

API Documentation

API Documentation

October 3, 2010

Contents

Contents	1
1 Package easyLDAP	2
1.1 Modules	2
1.2 Variables	2
2 Module easyLDAP.easyLDAP_bind	4
2.1 Functions	4
2.2 Variables	8
3 Module easyLDAP.easyLDAP_class_base	9
3.1 Variables	9
3.2 Class easyLDAP	9
3.2.1 Methods	9
4 Module easyLDAP.easyLDAP_class_cache	20
4.1 Variables	20
4.2 Class easyLDAP_cache_history	20
4.2.1 Methods	20
4.2.2 Class Variables	21
4.3 Class easyLDAP_cacheobject	21
4.3.1 Methods	21
4.3.2 Class Variables	22
4.4 Class easyLDAP_cachetree	23
4.4.1 Methods	23
4.4.2 Class Variables	25
5 Module easyLDAP.easyLDAP_class_object_base	26
5.1 Variables	26
5.2 Class easyLDAP_object	26
5.2.1 Methods	26
5.3 Class easyLDAP_object	39
5.3.1 Methods	39
6 Module easyLDAP.easyLDAP_class_tree	52
6.1 Variables	52
6.2 Class easyLDAP_tree	52
6.2.1 Methods	52

6.2.2	Class Variables	55
6.3	Class easyLDAP_tree	56
6.3.1	Methods	56
6.3.2	Class Variables	59
7	Module easyLDAP.easyLDAP_defaults	60
7.1	Functions	60
7.2	Variables	60
8	Module easyLDAP.easyLDAP_etc	61
8.1	Variables	61
9	Module easyLDAP.easyLDAP_exceptions'	62
9.1	Variables	62
9.2	Class easyLDAP_exceptions	62
9.2.1	Methods	62
10	Module easyLDAP.easyLDAP_utils	63
10.1	Functions	63
10.2	Variables	70
11	Module easyLDAP.easyLDAP_version	72
11.1	Variables	72

1 Package easyLDAP

Python easyLDAP aims to be an easy-to-use LDAP client API for your Python programmes.

The library currently provides the following features:

- read-write access to OpenLDAP databases, read-only access to ActiveDirectory
- LDAP schema based operations on data objects in an LDAP tree
- offline cache of LDAP tree, offline analysis and offline modification of data in your LDAP tree
- offline and online search for LDAP data in your tree
- flushing back of cache to the LDAP server
- LDAP tree is stored in a recursive data structure (i.e. also in a tree-like data object)
- LDAP methods and cache methods have been implemented in separate object classes (since v0.2.0)

The easyLDAP module currently still has some limitation:

- only LDAP simple bind methods are supported (neither Kerberos support, nor GSS-API support yet)

If you have any questions concerning Python easyLDAP, please visit our website and seek for contact data there: <http://das-netzwerkteam.de>

Version: 0.2.2

1.1 Modules

- **easyLDAP_bind** (*Section 2, p. 4*)
- **easyLDAP_class_base** (*Section 3, p. 9*)
- **easyLDAP_class_cache** (*Section 4, p. 20*)
- **easyLDAP_class_object_base** (*Section 5, p. 26*)
- **easyLDAP_class_tree** (*Section 6, p. 52*)
- **easyLDAP_defaults**: Default settings for easyLDAP.
(*Section 7, p. 60*)
- **easyLDAP_etc**: Make the easyLDAP configuration folder available to Python.
(*Section 8, p. 61*)
- **easyLDAP_exceptions** (*Section ??, p. ??*)
- **easyLDAP_exceptions'** (*Section 9, p. 62*)
- **easyLDAP_utils** (*Section 10, p. 63*)
- **easyLDAP_version**: Define the version of Python easyLDAP.
(*Section 11, p. 72*)

1.2 Variables

Name	Description
EASY_LDAP	Value: {'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ...}
SCHEMA_ATTRS	Value: []
SCHEMA_CLASS_MAPPING	Value: {}
__package__	Value: 'easyLDAP'
easyLDAP_ADDATTR	Value: 0

continued on next page

Name	Description
easyLDAP_REPLACEATTR	Value: 1
except_dict	Value: {'CANNOT_DELETE_BASEDN_OBJECT': 'The LDAP server\'s base ...
key	Value: 'NO_PARENTOBJECT_FOR_GIVEN_DN'

2 Module *easyLDAP.easyLDAP_bind*

2.1 Functions

```
easyLDAP_setDefaultCredentials(bind_dn, bind_pw=None,  
config_defaults={'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ...})
```

```
easyLDAP_setDefaultCredentials('myDN', 'myPW')
```

sets the easyLDAP default value dictionary entries as follows:

```
EASY_LDAP['BindDN'] = 'myDN'  
EASY_LDAP['BindPW'] = 'myPW'
```

Now you can call the functions

```
easyLDAPdefaultBind(obj)  
easyLDAPdefaultBind.norefresh(obj)
```

and the bind process will be performed automatically with the default values.

If 'myPW' is not specified, you will be prompted for it on STDOUT/STDIN.

```
easyLDAP_setDefaultBindCredentials(bind_pw=None, config_defaults={'AdminRDN':  
'cn=admin', 'AutomountRDN': 'ou=automount', ...})
```

```
easyLDAP_setDefaultBindCredentials('myPW')
```

sets the easyLDAP default value dictionary entries as follows:

```
EASY_LDAP['BindDN'] = 'myDN'  
EASY_LDAP['BindPW'] = 'myPW'
```

Now you can call the functions

```
easyLDAPdefaultBind(obj)  
easyLDAPdefaultBind.norefresh(obj)
```

and the bind process will be performed automatically with the default values.

If 'myPW' is not specified, you will be prompted for it on STDOUT/STDIN.

```
easyLDAP_setDefaultAdminCredentials(bind_pw=None,  
config_defaults={'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ...})
```

```
easyLDAP_setDefaultAdminCredentials('myAdminPW')
```

sets the easyLDAP default value dictionary entries as follows:

```
EASY_LDAP['BindDN'] = config_defaults['AdminRDN']+', '+  
                      config_defaults['BaseDN']  
EASY_LDAP['BindPW'] = 'myAdminPW'
```

Now you can call the functions

```
easyLDAPdefaultBind(my_easyLDAPobject)  
easyLDAPdefaultBind.norefresh(my_easyLDAPobject)
```

and the administrative bind process will be performed automatically with these default values.

If 'myAdminPW' is not specified and EASY_LDAP['BindPW'] contains an empty string, a password dialog will be initiated on STDIN/STDOUT. This dialog can be forced by passing a non-string as 'myAdminPW' (e.g. explicitly: None).

```
easyLDAP_defaultBind(INSTANCE)
```

binds the given object to the LDAP server. This helper function requires the dictionary entry EASY_LDAP['BindDN'] to be set.

If so, it will check the dict entry EASY_LDAP['BindPW'] for a non empty string. If so, it will bind the given easyLDAP object with the standard credentials, defined by the named dictionary entries. If the bind password is not globally specified, the function will ask for the password at the command line.

This function uses refreshes the easyLDAP object's cache. The result of the binding method will be returned by this function.

easyLDAP_defaultBind_norefresh(*INSTANCE*)

binds the given object to the LDAP server. This helper function requires the dictionary entry `EASY_LDAP['BindDN']` to be set.

If so, it will check the dict entry `EASY_LDAP['BindPW']` for a non-empty string. If so, it will bind the given easyLDAP object with the standard credentials, defined by the named dictionary entries. If the bind password is not globally specified, the function will ask for the password at the command line.

This function does *NOT* refresh the easyLDAP object's cache. Some values of the original LDAP object might be missing in the easyLDAP object cache as a consequence of restrictions of the previously used credentials.

The result of the binding method will be returned by this function.

easyLDAPsetDefaultCredentials(*bind_dn*, *bind_pw*=None, *config_defaults*={'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ...})

```
easyLDAP_setDefaultCredentials('myDN', 'myPW')
```

sets the easyLDAP default value dictionary entries as follows:

```
EASY_LDAP['BindDN'] = 'myDN'
EASY_LDAP['BindPW'] = 'myPW'
```

Now you can call the functions

```
easyLDAPdefaultBind(obj)
easyLDAPdefaultBind.norefresh(obj)
```

and the bind process will be performed automagically with the default values.

If 'myPW' is not specified, you will be prompted for it on STDOUT/STDIN.

```
easyLDAPsetDefaultBindCredentials(bind_pw=None, config_defaults={'AdminRDN':  
'cn=admin', 'AutomountRDN': 'ou=automount', ...})
```

```
easyLDAP_setDefaultBindCredentials('myPW')
```

sets the easyLDAP default value dictionary entries as follows:

```
EASY_LDAP['BindDN'] = 'myDN'  
EASY_LDAP['BindPW'] = 'myPW'
```

Now you can call the functions

```
easyLDAPdefaultBind(obj)  
easyLDAPdefaultBind.norefresh(obj)
```

and the bind process will be performed automagically with the default values.

If 'myPW' is not specified, you will be prompted for it on STDOUT/STDIN.

```
easyLDAPsetDefaultAdminCredentials(bind_pw=None, config_defaults={'AdminRDN':  
'cn=admin', 'AutomountRDN': 'ou=automount', ...})
```

```
easyLDAP_setDefaultAdminCredentials('myAdminPW')
```

sets the easyLDAP default value dictionary entries as follows:

```
EASY_LDAP['BindDN'] = config_defaults['AdminRDN']+',','+  
                      config_defaults['BaseDN']  
EASY_LDAP['BindPW'] = 'myAdminPW'
```

Now you can call the functions

```
easyLDAPdefaultBind(my_easyLDAPobject)  
easyLDAPdefaultBind.norefresh(my_easyLDAPobject)
```

and the administrative bind process will be performed automagically with these default values.

If 'myAdminPW' is not specified and EASY_LDAP['BindPW'] contains an empty string, a password dialog will be initiated on STDIN/STDOUT. This dialog can be forced by passing a non-string as 'myAdminPW' (e.g. explicitly: None).

easyLDAPdefaultBind(*INSTANCE*)

binds the given object to the LDAP server. This helper function requires the dictionary entry `EASY_LDAP['BindDN']` to be set.

If so, it will check the dict entry `EASY_LDAP['BindPW']` for a non empty string. If so, it will bind the given easyLDAP object with the standard credentials, defined by the named dictionary entries. If the bind password is not globally specified, the function will ask for the password at the command line.

This function uses refreshes the easyLDAP object's cache. The result of the binding method will be returned by this function.

easyLDAPdefaultBind_norefresh(*INSTANCE*)

binds the given object to the LDAP server. This helper function requires the dictionary entry `EASY_LDAP['BindDN']` to be set.

If so, it will check the dict entry `EASY_LDAP['BindPW']` for a non-empty string. If so, it will bind the given easyLDAP object with the standard credentials, defined by the named dictionary entries. If the bind password is not globally specified, the function will ask for the password at the command line.

This function does *NOT* refresh the easyLDAP object's cache. Some values of the original LDAP object might be missing in the easyLDAP object cache as a consequence of restrictions of the previously used credentials.

The result of the binding method will be returned by this function.

2.2 Variables

Name	Description
<code>EASY_LDAP</code>	Value: <code>{ 'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ... }</code>
<code>--package--</code>	Value: <code>'easyLDAP'</code>

3 Module *easyLDAP.easyLDAP_class_base*

3.1 Variables

Name	Description
<code>easyLDAP_ADDATTR</code>	Value: 0
<code>easyLDAP_REPLACEATTR</code>	Value: 1
<code>SCHEMA_CLASS_MAPPING</code>	Value: {}
<code>SCHEMA_ATTRS</code>	Value: []
<code>EASY_LDAP</code>	Value: {'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ...}
<code>__package__</code>	Value: 'easyLDAP'
<code>except_dict</code>	Value: {'CANNOT_DELETE_BASEDN_OBJECT': 'The LDAP server\'s base ...}
<code>key</code>	Value: 'NO_PARENTOBJECT_FOR_GIVEN_DN'

3.2 Class *easyLDAP*

Known Subclasses: *easyLDAP.easyLDAP_class_object_base.easyLDAP_object*, *easyLDAP.easyLDAP_class_tree.easyLDAP*

3.2.1 Methods

`__deepcopy__`(*self*, *memo*)

Creates a deepcopy of the *easyLDAP* class.

