ms_active_directory

Release 1.9.1

Azaria Zornberg

Dec 09, 2021

CONTENTS

1 Documentation vs. Examples **3**

2 Contents **5**

2.1 The ms_active_directory project 5

`ms_active_directory` is a pure Python client library for developing tools for and integrations with Microsoft Active Directory domains. It is mostly platform independent, with optional features that do have platform specific behavior.

It includes utilities for discovering and searching domains, as well as joining computers to them, modifying entities within them, and looking up information about users, groups, computers, and other objects.

It does its best to abstract away the nuances and quirks of Active Directory, and allow users to easily perform common operations in a highly efficient manner that is highly secure by default, while also being flexible enough for power users to perform complex operations not supported by the library in pre-made functions.

This library tries to conform to all Active Directory standard defaults in terms of object locations, entity object classes, encryption types used, DNS names used for computers, etc.

CHAPTER

ONE

DOCUMENTATION VS. EXAMPLES

If you're looking for examples of using the library, there's a good number of examples in the github repo's README file and the repo itself, which help to provide concrete demonstrations of how to use the functions documented here.

The documentation here is based on the docstrings in the repo and the type annotations in the repo, which means that it's incredibly detailed and thorough. A point of pride for this library is the complete type annotation of functions and highly descriptive docstrings for every user-facing function.

CHAPTER

TWO

CONTENTS

2.1 The ms_active_directory project

ms_active_directory is a library designed to make integrations with Active Directory domains, and tools for managing them, easier to write.

There are a large number of protocols that can be used to interact with Active Directory domains, but a lot of them can be difficult to use when designing a tool or integration from scratch. They can be confusing to use because the protocols in most cases were not designed specifically for Microsoft Active Directory, and so there will be behavioral quirks and slightly differences when using them with Active Directory.

The primary goal of this library is to allow users, whether they be SysAdmins, DevOps Engineers, or Software Engineers developing a new product that integrates with Active Directory, to abstract away the need to deeply understand the different options for integration and their quirks.

The secondary goal of this library is platform independence. There are a lot of tools for Active Directory that are windows-only, or that behave differently on different operating systems due to using system libraries. In order to achieve some amount of platform independence, this library works out of the box using pure python and builds primarily on other python packages that are pure python such as ldap3. However, certain optional features (e.g. Kerberos negotiation) will require python packages that build upon system libraries; this is done in order to avoid reimplementing complex security-related features, and to instead use well-trusted and verified implementations of them.

2.1.1 License

The ms_active_directory library is distributed under the MIT License.

This means that users of this library may freely use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software. The only condition is that the appropriate copyright notice be included with any copies or substantial portions of the library.

2.1.2 RFCs Compliance

This library largely utilizes the LDAP protocol for communication, as well as DNS. Utilization of those protocols is done via other python libraries. In particular, ldap3 is used for LDAP communication, and so all LDAP communication is compliant with RFCs 4510-4518.

Generation of kerberos keys for Active Directory, and parsing of kerberos keys, according to various kerberos encryption types is done in compliance with RFC4757, RFC1964, RFC8429, RFC3962, and a variety of other RFCs that define how kerberos keys are to be derived and encoded for different encryption types. However, actual kerberos negotiation relies on the underlying OS mechanism that implements GSSAPI, and so this library makes no claim to enforce specific RFC compliance in the actual negotiation, as even different versions of the same OS have significant differences (e.g. Ubuntu 14 vs. 18).

2.1.3 PEP8 Compliance

`ms_active_directory` is PEP8 compliant, excluding line length. PEP8 (https://www.python.org/dev/peps/pep-0008/) is the standard coding style guide for the Python Standard Library and for many other Python projects. It provides a consistent way of writing code for maintainability and readability following the principle that "software is more read than written".

Type hints are also utilized in all outwardly exposed classes and functions implemented in the library, and nearly all functions overall.

2.1.4 Home Page

The home page of the `ms_active_directory` project is https://github.com/zorn96/ms_active_directory

2.1.5 Documentation

Documentation is available at https://ms-active-directory.readthedocs.io/. You can download a PDF copy of the manual at https://media.readthedocs.org/pdf/ms-active-directory/stable/ms-active-directory.pdf

2.1.6 Documentation vs. Examples

If you're looking for examples of using the library, there's a good number of examples in the github repo's README file and the repo itself, which help to provide concrete demonstrations of how to use the functions documented here.

The documentation is based on the docstrings in the repo and the type annotations in the repo, which means that it's incredibly detailed and thorough. A point of pride for this library is the complete type annotation of functions and highly descriptive docstrings for every user-facing function.

2.1.7 Download

The `ms_active_directory` package can be downloaded at https://pypi.org/project/ms-active-directory/.

2.1.8 Install

Install with **pip install ms_active_directory**. If needed the library installs the `pyasn1` package, `ldap3`, `dnspython`, and `pycryptodome`. There are some other packages that may be installed but they're fairly standard (e.g. `six`). If you need Kerberos support you must install the `gssapi` package, or the `winkerberos` package if you're windows in a setup where `gssapi` does not work. These packages may require other system libraries be installed.

2.1.9 GIT repository

You can download the latest released source code at https://github.com/zorn96/ms_active_directory/tree/main

2.1.10 Contributing to this project

ms_active_directory source is hosted on github. You can contribute to the project on https://github.com/zorn96/ms_active_directory forking the project and submitting a *pull request* with your modifications.

2.1.11 Support

You can submit support tickets on https://github.com/zorn96/ms_active_directory/issues/new

2.1.12 Contact me

For information and suggestions you can contact me at a.zornberg96@gmail.com. You can also open a support ticket on https://github.com/zorn96/ms_active_directory/issues/new

2.1.13 Donate

If you want to keep this project up and running you can send me an Amazon gift card. I will use it to improve my skills in the Information and Communication technologies.

2.1.14 Acknowledgements and Shout-outs

- **Ilya Etingof**, the author of the pyasn1 package for his excellent work and support.
- **Giovanni Cannata** for his work on the ldap3 package, which is where I got my start on learning about this area, and which is an integral part of this package.
- **GitHub** for providing the *free source repository space and tools* used to develop this project.
- **VMWare** for providing the free licenses used to run windows VMs for developing and testing this library.

2.1.15 Documentation Contents

Primary Objects in ms_active_directory

The following are the references for objects that you will interact with in order to exercise the majority of the functionality of the library.

These pages will often reference each other as well, as they interact heavily and these objects can produce each other.

ADDomain Objects

Help on the ADDomain from module ms_active_directory.core.ad_domain follows.

Creating an ADDomain object

Discovery of a domain's resources and subsequent creation of sessions with the domain for the purposes of lookups, modifications, and such is done using an ADDomain object:

```
class ADDomain(builtins.object)

    __init__(self, domain: str, site: str = None,
             ldap_servers_or_uris: List = None,
             kerberos_uris: List[str] = None,
             encrypt_connections: bool = True,
             ca_certificates_file_path: str = None,
             discover_ldap_servers: bool = True,
             discover_kerberos_servers: bool = True,
             dns_nameservers: List[str] = None,
             source_ip: str = None,
             netbios_name: str = None,
             auto_configure_kerberos_client: bool = False)
        Initializes an interface for defining an AD domain and interacting with it.

        :param domain: The DNS name of the Active Directory domain that this object␣
↪represents.
        :param site: The Active Directory site to operate within. This is only relevant␣
↪if LDAP or
                     kerberos servers are discovered in DNS, as there's site-specific␣
↪records.
                     If set, only hosts within the specified site will be used.
        :param ldap_servers_or_uris: A list of either Server objects from the ldap3␣
↪library, or
                                     string LDAP uris. If specified, they will be used␣
↪to establish
                                     sessions with the domain.
        :param kerberos_uris: A list of string kerberos server uris. These can be IPs␣
↪(and the default
                              kerberos port of 88 will be used) or IP:port combinations.
        :param encrypt_connections: Whether or not LDAP connections with the domain will␣
↪be secured
                                    using TLS. This must be True for join functionality␣
↪to work,
                                    as passwords can only be set over secure connections.
                                    If not specified, defaults to True. If LDAP server␣
↪objects are
                                    provided with ssl enabled or ldaps:// uris are␣
↪provided, then
                                    connections to those servers will be encrypted␣
↪because of the
                                    inherent behavior of such configurations.
        :param ca_certificates_file_path: A path to CA certificates to be used to␣
↪establish trust
                                          with LDAP servers when securing connections.␣
↪If not
                                          specified, then TLS will not check the peer␣
↪certificate.
```

