records
>>> factory = LDAPServerFactory()
>>> store = ObjectStore(objects=data)
>>> factory.register(store)
>>> fake_ldap = FakeLDAP(factory)
```

Now any time your code does an ldap.initialize() to our patched version of that function, it will get a a *FakeLDAPObject* configured with a copy.deepcopy of the *ObjectStore* store, no matter what uri it passes to ldap.initialize().

To register a different ObjectStores that will be used for specific uris:

```
>>> from ldap_faker import ObjectStore, Servers
>>> data1 = [ ... ]   # some LDAP records
>>> factory = LDAPServerFactory()
>>> store1 = ObjectStore(objects=data1)
>>> factory.register(store1, uri='ldap://server1')
>>> data2 = [ ... ]   # some different LDAP records
>>> store2 = ObjectStore(objects=data2)
>>> factory.register(store2, uri='ldap://server2')
>>> fake_ldap = FakeLDAP(factory)
```

Now if your code does `ldap.initialize('ldap://server1')`, it will get a *FakeLDAPObject* configured with a `copy.deepcopy` of the *ObjectStore* object `store1`, while if it does `ldap.initialize('ldap://server2' )`, it will get a *FakeLDAPObject* configured with a `copy.deepcopy` of the *ObjectStore* object `store2`.

**load_from_file**(*filename: str*, *uri: Optional[str] = None*, *tags: Optional[List[str]] = None*) → None

Given a file path to a JSON file with the objects for an *ObjectStore*, create a new *ObjectStore*, load it with that JSON File and register it with uri of `uri`.

**Parameters**
**filename** – the full path to our JSON file

**Keyword Arguments**

- **uri** – the uri to assign to the *ObjectStore* we create
- **tags** – the list of tags to apply to the the *ObjectStore*

**Raises**

- **ValueError** – raised if a default is already configured while trying to register the *ObjectStore* with a specific `uri`
- **RuntimeWarning** – raised if we try to overwrite an already registered object store with our new one

**register**(*store:* ObjectStore, *uri: Optional[str] = None*) → None

Register a new *ObjectStore* to be used as our fake LDAP server for when we run our fake `initialize` function.

**Parameters**
**store** – a configured *ObjectStore*

**Keyword Arguments**
**uri** – the LDAP uri to associated with `directory`

**Raises**

- **ValueError** – raised if a default is already configured while trying to register an *ObjectStore* with a specific `uri`
- **RuntimeWarning** – raised if we try to overwrite an already registered object store with a new one

**get**(*uri: str*) → *ObjectStore*

Return a `copy.deepcopy` of the *ObjectStore* identified by `uri`.

**Parameters**
**uri** – use this uri to look up which *ObjectStore* to use

**Raises**
ldap.SERVER_DOWN – no *ObjectStore* could be found for uri

**Returns**
A copy.deepcopy of the *ObjectStore*

class ldap_faker.**ObjectStore**(*tags: Optional[List[str]] = None*)

This class represents our actual simulated LDAP object store. Copies of this will be used to configure *FakeLDAPObject* objects.

**raw_objects: ldap_faker.types.RawLDAPObjectStore**

LDAP records as they would have been returned by python-ldap`

**objects: ldap_faker.types.LDAPObjectStore**

LDAP records set up to make searching better

**tags: List[str]**

used when filtering hooks to apply

**controls: Dict[str, Any]**

can be used by hooks to store state

**operational_attributes: Set[str]**

list of attributes that have to be specifically requested

**convert_LDAPData**(*data: ldap_faker.types.LDAPData*) → ldap_faker.types.CILDAPData

Convert an incoming LDAPData` dict (``Dict[str, List[bytes]]) to a CILDAPData dict (CaseInsensitiveDict[str, List[str]])

We need the data dict to have values as List[str] so that our filtering works properly – ldap_filter.Filter.match only works with strings, not bytes.

**Parameters**
**data** – the LDAPData dict to convert

**Returns**
The convered CILDAPData dict.

**load_objects**(*filename: str*) → None

Load a list of LDAP records stored as JSON from a file into our internal database. Use this when setting up the data you will use to run your tests.

---

**Note:** One caveat with this method vs. *ObjectStore.register_objects* is that the records returned by python-ldap are of type Tuple[str, Dict[str, List[bytes]]] but JSON has no concept of bytes or tuple. Thus we will expect the LDAP records in the file to have type List[str, Dict[str, List[str]]] and we will convert them to Tuple[str, Dict[str, List[bytes]]] before saving to *raw_objects*

---

**Parameters**
**filename** – the path to the JSON file to load

**Raises**

- ldap.ALREADY_EXISTS – there is already an object in our object store with this dn
- ldap.INVALID_DN_SYNTAX – one of the object DNs is not well formed

**register_objects**(*objs: List[ldap_faker.types.LDAPRecord]*) → None

Load a list of LDAP records into our internal database. Use this when setting up the data you will use to run your tests. Each record in the list should be in exactly the format that `python-ldap` itself returns: a 2-tuple with dn as the first element and the attribute/value dict as the second element.

### Example

Adding a several PosixAccount objects:

```
>>> obj = [
    (
        'uid=user,ou=mydept,o=myorg,c=country',
        {
            'cn': [b'Firstname User1'],
            'uid': [b'user'],
            'uidNumber': [b'123'],
            'gidNumber': [b'456'],
            'homeDirectory': [b'/home/user'],
            'loginShell': [b'/bin/bash'],
            'userPassword': [b'the password'],
            'objectclass': [b'posixAccount', b'top']
        }
    ),
    (
        'uid=user2,ou=mydept,o=myorg,c=country',
        {
            'cn': [b'Firstname User2'],
            'uid': [b'user2'],
            'uidNumber': [b'124'],
            'gidNumber': [b'457'],
            'homeDirectory': [b'/home/user1'],
            'loginShell': [b'/bin/bash'],
            'userPassword': [b'the password'],
            'objectclass': [b'posixAccount', b'top']
        }
    )
]
>>> directory = ObjectStore()
>>> directory.register_objects(obj)
```

**Parameters**

**objs** – A list of LDAP records as they would have been returned by `ldap.ldapobject.LDAPObject.search_s()`. These are 2-tuples, where the first element is the *dn* (a `str`) and the second element is a dict where the keys are `str` and the values are lists of `bytes`.

**Raises**

- `ldap.ALREADY_EXISTS` – there is already an object in our object store with this dn
- `ldap.INVALID_DN_SYNTAX` – one of the object DNs is not well formed
- `TypeError` – the LDAPData portion for an object was not of type `Dict[str, List[bytes]]`

**register_object**(*obj: ldap_faker.types.LDAPRecord*) → None

Add an LDAP record our internal database. Use this to add a single record when setting up the data you will use to run your tests. The data should be in exactly the format that python-ldap itself returns: a 2-tuple with dn as the first element and the attribute/value dict as the second element.

### Example

Adding a PosixAccount object:

```
>>> obj = (
    'uid=user,ou=mydept,o=myorg,c=country',
    {
        'cn': [b'Firstname Lastname'],
        'uid': [b'user'],
        'uidNumber': [b'123'],
        'gidNumber': [b'456'],
        'homeDirectory': [b'/home/user'],
        'loginShell': [b'/bin/bash'],
        'userPassword': [b'the password']
        'objectclass': [b'posixAccount', b'top']
    }
)
>>> directory = ObjectStore()
>>> directory.register_object(obj)
```

**Parameters**
**obj** – An LDAP record as it would have been returned by `ldap.ldapobject.LDAPObject.search_s()`. This is a 2-tuple, where the first element is the *dn* (a `str`) and the second element is a dict where the keys are `str` and the values are lists of `bytes`.

**Raises**
- **ldap.ALREADY_EXISTS** – there is already an object in our object store with this dn
- **ldap.INVALID_DN_SYNTAX** – the DN is not well formed
- **TypeError** – the LDAPData portion was not of type `Dict[str, List[bytes]]`

**property count**

**exists**(*dn: str*, *validate: bool = True*) → bool

Test whether an object with dn **dn** exists.

**Parameters**
**dn** – the dn of the object to look for

**Keyword Arguments**
**validate** – if `True`, validate that **dn** is a valid dn

**Returns**
`True` if the object exists, `False` otherwise.

**get**(*dn: str*) → ldap_faker.types.LDAPData

Return all data for an object from our object store.

**Parameters**
**dn** – the dn of the object to copy.

**Raises**
ldap.NO_SUCH_OBJECT – no object with dn of dn exists in our object store

**Returns**
The data for an LDAP object

**copy**(*dn: str*) → ldap_faker.types.LDAPData

Return a copy of the data for an object from our object store.

**Parameters**
**dn** – the dn of the object to copy.

**Raises**
ldap.NO_SUCH_OBJECT – no object with dn of dn exists in our object store

**Returns**
The data for an LDAP object

**set**(*dn: str*, *data: ldap_faker.types.LDAPData*, *bind_dn: Optional[str] = None*) → None

Add or update data for the object with dn dn.

**Parameters**

- **dn** – the dn of the object to copy.
- **data** – the dict of data for this object

**Keyword Arguments**
**bind_dn** – the dn of the user doing the set, if any

**Raises**

- ldap.INVALID_DN_SYNTAX – the DN is not well formed
- TypeError – the LDAPData portion was not of type `Dict[str, List[bytes]]`

**update**(*dn: str*, *modlist: ldap_faker.types.ModList*, *bind_dn: Optional[str] = None*) → None

Modify the object with dn of dn using the modlist modlist.

Each element in the list modlist should be a tuple of the form `(mod_op: int, mod_type: str, mod_vals: Union[bytes, List[bytes]])`, where mod_op indicates the operation (one of ldap.MOD_ADD, ldap.MOD_DELETE, or ldap.MOD_REPLACE, mod_type is a string indicating the attribute type name, and mod_vals is either a bytes value or a list of bytes values to add, delete or replace respectively. For the delete operation, mod_vals may be None indicating that all attributes are to be deleted.

---

**Note:** ldap.modlist.modifyModlist MAY be your friend here for generating modlists. Do read the note in those docs about ldap.MOD_DELETE / ldap.MOD_ADD vs. ldap.MOD_REPLACE to see whether that will affect you poorly.

---

### Example

Here is an example of constructing a modlist for `modify_s`:

```
>>> import ldap
>>> modlist = [
    (ldap.MOD_ADD, 'mail', [b'user@example.com', b'user+foo@example.com']),
    (ldap.MOD_REPLACE, 'cn', [b'My Name']),
    (ldap.MOD_DELETE, 'gecos', None)
]
```

**Parameters**

- **dn** – the dn of the object to delete
- **modlist** – a modlist suitable for `modify_s`

**Keyword Arguments**

**bind_dn** – the dn of the user doing the update, if any

**Raises**

- `ldap.INVALID_DN_SYNTAX` – the dn was not well-formed
- `ldap.NO_SUCH_OBJECT` – no object with dn of `dn` exists in our object store
- `ldap.TYPE_OR_VALUE_EXISTS` – you tried to add an value to an attribute, but it was already in the value list
- `ldap.INSUFFICIENT_ACCESS` – you need to do a non-anonymous bind before doing this

**create**(*dn: str*, *modlist: ldap_faker.types.AddModList*, *bind_dn: Optional[str] = None*) → None

Create an object in our store with dn of `dn`.

`modlist` is similar the one passed to `modify_s`, except that the operation integer is omitted from the tuples in `modlist`. You might want to look into sub-module ldap.modlist for generating the modlist.

### Example

Here is an example of constructing a modlist for `create`:

```
>>> modlist = [
    ('uid', [b'user']),
    ('gidNumber', [b'1000']),
    ('uidNumber', [b'1000']),
    ('loginShell', [b'/bin/bash']),
    ('homeDirectory', [b'/home/user']),
    ('userPassword', [b'the password']),
    ('cn', [b'My Name']),
    ('objectClass', [b'top', b'posixAccount']),
]
```

**Parameters**

- **dn** – the dn of the object to add
- **modlist** – the add modlist

**Keyword Arguments**
**bind_dn** – the dn of the user doing the create, if any

**Raises**

- **ldap.INVALID_DN_SYNTAX** – the dn was not well-formed
- **ldap.ALREADY_EXISTS** – an object with dn of **dn** already exists in our object store
- **ldap.INSUFFICIENT_ACCESS** – you need to do a non-anonymous bind before doing this

**delete**(*dn: str*, *bind_dn: Optional[str] = None*) → None

Delete an object from our objects directory.

**Parameters**
**dn** – the dn of the object to delete

**Keyword Arguments**
**bind_dn** – the dn of the user doing the delete, if any

**Raises**
**ldap.INVALID_DN_SYNTAX** – the dn was not well-formed

**search_base**(*base: str*, *filterstr: str*, *attrlist: Optional[List[str]] = None*) → ldap_faker.types.LDAPSearchResult

Do a `ldap.SCOPE_BASE` search. Return the requested attributes of the object in our object store with **dn** of `base` that also matches `filterstr`.

---

**Note:** We return a `copy.deepcopy` of the object, not the actual object. This ensures that if the caller modifies the object they don't update the objects in us unintentionally.

---

---

**Note:** Some attributes are "operational" and are not returned by default They must be named specifically if you want them. Example:

```
>>> store.search_base('thebasedn', '(objectclass=*)', ['*', 'createTimestamp'])
```

---

**Parameters**

- **base** – the dn of the object to return
- **filterstr** – the ldap filter string

**Keyword Arguments**
**attrlist** – the list of attributes to return for each object

**Raises**

- **ldap.INVALID_DN_SYNTAX** – `base` was not a well-formed DN
- **ldap.FILTER_ERROR** – `filterstr` is has bad filter syntax
- **ldap.NO_SUCH_OBJECT** – no object with dn of `base` exists in the object store

**Returns**
A list with one element – the object with dn of `base`.

**search_onelevel**(*base: str*, *filterstr: str*, *attrlist: Optional[List[str]] = None*) → ldap_faker.types.LDAPSearchResult

Do a `ldap.SCOPE_ONELEVEL` search, for objects directly under basedn `base` that match `filterstr`.

---

**Note:** We return a `copy.deepcopy` of each object, not the actual object. This ensures that if the caller modifies the object they don't update the objects in us unintentionally.

---

**Parameters**

- **base** – the dn of the object to return
- **filterstr** – the ldap filter string

**Keyword Arguments**

**attrlist** – the list of attributes to return for each object

**Raises**

- **ldap.INVALID_DN_SYNTAX** – `base` was not a well-formed DN
- **ldap.FILTER_ERROR** – `filterstr` is has bad filter syntax

**Returns**

A list of LDAP objects – 2-tuples of `(dn, data)`.

**search_subtree**(*base: str*, *filterstr: str*, *attrlist: Optional[List[str]] = None*, *include_operational_attributes: bool = False*) → ldap_faker.types.LDAPSearchResult

Do a `ldap.SCOPE_SUBTREE` search, for objects under basedn `base` that match `filterstr`.

**Parameters**

- **base** – the dn of the object to return
- **filterstr** – the ldap filter string

---

**Note:** We return a `copy.deepcopy` of each object, not the actual object. This ensures that if the caller modifies the object they don't update the objects in us unintentionally.

---

**Keyword Arguments**

- **attrlist** – the list of attributes to return for each object
- **include_operational_attributes** – include all operational attributes even if they are not named in `attrlist`

**Raises**

- **ldap.INVALID_DN_SYNTAX** – `base` was not a well-formed DN
- **ldap.FILTER_ERROR** – `filterstr` is has bad filter syntax

**Returns**

A list of LDAP objects – 2-tuples of `(dn, data)`.

**class** ldap_faker.**OptionStore**

We use this to store options set via `set_option`.

**set**(*option: int*, *invalue: ldap_faker.types.LDAPOptionValue*) → None

Set an option.

**Parameters**

- **option** – the code for the option (e.g. `ldap.OPT_X_TLS_NEWCTX`)
- **value** – the value we want the option to be set to

**Raises**

**ValueError** – `option` is not a valid `python-ldap` option

**get**(*option: int*) → ldap_faker.types.LDAPOptionValue

Get the value for a previosly set option that was set via *OptionStore.set*.

**Parameters**

**option** – the code for the option (e.g. `ldap.OPT_X_TLS_NEWCTX`)

**Raises**

**ValueError** – `option` is not a valid `python-ldap` option

**Returns**

The value for the option, or the default.

### 3.6.4 Hook management

ldap_faker.**hooks = <ldap_faker.hooks.HookRegistry object>**

**class** ldap_faker.**Hook**(*func: Callable*, *tags: List[str]*)

**func:** Callable

**tags:** List[str]

**class** ldap_faker.**HookDefinition**(*name: str*, *signature: str*)

The definition for a hook. This is comprised of a name and a signature.

#### Example