`__del__`(*self*)

`__init__`(*self*, *bind_dn*=None, *bind_pw*=None, *config_defaults*={'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ...}, *update_schema_cache*=False, *ldapsession*=None)

`adminbind`(*self*, *admin_bind_pw*, *refresh_cache*=True)

`thisClass.adminbind('myADMINPW')`

binds to the LDAP server with administrative access. The LDAP server's admin DN is stored in `thisClass.AdminDN`, the administrative password is given through the argument 'myADMINPW'.

`adminbind_norefresh`(*self*, *admin_bind_pw*)

`thisClass.bind_norefresh('myDN', 'myPW')`

as the *easyLDAP* class does not load any objects into the *easyLDAP* cache, this method is just an alias for `this_class.bind()`

anonymous_bind(*thisClass*)

tries an anonymous bind to the LDAP server.

anonymous_bind_norefresh(*thisClass*)

tries an anonymous bind to the LDAP server.

attrtype_needs_objectclass(*self*, *attr*)

attrtype_needs_objectclass('myATTRIBUTE')

returns True if the easyLDAP object in cache needs an additional objectClass before adding the LDAP attribute 'myATTRIBUTE'. If the objectClass dependencies are already all solved, it returns False.

The pseudo-boolean function returns None, if there is no such attribute in the server's LDAP schema.

bind(*self*, *bind_dn*, *bind_pw*, *refresh_cache=True*)

thisClass.bind('myDN', 'myPW')

binds to the LDAP server with the given credentials 'myDN' and 'myPW'.

bind_norefresh(*self*, *bind_dn*, *bind_pw*)

thisClass.bind_norefresh('myDN', 'myPW')

as the easyLDAP class does not load any objects into the easyLDAP cache, this method is just an alias for this_class.bind()

check_objectclass_dependencies(*self*, *objectclass_list*)

```
check_objectclass_dependencies ('myOBJECT_CLASS'|['myOBJECT_CLASS1',
                                                    'myOBJECT_CLASS2'])
```

checks if and what objectClasses are needed before adding them to an LDAP directory. The dependencies are derived from the server's LDAP schema.

The method returns a list of ObjectClasses, including those that were given to the method and those that are needed to complete the set.

cn_bind(*self*, *bind_cn*, *bind_pw*, *refresh_cache=True*)

thisClass.cnbind('myCN', 'myPW')

binds to the LDAP server. This method tries to find the cn's corresponding DN in the LDAP tree starting at thisClass.BaseDN.

Binding by CN (Common Name) will fail if the given cn 'myCN' is not unique to the specified LDAP tree.

cn_bind.norefresh(*self*, *bind_cn*, *bind_pw*)

thisClass.uid_bind.norefresh('myDN', 'myPW')

as the easyLDAP class does not load any objects into the easyLDAP cache, this method is just an alias for this_class.bind()

find_newGidNumber(*self*, *minGID*, *maxGID=None*, *myGROUPSBASEDN=None*)

searches the connected LDAP server for the lowest vacant gidNumber between minGID and maxGID.

If myGROUPSBASEDN is not specified, the easyLDAP thisClass.GroupBaseDN is presumed.

The method only checks vacant gidNumber in the connected LDAP tree. Other sources for posixGroupIDs (/etc/groups, NIS, etc.) are not taken into consideration!

If an error occurs or no vacant gidNumber can be found, the method returns '-1'.

find_newUidNumber(*self*, *min_uidNumber=None*, *max_uidNumber=None*, *searchbase=None*, *cached_searchresult=None*)

self.find_newUidNumber (minUID,maxUID,myPEOPLEBASEDN)

live searches the connected LDAP server for the lowest vacant uidNumber between minUID and maxUID.

If myPEOPLEBASEDN is not specified, the easyLDAP thisClass.PeoplebaseDN is presumed.

The method only checks vacant uidNumber in the connected LDAP tree. Other sources for posixUserIDs (/etc/passwd, NIS, etc.) are not taken into consideration!

If an error occurs or no vacant uidNumber can be found, the method returns '-1'.

generate_attributetypes_dict(*self*)

thisClass.generate_attributetypes_dict ()

is for class-internal use only. It is evoked once from thisClass.__init__() to generate the python dictionary thisClass.attributeTypesDict.

You should not need to call this method.

generate_ldapsyntaxes_dict(*self*)

thisClass.generate_ldapsyntaxes_dict ()

is for class-internal use only. It is evoked once from thisClass.__init__() to generate the python dictionary thisClass.ldapSyntaxesDict.

You should not need to call this method.

generate_objectclass_dict(*thisClass*)

is for class-internal use only. It is evoked once from thisClass.__init__() to generate the python dictionary thisClass.objectClassesDict.

You should not need to call this method.

generate_reverse_attributetypes_dict(*self*)**get_admin_dn(*self*)**

thisClass.get_admin_dn ()

get_attributetypes(*thisClass*)

returns a list of all attributeTypes that are known the server's LDAP schema.

get_attrtype_aliases(*self*, *attrtype*, *capitalize=False*)

thisClass.get_attrtypealiases('myATTRTYPE')

returns a list of attribute types that are treated as aliases in the LDAP server's schema.

get_basedn(*self*)

thisClass.get_basedn ()

returns the absolute DN of the currently cached easyLDAP object/tree/subtree from the LDAP server's directory tree.

get_cache_reference(*thisClass*)

commits a search operation on the LDAP server with starting with thisClass.ObjectCacheBaseDN as the base DN. The result of this operation can be compared to the current content of the easyLDAP cache to figure out what changes have been made to the object in cache by easyLDAP methods.

If an error occurs, an empty list will be returned.

get_groups_basedn(*self*)

thisClass.get_groups_basedn ()

get_hosts_basedn(*self*)

thisClass.get_hosts_basedn ()

get_objectclasses(*thisClass*)

returns a list of all objectClasses that are known to the server's LDAP schema.

get_parent_dn(*self, dn*)

thisClass.get_parent_dn ('myDN')

returns the parent absolute DN of 'myDN' from the LDAP server's directory tree.

get_parent_rdn(*self, dn*)

thisClass.get_parent_rdn ('myDN')

returns the parent relative DN of 'myDN' from the LDAP server's directory tree.

get_people_basedn(*self*)

thisClass.get_people_basedn ()

get_samba3_algorithmicRidBase(*self, this_domain*)

thisClass.get_samba3_algorithmicRidBase('myDOMAIN')

searches the LDAP tree starting at thisClass.BaseDN for the algorithmic RID base of Samba domain 'myDOMAIN'.

get_samba3_domainSID(*self, domain*)

thisClass.get_samba3_domainSID('myDOMAIN')

searches the LDAP tree starting at thisClass.BaseDN for the Samba PDC's sambaDomainName entry. When the method can find it, then it will extract the domain's SID.

get_samelevel_dns(*self, dn, use_cache=False*)

thisClass.get_samelevel_dns ('myDN')

returns a list with distinguished names that exist on the same level as the given DN 'myDN'.

If the DN 'myDN' is alone on its level, it will be the only item in the list.

If the returned list is empty, an error occurred.

get_sublevel_dns(*self*, *dn*, *use_cache=False*)

thisClass.get_subleveldns ('myDN')

returns a list with distinguished names that exist exactly below the given DN 'myDN'.

If the easyLDAP object has no objects on its sublevel, the result will be an empty list. If an LDAP error occurs, the list will also be empty!

get_sublevel_rdns(*self*, *dn*, *use_cache=False*)

thisClass.get_sublevelrdns ('myDN')

returns a list with reduced distinguished names that exist exactly below the given DN 'myDN'.

If the easyLDAP object has no objects on its sublevel, the result will be an empty list. If an LDAP error occurs, the list will also be empty!

get_subtree_dns(*self*, *dn*, *use_cache=False*)

thisClass.get_subtree_dns (this_dn)

returns a tree of all distinguished names that exists below the given DN 'myDN'.

If the easyLDAP object has no objects on its sublevel, the result will be an empty list. If an LDAP error occurs, the list will also be empty!

get_used_attroptions(*self*, *attrdesc*, *pyldapobject*)

thisClass.get_used_attroptions('myATTRDESC', [mySINGLELDAPOBJECT])

returns a list of all attribute options used with attribute type same as in 'myATTRDESC'. The comparison is performed on an OID basis. So, if an alias of an 'myATTRDESC' is used, it will be taken into account.

get_used_attrtype_oids(*thisClass*)

returns a list of those attribute OIDs, that are currently used in the cached easyLDAP object. This list is taken from the server's LDAP schema. If - by some reason - the easyLDAP object cache is, an empty list is returned.

Another LDAP object than the one currently cached can be analyzed by passing 'mySINGLELDAPOBJECT'. The lengths of the passed LDAP object has to be 1.

get_used_attrtypes(*thisClass*, *mySINGLELDAPOBJECT=...*)

returns a list of attribute types that are used in the current easyLDAP object. If - by some reason - the cache is empty, an empty list is returned.

Another LDAP object than the one currently cached can be analyzed by passing 'mySINGLELDAPOBJECT'. The lengths of the passed LDAP object has to be 1.

has_dn(*self*, *dn*, *searchbase*=None, *scope*=2)

`thisClass.has_dn ('myDN')`

checks, if the given distinguished name 'myDN' exists in the server's LDAP directory. The searches starts with the `thisClass.BaseDN`.

has_oid_set(*self*, *oid*, *pyldapobject*)

`thisClass.has_oid_set('myOID'[, mySINGLELDAPOBJECT])`

checks if the given OID 'myOID' is set in the cached easyLDAP object. If so, the method's result is True.

Another LDAP object than the one currently cached can be analyzed by passing 'mySINGLELDAPOBJECT'. The lengths of the passed LDAP object has to be 1.

has_parent_dn(*self*, *dn*)

`thisClass.has_parent_dn ('myDN')`

tests the existence of the referred object with a live search request to the LDAP server.

has_samba2_schema(*self*)

has_samba3_schema(*self*)

has_valid_attrtype(*self*, *attrdesc*)

`thisClass.has_valid_attrtype('myATTRDESC')`

tests if the given attribute 'myATTRDESC' has a valid attributeType according to the server's LDAP schema.

is_collectiveattr(*self*, *this_attr*)

`thisClass.is_collectiveattr('myATTRIBUTE')`

tests if the COLLECTIVE flag is set for the LDAP attribute 'myATTRIBUTE'.

is_nousermodattr(*self*, *this_attr*)

`thisClass.is_nousermodattr('myATTRIBUTE')`

tests if the NO-USER-MODIFICATION flag is set for the LDAP attribute 'myATTRIBUTE'.

is_obsoleteattr(*self*, *this_attr*)

`thisClass.is_obsoleteattr('myATTRIBUTE')`

tests if the OBSOLETE flag is set for the LDAP attribute 'myATTRIBUTE'.

is_pyldapobject(*self*, *ldap_data_dict*)

`thisClass.is_ldapobject('myLDAP_DATA')`

returns True, if the given data structure conforms to the python-ldap data format and only contains data of a single DN.

is_pyldaptree(*self*, *ldap_data_dict*)

`thisClass.is_ldaptree('myLDAP_DATA')`

returns True, if the given data structure conforms to the python-ldap data format and the hierarchical structure is tree-like.

is_singlevalueattr(*self*, *this_attr*)

`thisClass.is_singlevalueattr('myATTRIBUTE')`

tests if the SINGLE-VALUE flag is set for the LDAP attribute 'myATTRIBUTE'.

is_valid_attrtype(*self*, *attrtype*)

`thisClass.is_valid_attrtype('myATTRTYPE')`

tests if the given attribute type 'myATTRTYPE' is a valid attribute type according to the server's LDAP schema.

is_valid_objectclass(*self*, *objectclass*)

`thisClass.is_valid_objectclass('myOBJECTCLASS')`

tests if the given objectClass 'myOBJECTCLASS' is a valid objectClass according to the server's LDAP schema.