(continues on next page)

(continued from previous page)

```
                                          If LDAP server objects are specified, then␣
↪their TLS
                                          settings will be used rather than anything set␣
↪in this
                                          variable. It is only used when discovering␣
↪servers or
                                          using string URIs, so Server objects can be␣
↪used if
                                          different CAs sign different servers'␣
↪certificates
                                          due to regional CAs or something similar.
                                          If not specified, defaults to None.
        :param discover_ldap_servers: If true, and LDAP servers/uris are not specified,␣
↪then LDAP
                                      servers for the domain will be discovered in DNS.
                                      If not specified, defaults to True.
        :param discover_kerberos_servers: If true, and kerberos uris are not specified,␣
↪then kerberos
                                          servers for the domain will be discovered in␣
↪DNS.
                                          If not specified, defaults to True.
        :param dns_nameservers: A list of strings indicating the IP addresses of DNS␣
↪servers to use
                                when discovering servers for the domain. These may be␣
↪IPv4 or IPv6
                                addresses.
                                If not specified, defaults to what's configured in /etc/
↪resolv.conf on
                                POSIX systems, and extracting nameservers from registry␣
↪keys on windows.
                                Defaults to None.
        :param source_ip: A source IP address to use for both DNS and LDAP connections␣
↪established for
                          this domain. If not specified, defaults to automatic␣
↪assignment of IP using
                          underlying system networking.
                          Defaults to None.
        :param netbios_name: The netbios name of this domain, which is relevant for a␣
↪variety of functions.
                             If this is set, then we won't search the domain for the␣
↪information.
                             This can be set by users, but isn't needed. It's primarily␣
↪here to avoid
                             extra lookups when creating ADDomain objects from␣
↪ADTrustedDomain objects, as
                             the netbios name is already known.
        :param auto_configure_kerberos_client: If true, automatically configure the␣
↪local system to enable kerberos
                                               communication with the domain.
```

As can be seen, creating a domain is fairly flexible. The only actual *required* parameter is the domain's dns name. But if you need to specify a site to confine searches you can. If you're running in a container in a multi-tenant network environment, you can configure your dns nameservers and source IP as needed. You can specify what servers you want to connect to, or let the library discover the closest servers. There's options for security.

It's very simple at its simplest, while still being very flexible.

Creating a connection with the ADDomain

Once you have an ADDomain, you probably want to create a connection to it. Connections can be made as a user or as a computer. Functionally, computers act as users, but the functions to create connections as a computer provide some additional helpful checks based on restrictions that AD applies to computers.

There's two ways you can create a connection. The first is creating an ADSession object, which is a wrapper around an LDAP connection that provides a lot of useful functions, like those for finding users, groups, etc. It's recommended that you use this for most use cases, as it abstracts away many complexities:

```
create_session_as_computer(self, computer_name: str, computer_password: str = None,
                           check_name_format: bool = True, authentication_mechanism: str␣
↪= 'GSSAPI',
                           **kwargs) -> ms_active_directory.core.ad_session.ADSession
    Create a session with AD domain authenticated as the specified computer.

    :param computer_name: The name of the computer to use when authenticating with the␣
↪domain.
    :param computer_password: Optional, the password of the computer to use when␣
↪authenticating with the domain.
                              If using an authentication mechanism like NTLM, this must␣
↪be specified. But for
                              authentication mechanisms such as kerberos or external,␣
↪either `sasl_credentials`
                              can be specified as a keyword argument or default system␣
↪credentials will be used
                              in accordance with the auth mechanism.
    :param check_name_format: If True, the `computer_name` will be processed to try and␣
↪format it based on the
                              authentication mechanism in use. For NTLM we will try to␣
↪format it as
                              `domain`\`computer_name`, and for Kerberos/GSSAPI we will␣
↪try to format is ass
                              `computer_name`@`domain`.
                              Defaults to True.
    :param authentication_mechanism: An LDAP authentication mechanism or SASL mechanism.␣
↪If 'SASL' is specified,
                                     then the keyword argument `sasl_mechanism` must␣
↪also be specified. Valid values
                                     include all authentication mechanisms and SASL␣
↪mechanisms from the ldap3
                                     library, such as SIMPLE, NTLM, KERBEROS, etc.
    :returns: An ADSession object representing a connection with the domain.


create_session_as_user(self, user: str = None, password: str = None, authentication_
↪mechanism: str = None,
                       **kwargs) -> ms_active_directory.core.ad_session.ADSession
    Create a session with AD domain authenticated as the specified user.
```

(continues on next page)

(continued from previous page)

```
    :param user: The name of the user to use when authenticating with the domain. This␣
↪should be formatted based
                 on the authentication mechanism. For example, kerberos authentication␣
↪expects username@domain,
                 NTLM expects domain\\username, and simple authentication can use a␣
↪distinguished name,
                 username@domain, or other formats based on your domain's settings.
                 If not specified, anonymous authentication will be used. If specified,␣
↪SIMPLE authentication
                 will be used by default if authentication_mechanism is not specified.
    :param password: The password to use when authenticating with the domain.
                     If not specified, anonymous authentication will be used. If␣
↪specified, SIMPLE authentication
                     will be used by default if authentication_mechanism is not␣
↪specified.
    :param authentication_mechanism: An LDAP authentication mechanism or SASL mechanism.␣
↪If 'SASL' is specified,
                                     then the keyword argument `sasl_mechanism` must␣
↪also be specified. Valid values
                                     include all authentication mechanisms and SASL␣
↪mechanisms from the ldap3
                                     library, such as SIMPLE, NTLM, KERBEROS, etc.
    :param kwargs: Additional keyword arguments can be specified for any of the␣
↪arguments to an ldap3 Connection
                   object and they will be used. This can be used to set things like␣
↪`client_strategy` or
                   `pool_name`.
    :return: An ADSession object representing a connection with the domain.
```

However, you can also create a simple LDAP connection - this will return a `ldap3.Connection` object. You can then treat it like any other LDAP connection, and you'll need to form filters and such yourself. If you do this, you should consult the `ldap3` documentation on how `Connection` objects are used. To do this you can call either of the following functions:

```
create_ldap_connection_as_computer(self, computer_name: str, computer_password: str =␣
↪None,
                                   check_name_format: bool = True, authentication_
↪mechanism: str = 'GSSAPI',
                                   **kwargs) -> ldap3.core.connection.Connection
    Create an LDAP connection with AD domain authenticated as the specified computer.

    :param computer_name: The name of the computer to use when authenticating with the␣
↪domain.
    :param computer_password: Optional, the password of the computer to use when␣
↪authenticating with the domain.
                              If using an authentication mechanism like NTLM, this must␣
↪be specified. But for
                              authentication mechanisms such as kerberos or external,␣
↪either `sasl_credentials`
                              can be specified as a keyword argument or default system␣
↪credentials will be used
```

(continues on next page)

(continued from previous page)

```
                              in accordance with the auth mechanism.
    :param check_name_format: If True, the `computer_name` will be processed to try and␣
↪format it based on the
                              authentication mechanism in use. For NTLM we will try to␣
↪format it as
                              `domain`\`computer_name`, and for Kerberos/GSSAPI we will␣
↪try to format is ass
                              `computer_name`@`domain`.
                              Defaults to True.
    :param authentication_mechanism: An LDAP authentication mechanism or SASL mechanism.␣
↪If 'SASL' is specified,
                                     then the keyword argument `sasl_mechanism` must␣
↪also be specified. Valid values
                                     include all authentication mechanisms and SASL␣
↪mechanisms from the ldap3
                                     library, such as SIMPLE, NTLM, KERBEROS, etc.
    :returns: A Connection object representing a ldap connection with the domain.



create_ldap_connection_as_user(self, user: str = None, password: str = None,␣
↪authentication_mechanism: str = None,
                               **kwargs) -> ldap3.core.connection.Connection
    Create an LDAP connection with AD domain authenticated as the specified user.

    :param user: The name of the user to use when authenticating with the domain. This␣
↪should be formatted based
                 on the authentication mechanism. For example, kerberos authentication␣
↪expects username@domain,
                 NTLM expects domain\\username, and simple authentication can use a␣
↪distinguished name,
                 username@domain, or other formats based on your domain's settings.
                 If not specified, anonymous authentication will be used. If specified,␣
↪SIMPLE authentication
                 will be used by default if authentication_mechanism is not specified.
    :param password: The password to use when authenticating with the domain.
                     If not specified, anonymous authentication will be used. If␣
↪specified, SIMPLE authentication
                     will be used by default if authentication_mechanism is not␣
↪specified.
    :param authentication_mechanism: An LDAP authentication mechanism or SASL mechanism.␣
↪If 'SASL' is specified,
                                     then the keyword argument `sasl_mechanism` must␣
↪also be specified. Valid values
                                     include all authentication mechanisms and SASL␣
↪mechanisms from the ldap3
                                     library, such as SIMPLE, NTLM, KERBEROS, etc.
    :param kwargs: Additional keyword arguments can be specified for any of the␣
↪arguments to an ldap3 Connection
                   object and they will be used. This can be used to set things like␣
↪`client_strategy` or
                   `pool_name`.
    :return: An ldap3 Connection object representing a connection with the domain.
```