```
>>> hook_def = HookDefinition(
    name='pre_save",
    signature="Callable[[ObjectStore, LDAPRecord], None]
)
>>> hook_def.name
"pre_save"
>>> hook_def.signature
"Callable[[ObjectStore, LDAPRecord], None]"
```

**name**

the name of the hook, e.g. “pre_save”

**Type**

str

**signature**

the python type annotation signature that the hook should implement, e.g. "Callable[[ObjectStore, LDAPRecord], None]"

**Type**
str

**name:** *str*

**signature:** *str*

class ldap_faker.**HookRegistry**

**property definitions:** *List[HookDefinition]*

Return a list of known hooks definitions as

**register_hook_definition**(*hook_name: str*, *signature: str*) → None

Register a hook definition. Hook definitions define what hooks exist, and what their function signature must be.

### Example

```
>>> hooks = HookRegistry()
>>> hooks.register_definition('pre_set', 'Callable[[ObjectStore, LDAPRecord],␣
↪None]')
```

**Parameters**

- **hook_name** – the name of the hook
- **signature** – A string in Python type annotation format describing the signature the hook must have

**register_hook**(*hook_name: str*, *func: Callable*, *tags: Optional[List[str]] = None*) → None

Register a hook for this object store. Hooks are functions with this signature:

```
def myhook(store: ObjectStore, record: LDAPRecord) -> None:
```

Use hooks to implement side-effects on select *ObjectStore* methods.

### Example

To register a hook that updates a an attribute named ``modifyTimestamp` before saving a record to the object store, you could define the hook like so:

**def update_modifyTimestamp(store: ObjectStore, record: LDAPRecord) -> None:**
record[1]['modifyTimestamp'] = datetime.datetime.utcnow().strftime('%Y%m%d%H%M%SZ')

and register it as a *pre_modify* method like so:

```
>>> store = ObjectStore()
>>> store.register_hook('pre_set', update_modifyTimestamp)
```

**Note:** Hooks for a particular hook_name are applied in the order they are registered.

**Parameters**

- **hook_name** – the name of the known hook to which register this `func`
- **func** – the hook function

**Raises**

**ValueError** – `hook_name` is not a known hook

**get**(*hook_name: str*, *tags: Optional[List[str]] = None*) → List[Callable]

Get a list of hook callables for the hook named by `name`, possibly filtering hooks by tag.

Tag filtering rules:

- If a hook has no tags associated with it, it always applies.
- Otherwise, if at least one of the hooks tags are present in `tags`, the hook applies.

**Parameters**

**hook_name** – the name of the hook for which to return functions

**Keyword Arguments**

**tags** – if provided, filter the available hook functions to include only those with tags listed in `tags`

**Raises**

**ValueError** – there is no known hook with name `hook_name`

**Returns**

A list of callables.

### 3.6.5 Type Aliases

ldap_faker.types.**LDAPOptionValue**

The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of Union[int, str]

ldap_faker.types.**LDAPData**

The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of Dict[str, List[bytes]]

ldap_faker.types.**LDAPRecord**

The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of Tuple[str, Dict[str, List[bytes]]]

ldap_faker.types.**LDAPSearchResult**

The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of List[Tuple[str, Dict[str, List[bytes]]]]

ldap_faker.types.**ModList**

The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of List[Tuple[int, str, List[bytes]]]

ldap_faker.types.**AddModList**

The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of List[Tuple[str, List[bytes]]]

ldap_faker.types.**LDAPFixtureList**

The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of Union[str, Tuple[str, List[str]], List[Tuple[str, str, List[str]]]]

# PYTHON MODULE INDEX

# INDEX