map_attrtype2ldapsyntax(*self*, *this_attrtype*)

`thisClass.map_attrtype2ldapsyntax('myATTRTYPE')`

returns the LDAPSyntax for the given attribute 'myATTRTYPE'. Using the dictionary `thisClass.templateLDAPSyntaxValues` will help you to generate a default value for new attribute values:

```
print thisClass.templateLDAPSyntaxValues[
    thisClass.map_attrtype2ldapsyntax['myATTRTYPE']
]
```

map_attrtype2ldapsyntaxname(*self, this_attrtype*)

`thisClass.map_attrtype2ldapsyntax('myATTRTYPE')`

returns the LDAPSyntax for the given attribute 'myATTRTYPE'. Using the dictionary `thisClass.templateLDAPSyntaxValues` will help you to generate a default value for new attribute values:

```
print thisClass.templateLDAPSyntaxValues[
    thisClass.map_attrtype2ldapsyntax['myATTRTYPE']
]
```

map_attrtype2ldapsyntaxoid(*self, this_attrtype*)

`thisClass.map_attrtype2ldapsyntaxoid('myATTRTYPE')`

returns the LDAPSyntax's OID for the given attribute type 'myATTRTYPE'.

map_attrtype2objectclasses(*self, attrtype*)

`thisClass.map_attrtype2objectclass ('myATTRTYPE')`

returns a lists of objectClasses that the given attribute 'myATTRTYPE' occurs in. This list is derived from the server's LDAP schema.

map_attrtype2valuemaxlen(*self, this_attrtype*)

`thisClass.map_attrtype2ldapsyntaxoid('myATTRTYPE')`

returns the LDAPSyntax's OID for the given attribute type 'myATTRTYPE'.

map_attrtypes2oids(*self, attribute_types_list*)

`thisClass.map_attrtypes2oids('myATTRIBUTELIST')`

maps the given attributes in 'myATTRIBUTELIST' to a list of their unequivocal OIDs. If the mapping fails, an empty list is returned.

map_cn2dn(*self, cn, searchbase=None, scope=2*)

`thisClass.map_cn2dn ('myCN')`

transforms the given common name 'myCN' - if unique - to the corresponding DN in the LDAP tree. The search is started at base DN `thisClass.BaseDN`.

map_dn2ufn(*self, dn, searchbase=None*)

`thisClass.map_dn2ufn ('myDN')`:

tries to map a given 'myDN' to a user-friendly DN. This method is taken directly from `python-ldap`.

map_gid2dn(*self*, *gid*, *searchbase=None*, *scope=2*)

thisClass.map_gid2dn ('myGID')

transforms the given gid 'myGID' - if unique - to the corresponding DN in the LDAP tree. The search is started at base DN thisClass.BaseDN.

map_gid2gidNumber(*self*, *gid*)

thisClass.map_gid2gidNumber ('myGID')

transforms the given gid 'myGID' to the corresponding gidNumber. The search is started at base DN thisClass.BaseDN.

map_gidNumber2gid(*self*, *gid_number*)

thisClass.map_gidNumber2gid ('myGIDNUMBER')

transforms the given gidNumber 'myGIDNUMBER' to the corresponding gid. The search is started at base DN thisClass.BaseDN.

map_objectclass2attr(*self*, *objectclass*)

thisClass.map_objectclass2attr ('myOBJECT_CLASS')

returns all attributes that are mentioned in the server's LDAP schema for objectClass 'myOBJECT_CLASS'.

map_oid2attrtypes(*self*, *oid*)

thisClass.map_oid2attrtypes('myOID')

maps the given OID 'myOID' to a list of their possible ATTRIBUTES according to the LDAP server's LDAP schema. If the mapping fails, an empty list is returned.

map_ou2dn(*self*, *ou*, *searchbase=None*, *scope=2*)

thisClass.map_ou2dn ('myOU', 'mySUBTREE')

transforms the given organizationalUnit 'myOU' - if unique - to the corresponding DN in the LDAP tree. The search is started at base DN thisClass.BaseDN.

map_uid2dn(*self*, *uid*, *searchbase=None*, *scope=2*)

thisClass.map_uid2dn ('myUID')

transforms the given uid 'myUID' - if unique - to the corresponding DN in the LDAP tree. The search is started at base DN thisClass.BaseDN.

map_uid2posixgroups(*self*, *uid*, *searchbase=None*, *scope=2*)

thisClass.map_uid2posixgroups ('myUID',['mySUBTREE'])

find all ldap groups that the given uid 'myUID' is a posix member of. As a result the method returns a list of posixGroup DNs. The search is started at base DN thisClass.BaseDN if not otherwise specified in 'mySUBTREE'.

map_uid2uniquemembergroups(*self*, *uid*, *searchbase=None*, *user_searchbase=None*, *group_searchbase=None*, *scope=2*)

thisClass.map_uid2uniquemembergroups ('myUID',['mySUBTREE'])

find all groups that the given uid 'myUID' is a unique member of. As a result the method returns a list of posixGroup DNs. The search is started at base DN thisClass.BaseDN if not otherwise specified in 'mySUBTREE'.

map_uidNumber2uid(*self*, *uid_number*)

thisClass.map_uidNumber2uid ('myUIDNUMBER')

transforms the given uidNumber 'myUIDNUMBER' to the corresponding uid. The search is started at base DN thisClass.BaseDN.

refresh_cache(*self*)

dummy method, will be used in classes easyLDAP_object, easyLDAP_tree, etc.

search(*thisClass*, *filter='MyFILTER'*, *scope=myLDAPSCOPE*, *base='myLDAPSEARCHBASEDN'*)

performs a raw ldap.search request and returns the DNs of objects the search filter applies to.

show_objectclass(*self*, *this_objectclass*)

thisClass.show_objectclass ('myOBJECT_CLASS'):

displays an objectClass's schema in a user-friendly manner...

uid_bind(*self*, *bind_uid*, *bind_pw*, *refresh_cache=True*)

thisClass.uid_bind('myUID', 'myPW')

binds to the LDAP server. This method tries to find the UID's corresponding DN in the LDAP tree starting at thisClass.BaseDN.

Binding by UID will fail if the given UID 'myUID' is not unique to the specified LDAP tree.

uid_bind_norefresh(*self*, *bind_uid*, *bind_pw*)

thisClass.uid_bind_norefresh('myUID', 'myPW')

as the easyLDAP class does not load any objects into the easyLDAP cache, this method is just an alias for this_class.bind()

4 Module *easyLDAP.easyLDAP_class.cache*

4.1 Variables

Name	Description
EASY_LDAP	Value: {'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ...}
__package__	Value: 'easyLDAP'
except_dict	Value: {'CANNOT_DELETE_BASEDN_OBJECT': 'The LDAP server\'s base ...}
key	Value: 'NO_PARENTOBJECT_FOR_GIVEN_DN'

4.2 Class *easyLDAP.cache.history*

Known Subclasses: *easyLDAP.easyLDAP_class.cache.easyLDAP.cacheobject*, *easyLDAP.easyLDAP_class.cache.easyLDAP...*

4.2.1 Methods

```
__init__(self, config_defaults={'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ...})
```

```
__deepcopy__(self, memo)
```

Creates a deepcopy of the *easyLDAP* cache class, especially of the undo/redo stack.

```
clear_cache_history(thisClass)
```

clears the *easyLDAP* cache. This method can be used to completely change an existing LDAP object or in order to erase the object entirely from the LDAP directory. Changes will, of course, only be synced to the LDAP directory, when the method *thisClass.flush.cache()* is called.

```
resize_cache_history(thisClass, mySIZE)
```

sets a new size for the *easyLDAP* undo stack. If the new given stack size *mySIZE* is less than the former one, the undo stack will be reduced with the next *thisClass.push()* operation.

```
undo(thisClass)
```

undoes the last modification to the *easyLDAP* object cache.

```
redo(thisClass)
```

redoes the former modification to the *easyLDAP* object cache (if any).

```
enable_cache_history(self)
```

disable_cache_history(*self*)

4.2.2 Class Variables

Name	Description
<code>ldap_cachetree</code>	Value: None
<code>ldap_cache_undo_history</code>	Value: []
<code>ldap_cache_redo_history</code>	Value: []
<code>ldap_cache_history_size</code>	Value: 50
<code>use_cache_history</code>	Value: True

4.3 Class *easyLDAP_cacheobject*

`easyLDAP.easyLDAP_class_cache.easyLDAP_cache_history` — `easyLDAP.easyLDAP_class_cache.easyLDAP_cacheobject`

An *easyLDAP_cacheobject* stores and accesses the python-ldap like LDAP object dictionary.

4.3.1 Methods

__init__(*self*, *ldap_cacheobject*, *use_cache_history*=True, *config_defaults*={'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ...})
 Overrides: `easyLDAP.easyLDAP_class_cache.easyLDAP_cache_history.__init__`

__deepcopy__(*self*, *memo*)
 Creates a deepcopy of the *easyLDAP* cache class, especially of the undo/redo stack.
 Overrides: `easyLDAP.easyLDAP_class_cache.easyLDAP_cache_history.__deepcopy__`
 extit(inherited documentation)

__len__(*self*)

__call__(*self*)

get_cacheobject(*self*)

get_cacheobject_dn(*self*)

clear_cacheobject(*self*)

set_cacheobject(*self*, *ldap_cacheobject*)

```
set_cacheobject_dn(self, dn)
```

```
get_cacheobject_attrdesc(self, attrdesc, ignore_case=False)
```

```
has_cacheobject_attrdesc_set(self, attrdesc, ignore_case=False)
```

```
set_cacheobject_attrdesc(self, attrdesc, values)
```

```
del_cacheobject_attrdesc(self, attrdesc)
```

```
get_cacheobject_attrtype(self, attrtype)
```

```
has_cacheobject_attrtype_set(self, attrtype)
```

```
del_cacheobject_attrtype(self, attrtype)
```

```
has_cacheobject_attrdesc_value_set_in_values(self, attrdesc, value)
```

```
has_cacheobject_attrdesc_values_set(self, attrdesc, values)
```

```
add_cacheobject_attrdesc_values(self, attrdesc, values)
```

```
del_cacheobject_attrdesc_values(self, attrdesc, values, ignore_case=False)
```

Inherited from easyLDAP.easyLDAP_class_cache.easyLDAP_cache_history (Section 4.2)

`clear_cache_history()`, `disable_cache_history()`, `enable_cache_history()`, `redo()`, `resize_cache_history()`, `undo()`

4.3.2 Class Variables

Name	Description
<code>ldap.cacheobject</code>	Value: <code>['', {}]</code>
<code>ldap.cacheobject_dn</code>	Value: <code>''</code>
<i>Inherited from easyLDAP.easyLDAP_class_cache.easyLDAP_cache_history (Section 4.2)</i>	
<code>ldap.cache_history_size</code> , <code>ldap.cache_redo_history</code> , <code>ldap.cache_undo_history</code> , <code>ldap.cachetree</code> , <code>use_cache_history</code>	

4.4 Class *easyLDAP_cachetree*

easyLDAP.easyLDAP_class_cache.easyLDAP_cache_history — *easyLDAP.easyLDAP_class_cache.easy*

An *easyLDAP_cachetree* object stores and accesses the python-ldap like LDAP object dictionary.

4.4.1 Methods

```
__init__(self, ldap_cachetree_linear_or_recursive, basedn=None,
use_cache_history=True, config_defaults={'AdminRDN': 'cn=admin',
'AutomountRDN': 'ou=automount', ...})
```

Overrides: *easyLDAP.easyLDAP_class_cache.easyLDAP_cache_history.__init__*

```
__len__(self)
```

```
__deepcopy__(self, memo)
```

Creates a deepcopy of the *easyLDAP* cache class, especially of the undo/redo stack.