Discovering domain properties

ADDomain objects provide a number of functions for discovering basic information about a domain. Most of these can be done without authenticating with the domain as a user or computer (though you can reuse such authentication if desired) because they may inform your decisions on how to authenticate.

For example, you can check the time of the domain, and there's a helper for seeing if your local system time is close to the domain's time, which is important for kerberos authentication. You can also discover supported SASL mechanisms, the domain's functional level, etc.

Note: All of these functions *also* have equivalents within the ADSession object that can be called, so if you're unsure what information is guarded by authentication requirements within your domain, you can use your authenticated ADSession instead of these.

The functions are as follows:

```
find_current_time(self, ldap_connection: ldap3.core.connection.Connection = None) ->␣
↪datetime.datetime
    Find the current time for this domain. This is useful for detecting drift that can␣
↪cause
    Kerberos and TLS issues.
    Optionally, an existing connection can be used. If one is not specified, an␣
↪anonymous LDAP
    connection will be created and used.
    :param ldap_connection: An ldap3 connection to the domain, optional.
    :returns: A datetime object representing the time.


find_functional_level(self, ldap_connection: ldap3.core.connection.Connection = None) ->
↪'domainFunctionality'
    Find the functional level for this domain.
    Optionally, an existing connection can be used. If one is not specified, an␣
↪anonymous LDAP
    connection will be created and used.
    :param ldap_connection: An ldap3 connection to the domain, optional.
    :returns: An ADVersion enum indicating the functional level.


find_netbios_name(self, ldap_connection: ldap3.core.connection.Connection = None, force_
↪refresh: bool = False) -> str
    Find the netbios name for this domain. Renaming a domain is a huge task and is␣
↪incredibly rare,
    so this information is cached when first read, and it only re-read if specifically␣
↪requested.
    Optionally, an existing connection can be used. If one is not specified, an␣
↪anonymous LDAP
    connection will be created and used.

    :param ldap_connection: An ldap3 connection to the domain, optional.
    :param force_refresh: If set to true, the domain will be searched for the␣
↪information even if
                          it is already cached. Defaults to false.
    :returns: A string indicating the netbios name of the domain.
```

(continues on next page)

(continued from previous page)

```
find_supported_sasl_mechanisms(self, ldap_connection: ldap3.core.connection.Connection =␣
↪None) -> List[str]
    Find the supported SASL mechanisms for this domain.
    Optionally, an existing connection can be used. If one is not specified, an␣
↪anonymous LDAP
    connection will be created and used.
    :param ldap_connection: An ldap3 connection to the domain, optional.
    :returns: A list of strings indicating the supported SASL mechanisms for the domain.
              ex: ['GSSAPI', 'GSS-SPNEGO', 'EXTERNAL']


find_trusted_domains(self, ldap_connection: ldap3.core.connection.Connection = None) ->␣
↪List[ForwardRef('ADTrustedDomain')]
    Find the trusted domains for this domain.
    An LDAP connection is technically optional, as some domains allow enumeration of␣
↪trust
    relationships by anonymous users, but a connection is likely needed. If one is not␣
↪specified,
    an anonymous LDAP connection will be created and used.

    :param ldap_connection: An ldap3 connection to the domain, optional.
    :returns: A list of ADTrustedDomain objects

is_close_in_time_to_localhost(self, ldap_connection: ldap3.core.connection.Connection =␣
↪None, allowed_drift_seconds: int = None) -> bool
    Check if we're close in time to the domain.
    This is primarily useful for kerberos and TLS negotiation health.
    Optionally, an existing connection can be used. If one is not specified, an␣
↪anonymous LDAP
    connection will be created and used.
    :param ldap_connection: An ldap3 connection to the domain, optional.
    :param allowed_drift_seconds: The number of seconds considered "close", defaults to␣
↪5 minutes.
                                  5 minutes is the standard allowable drift for kerberos.
    :returns: A boolean indicating whether we're within allowed_drift_seconds seconds of␣
↪the domain time.
```

Managing discovered domain resources

If you relied on auto-discovery to find kerberos and LDAP servers in the domain, you can retrieve the information on what was discovered or redo the discovery if you believe network conditions may have changed or new servers may have been added.

You can retrieve URIs for both, and for LDAP servers you can also retrieve `ldap3.Server` objects if desired. You can also *set* the LDAP or kerberos servers for the domain if you wish to manually filter out or add in specific servers or are generally controlling the servers yourself.

The functions to do so are as follows:

```
get_kerberos_uris(self) -> List[str]
```

(continues on next page)

(continued from previous page)

```
get_ldap_servers(self) -> List[ldap3.core.server.Server]

get_ldap_uris(self) -> List[str]

refresh_kerberos_server_discovery(self)
    Re-discover Kerberos servers in DNS for the domain and redo the sorting by RTT.
    This can update our list of KDCs for future use by callers, allowing faster servers␣
↪to be
    moved up in priority, unavailable servers to be removed from the list, and␣
↪previously unavailable
    servers to be added.

refresh_ldap_server_discovery(self)
    Re-discover LDAP servers in DNS for the domain and redo the sorting by RTT.
    This can update our list of LDAP servers for future connections, allowing faster␣
↪servers to be
    moved up in priority, unavailable servers to be removed from the list, and␣
↪previously unavailable
    servers to be added.

set_kerberos_uris(self, kerberos_uris: List)
    Sets our kerberos server uris

set_ldap_servers_or_uris(self, ldap_servers_or_uris: List)
    Set our list of LDAP servers or LDAP URIs. The list provided can be a list of
    Server objects, URIs, or a mixture.
```

Joining a domain

You can join the local machine to a domain using an ADDomain object. This action will create a computer object in the domain representing the local machine.

You can specify a lot of properties about the computer to be created, but by default it will be named after the local machine's hostname (if it's a valid AD name) and created in AD's default Computers container. A strong password is set for the computer that is 120 characters long and random, strong encryption types are enabled, and Kerberos keys will be generated for the computer and written to the standard default system location (/etc/krb5.keytab).

A ManagedADComputer object is returned which has many helper functions for reading information about the created computer and managing its keys.

To join a domain and create a new computer, use the following function:

```
join(self, admin_username: str, admin_password: str, authentication_mechanism: str =
↪'SIMPLE',
     computer_name: str = None, computer_location: str = None, computer_password: str =␣
↪None,
     computer_encryption_types: List[Union[str, ms_active_directory.environment.security.
↪security_config_constants.ADEncryptionType]] = None,
     computer_hostnames: List[str] = None, computer_services: List[str] = None,
     supports_legacy_behavior: bool = False, computer_key_file_path: str = '/etc/krb5.
↪keytab',
     **additional_account_attributes) -> ms_active_directory.core.managed_ad_objects.
↪ManagedADComputer
```

(continues on next page)

(continued from previous page)

```
    A super simple 'join the domain' function that requires minimal input - just admin␣
↪user credentials
    to use in the join process.
    Given those basic inputs, the domain's settings are used to establish a connection,␣
↪and an account is made
    with strong security settings. The account's attributes follow AD naming conventions␣
↪based on the computer's
    hostname by default.
    :param admin_username: The username of a user or computer with the rights to create␣
↪the computer.
                           This username should be formatted based on the authentication␣
↪protocol being used.
                           For example, DOMAIN\username for NTLM as opposed to␣
↪username@DOMAIN for GSSAPI, or
                           a distinguished name for SIMPLE.
                           If `old_computer_password` is specified, then this account␣
↪only needs permission to
                           change the password of the computer being taken over, which␣
↪is different from the reset
                           password permission.
    :param admin_password: The password for the user. Optional, as SASL authentication␣
↪mechanisms can use
                           `sasl_credentials` specified as a keyword argument, and␣
↪things like KERBEROS will use
                           default system kerberos credentials if they're available.
    :param authentication_mechanism: An LDAP authentication mechanism or SASL mechanism.␣
↪If 'SASL' is specified,
                                     then the keyword argument `sasl_mechanism` must␣
↪also be specified. Valid values
                                     include all authentication mechanisms and SASL␣
↪mechanisms from the ldap3
                                     library, such as SIMPLE, NTLM, KERBEROS, etc.
    :param computer_name: The name of the computer to take over in the domain. This␣
↪should be the sAMAccountName
                          of the computer, though if computer has a trailing $ in its␣
↪sAMAccountName and that is
                          omitted, that's ok. If not specified, we will attempt to find␣
↪a computer with a name
                          matching the local system's hostname.
    :param computer_location: The location in which to create the computer. This may be␣
↪specified as an LDAP-style
                              relative distinguished name (e.g. OU=ServiceMachines,
↪OU=Machines) or a windows path
                              style canonical name (e.g. example.com/Machines/
↪ServiceMachines).
                              If not specified, defaults to CN=Computers which is the␣
↪standard default for AD.
    :param computer_password: The password to set for the computer when taking it over.␣
↪If not specified, a random
                              120 character password will be generated and set.
    :param computer_encryption_types: A list of encryption types, based on the␣
↪ADEncryptionType enum, to enable on
```

(continues on next page)

(continued from previous page)

```
                                     the account created. These may be strings or enums;
↪ if they are strings,
                                     they should be strings of the encryption types as␣
↪written in kerberos
                                     RFCs or in AD management tools, and we will try to␣
↪map them to enums and
                                     raise an error if they don't match any supported␣
↪values.
                                     AES256-SHA1, AES128-SHA1, and RC4-HMAC encryption␣
↪types are supported. DES
                                     encryption types aren not.
                                     If not specified, defaults to [AES256-SHA1].
   :param computer_hostnames: Hostnames to set for the computer. These will be used to␣
↪set the dns hostname
                              attribute in AD. If not specified, the computer hostnames␣
↪will default to
                              [`computer_name`, `computer_name`.`domain`] which is the␣
↪AD standard default.
   :param computer_services: Services to enable on the computers hostnames. These␣
↪services dictate what clients
                             can get kerberos tickets for when communicating with this␣
↪computer, and this property
                             is used with `computer_hostnames` to set the service␣
↪principal names for the computer.
                             For example, having `nfs` specified as a service principal␣
↪is necessary if you want
                             to run an NFS server on this computer and have clients get␣
↪kerberos tickets for
                             mounting shares; having `ssh` specified as a service␣
↪principal is necessary for
                             clients to request kerberos tickets for sshing to the␣
↪computer.
                             If not specified, defaults to `HOST` which is the standard␣
↪AD default service.
                             `HOST` covers a wide variety of services, including `cifs`,
↪ `ssh`, and many others
                             depending on your domain. Determining exactly what␣
↪services are covered by `HOST`
                             in your domain requires checking the aliases set on a␣
↪domain controller.
   :param supports_legacy_behavior: If `True`, then an error will be raised if the␣
↪computer name is longer than
                                    15 characters (not including the trailing $). This␣
↪is because various older
                                    systems such as NTLM, certain UNC path applications,
↪ Netbios, etc. cannot
                                    use names longer than 15 characters. This name␣
↪cannot be changed after
                                    creation, so this is important to control at␣
↪creation time.
                                    If not specified, defaults to `False`.
   :param computer_key_file_path: The path of where to write the keytab file for the␣
↪computer after taking it over.
```

(continues on next page)

(continued from previous page)

```
                                   This will include keys for both user and server keys␣
↪for the computer.
                                   If not specified, defaults to /etc/krb5.keytab
    :param additional_account_attributes: Additional keyword argument can be specified␣
↪to set other LDAP attributes
                                          of the computer that are not covered above, or␣
↪where the above controls
                                          are not sufficiently granular. For example,␣
↪`userAccountControl` could
                                          be used to set the user account control values␣
↪for the computer if it's
                                          desired to set it differently from the default␣
↪(e.g. create a computer
                                          in a disabled state and enable it later).
    :returns: A ManagedADComputer object representing the computer created.
```

A domain can also be joined by taking over an existing computer. This is convenient for setups where the computer is pre-created with a lot of settings so that the machines joining don't need to know what attribute values to set.

Taking over an existing computer returns the same form of `ManagedADComputer` object, and still writes kerberos keys to the local file system and such, but there's no option to specify things like services and dns hostnames as those are read from the existing computer.

To take over a computer in this way, use the following function:

```
join_by_taking_over_existing_computer(self, admin_username: str, admin_password: str =␣
↪None,
                                      authentication_mechanism: str = 'SIMPLE', computer_
↪name: str = None,
                                      computer_password: str = None, old_computer_
↪password: str = None,
                                      computer_key_file_path: str = '/etc/krb5.keytab',
                                      **additional_connection_attributes) -> ms_active_
↪directory.core.managed_ad_objects.ManagedADComputer

    A super simple 'join the domain' function that requires minimal input - just admin␣
↪user credentials
    to use in the join process.
    Given those basic inputs, the domain's settings are used to establish a connection,␣
↪and an account is taken over
    based on inputs. The account's attributes are then read and used to generate␣
↪kerberos keys and set other attributes
    of the returned object.
    :param admin_username: The username of a user or computer with the rights to reset␣
↪the password of the computer
                           being taken over.
                           This username should be formatted based on the authentication␣
↪protocol being used.
                           For example, DOMAIN\username for NTLM as opposed to␣
↪username@DOMAIN for GSSAPI, or
                           a distinguished name for SIMPLE.
                           If `old_computer_password` is specified, then this account␣
↪only needs permission to
```

(continues on next page)

(continued from previous page)

```
                           change the password of the computer being taken over, which␣
↪is different from the reset
                           password permission.
    :param admin_password: The password for the user. Optional, as SASL authentication␣
↪mechanisms can use
                           `sasl_credentials` specified as a keyword argument, and␣
↪things like KERBEROS will use
                           default system kerberos credentials if they're available.
    :param authentication_mechanism: An LDAP authentication mechanism or SASL mechanism.␣
↪If 'SASL' is specified,
                                     then the keyword argument `sasl_mechanism` must␣
↪also be specified. Valid values
                                     include all authentication mechanisms and SASL␣
↪mechanisms from the ldap3
                                     library, such as SIMPLE, NTLM, KERBEROS, etc.
    :param computer_name: The name of the computer to take over in the domain. This␣
↪should be the sAMAccountName
                          of the computer, though if computer has a trailing $ in its␣
↪sAMAccountName and that is
                          omitted, that's ok. If not specified, we will attempt to find␣
↪a computer with a name
                          matching the local system's hostname.
    :param computer_password: The password to set for the computer when taking it over.␣
↪If not specified, a random
                              120 character password will be generated and set.
    :param old_computer_password: The current password of the computer being taken over.␣
↪If specified, the action
                                  of taking over the computer will use a "change password
↪" operation, which is less
                                  privileged than a "reset password" operation. So␣
↪specifying this reduces the
                                  permissions needed by the user specified.
    :param computer_key_file_path: The path of where to write the keytab file for the␣
↪computer after taking it over.
                                   This will include keys for both user and server keys␣
↪for the computer.
                                   If not specified, defaults to /etc/krb5.keytab
    :param additional_connection_attributes: Additional keyword arguments may be␣
↪specified for any properties of
                                             the `Connection` object from the `ldap3`␣
↪library that is desired to
                                             be set on the connection used in the␣
↪session created for taking over
                                             the computer. Examples include `sasl_
↪credentials`, `client_strategy`,
                                             `cred_store`, and `pool_lifetime`.
    :returns: A ManagedADComputer object representing the computer taken over.
```

Help on class ADTrustedDomain in module ms_active_directory.core.ad_domain:

class ADTrustedDomain(builtins.object)

ADTrustedDomain(primary_domain: ms_active_directory.core.ad_domain.ADDomain, trust_ldap_attributes: dict)

Methods defined here:

__init__(self, primary_domain: ms_active_directory.core.ad_domain.ADDomain, trust_ldap_attributes: dict)

ADTrustedDomain objects represent a trustedDomain object found within an ADDomain.