Overrides:

easyLDAP.easyLDAP_class_cache.easyLDAP_cache_history.__deepcopy__
exitit(inherited documentation)

```
__call__(self, linear=False, retrieve_data=False)
```

```
__delitem__(self, dn)
```

```
is_ldap_cachetree(self, cachetree)
```

```
get_cachetree_basedn(self)
```

```
set_cachetree_basedn(self, new_basedn)
```

```
is_childdn_of_dn(self, child_dn, parent_dn)
```

```
is_childdn_of_cachetree_basedn(self, child_dn)
```



```
is_childdn_of_ldapserver_basedn(self, child_dn)
```

```
strip_off_dn_from_dn(self, parent_dn, child_dn)
```

```
get_cachetree(self, linear=False, retrieve_data=False, hashed=False)
```

retrieve_data works only on linear operations

```
get_cachetree_hash(self, retrieve_data=False)
```

```
get_cachetree_pyldap(self)
```

```
get_cachetree_dnlist(self)
```

```
get_cachetree_depth(self)
```

```
get_cachetree_subtree(self, dn, deepcopy=False)
```

```
get_cachetree_parenttree(self, dn, deepcopy=False)
```

```
get_cachetree_subtree_data(self, dn)
```

```
has_dn(self, dn)
```

```
has_children(self, dn)
```

```
get_cacheobject_from_cachetree(self, object_dn)
```

```
get_cacheobject_from_cachetree('DN')
```

returns the data of a single cache object from the cached cachetree.

```
remove_cacheobject_from_cachetree(self, object_dn)
```

```
has_parent_dn(self, dn)
```

```
new_cachetree_object(self, object_data)
```

```
del_cachesubtree_from_cachetree(self, subtree_basedn)
```

```
move_cachesubtree_in_cachetree(self, from_dn, to_dn)
```

Inherited from `easyLDAP.easyLDAP_class_cache.easyLDAP_cache_history` (Section 4.2)

`clear_cache_history()`, `disable_cache_history()`, `enable_cache_history()`, `redo()`, `resize_cache_history()`, `undo()`

4.4.2 Class Variables

Name	Description
<code>ldap_server_basedn</code>	Value: ''
<code>ldap_cachetree_basedn</code>	Value: None
<i>Inherited from <code>easyLDAP.easyLDAP_class_cache.easyLDAP_cache_history</code> (Section 4.2)</i>	
<code>ldap_cache_history_size</code> , <code>ldap_cache_redo_history</code> , <code>ldap_cache_undo_history</code> , <code>ldap_cachetree</code> , <code>use_cache_history</code>	

5 Module *easyLDAP.easyLDAP_class_object_base*

5.1 Variables

Name	Description
EASY_LDAP	Value: {'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ...}
SCHEMA_ATTRS	Value: []
SCHEMA_CLASS_MAPPING	Value: {}
--package--	Value: 'easyLDAP'
easyLDAP_ADDATTR	Value: 0
easyLDAP_REPLACEATTR	Value: 1
except_dict	Value: {'CANNOT_DELETE_BASEDN_OBJECT': 'The LDAP server\'s base ...}
key	Value: 'NO_PARENTOBJECT_FOR_GIVEN_DN'

5.2 Class *easyLDAP_object*

easyLDAP.easyLDAP_class_base.easyLDAP  easyLDAP.easyLDAP_class_object_base.easyLDAP

5.2.1 Methods

```
__init__(self, dn_or_cacheobject, bind_dn=None, bind_pw=None,
config_defaults={'AdminRDN': 'cn=admin', 'AutomountRDN':
'ou=automount', ..., update_schema_cache=False, use_cache_history=True,
ldapsession=None)
```

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.__init__*

```
__deepcopy__(self, memo)
```

Creates a deepcopy of the *easyLDAP* object class.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.__deepcopy__*

```
__call__(self)
```

```
set_object_dn(self, dn)
```

add_attrdesc_values(*self*, *attrdesc*, *values*)**get_object**(*self*)**get_parent_dn**(*self*)*thisClass*.get_parent_dn ('myDN')

returns the parent absolute DN of 'myDN' from the LDAP server's directory tree.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.get_parent_dn*
exitit(inherited documentation)

get_object_dn(*self*)**get_object_rdn**(*thisClass*)

returns the relative DN of the cached *easyLDAPobject* (i.e. the first portion of the object's DN).

get_used_attrdescs_aliases(*self*, *attrdescs*, *capitalize=True*)**get_attrdesc_values**(*self*, *attrdesc*)**get_attrtype_values**(*self*, *attrtype*)*get_attrtype_values*('myATTRTYPE')

returns a dictionary that contains all attribute descriptions in the *easyLDAP* object cache that contain the attribute type 'myATTRTYPE'.

This method is handy for e.g. viewing all attribute type's translations.

On failure, the method returns an empty dictionary.

get_used_attrtypes(*self*, *pyldapobject*=None)

returns a list of attribute types that are used in the current easyLDAP object. If - by some reason - the cache is empty, an empty list is returned.

Another LDAP object than the one currently cached can be analyzed by passing 'mySINGLELDAPOBJECT'. The lengths of the passed LDAP object has to be 1.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.get_used_attrtypes*
 extit(inherited documentation)

get_used_attroptions(*self*, *attrdesc*)

thisClass.get_used_attroptions('myATTRDESC', [*mySINGLELDAPOBJECT*])

returns a list of all attribute options used with attribute type same as in 'myATTRDESC'. The comparison is performed on an OID basis. So, if an alias of an 'myATTRDESC' is used, it will be taken into account.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.get_used_attroptions*
 extit(inherited documentation)

get_used_objectclasses(*thisClass*, *mySINGLELDAPOBJECT*=...)

returns a list of objectClasses that are used in the current easyLDAP object. If - by some reason - the cache is empty, an empty list is returned.

Another LDAP object than the one currently cached can be analyzed by passing 'mySINGLELDAPOBJECT'. The lengths of the passed LDAP object has to be 1.

get_used_attrdescs(*thisClass*)

returns a list of all attribute descriptions that are used in the current easyLDAP object.

has_objectclass_set(*self*, *objectclass*)

thisClass.has_objectclass_set('myOBJECTCLASS'):

returns True, if the objectClass 'myOBJECTCLASS' is set in the cached easyLDAP object.

has_attrtype_set(*self*, *attrtype*)

thisClass.has_attrtype_set('myATTRTYPE'):

returns True, if the attribute type 'myATTRTYPE' is set in the cached easyLDAP object. The attribute type is checked on an OID basis.

has_attrdesc_set(*self*, *attrdesc*)

thisClass.has_attrdesc_set('myATTRDESC'):

returns True, if the attribute 'myATTRDESC' is set in the cached easyLDAP object. The attribute type is checked on an OID basis.

has_value(*self*, *value*)

thisClass.has_value('myVALUE'):

returns True, if the value 'myVALUE' is set in the cached easyLDAP object, regardless from the attribute it is set for.

has_attrdesc_values_set(*self*, *attrdesc*, *values*)

thisClass.has_attrdesc_values_set('myATTRDESC', 'myVALUE'):

returns True, if the list of values 'myVALUES' is identical to those found in the cached easyLDAP object's attribute description 'myATTRDESC'.

has_attrdesc_value_set_in_values(*self*, *attrdesc*, *value*)

thisClass.has_attrdesc_value_set_in_values('myVALUE', 'myATTRDESC'):

returns True, if the value 'myVALUE' is set in the cached easyLDAP object's attribute description 'myATTRDESC' (amongst other possible values).

refresh_cache(*thisClass*)

loads/refreshes the easyLDAP object cache. This method should be the only easyLDAP module code that performs a read operation on the actual LDAP directory server.

BEWARE: This method is called, when *thisClass* is initiated.

If the given DN (at `__init__` time) exists in the LDAP directory, this method:

- reads the DN's data LDAP directory from the LDAP directory,
- checks and repairs objectClass dependencies violations,
- moves the data in the easyLDAP cache.

If the given DN is new to the LDAP directory, this method:

- creates an empty easyLDAP object cache,
- marks the object as a new object, so the first `thisClass.flush_cache()` operation will be `'ldap.add_s'` as opposed to `'ldap.modify_s'`

Once the easyLDAP cache is filled with the DN data all modification will be performed in the easyLDAP object cache. Once you are done, use `thisClass.flush_cache()` to update the LDAP directory's data.

WARNING: this method tries to repair buggy LDAP objects. If it finds an attribute that does not have an objectClass set, the method will try to find a usable objectClass (it will take the first it finds, which might not be wanted). If there is no such objectClass in the server's LDAP schema, the invalid attributes will be purged from the object. This is a repair functionality and should not come to use...

Overrides: `easyLDAP.easyLDAP_class_base.easyLDAP.refresh_cache`

map_diff2modlist(*thisClass*)

returns a list of changes that have been applied to the easyLDAP object chache. The format of this list is conformant with the modlist structure in python-ldap that is used for the methods

`ldap.modify()` / `ldap.modify_s()`

map_reverse_diff2modlist(*thisClass*)

returns a list of changes applied to the easyLDAP object cache that these changes can be undone with. The format of this list is conformant with the modlist structure in python-ldap that is used for the methods

ldap.modify() / ldap.modify_s()

map_diff2ldif(*thisClass*)

returns the changes applied to the easyLDAP object cache in ldif format. The method returns a python list in which each item represents a line of the text base ldif format.

map_reverse_diff2ldif(*thisClass*)

returns ldif output that can undo the changes applied to the easyLDAP object cache. The method returns a python list in which each item represents a line of the text base ldif format.

flush_cache(*thisClass*)

writes the easyLDAP object cache to the LDAP directory. This method should be the only easyLDAP module code that performs a write operation on the actual LDAP directory server.

If the given DN existed in the LDAP directory at `__init__` time, this method:

- checks, if the cached DN still exists in the LDAP directory (otherwise, it switches to object creation mode).
- flushes modifications and additions to the easyLDAP object cache to the server's LDAP directory.

If the given DN was new to the LDAP directory at `__init__` time, this method:

- checks, if the cached DN is still new to the LDAP directory (otherwise, it switches to object modification mode).
- checks, if the underlying DN still exists in the directory. If not the `flush_cache()` operation will fail.
- adds easyLDAP object cache to the LDAP directory.

If the easyLDAP object cache does not need a `flush_cache()` operation (as it is up-to-date), the flush will not be performed (i.e. fail).

add_objectclasses(*self*, *objectclasses*)

```
thisClass.add_objectclass('myOBJECTCLASS'|['myOBJECTCLASS1',  
                                           'myOBJECTCLASS2', ...])
```

adds one ore more objectClass(es) and all needed SUPERior objectClasses to the cached easyLDAP object. Attributes that - according to the server's LDAP schema - MUST be present will be set to default values regarding their proposed syntaxes in thisClass.LDAPSyntaxes.

The default values are stored in thisClass.templateLDAPSyntaxValues.

get_used_attrtype_oids(*thisClass*)

returns a list of those attribute OIDs, that are used the LDAP object 'mySINGLEOBJECT'. This list is taken from the server's LDAP schema.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.get_used_attrtype_oids*

has_oid_set(*self*, *oid*, *pyldapobject*=None)

```
thisClass.has_oid_set('myOID')
```

checks if the given OID 'myOID' is set in the LDAP object 'mySINGLELDAPOBJECT'. If so, the method's result is True.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.has_oid_set*

del_objectclass(*self*, *del_these_objectClasses*)

```
thisClass.del_objectclass('myOBJECTCLASS'|['myOBJECTCLASS1',  
                                           'myOBJECTCLASS2', ...])
```

deletes one ore more objectClass(es) from the cached easyLDAP object. All attributes that only occur in the deleted objectClass(es) are also removed - so be careful!

The method will abort ...

if one of the given objectClass names are not set in the easyLDAP object cache.

The method ignores ...

objectClasses that are to be deleted, but are needed as a SUPerior objectClass by another objectClass.

del_attrdesc_values(*self*, *attrdesc*, *value*, *ignore_case=False*, *by_oid=False*)

```
thisClass.del_attrdesc_value('myATTRDESC', 'myVALUE'  
ignore_case={True|FALSE})
```

deletes the value 'myVALUE' from the LDAP attribute description 'myATTRDESC' in the currently cached object.

If 'myVALUE' is the only value for 'myATTRDESC', the whole attribute is removed from the cached easyLDAP object.