:param primary_domain: An ADDomain object representing the domain where this trusted domain object was found.
:param trust_ldap_attributes: A dictionary of LDAP attributes for the trustedDomain.

__repr__(self)

Return repr(self).

__str__(self)

Return str(self).

convert_to_ad_domain(self, site: str = None, ldap_servers_or_uris: List = None, kerberos_uris: List[str] = None, encrypt_connections: bool = True, ca_certificates_file_path: str = None, discover_ldap_servers: bool = True, discover_kerberos_servers: bool = True, dns_nameservers: List[str] = None, source_ip: str = None) -> ms_active_directory.core.ad_domain.ADDomain

Convert this AD domain trust to an ADDomain object. This takes all of the same keyword arguments as creating an ADDomain object, and use the attributes of the primary domain where appropriate for network settings.

:param site: The Active Directory site to operate within. This is only relevant if LDAP or
kerberos servers are discovered in DNS, as there's site-specific records.
If set, only hosts within the specified site will be used.
:param ldap_servers_or_uris: A list of either Server objects from the ldap3 library, or
string LDAP uris. If specified, they will be used to establish
sessions with the domain.
:param kerberos_uris: A list of string kerberos server uris. These can be IPs (and the default
kerberos port of 88 will be used) or IP:port combinations.
:param encrypt_connections: Whether or not LDAP connections with the domain will be secured
using TLS. This must be True for join functionality to work,
as passwords can only be set over secure connections.
If not specified, defaults to True. If LDAP server objects are
provided with ssl enabled or ldaps:// uris are provided, then
connections to those servers will be encrypted because of the
inherent behavior of such configurations.
:param ca_certificates_file_path: A path to CA certificates to be used to establish trust
with LDAP servers when securing connections. If not
specified, then TLS will not check the peer certificate.
If LDAP server objects are specified, then their TLS
settings will be used rather than anything set in this
variable. It is only used when discovering servers or
using string URIs, so Server objects can be used if
different CAs sign different servers' certificates

due to regional CAs or something similar.
If not specified, defaults to None.

:param discover_ldap_servers: If true, and LDAP servers/uris are not specified, then LDAP
servers for the domain will be discovered in DNS.
If not specified, defaults to True.

:param discover_kerberos_servers: If true, and kerberos uris are not specified, then kerberos
servers for the domain will be discovered in DNS.
If not specified, defaults to True.

:param dns_nameservers: A list of strings indicating the IP addresses of DNS servers to use
when discovering servers for the domain. These may be IPv4 or IPv6
addresses.
If not specified, defaults to the DNS nameservers configured in the
primary domain where this trusted domain was found because domains
that trust each other are mutually discoverable in each others'
DNS or must use a DNS that contains both of them.
If not specified and the primary domain has no nameservers set,
defaults to what's configured in /etc/resolv.conf on POSIX systems,
and extracting nameservers from registry keys on windows.
Can be set to an empty list to force use of the system defaults even
when the primary domain has dns_nameservers set.

:param source_ip: A source IP address to use for both DNS and LDAP connections established for
this domain.
If not specified, defaults to the source IP used for the primary where
this trusted domain was found, because domains that trust each other are
mutually routable, and so the source IP used to talk to the primary domain
is assumed to also be the right default network identity for talking to
this domain.
If not specified and the primary domain has no source ip set, defaults to
automatic assignment of IP using underlying system networking.
Can be set to an empty string to force use of the system defaults even
when the primary domain has source_ip set.

:returns: An ADDomain object representing this trusted domain as a complete domain with the
corresponding functionality.

create_transfer_session_to_trusted_domain(self, ad_session: ms_active_directory.core.ad_session.ADSession, converted_ad_domain: ms_active_directory.core.ad_domain.ADDomain = None, skip_validation: bool = False) -> ms_active_directory.core.ad_session.ADSession

Create a session with this trusted domain that functionally transfers the authentication of a given session.
This is useful for transferring a kerberos/ntlm session to create new sessions for querying in trusted domains
without needing to provide credentials ever time.

:param ad_session: The active directory session to transfer. This session will not be altered.

:param converted_ad_domain: Optional. If a caller wants to specify information like an AD site, or ldap
server preferences, or if the caller simply wants to avoid having DNS lookups
and RTT measurements done every single time they transfer a session because they
have a lot of sessions to transfer, then they can specify an ADDomain object

that represents the converted ADTrustedDomain.

If not specified, an ADDomain will be created for the trusted domain during transfer.

:param skip_validation: Optional. If set to False, validation checks about the trusted domain being an AD domain

or the trusted domain trusting the primary domain for users originating from the primary domain will be skipped. This can be set to True in scenarios where the trust has been reconfigured on the trusted domain, but the primary domain has stale info, to avoid needing to wait for changes to propagate to make use of the new trust. If not specified, defaults to True.

:returns: An ADSession representing the transferred authentication to the trusted domain.

:raises: SessionTransferException If any validation fails when transferring the session.

:raises: Other LDAP exceptions if the attempt to bind the transfer session in the trusted domain fails due to

authentication issues (e.g. trying to use a non-transitive trust when transferring a user that is not from the primary domain, transferring across a one-way trust when skipping validation, transferring to a domain using SID filtering to restrict cross-domain users)

get_fqdn(self) -> str

Returns the FQDN of the trusted domain.

get_netbios_name(self) -> str

Returns the netbios name of the trusted domain.

get_posix_offset(self) -> int

Returns the posix offset for the trust relationship. This is specific to the primary domain.

get_raw_trust_attributes_value(self) -> int

Returns the raw trust attributes value, which is a bitstring indicating properties of the trust.

is_active_directory_domain_trust(self) -> bool

Returns True if the trusted domain is an Active Directory domain.

is_bidirectional_trust(self) -> bool

Returns True if the trust is mutual, meaning the primary domain trusts users from the trusted domain, and the trusted domain trusts users from the primary domain.

is_cross_forest_trust(self) -> bool

Returns True if the trust relationship is a cross-forest trust.

is_cross_organization_trust(self) -> bool

Returns True if the trust relationship is a cross-organization trust.

is_disabled(self) -> bool

Returns True if the trust relationship has been disabled.

is_findable_via_netlogon(self) -> bool

Returns True if the trusted domain is findable in netlogon and the trust works there.

is_in_same_forest_as_primary_domain(self) -> bool

Returns True if the trusted domain is in the same forest as the primary domain. For example, both “americas.my-corp.net” and “emea.my-corp.net” might be subdomains within the “my-corp.net” forest.

is_mit_trust(self) -> bool

Returns True if the trusted domain is an MIT Kerberos Realm.

is_non_active_directory_windows_trust(self) -> bool

Returns True if the trusted domain is a non-Active Directory windows domain.

is_transitive_trust(self) -> bool

Returns True if the trust relationship is transitive. If a relationship is transitive, then that means that if A trusts principals from B, and B trusts principals from C, then A will also trust principals from C even if it doesn’t explicitly know that C exists.

Cross-forest trusts are inherently transitive unless transitivity is disabled. Cross-domain trusts are not inherently transitive.

is_trusted_by_primary_domain(self) -> bool

Returns True if the primary domain trusts users originating in the trusted domain.

mit_trust_uses_rc4_hmac_for(self) -> bool

Returns True to indicate that this trusted MIT Kerberos Realm can use RC4-HMAC encryption.

This is only relevant for MIT Kerberos Realms, and is a legacy attribute from a time when RC4-HMAC was not widely adopted, AES128/AES256 weren’t standard in AD, and only the less secure single-DES encryption mechanisms were shared between MIT and AD by default.

should_treat_as_external_trust(self) -> bool

Returns True if the trusted domain is configured such that it should be explicitly treated as if the trusted domain is external to the forest of the primary domain, despite being within it.

trusts_primary_domain(self) -> bool

Returns True if the trusted domain trusts users originating in the primary domain.

uses_sid_filtering(self) -> bool

Returns True if this relationship employs SID filtering. This is common in forest trusts/transitive trusts in order to ensure some level of control over which users from other domains are allowed to operate within

the primary domain.

——————————————————————-

Data descriptors defined here:

__dict__

dictionary for instance variables (if defined)

__weakref__
list of weak references to the object (if defined)

ADSession Objects

Help for the class ADSession in module ms_active_directory.core.ad_session follows:

Manually creating an ADSession

While it's recommended that you create ADSession objects from ADDomain objects, you can manually create them given a domain and an LDAP connection to it.

The function to do so is as follows:

```
class ADSession(builtins.object)
    ADSession(ldap_connection: ldap3.core.connection.Connection, domain: 'ADDomain',␣
↪search_paging_size: int = 100, trusted_domain_cache_lifetime_seconds: int = 86400)

    Methods defined here:

    __init__(self, ldap_connection: ldap3.core.connection.Connection, domain: 'ADDomain',
↪ search_paging_size: int = 100, trusted_domain_cache_lifetime_seconds: int = 86400)
        Create a session object for a connection to an AD domain.
        Given an LDAP connection, a domain, and optional parameters relating to searches␣
↪and multi-domain
        functionality, create an ADSession object.

        :param ldap_connection: An ldap3 Connection object representing the connection␣
↪to LDAP servers within
                                the domain.
        :param domain: An ADDomain object representing the domain that we're␣
↪communicating with.
        :param search_paging_size: Optional. The page size for paginated searches. If a␣
↪search is expected to
                                   be able to have more than this many results, a␣
↪paginated search will be
                                   performed. This is used as the page size in such␣
↪searches. Changing this
                                   affects the balance between the number of queries␣
↪made and the size of
                                   each query response in a large scale environment, and␣
↪so it can be used
                                   to optimize behavior based on network topology and␣
↪traffic.
                                   If not specified, defaults to 100.
        :param trusted_domain_cache_lifetime_seconds: Optional. How long to maintain our␣
↪trusted domain cache in
                                                      seconds. The cache of trusted␣
↪domain information exists because
                                                      trust relationships change␣
↪infrequently, but will be used a lot
```

(continues on next page)

(continued from previous page)

```
                                                      in searches and such when␣
↪automatic traversal of trusts is
                                                      supported. Can be set to 0 to␣
↪disable the cache.
                                                      If not specified, defaults to 24␣
↪hours.
```

Some attributes of the session can changed later. For example, paging size or domain cache lifetimes can be adjusted in response to observed response sizes and network conditions.

```
set_domain_search_base(self, search_base: str)
    Set the search base to use for 'find' queries within the domain made by this session.
    This can be used to confine our search to a sub-container within the domain. This␣
↪can improve
    the performance of lookups, avoid permissioning issues, and remove issues around␣
↪duplicate
    records with the same common name.

set_search_paging_size(self, new_size: int)

set_trusted_domain_cache_lifetime_seconds(self, new_lifetime_in_seconds: int)
```

There are functions for finding domain resources, such as DNS servers, CA certificates, policies time, etc.

```
find_certificate_authorities_for_domain(self, pem_format: bool = True, controls:␣
↪List[ldap3.protocol.rfc4511.Control] = None) -> Union[List[str], List[bytes]]
    Attempt to discover the CAs within the domain and return info on their certificates.
    If a session was first established using an IP address or blind trust TLS, but we␣
↪want to bootstrap our
    sessions to establish stronger trust, or write the CA certificates to a local␣
↪truststore for other
    non-LDAP applications to use (e.g. establishing roots of trust for https or syslog␣
↪over TLS), then it's
    helpful to grab the certificate authorities in the domain and their signing␣
↪certificates.
    Not all domains run certificate authorities; some use public CAs or get certs from␣
↪other PKI being run,
    so this isn't useful for everyone. But a lot of people do run CAs in their AD␣
↪domains, and this is useful
    for them.

    :param pem_format: If True, return the certificates as strings in PEM format.␣
↪Otherwise, return the
                       certificates as bytestrings in DER format. Defaults to True.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: A list of either PEM-formatted certificate strings or DER-formatted␣
↪certificate byte strings,
              representing the CA certificates of the CAs within the domain.
```

(continues on next page)

(continued from previous page)

```
find_current_time_for_domain(self) -> datetime.datetime
    Get the current time for the domain as a datetime object.
    Just calls the parent domain function and returns that. This is included here for␣
↪completeness.
    :returns: A datetime object representing the current time in the domain.

find_dns_servers_for_domain(self, controls: List[ldap3.protocol.rfc4511.Control] = None)␣
↪-> Dict[str, str]
    Attempt to discover the DNS servers within the domain and return info on them.
    If a session was first established using an IP address or blind trust TLS, but we␣
↪want to bootstrap our
    sessions to use kerberos or TLS backed by CA certificates, we need proper DNS␣
↪configured. For private
    domains (e.g. in a datacenter), we may run DNS servers within the domain. This␣
↪function discovers
    computers with a "DNS/" service principal name, tries to look up IP addresses for␣
↪them, and then
    returns that information.
    This won't always be useful, as DNS isn't always part of the AD domain, but it can␣
↪help if we're bootstrapping
    a computer with manufacturer configurations to use the AD domain for everything␣
↪based on a minimal starting
    configuration.

    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: A dictionary mapping DNS hostnames of DNS servers to IP addresses. The␣
↪hostnames are provided in case
              a caller is configuring DNS-over-TLS. If no IP address can be resolved for␣
↪a hostname, it will map to
              a None value.
              https://datatracker.ietf.org/doc/html/rfc8310

find_forest_schema_version(self) -> ms_active_directory.environment.constants.ADVersion
    Attempt to determine the version of Windows Server set in the forest's schema.
    :returns: An Enum of type ADVersion indicating the schema version.

find_functional_level_for_domain(self) -> ms_active_directory.environment.constants.
↪ADFunctionalLevel
    Attempt to discover the functional level of the domain and return it.
    This will indicate if the domain is operating at the level of a 2008, 2012R2, 2016,␣
↪etc. domain.
    The functional level of a domain influences what functionality exists (e.g. 2003␣
↪cannot issue AES keys,
    2012 cannot use many TLS ciphers introduced with TLS1.3) and so it can be useful for␣
↪determining what
    to do.
    :returns: An Enum of type ADFunctionalLevel indicating the functional level.
```

(continues on next page)

(continued from previous page)

```
find_netbios_name_for_domain(self, force_refresh: bool = False) -> str
    Find the netbios name for this domain. Renaming a domain is a huge task and is␣
↪incredibly rare,
    so this information is cached when first read, and it only re-read if specifically␣
↪requested.

    :param force_refresh: If set to true, the domain will be searched for the␣
↪information even if
                          it is already cached. Defaults to false.
    :returns: A string indicating the netbios name of the domain.

find_policies_in_domain(self) -> List[ADGroupPolicy]:
    Find all of the policy objects in this domain. The number of policies is often less␣
↪than the
    number of things affected by them, so querying all of them once and handling mapping␣
↪locally is
    more desirable than re-querying policies every time a container that bears policies␣
↪is queried.

    :returns: A list of ADGroupPolicy objects representing the policies in the domain.

def find_sid_for_domain(self) -> str:
    Returns the SID identifier for the domain as a string. This will be unique even if␣
↪multiple different
    domains exist with the same DNS name (e.g. a domain was cloned from one data center␣
↪to another) and
    is a part of the SIDs of domain members.
    This will be cached after one lookup because the domain SID doesn't change.

    :returns: A string representing the domain SID.

find_supported_sasl_mechanisms_for_domain(self) -> List[str]
    Attempt to discover the SASL mechanisms supported by the domain and return them.
    This just builds upon the functionality that the domain has for this, as you don't␣
↪need
    to be authenticated as anything other than anonymous to read this information (since␣
↪it's
    often used to figure out how to authenticate).
    This is included in the session object for completeness.
    :returns: A list of strings indicating the supported SASL mechanisms for the domain.
              ex: ['GSSAPI', 'GSS-SPNEGO', 'EXTERNAL']

def find_upn_suffixes_for_domain(self) -> List[str]:
    Get the user principal name suffixes for the domain. These are alternate domains␣
↪that users might have
    in their user principal name, and use for logging on.
    For example, a domain that has a read-only domain controller exposed to the internet␣
↪might support logon
    using both "company.local" and "company.com" so that people can use their␣
↪username@company.com emails to
    login.
```