The method returns True on successful deletion.

del_attrdesc(*self*, *attrdesc*, *by_oid=False*)

```
thisClass.del_attrdesc('myATTRDESC')
```

deletes the whole entry for attribute description 'myATTRDESC' from the cached easyLDAP object.

The method returns True on successful deletion.

del_attrtype(*self*, *attrtype*, *by_oid=False*)

`thisClass.del_attrtype('myATTRTYPE')`

deletes all entries for attribute type 'myATTRTYPE' from the cached easyLDAP object. Be aware that all subentries of the attribute type 'myATTRTYPE' will be removed (e.g. all language options with one shot).

The method returns True on successful deletion.

set_attrdesc(*self*, *attrdesc*, *value*)

`thisClass.set_attr('myLDAP_ATTRDESC',
 'myVALUE'|['myVALUE1', 'myVALUE2', ...])`

is a wrapper for

`thisClass.modify_attrdesc('myLDAP_ATTRDESC',
 'myVALUE'|['myVALUE1', 'myVALUE2', ...],
 easyLDAP_REPLACEATTR)`

or

`thisClass.del_attrdesc('myLDAP_ATTRIBUTE')`

depending on the attribute description's context...

```
set_attrtype(self, attrtype_dict)
```

```
thisClass.set_attr('myATTRDESCDICT')
```

helps copy+pasting a whole attribute type from one easyLDAP object into another easyLDAP object.

Little HOWTO:

- o connect to two easyLDAP objects. Let's call them „source" and „target".
- o retrieve all attribute descriptions (including all subentries) of a certain attribute type 'myATTRTYPE' from easyLDAPobject „source" with

```
my_attrtype = source.get_attrtype_values('myATTRTYPE')
```

- o paste the retrieved data (a python dictionary) into easyLDAPobject „target" with

```
target.set_attrtype(my_attrtype)
```

- o flush the easyLDAP object cache of „target" with

```
target.flush_cache()
```

If necessary objectClasses can be detected automagically they will be added silently. If not, the method returns a list of possible objectClasses that all provide the attribute type of your interest.

The structure of myATTRDESCDICT is expected to be an excerpt from a python-ldap object

```
{
  'attrtype;attropt1': ['val1','val2'],
  'attrtype;attropt2': [...],
  ...
}
```

Keys of myATTRDESCDICT must always have the same attribute type. The attribute options must be supported by LDAPv3 protocol. Attribute description values must be given as multi-valued or single-valued lists (depending on the attribute's properties in the server's schema).

add_attrdesc(*self*, *attrdesc*, *value*)

```
thisClass.add_attrdesc('myLDAP_ATTRDESC',  
    'myVALUE'|['myVALUE1', 'myVALUE2', ...])
```

is a wrapper for

```
thisClass._modify_attrdesc('myLDAP_ATTRDESC',  
    'myVALUE'|['myVALUE1', 'myVALUE2', ...],  
    easyLDAP_ADDATTR)
```

push_cache_history(*self*)**show_cache**(*thisClass*)

will display the cached easyLDAP object just like the console tool 'ldapsearch' does... unset but possible attributes are not listed.

clear_cache(*self*)

new_object(*self*, *objectclasses*, *attrdesc_dict*)

```
thisClass.new_object(['myOBJ_CLASS1', 'myOBJ_CLASS2'],
    {
        'myATTR1': ['myVAL1_1', 'myVAL1_2', ...],
        'myATTR2': ['myVAL2_1', 'myVAL2_2', ...], '
        ...
    }
)
```

creates an empty object in the easyLDAP object cache. The object will only be uploaded to the LDAP directory, until the method `thisClass.flush_cash()` is evoked.

This action overwrites the current easyLDAP object cache and creates a new cache object if the used DN was not existent in the LDAP directory at the time of `thisClass.__init__` ('myDN').

This newly created object can be modified by several easyLDAP methods. Use `dir(thisClass)`.

To view the contents of the new easyLDAP object cache, use `thisClass.show()`.

undo(*self*)**redo**(*self*)

```
set_userPassword(self, pw=None, HASH='CRYPT')
```

```
thisClass.set_userPassword('myUNIXPASSWORD', 'myHASH')
```

sets the user's unix password to 'myUNIXPASSWORD'.
As encryption hash 'myHASH' you can additionally pass

```
CLEAR
```

```
CRYPT
```

The default hash is 'CRYPT'.

If the easyLDAP object has the shadowAccount objectclass set, the attribute shadowLastChange will be set to the current date (in days since 1970).

```
set_naming_attrdesc(self, naming_attrdesc)
```

```
is_naming_attrdesc(self, naming_attrdesc)
```

Inherited from *easyLDAP.easyLDAP_class_base.easyLDAP* (Section 3.2)

```
__del__(), adminbind(), adminbind_norefresh(), anonymous_bind(), anonymous_bind_norefresh(),
attrtype_needs_objectclass(), bind(), bind_norefresh(), check_objectclass_dependencies(),
cn_bind(), cn_bind_norefresh(), find_newGidNumber(), find_newUidNumber(), generate_attrtypes_dict(),
generate_ldapsyntaxes_dict(), generate_objectclass_dict(), generate_reverse_attrtypes_dict(),
get_admin_dn(), get_attrtypes(), get_attrtype_aliases(), get_basedn(), get_cache_reference(),
get_groups_basedn(), get_hosts_basedn(), get_objectclasses(), get_parent_rdn(), get_people_basedn(),
get_samba3_algorithmicRidBase(), get_samba3_domainSID(), get_samelevel_dns(), get_sublevel_dns(),
get_sublevel_rdns(), get_subtree_dns(), has_dn(), has_parent_dn(), has_samba2_schema(),
has_samba3_schema(), has_valid_attrtype(), is_collectiveattr(), is_nousermodattr(),
is_obsoleteattr(), is_pyldapobject(), is_pyldaptree(), is_singlevalueattr(), is_valid_attrtype(),
is_valid_objectclass(), map_attrtype2ldapsyntax(), map_attrtype2ldapsyntaxname(),
map_attrtype2ldapsyntaxoid(), map_attrtype2objectclasses(), map_attrtype2valuemaxlen(),
map_attrtypes2oids(), map_cn2dn(), map_dn2ufn(), map_gid2dn(), map_gid2gidNumber(),
map_gidNumber2gid(), map_objectclass2attr(), map_oid2attrtypes(), map_ou2dn(),
map_uid2dn(), map_uid2posixgroups(), map_uid2uniquemembergid(), map_uidNumber2uid(),
search(), show_objectclass(), uid_bind(), uid_bind_norefresh()
```

5.3 Class *easyLDAP_object*

easyLDAP.easyLDAP_class_base.easyLDAP — *easyLDAP.easyLDAP_class_object_base.easyLDAP*

5.3.1 Methods

```
__init__(self, dn_or_cacheobject, bind_dn=None, bind_pw=None,
config_defaults={'AdminRDN': 'cn=admin', 'AutomountRDN':
'ou=automount', ..., update_schema_cache=False, use_cache_history=True,
ldapsession=None)
```

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.__init__*

```
__deepcopy__(self, memo)
```

Creates a deepcopy of the *easyLDAP* object class.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.__deepcopy__*

```
__call__(self)
```

```
set_object_dn(self, dn)
```

```
add_attrdesc_values(self, attrdesc, values)
```

```
get_object(self)
```

```
get_parent_dn(self)
```

thisClass.get_parent_dn ('myDN')

returns the parent absolute DN of 'myDN' from the LDAP server's directory tree.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.get_parent_dn*
exitit(inherited documentation)

```
get_object_dn(self)
```


get_object_rdn(*thisClass*)

returns the relative DN of the cached easyLDAPobject (i.e. the first portion of the object's DN).

get_used_attrdescs_aliases(*self*, *attrdescs*, *capitalize=True*)**get_attrdesc_values**(*self*, *attrdesc*)**get_attrtype_values**(*self*, *attrtype*)

get_attrtype_values('myATTRTYPE')

returns a dictionary that contains all attribute descriptions in the easyLDAP object cache that contain the attribute type 'myATTRTYPE'.

This method is handy for e.g. viewing all attribute type's translations.

On failure, the method returns an empty dictionary.

get_used_attrtypes(*self*, *pyldapobject=None*)

returns a list of attribute types that are used in the current easyLDAP object. If - by some reason - the cache is empty, an empty list is returned.

Another LDAP object than the one currently cached can be analyzed by passing 'mySINGLELDAPOBJECT'. The length of the passed LDAP object has to be 1.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.get_used_attrtypes*
extit(inherited documentation)

get_used_attroptions(*self*, *attrdesc*)

thisClass.get_used_attroptions('myATTRDESC', [mySINGLELDAPOBJECT])

returns a list of all attribute options used with attribute type same as in 'myATTRDESC'. The comparison is performed on an OID basis. So, if an alias of an 'myATTRDESC' is used, it will be taken into account.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.get_used_attroptions*
extit(inherited documentation)

get_used_objectclasses(*thisClass*, *mySINGLELDAPOBJECT*=...)

returns a list of objectClasses that are used in the current easyLDAP object. If - by some reason - the cache is empty, an empty list is returned.

Another LDAP object than the one currently cached can be analyzed by passing 'mySINGLELDAPOBJECT'. The lengths of the passed LDAP object has to be 1.

get_used_attrdescs(*thisClass*)

returns a list of all attribute descriptions that are used in the current easyLDAP object.

has_objectclass_set(*self*, *objectclass*)

thisClass.has_objectclass_set('myOBJECTCLASS'):

returns True, if the objectClass 'myOBJECTCLASS' is set in the cached easyLDAP object.

has_attrtype_set(*self*, *attrtype*)

thisClass.has_attrtype_set('myATTRTYPE'):

returns True, if the attribute type 'myATTRTYPE' is set in the cached easyLDAP object. The attribute type is checked on an OID basis.

has_attrdesc_set(*self*, *attrdesc*)

thisClass.has_attrdesc_set('myATTRDESC'):

returns True, if the attribute 'myATTRDESC' is set in the cached easyLDAP object. The attribute type is checked on an OID basis.

has_value(*self*, *value*)

thisClass.has_value('myVALUE'):

returns True, if the value 'myVALUE' is set in the cached easyLDAP object, regardless from the attribute it is set for.

has_attrdesc_values_set(*self*, *attrdesc*, *values*)

thisClass.has_attrdesc_values_set('myATTRDESC', 'myVALUE'):

returns True, if the list of values 'myVALUES' is identical to those found in the cached easyLDAP object's attribute description 'myATTRDESC'.

has_attrdesc_value_set_in_values(*self*, *attrdesc*, *value*)

thisClass.has_attrdesc_value_set_in_values('myVALUE','myATTRDESC'):

returns True, if the value 'myVALUE' is set in the cached easyLDAP object's attribute description 'myATTRDESC' (amongst other possible values).

refresh_cache(*thisClass*)

loads/refreshes the easyLDAP object cache. This method should be the only easyLDAP module code that performs a read operation on the actual LDAP directory server.

BEWARE: This method is called, when thisClass is initiated.

If the given DN (at __init__ time) exists in the LDAP directory, this method:

- reads the DN's data LDAP directory from the LDAP directory,
- checks and repairs objectClass dependencies violations,
- moves the data in the easyLDAP cache.

If the given DN is new to the LDAP directory, this method:

- creates an empty easyLDAP object cache,
- marks the object as a new object, so the first thisClass.flush_cache() operation will be 'ldap.add_s' as opposed to 'ldap.modify_s'

Once the easyLDAP cache is filled with the DN data all modification will be performed in the easyLDAP object cache. Once you are done, use thisClass.flush_cache() to update the LDAP directory's data.

WARNING: this method tries to repair buggy LDAP objects. If it finds an attribute that does not have an objectClass set, the method will try to find a usable objectClass (it will take the first it finds, which might not be wanted). If there is no such objectClass in the server's LDAP schema, the invalid attributes will be purged from the object. This is a repair functionality and should not come to use...