(continues on next page)

(continued from previous page)

```
    Similarly, after a merger/acquisition, this may be used to support a smooth␣
↪transition if trust relationships
    aren't used - where the domains are merged while still allowing "@company1.local"␣
↪and "@company2.local"
    emails to be used for login (since many other services may be using AD for auth).

is_domain_close_in_time_to_localhost(self, allowed_drift_seconds=None) -> bool
    Get whether the domain time is close to the current local time.
    Just calls the parent domain function and returns that. This is included here for␣
↪completeness.
    :param allowed_drift_seconds: The number of seconds considered "close", defaults to␣
↪5 minutes.
                                  5 minutes is the standard allowable drift for kerberos.
    :returns: A boolean indicating whether we're within allowed_drift_seconds seconds of␣
↪the domain time.
```

Finding Users, Groups, Computers, and other objects

There are functions for finding users, groups, and computers by a variety of properties. These properties range from unique distinguishers, like canonical name, SID, or sAMAccountName, to generic descriptors that can find multiple records, like common name and searching for any records with a specific attribute value.

You can also look up attributes about the things you look up by specifying a list of LDAP attributes to query.

```
find_computer_by_distinguished_name(self, computer_dn: str, attributes_to_lookup:␣
↪List[str] = None, controls: List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_
↪active_directory.core.ad_objects.ADComputer, NoneType]
    Find a Computer in AD based on a specified distinguished name and return it along␣
↪with any
    requested attributes.
    :param computer_dn: The distinguished name of the computer.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪computer. Regardless of
                                 what's specified, the computer's name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADComputer object or None if the computer does not exist.

find_computer_by_name(self, computer_name: str, attributes_to_lookup: List[str] = None,␣
↪controls: List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_active_directory.
↪core.ad_objects.ADComputer, NoneType]
    Find a Computer in AD based on a provided name.
    This function takes in a generic name which can be either a distinguished name, a␣
↪common name, or a
    sAMAccountName, and tries to find a unique computer identified by it and return␣
↪information on the computer.
    :param computer_name: The name of the computer, which may be a DN, common name, or␣
↪sAMAccountName.
```

(continues on next page)

(continued from previous page)

```
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪computer. Regardless of
                                 what's specified, the computer's name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADComputer object or None if the computer does not exist.
    :raises: a DuplicateNameException if more than one entry exists with this name.

find_computer_by_sam_name(self, computer_name: str, attributes_to_lookup: List[str] =␣
↪None, controls: List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_active_
↪directory.core.ad_objects.ADComputer, NoneType]
    Find a Computer in AD based on a specified sAMAccountName name and return it along␣
↪with any
    requested attributes.
    :param computer_name: The sAMAccountName name of the computer. Because a lot of␣
↪people get a bit confused on
                          what a computer name, as many systems leave out the trailing $␣
↪that's common to many
                          computer sAMAccountNames when showing it, if computer_name␣
↪does not end in a trailing $
                          and no computer can be found with computer_name, a lookup will␣
↪be attempted for the
                          computer_name with a trailing $ added.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪computer. Regardless of
                                 what's specified, the computer's name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADComputer object or None if the computer does not exist.

find_computer_by_sid(self, computer_sid: Union[ms_active_directory.environment.security.
↪security_config_constants.WellKnownSID, str, ms_active_directory.environment.security.
↪security_descriptor_utils.ObjectSid], attributes_to_lookup: List[str] = None,␣
↪controls: List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_active_directory.
↪core.ad_objects.ADComputer, NoneType]
    Find a Computer in AD given its SID.
    This function takes in a computer's objectSID and then looks up the computer in AD␣
↪using it. SIDs are unique
    so only a single entry can be found at most.
    The computer SID can be in many formats (well known SID enum, ObjectSID object,␣
↪canonical SID format,
    or bytes) and so all 4 possible formats are handled.
    :param computer_sid: The computer SID. This may either be a well-known SID enum, an␣
↪ObjectSID object, a string
```

(continues on next page)

(continued from previous page)

```
                         SID in canonical format (e.g. S-1-1-0), object SID bytes, or␣
↪the hex representation of
                         such bytes.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪computer. Regardless of
                                 what's specified, the computer's name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADComputer object or None if the computer does not exist.

find_computers_by_attribute(self, attribute_name: str, attribute_value, attributes_to_
↪lookup: List[str] = None, size_limit: int = 0, controls: List[ldap3.protocol.rfc4511.
↪Control] = None) -> List[ms_active_directory.core.ad_objects.ADComputer]
    Find all computers that possess the specified attribute with the specified value,␣
↪and return a list of
    ADComputer objects.

    :param attribute_name: The LDAP name of the attribute to be used in the search.
    :param attribute_value: The value that returned computers should possess for the␣
↪attribute.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪computers. Regardless of
                                 what's specified, the computers' name and object class␣
↪attributes will be queried.
    :param size_limit: An integer indicating a limit to place the number of results the␣
↪search will return.
                       If not specified, defaults to 0, meaning unlimited.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: a list of ADComputer objects representing computers with the specified␣
↪value for the specified
              attribute.

find_computers_by_common_name(self, computer_name: str, attributes_to_lookup: List[str]␣
↪= None, controls: List[ldap3.protocol.rfc4511.Control] = None) -> List[ms_active_
↪directory.core.ad_objects.ADComputer]
    Find all computers with a given common name and return a list of ADComputer objects.
    This is particularly useful when you have multiple computers with the same name in␣
↪different OUs
    as a result of a migration, and want to find them so you can combine them.

    :param computer_name: The common name of the computer(s) to be looked up.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪computers. Regardless of
                                 what's specified, the computers' name and object class␣
↪attributes will be queried.
```

(continues on next page)

(continued from previous page)

```
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: a list of ADComputer objects representing computers with the specified␣
↪common name.


find_group_by_distinguished_name(self, group_dn: str, attributes_to_lookup: List[str] =␣
↪None, controls: List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_active_
↪directory.core.ad_objects.ADGroup, NoneType]
    Find a group in AD based on a specified distinguished name and return it along with␣
↪any
    requested attributes.
    :param group_dn: The distinguished name of the group.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪group. Regardless of
                                 what's specified, the group's name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADGroup object or None if the group does not exist.

find_group_by_name(self, group_name: str, attributes_to_lookup: List[str] = None,␣
↪controls: List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_active_directory.
↪core.ad_objects.ADGroup, NoneType]
    Find a Group in AD based on a provided name.
    This function takes in a generic name which can be either a distinguished name, a␣
↪common name, or a
    sAMAccountName, and tries to find a unique group identified by it and return␣
↪information on the group.
    :param group_name: The name of the group, which may be a DN, common name, or␣
↪sAMAccountName.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪group. Regardless of
                                 what's specified, the group's name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADGroup object or None if the group does not exist.
    :raises: a DuplicateNameException if more than one entry exists with this name.

def find_computer_by_principal_name(self, computer_name: str, attributes_to_lookup:␣
↪List[str] = None,
                                    controls: List[Control] = None) ->␣
↪Optional[ADComputer]:
```

(continues on next page)

(continued from previous page)

```
    Find a Computer in AD based on a specified userPrincipalName and return it along␣
↪with any
    requested attributes.
    :param computer_name: The userPrincipalName name of the computer.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪computer. Regardless of
                                 what's specified, the computer's name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADComputer object or None if the computer does not exist.

find_group_by_sam_name(self, group_name: str, attributes_to_lookup: List[str] = None,␣
↪controls: List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_active_directory.
↪core.ad_objects.ADGroup, NoneType]
    Find a Group in AD based on a specified sAMAccountName name and return it along with␣
↪any
    requested attributes.
    :param group_name: The sAMAccountName name of the group.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪group. Regardless of
                                 what's specified, the group's name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADGroup object or None if the group does not exist.

find_group_by_sid(self, group_sid: Union[ms_active_directory.environment.security.
↪security_config_constants.WellKnownSID, str, ms_active_directory.environment.security.
↪security_descriptor_utils.ObjectSid], attributes_to_lookup: List[str] = None,␣
↪controls: List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_active_directory.
↪core.ad_objects.ADGroup, NoneType]
    Find a Group in AD given its SID.
    This function takes in a group's objectSID and then looks up the group in AD using␣
↪it. SIDs are unique
    so only a single entry can be found at most.
    The group SID can be in many formats (well known SID enum, ObjectSID object,␣
↪canonical SID format,
    or bytes) and so all 4 possible formats are handled.
    :param group_sid: The group SID. This may either be a well-known SID enum, an␣
↪ObjectSID object, a string SID
                      in canonical format (e.g. S-1-1-0), object SID bytes, or the hex␣
↪representation of such bytes.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪group. Regardless of
                                 what's specified, the group's name and object class␣
↪attributes will be queried.
```