Overrides: easyLDAP.easyLDAP_class_base.easyLDAP.refresh_cache

map_diff2modlist(*thisClass*)

returns a list of changes that have been applied to the easyLDAP object cache. The format of this list is conformant with the modlist structure in python-ldap that is used for the methods

ldap.modify() / ldap.modify_s()

map_reverse_diff2modlist(*thisClass*)

returns a list of changes applied to the easyLDAP object cache that these changes can be undone with. The format of this list is conformant with the modlist structure in python-ldap that is used for the methods

ldap.modify() / ldap.modify_s()

map_diff2ldif(*thisClass*)

returns the changes applied to the easyLDAP object cache in ldif format. The method returns a python list in which each item represents a line of the text base ldif format.

map_reverse_diff2ldif(*thisClass*)

returns ldif output that can undo the changes applied to the easyLDAP object cache. The method returns a python list in which each item represents a line of the text base ldif format.

flush_cache(*thisClass*)

writes the easyLDAP object cache to the LDAP directory. This method should be the only easyLDAP module code that performs a write operation on the actual LDAP directory server.

If the given DN existed in the LDAP directory at `__init__` time, this method:

- checks, if the cached DN still exists in the LDAP directory (otherwise, it switches to object creation mode).
- flushes modifications and additions to the easyLDAP object cache to the server's LDAP directory.

If the given DN was new to the LDAP directory at `__init__` time, this method:

- checks, if the cached DN is still new to the LDAP directory (otherwise, it switches to object modification mode).
- checks, if the underlying DN still exists in the directory. If not the `flush_cache()` operation will fail.
- adds easyLDAP object cache to the LDAP directory.

If the easyLDAP object cache does not need a `flush_cache()` operation (as it is up-to-date), the flush will not be performed (i.e. fail).

add_objectclasses(*self*, *objectclasses*)

```
thisClass.add_objectclass('myOBJECTCLASS'|['myOBJECTCLASS1',  
                                           'myOBJECTCLASS2', ...])
```

adds one or more `objectClass(es)` and all needed `SUPERior objectClasses` to the cached easyLDAP object. Attributes that - according to the server's LDAP schema - MUST be present will be set to default values regarding their proposed syntaxes in `thisClass.LDAPSyntaxes`.

The default values are stored in `thisClass.templateLDAPSyntaxValues`.

get_used_attrtype_oids(*thisClass*)

returns a list of those attribute OIDs, that are used the LDAP object 'mySINGLEOBJECT'. This list is taken from the server's LDAP schema.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.get_used_attrtype_oids*

has_oid_set(*self, oid, pyldapobject=None*)

thisClass.has_oid_set('myOID')

checks if the given OID 'myOID' is set in the LDAP object 'mySINGLELDAPOBJECT'. If so, the method's result is True.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.has_oid_set*

del_objectclass(*self, del_these_objectClasses*)

```
thisClass.del_objectclass('myOBJECTCLASS'|['myOBJECTCLASS1',  
                                             'myOBJECTCLASS2', ...])
```

deletes one ore more *objectClass(es)* from the cached *easyLDAP* object. All attributes that only occur in the deleted *objectClass(es)* are also removed - so be careful!

The method will abort ...

if one of the given *objectClass* names are not set in the *easyLDAP* object cache.

The method ignores ...

objectClasses that are to be deleted, but are needed as a *SUPERior objectClass* by another *objectClass*.

del_attrdesc_values(*self*, *attrdesc*, *value*, *ignore_case=False*, *by_oid=False*)

thisClass.del_attrdesc_value('myATTRDESC', 'myVALUE'
ignore_case={True|FALSE})

deletes the value 'myVALUE' from the LDAP attribute description 'myATTRDESC' in the currently cached object.

If 'myVALUE' is the only value for 'myATTRDESC', the whole attribute is removed from the cached easyLDAP object.

The method returns True on successful deletion.

del_attrdesc(*self*, *attrdesc*, *by_oid=False*)

thisClass.del_attrdesc('myATTRDESC')

deletes the whole entry for attribute description 'myATTRDESC' from the cached easyLDAP object.

The method returns True on successful deletion.

del_attrtype(*self*, *attrtype*, *by_oid=False*)

thisClass.del_attrtype('myATTRTYPE')

deletes all entries for attribute type 'myATTRTYPE' from the cached easyLDAP object. Be aware that all subentries of the attribute type 'myATTRTYPE' will be removed (e.g. all language options with one shot).

The method returns True on successful deletion.

```
set_attrdesc(self, attrdesc, value)
```

```
thisClass.set_attr('myLDAP_ATTRDESC',  
                  'myVALUE'|['myVALUE1', 'myVALUE2', ...])
```

is a wrapper for

```
thisClass.modify_attrdesc('myLDAP_ATTRDESC',  
                          'myVALUE'|['myVALUE1', 'myVALUE2', ...],  
                          easyLDAP_REPLACEATTR)
```

or

```
thisClass.del_attrdesc('myLDAP_ATTRIBUTE')
```

depending on the attribute description's context...


```
set_attrtype(self, attrtype_dict)
```

```
thisClass.set_attr('myATTRDESCDICT')
```

helps copy+pasting a whole attribute type from one easyLDAP object into another easyLDAP object.

Little HOWTO:

- o connect to two easyLDAP objects. Let's call them „source" and „target".
- o retrieve all attribute descriptions (including all subentries) of a certain attribute type 'myATTRTYPE' from easyLDAPobject „source" with

```
my_attrtype = source.get_attrtype_values('myATTRTYPE')
```

- o paste the retrieved data (a python dictionary) into easyLDAPobject „target" with

```
target.set_attrtype(my_attrtype)
```

- o flush the easyLDAP object cache of „target" with

```
target.flush_cache()
```

If necessary objectClasses can be detected automagically they will be added silently. If not, the method returns a list of possible objectClasses that all provide the attribute type of your interest.

The structure of myATTRDESCDICT is expected to be an excerpt from a python-ldap object

```
{
  'attrtype;attropt1': ['val1','val2'],
  'attrtype;attropt2': [...],
  ...
}
```

Keys of myATTRDESCDICT must always have the same attribute type. The attribute options must be supported by LDAPv3 protocol. Attribute description values must be given as multi-valued or single-valued lists (depending on the attribute's properties in the server's schema).

add_attrdesc(*self*, *attrdesc*, *value*)

```
thisClass.add_attrdesc('myLDAP_ATTRDESC',  
    'myVALUE'|['myVALUE1', 'myVALUE2', ...])
```

is a wrapper for

```
thisClass._modify_attrdesc('myLDAP_ATTRDESC',  
    'myVALUE'|['myVALUE1', 'myVALUE2', ...],  
    easyLDAP_ADDATTR)
```

push_cache_history(*self*)**show_cache**(*thisClass*)

will display the cached easyLDAP object just like the console tool 'ldapsearch' does... unset but possible attributes are not listed.

clear_cache(*self*)

new_object(*self*, *objectclasses*, *attrdesc_dict*)

```
thisClass.new_object(['myOBJ_CLASS1', 'myOBJ_CLASS2'],
    {
        'myATTR1': ['myVAL1_1', 'myVAL1_2', ...],
        'myATTR2': ['myVAL2_1', 'myVAL2_2', ...], '
        ...
    }
)
```

creates an empty object in the easyLDAP object cache. The object will only be uploaded to the LDAP directory, until the method `thisClass.flush_cache()` is evoked.

This action overwrites the current easyLDAP object cache and creates a new cache object if the used DN was not existent in the LDAP directory at the time of `thisClass.__init__` ('myDN').

This newly created object can be modified by several easyLDAP methods. Use `dir(thisClass)`.

To view the contents of the new easyLDAP object cache, use `thisClass.show()`.

undo(*self*)**redo**(*self*)

```
set_userPassword(self, pw=None, HASH='CRYPT')
```

```
thisClass.set_userPassword('myUNIXPASSWORD', 'myHASH')
```

sets the user's unix password to 'myUNIXPASSWORD'.
As encryption hash 'myHASH' you can additionally pass

```
CLEAR
CRYPT
```

The default hash is 'CRYPT'.

If the easyLDAP object has the shadowAccount objectclass set, the attribute shadowLastChange will be set to the current date (in days since 1970).

```
set_naming_attrdesc(self, naming_attrdesc)
```

```
is_naming_attrdesc(self, naming_attrdesc)
```

Inherited from easyLDAP.easyLDAP_class_base.easyLDAP(Section 3.2)

```
__del__(), adminbind(), adminbind_norefresh(), anonymous_bind(), anonymous_bind_norefresh(),
attrtype_needs_objectclass(), bind(), bind_norefresh(), check_objectclass_dependencies(),
cn_bind(), cn_bind_norefresh(), find_newGidNumber(), find_newUidNumber(), generate_attrtypes_dict(),
generate_ldapsyntaxes_dict(), generate_objectclass_dict(), generate_reverse_attrtypes_dict(),
get_admin_dn(), get_attrtypes(), get_attrtype_aliases(), get_basedn(), get_cache_reference(),
get_groups_basedn(), get_hosts_basedn(), get_objectclasses(), get_parent_rdn(), get_people_basedn(),
get_samba3_algorithmicRidBase(), get_samba3_domainSID(), get_samelevel_dns(), get_sublevel_dns(),
get_sublevel_rdns(), get_subtree_dns(), has_dn(), has_parent_dn(), has_samba2_schema(),
has_samba3_schema(), has_valid_attrtype(), is_collectiveattr(), is_nousermodattr(),
is_obsoleteattr(), is_pyldapobject(), is_pyldaptree(), is_singlevalueattr(), is_valid_attrtype(),
is_valid_objectclass(), map_attrtype2ldapsyntax(), map_attrtype2ldapsyntaxname(),
map_attrtype2ldapsyntaxoid(), map_attrtype2objectclasses(), map_attrtype2valuemaxlen(),
map_attrtypes2oids(), map_cn2dn(), map_dn2ufn(), map_gid2dn(), map_gid2gidNumber(),
map_gidNumber2gid(), map_objectclass2attr(), map_oid2attrtypes(), map_ou2dn(),
map_uid2dn(), map_uid2posixgroups(), map_uid2uniquemembergid(), map_uidNumber2uid(),
search(), show_objectclass(), uid_bind(), uid_bind_norefresh()
```

6 Module `easyLDAP.easyLDAP_class_tree`

6.1 Variables

Name	Description
<code>EASY_LDAP</code>	Value: <code>{'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ...}</code>
<code>SCHEMA_ATTRS</code>	Value: <code>[]</code>
<code>SCHEMA_CLASS_MAPPING</code>	Value: <code>{}</code>
<code>--package--</code>	Value: <code>'easyLDAP'</code>
<code>easyLDAP_ADDATTR</code>	Value: <code>0</code>
<code>easyLDAP_REPLACEATTR</code>	Value: <code>1</code>
<code>except_dict</code>	Value: <code>{'CANNOT_DELETE_BASEDN_OBJECT': 'The LDAP server\'s base ...}</code>
<code>key</code>	Value: <code>'NO_PARENTOBJECT_FOR_GIVEN_DN'</code>

6.2 Class `easyLDAP_tree`

`easyLDAP.easyLDAP_class_base.easyLDAP` — `easyLDAP.easyLDAP_class_tree.easyLDAP_tree`

6.2.1 Methods

```
__init__(self, dn_or_cachetree, bind_dn=None, bind_pw=None,
filter='objectClass=*', config_defaults={'AdminRDN': 'cn=admin',
'AutomountRDN': 'ou=automount', ..., update_schema_cache=False,
use_cache_history=True, ldapsession=None)
```

Overrides: `easyLDAP.easyLDAP_class_base.easyLDAP.__init__`

```
__len__(self)
```

```
__call__(self, **kwargs)
```

```
__deepcopy__(self, memo)
```

Creates a deepcopy of the `easyLDAP` tree class.

Overrides: `easyLDAP.easyLDAP_class_base.easyLDAP.__deepcopy__`

__delitem__(*self*, *dn*)

refresh_cache(*self*)

dummy method, will be used in classes *easyLDAP_object*, *easyLDAP_tree*, etc.
 Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.refresh_cache*
 extit(inherited documentation)

get_tree(*self*, *linear=False*, *retrieve_data=False*, *hashed=False*)

get_tree('DN')

returns the data of a single ldap object from the cached *easyLDAP* tree.

get_object_from_tree(*self*, *dn*)

has_dn(*self*, *dn*)

thisClass.has_dn ('myDN')

checks, if the given distinguished name 'myDN' exists in the server's LDAP directory. The searches starts with the *thisClass.BaseDN*.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.has_dn* extit(inherited documentation)

get_tree_basedn(*thisClass*)

returns the base DN of the cached *easyLDAPtree*.

get_tree_depth(*self*)

get_tree_dnlist(*self*)

get_tree_hash(*self*)

get_tree_pyldap(*self*)

get_subtree_data(*self*, *dn*)

get_subtree_cachetree(*self*, *dn*)

get_subtree(*self*, *dn*)

has_parent_dn(*self*, *dn*)

thisClass.has_parent_dn ('myDN')

tests the existence of the referred object with a live search request to the LDAP server.

Overrides: easyLDAP.easyLDAP_class_base.easyLDAP.has_parent_dn

new_object(*self*, *object_dn*, *object_data*=None)

remove_object(*self*, *object_dn*)

clear_cache(*self*)

show_cache(*self*)

flush_cache(*self*)

push_cache_history(*self*)

undo(*self*)

redo(*self*)

find_newUidNumber(*self*, *min_uidNumber*=None, *max_uidNumber*=None, *searchbase*=None)

self.find_newUidNumber (minUID,maxUID,myPEOPLEBASEDN)

live searches the connected LDAP server for the lowest vacant uidNumber between minUID and maxUID.

If myPEOPLEBASEDN is not specified, the easyLDAP thisClass.PeoplebaseDN is presumed.

The method only checks vacant uidNumber in the connected LDAP tree. Other sources for posixUserIDs (/etc/passwd, NIS, etc.) are not taken into consideration!

If an error occurs or no vacant uidNumber can be found, the method returns '-1'.

Overrides: easyLDAP.easyLDAP_class_base.easyLDAP.find_newUidNumber

```
find_newGidNumber(self, min_gidNumber=None, max_gidNumber=None,
searchbase=None)
```

searches the connected LDAP server for the lowest vacant gidNumber between minGID and maxGID.

If myGROUPSBASEDN is not specified, the easyLDAP
thisClass.GroupBaseDN is presumed.

The method only checks vacant gidNumber in the connected LDAP tree.
Other sources for posixGroupIDs (/etc/groups, NIS, etc.) are not taken into
consideration!

If an error occurs or no vacant gidNumber can be found, the method returns
'-1'.

Overrides: easyLDAP.easyLDAP_class_base.easyLDAP.find_newGidNumber

Inherited from *easyLDAP.easyLDAP_class_base.easyLDAP* (Section 3.2)

`__del__()`, `adminbind()`, `adminbind_norefresh()`, `anonymous_bind()`, `anonymous_bind_norefresh()`,
`attrtype_needs_objectclass()`, `bind()`, `bind_norefresh()`, `check_objectclass_dependencies()`,
`cn_bind()`, `cn_bind_norefresh()`, `generate_attrtypes_dict()`, `generate_ldapsyntaxes_dict()`,
`generate_objectclass_dict()`, `generate_reverse_attrtypes_dict()`, `get_admin_dn()`,
`get_attrtypes()`, `get_attrtype_aliases()`, `get_basedn()`, `get_cache_reference()`, `get_groups_basedn()`,
`get_hosts_basedn()`, `get_objectclasses()`, `get_parent_dn()`, `get_parent_rdn()`, `get_people_basedn()`,
`get_samba3_algorithmicRidBase()`, `get_samba3_domainSID()`, `get_samelevel_dns()`,
`get_sublevel_dns()`, `get_sublevel_rdns()`, `get_subtree_dns()`, `get_used_attroptions()`,
`get_used_attrtype_oids()`, `get_used_attrtypes()`, `has_oid_set()`, `has_samba2_schema()`,
`has_samba3_schema()`, `has_valid_attrtype()`, `is_collectiveattr()`, `is_nousermodattr()`,
`is_obsoleteattr()`, `is_pyldapobject()`, `is_pyldaptree()`, `is_singlevalueattr()`, `is_valid_attrtype()`,
`is_valid_objectclass()`, `map_attrtype2ldapsyntax()`, `map_attrtype2ldapsyntaxname()`,
`map_attrtype2ldapsyntaxoid()`, `map_attrtype2objectclasses()`, `map_attrtype2valuemaxlen()`,
`map_attrtypes2oids()`, `map_cn2dn()`, `map_dn2ufn()`, `map_gid2dn()`, `map_gid2gidNumber()`,
`map_gidNumber2gid()`, `map_objectclass2attr()`, `map_oid2attrtypes()`, `map_ou2dn()`,
`map_uid2dn()`, `map_uid2posixgroups()`, `map_uid2uniquemembergroups()`, `map_uidNumber2uid()`,
`search()`, `show_objectclass()`, `uid_bind()`, `uid_bind_norefresh()`

6.2.2 Class Variables

Name	Description
SearchFilter	Value: 'objectClass=*

6.3 Class *easyLDAP_tree*

easyLDAP.easyLDAP_class_base.easyLDAP — *easyLDAP.easyLDAP_class_tree.easyLDAP_tree*

6.3.1 Methods

```
__init__(self, dn_or_cachetree, bind_dn=None, bind_pw=None,
        filter='objectClass=*', config_defaults={'AdminRDN': 'cn=admin',
        'AutomountRDN': 'ou=automount', ...}, update_schema_cache=False,
        use_cache_history=True, ldapsession=None)
```

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.__init__*

```
__len__(self)
```

```
__call__(self, **kwargs)
```

```
__deepcopy__(self, memo)
```

Creates a deepcopy of the *easyLDAP* tree class.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.__deepcopy__*

```
__delitem__(self, dn)
```

```
refresh_cache(self)
```

dummy method, will be used in classes *easyLDAP_object*, *easyLDAP_tree*, etc.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.refresh_cache*
 exitit(inherited documentation)

```
get_tree(self, linear=False, retrieve_data=False, hashed=False)
```

```
get_tree('DN')
```

returns the data of a single ldap object from the cached *easyLDAP* tree.

```
get_object_from_tree(self, dn)
```

has_dn(*self*, *dn*)*thisClass*.has_dn ('myDN')

checks, if the given distinguished name 'myDN' exists in the server's LDAP directory. The searches starts with the *thisClass*.BaseDN.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.has_dn* extit(inherited documentation)

get_tree_basedn(*thisClass*)

returns the base DN of the cached *easyLDAPtree*.

get_tree_depth(*self*)**get_tree_dnlist**(*self*)**get_tree_hash**(*self*)**get_tree_pyldap**(*self*)**get_subtree_data**(*self*, *dn*)**get_subtree_cachetree**(*self*, *dn*)**get_subtree**(*self*, *dn*)**has_parent_dn**(*self*, *dn*)*thisClass*.has_parent_dn ('myDN')

tests the existence of the referred object with a live search request to the LDAP server.

Overrides: *easyLDAP.easyLDAP_class_base.easyLDAP.has_parent_dn*

new_object(*self*, *object_dn*, *object_data*=None)**remove_object**(*self*, *object_dn*)**clear_cache**(*self*)

show_cache(*self*)

flush_cache(*self*)

push_cache_history(*self*)

undo(*self*)

redo(*self*)

find_newUidNumber(*self*, *min_uidNumber*=None, *max_uidNumber*=None, *searchbase*=None)

self.find_newUidNumber (minUID,maxUID,myPEOPLEBASEDN)

live searches the connected LDAP server for the lowest vacant uidNumber between minUID and maxUID.

If myPEOPLEBASEDN is not specified, the easyLDAP thisClass.PeoplebaseDN is presumed.

The method only checks vacant uidNumber in the connected LDAP tree. Other sources for posixUserIDs (/etc/passwd, NIS, etc.) are not taken into consideration!

If an error occurs or no vacant uidNumber can be found, the method returns '-1'.

Overrides: easyLDAP.easyLDAP_class_base.easyLDAP.find_newUidNumber

```
find_newGidNumber(self, min_gidNumber=None, max_gidNumber=None,
searchbase=None)
```

searches the connected LDAP server for the lowest vacant gidNumber between minGID and maxGID.

If myGROUPSBASEDN is not specified, the easyLDAP thisClass.GroupBaseDN is presumed.

The method only checks vacant gidNumber in the connected LDAP tree. Other sources for posixGroupIDs (/etc/groups, NIS, etc.) are not taken into consideration!

If an error occurs or no vacant gidNumber can be found, the method returns '-1'.

Overrides: easyLDAP.easyLDAP_class_base.easyLDAP.find_newGidNumber

Inherited from *easyLDAP.easyLDAP_class_base.easyLDAP* (Section 3.2)

`__del__()`, `adminbind()`, `adminbind_norefresh()`, `anonymous_bind()`, `anonymous_bind_norefresh()`, `attrtype_needs_objectclass()`, `bind()`, `bind_norefresh()`, `check_objectclass_dependencies()`, `cn_bind()`, `cn_bind_norefresh()`, `generate_attrtypes_dict()`, `generate_ldapsyntaxes_dict()`, `generate_objectclass_dict()`, `generate_reverse_attrtypes_dict()`, `get_admin_dn()`, `get_attrtypes()`, `get_attrtype_aliases()`, `get_basedn()`, `get_cache_reference()`, `get_groups_basedn()`, `get_hosts_basedn()`, `get_objectclasses()`, `get_parent_dn()`, `get_parent_rdn()`, `get_people_basedn()`, `get_samba3_algorithmicRidBase()`, `get_samba3_domainSID()`, `get_samelevel_dns()`, `get_sublevel_dns()`, `get_sublevel_rdns()`, `get_subtree_dns()`, `get_used_attroptions()`, `get_used_attrtype_oids()`, `get_used_attrtypes()`, `has_oid_set()`, `has_samba2_schema()`, `has_samba3_schema()`, `has_valid_attrtype()`, `is_collectiveattr()`, `is_nousermodattr()`, `is_obsoleteattr()`, `is_pyldapobject()`, `is_pyldaptree()`, `is_singlevalueattr()`, `is_valid_attrtype()`, `is_valid_objectclass()`, `map_attrtype2ldapsyntax()`, `map_attrtype2ldapsyntaxname()`, `map_attrtype2ldapsyntaxoid()`, `map_attrtype2objectclasses()`, `map_attrtype2valuemaxlen()`, `map_attrtypes2oids()`, `map_cn2dn()`, `map_dn2ufn()`, `map_gid2dn()`, `map_gid2gidNumber()`, `map_gidNumber2gid()`, `map_objectclass2attr()`, `map_oid2attrtypes()`, `map_ou2dn()`, `map_uid2dn()`, `map_uid2posixgroups()`, `map_uid2uniquemembergroups()`, `map_uidNumber2uid()`, `search()`, `show_objectclass()`, `uid_bind()`, `uid_bind_norefresh()`

6.3.2 Class Variables

Name	Description
SearchFilter	Value: 'objectClass=*

7 Module `easyLDAP.easyLDAP_defaults`

Default settings for easyLDAP.

7.1 Functions

`print_easyLDAP_defaults()`

Prints out a beautiful overview on the content of the EASY_LDAP dictionary.

The EASY_LDAP dictionary is used to pre-define the contact information for your LDAP server before calling any of the `easyLDAP_*` constructors.

7.2 Variables

Name	Description
EASY_LDAP	EASY_LDAP contains all the easyLDAP default values. Before calling an <code>easyLDAP_*</code> constructor, make sure EASY_LDAP contains all the information needed to contact your LDAP server. Value: <code>{ 'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ... }</code>
<code>__package__</code>	Value: <code>'easyLDAP'</code>

8 Module `easyLDAP.easyLDAP_etc`

Make the easyLDAP configuration folder available to Python.

8.1 Variables

Name	Description
<code>--package--</code>	Value: <code>'easyLDAP'</code>

9 Module `easyLDAP.easyLDAP_exceptions`

9.1 Variables

Name	Description
<code>EASY_LDAP</code>	Value: <code>{'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ...}</code>
<code>__package__</code>	Value: <code>'easyLDAP'</code>
<code>except_dict</code>	Value: <code>{'CANNOT_DELETE_BASEDN_OBJECT': 'The LDAP server\'s base ...}</code>
<code>key</code>	Value: <code>'NO_PARENTOBJECT_FOR_GIVEN_DN'</code>

9.2 Class `easyLDAP_exceptions`

9.2.1 Methods

<code>__init__()</code>

10 Module *easyLDAP.easyLDAP_utils*

10.1 Functions

is_uppercase(*test_str*)

Checks, whether all alphabetical characters in a given string are upper case characters.

Parameters

test_str: string that shall be tested for uppercase characters
(*type=*str)

is_lowercase(*test_str*)

Checks, whether all alphabetical characters in a given string are lower case characters.

Parameters

test_str: string that shall be tested for lowercase characters
(*type=*str)

is_mingledcase(*test_str*)

Checks, whether lower as well as upper case characters appear in a given string.

Parameters

test_str: string that shall be tested for mixed lowercase and uppercase characters
(*type=*str)

ia5string(*non_ia5_str*)

Tries to map special characters in a human readable string to IA5 string characters.

Parameters

non_ia5_str: string that shall be mapped to an IA5 string
(*type=*str)

days_since_1970()

Calculate number of days since 1970.

shadowAgeToday()

Calculate number of days since 1970.

crypthash(*password*)

Generate a crypt password hash for a given plain text password.

Parameters

password: plain text password that is to be hashed with the Crypt algorithm
(*type=**str*)

nthash(*password*)

Generate a smbpasswd-style NT hash.

Parameters

password: plain text password that is to be hashed with the NT hash algorithm (MD4)
(*type=**str*)

lmhash(*password*)

Generate a smbpasswd-style LanManager (Win9x) hash.

Parameters

password: plain text password that is to be hashed with the LanManager hash algorithm
(*type=**str*)

changeAttrTypeInAttrDescDict(*attrdesc_dict*, *attrtype_alias*)

This function is used to replace aliases of attribute types in the easyLDAP internal data structure.

The function expects a dictionary of attribute descriptions and their values (i.e. dictionary keys of the form <attrType>;<attrOption>).

The function will fail if the attribute descriptions (i.e. the dictionary keys) are not all of the same attribute type. It furtheron expects an alias of the attribute type name that shall replace the commonly used attribute type in the formerly named dictionary.

The attribute options, however, must be supported by the LDAPv3 protocol. Attribute description values must be given as multi-valued or single-valued lists (depending on the attribute's properties in the server's LDAP schema).

Example:

```
attrdesc_dict = { 'ou': ['network administration team'],
                  'ou;lang-de': ['NETZWERKTEAM'], }

attrtype_alias = 'organizationalUnit'

result = { 'organizationalUnit': ['network administration team'],
           'organizationalUnit;lang-de': ['NETZWERKTEAM'], }
```

If the function fails, the value `None` is returned.

Parameters

attrdesc_dict: attribute description dictionary (pre-requisites, see above)
(*type=*`dict`)

attrtype_alias: alias attribute type name that will replace attribute type names in the dictionary keys of `attrdesc_dict`

is_pyldapobject(*pyldapobject*, *strict=True*)

Return **True**, if the given data structure (a single Python dictionary in a Python list) conforms to the Python LDAP data format and only contains data of a single DN.

Parameters

pyldapobject: the LDAP data structure to be tested for Python LDAP compliancy
(*type=list*)

strict: test if the LDAP object also contains objectClass values
(*type=bool*)

is_pyldaptree(*pyldaptree*)

Return **True**, if the given data structure (several Python dictionaries in a Python list) conforms to the Python LDAP data format and all the contained LDAP DNs form a coherent LDAP (sub)tree.

Parameters

pyldaptree: the LDAP data structure to be tested for Python LDAP compliancy
(*type=list*)

to_list_of_strings(*values*)

Convert any object of a standard Pythonian variable type to a list of strings.

Parameters

values: value(s) to be converted @type values; tuple, list

generate_ldif(*from_single_pyldapobject, to_single_pyldapobject*)

Generate an ldif formatted text, that can be used with official OpenLDAP Utils (ldapadd, ldapmodify) or any other LDIF capable LDAP tool.

The method expects two Python LDAP objects that only contain a single DN each. To create an LDIF output it is also a pre-requisite that both LDAP objects share the same DN.

The method returns a list, each item represents a line of the LDIF output format.

If this function returns an empty list it means that both Python LDAP objects are identical.

If this function returns `None` it means that we met an error on our way.

Parameters

`from_single_pyldapobject`: current LDAP object in database
(*type=list*)

`to_single_pyldapobject`: LDAP object containing the database changes that shall be represented in the expected LDIF output text
(*type=list*)

generate_modlist(*from_single_pyldapobject, to_single_pyldapobject*)

Generate a modification list, that can be used with the Python LDAP methods:

```
ldap.ldap_modify()  
ldap.modify_s()
```

The method expects two Python LDAP objects that only contain a single DN each. To create an LDAP modlist it is also a pre-requisite that both LDAP objects share the same DN.

If this function returns an empty list it means that both Python LDAP objects are identical.

If this function returns `None` it means that we met an error on our way.

Parameters

from_single_pyldapobject: current LDAP object in database
(*type=list*)

to_single_pyldapobject: LDAP object containing the database changes that shall be represented in the expected Python LDAP stylish list of modifications
(*type=list*)

get_onelevel_dnlist(*dn_list, level*)

Return LDAP DN's that have all the same depth level in an LDAP tree hierarchy.

Parameters

dn_list: list of LDAP DN strings
(*type=list*)

level: depth level in the LDAP tree
(*type=int*)

get_sorted_dnlist(*dn_list*)

Sort a DN list so that it can be looked at as an LDAP tree after sorting.

Parameters

dn_list: list of LDAP DN strings
(*type=list*)

```
genpasswd(length=8,  
chars='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123...')
```

Generate a random password.

Parameters

length: password length
(*type=int*)

chars: choice of characters for the random password
(*type=str*)

```
split_dn(dn, basedn=None)
```

Split a DN into its RDN components.

Parameters

dn: DN to be split
(*type=str*)

basedn: if given, the base DN at the end of the DN will not be split into RDNs
(*type=str*)

```
get_parent_dn(dn, basedn=None)
```

Return the parent DN of a given DN.

If the given DN already is the base DN of the LDAP tree, then the Base DN is returned.

Parameters

dn: child DN
(*type=str*)

basedn: if the given DN equals the base DN then the base DN will be returned
(*type=str*)

get_rdn(*dn*, *basedn*=None)

Return the RDN of a given DN.

If the given DN already is the base DN of the LDAP tree, then the Base DN is returned.

Parameters

dn: find out RDN of this DN

(*type=*str)

basedn: if the given DN equals the base DN then the base DN will be returned

(*type=*str)

is_dn_syntax(*dn*)

Check if DN has a valid syntax.

Parameters

dn: DN to be checked for valid syntax

(*type=*str)

get_naming_attribute_value(*dn*)

Detect the naming attribute type of a given DN.

Parameters

dn: detect naming attribute type from this DN

(*type=*str)

intersect(*l1*, *l2*)

Find the (mathematical) intersection of two different sets of items.

Parameters

l1: first list of items for intersection

(*type=*list)

l2: second list of items for intersection type l2: list

10.2 Variables

Name	Description
EASY_LDAP	Value: {'AdminRDN': 'cn=admin', 'AutomountRDN': 'ou=automount', ...}

continued on next page

Name	Description
<code>--package--</code>	Value: <code>'easyLDAP'</code>
<code>except_dict</code>	Value: <code>{ 'CANNOT_DELETE_BASEDN_OBJECT' :</code> <code>'The LDAP server\'s base ...</code>
<code>key</code>	Value: <code>'NO_PARENTOBJECT_FOR_GIVEN_DN'</code>

11 Module `easyLDAP.easyLDAP_version`

Define the version of Python `easyLDAP`.

11.1 Variables

Name	Description
<code>--package--</code>	Value: <code>'easyLDAP'</code>

Index

- easyLDAP (*package*), 2–3
 - easyLDAP.easyLDAP_bind (*module*), 4–8
 - easyLDAP.easyLDAP_bind.easyLDAP_defaultBind (*function*), 5, 7
 - easyLDAP.easyLDAP_bind.easyLDAP_defaultBind.refresh (*function*), 5, 8
 - easyLDAP.easyLDAP_bind.easyLDAP_setDefaultBindCredentials (*function*), 4, 7
 - easyLDAP.easyLDAP_bind.easyLDAP_setDefaultBindCredentials.BindCredentials (*function*), 4, 6
 - easyLDAP.easyLDAP_bind.easyLDAP_setDefaultBindCredentials.Credentials (*function*), 4, 6
 - easyLDAP.easyLDAP_class_base (*module*), 9–19
 - easyLDAP.easyLDAP_class_base.easyLDAP (*class*), 9–19
 - easyLDAP.easyLDAP_class_cache (*module*), 20–25
 - easyLDAP.easyLDAP_class_cache.easyLDAP_cache.history (*class*), 20–21
 - easyLDAP.easyLDAP_class_cache.easyLDAP_cache.history.jc69 (*class*), 21–22
 - easyLDAP.easyLDAP_class_cache.easyLDAP_cache.history.jc69.history (*class*), 22–25
 - easyLDAP.easyLDAP_class_object_base (*module*), 26–51
 - easyLDAP.easyLDAP_class_object_base.easyLDAP_class_object_base.jc69 (*class*), 26–51
 - easyLDAP.easyLDAP_class_tree (*module*), 52–59
 - easyLDAP.easyLDAP_class_tree.easyLDAP_tree (*function*), 63
 - easyLDAP.easyLDAP_defaults (*module*), 60
 - easyLDAP.easyLDAP_defaults.print_easyLDAP_defaults (*function*), 60
 - easyLDAP.easyLDAP_etc (*module*), 61
 - easyLDAP.easyLDAP_exceptions' (*module*), 62
 - easyLDAP.easyLDAP_exceptions'.easyLDAP_exceptions.LDAPException (*class*), 62
 - easyLDAP.easyLDAP_utils (*module*), 63–71
 - easyLDAP.easyLDAP_utils.changeAttrTypeInAttrDefn (*function*), 64
 - easyLDAP.easyLDAP_utils.crypthash (*function*), 64
 - easyLDAP.easyLDAP_utils.days_since_1970 (*function*), 63
 - easyLDAP.easyLDAP_utils.generate_ldif (*function*), 66
 - easyLDAP.easyLDAP_utils.generate_modlist (*function*), 67
 - easyLDAP.easyLDAP_utils.genpasswd (*function*), 68
 - easyLDAP.easyLDAP_utils.get_naming_attribute_value (*function*), 70
 - easyLDAP.easyLDAP_utils.get_onelevel_dnlist (*function*), 68
 - easyLDAP.easyLDAP_utils.get_parent_dn (*function*), 69
 - easyLDAP.easyLDAP_utils.get_rdn (*function*), 69
 - easyLDAP.easyLDAP_utils.get_sorted_dnlist (*function*), 68
 - easyLDAP.easyLDAP_utils.ia5string (*function*), 63
 - easyLDAP.easyLDAP_utils.intersect (*function*), 70
 - easyLDAP.easyLDAP_utils.is_dn_syntax (*function*), 70
 - easyLDAP.easyLDAP_utils.is_lowercase (*function*), 63
 - easyLDAP.easyLDAP_utils.is_mingledcase (*function*), 63
 - easyLDAP.easyLDAP_utils.is_pyldapobject (*function*), 65
 - easyLDAP.easyLDAP_utils.is_pyldaptree (*function*), 66
 - easyLDAP.easyLDAP_utils.is_uppercase (*function*), 63
 - easyLDAP.easyLDAP_utils.lmhash (*function*), 64

easyLDAP.easyLDAP_utils.nthash (*function*), 64
easyLDAP.easyLDAP_utils.split_dn (*function*), 69
easyLDAP.easyLDAP_utils.to_list_of_strings
(*function*), 66
easyLDAP.easyLDAP_version (*module*), 72