(continues on next page)

(continued from previous page)

```
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADGroup object or None if the group does not exist.

find_groups_by_attribute(self, attribute_name: str, attribute_value, attributes_to_
↪lookup: List[str] = None, size_limit: int = 0, controls: List[ldap3.protocol.rfc4511.
↪Control] = None) -> List[ms_active_directory.core.ad_objects.ADGroup]
    Find all groups that possess the specified attribute with the specified value, and␣
↪return a list of ADGroup
    objects.

    :param attribute_name: The LDAP name of the attribute to be used in the search.
    :param attribute_value: The value that returned groups should possess for the␣
↪attribute.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪group. Regardless of
                                 what's specified, the groups' name and object class␣
↪attributes will be queried.
    :param size_limit: An integer indicating a limit to place the number of results the␣
↪search will return.
                       If not specified, defaults to 0, meaning unlimited.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: a list of ADGroup objects representing groups with the specified value for␣
↪the specified attribute.

find_groups_by_common_name(self, group_name: str, attributes_to_lookup: List[str] = None,
↪ controls: List[ldap3.protocol.rfc4511.Control] = None) -> List[ms_active_directory.
↪core.ad_objects.ADGroup]
    Find all groups with a given common name and return a list of ADGroup objects.
    This is particularly useful when you have multiple groups with the same name in␣
↪different OUs
    as a result of a migration, and want to find them so you can combine them.

    :param group_name: The common name of the group(s) to be looked up.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪group. Regardless of
                                 what's specified, the groups' name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: a list of ADGroup objects representing groups with the specified common␣
↪name.
```

(continues on next page)

(continued from previous page)

```
find_object_by_canonical_name(self, canonical_name: str, attributes_to_lookup: List[str]␣
↪= None, controls: List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_active_
↪directory.core.ad_objects.ADObject, ms_active_directory.core.ad_objects.ADUser, ms_
↪active_directory.core.ad_objects.ADGroup, ms_active_directory.core.ad_objects.
↪ADComputer, NoneType]
    Find an object in the domain using a canonical name, also called a 'windows path␣
↪style' name.

    :param canonical_name: A windows path style name representing an object in the␣
↪domain. This may be either a
                           fully canonical name (e.g. example.com/Users/Administrator)␣
↪or a relative canonical
                           name (e.g. /Users/Administrator).
    :param attributes_to_lookup: Attributes to look up about the object. Regardless of␣
↪what's specified,
                                 the object's name and object class attributes will be␣
↪queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADObject object or None if the distinguished name does not exist. If␣
↪the object can be cast to
              a more specific subclass, like ADUser, then it will be.

find_object_by_distinguished_name(self, distinguished_name: str, attributes_to_lookup:␣
↪List[str] = None, controls: List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_
↪active_directory.core.ad_objects.ADObject, ms_active_directory.core.ad_objects.ADUser,␣
↪ms_active_directory.core.ad_objects.ADGroup, ms_active_directory.core.ad_objects.
↪ADComputer, NoneType]
    Find an object in the domain using a relative distinguished name or full␣
↪distinguished name.

    :param distinguished_name: A relative or absolute distinguished name within the␣
↪domain to look up.
    :param attributes_to_lookup: Attributes to look up about the object. Regardless of␣
↪what's specified,
                                 the object's name and object class attributes will be␣
↪queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADObject object or None if the distinguished name does not exist. If␣
↪the object can be cast to
              a more specific subclass, like ADUser, then it will be.

find_object_by_sid(self, sid: Union[ms_active_directory.environment.security.security_
↪config_constants.WellKnownSID, str, ms_active_directory.environment.security.security_
↪descriptor_utils.ObjectSid], attributes_to_lookup: List[str] = None, object_class: str
↪= None, return_type=None, controls: List[ldap3.protocol.rfc4511.Control] = None) ->␣
↪Union[ms_active_directory.core.ad_objects.ADObject, ms_active_directory.core.ad_
↪objects.ADUser, ms_active_directory.core.ad_objects.ADGroup, ms_active_directory.core.
↪ad_objects.ADComputer, NoneType]
```

(continues on next page)

(continued from previous page)

```
    Find any object in AD given its SID.
    This function takes in a user's objectSID and then looks up the user in AD using it.␣
↪SIDs are unique
    so only a single entry can be found at most.
    The user SID can be in many formats (well known SID enum, ObjectSID object,␣
↪canonical SID format,
    or bytes) and so all 4 possible formats are handled.
    :param sid: The object's SID. This may either be a well-known SID enum, an ObjectSID␣
↪object, a string SID
                in canonical format (e.g. S-1-1-0), object SID bytes, or the hex␣
↪representation of such bytes.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪object. Regardless of
                                 what's specified, the object's name and object class␣
↪attributes will be queried.
    :param object_class: Optional. The object class to filter on when searching.␣
↪Defaults to 'top' which will
                         include all objects in AD.
    :param return_type: Optional. The class to use to represent the returned objects.␣
↪Defaults to ADObject.
                        If a generic search is being done, or an object class is used␣
↪that is not yet supported
                        by this library, using ADObject is recommended.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADObject object or None if the group does not exist.

find_objects_with_attribute(self, attribute_name: str, attribute_value, attributes_to_
↪lookup: List[str] = None, size_limit: int = 0, object_class: str = None, return_
↪type=None, controls: List[ldap3.protocol.rfc4511.Control] = None) -> List[Union[ms_
↪active_directory.core.ad_objects.ADUser, ms_active_directory.core.ad_objects.
↪ADComputer, ms_active_directory.core.ad_objects.ADObject, ms_active_directory.core.ad_
↪objects.ADGroup]]
    Find all AD objects that possess the specified attribute with the specified value␣
↪and return them.

    :param attribute_name: The LDAP name of the attribute to be used in the search.
    :param attribute_value: The value that returned objects should possess for the␣
↪attribute.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪group. Regardless of
                                 what's specified, the groups' name and object class␣
↪attributes will be queried.
    :param size_limit: An integer indicating a limit to place the number of results the␣
↪search will return.
                       If not specified, defaults to 0, meaning unlimited.
    :param object_class: Optional. The object class to filter on when searching.␣
↪Defaults to 'top' which will
                         include all objects in AD.
```

(continues on next page)

(continued from previous page)

```
    :param return_type: Optional. The class to use to represent the returned objects.␣
↪Defaults to ADObject.
                        If a generic search is being done, or an object class is used␣
↪that is not yet supported
                        by this library, using ADObject is recommended.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: a list of ADObject objects representing groups with the specified value␣
↪for the specified attribute.

find_user_by_distinguished_name(self, user_dn: str, attributes_to_lookup: List[str] =␣
↪None, controls: List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_active_
↪directory.core.ad_objects.ADUser, NoneType]
    Find a User in AD based on a specified distinguished name and return it along with␣
↪any
    requested attributes.
    :param user_dn: The distinguished name of the user.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪user. Regardless of
                                 what's specified, the user's name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADUser object or None if the user does not exist.

find_user_by_name(self, user_name: str, attributes_to_lookup: List[str] = None,␣
↪controls: List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_active_directory.
↪core.ad_objects.ADUser, NoneType]
    Find a User in AD based on a provided name.
    This function takes in a generic name which can be either a distinguished name, a␣
↪common name, or a
    sAMAccountName, and tries to find a unique user identified by it and return␣
↪information on the user.
    :param user_name: The name of the user, which may be a DN, common name, or␣
↪sAMAccountName.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪user. Regardless of
                                 what's specified, the user's name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADUser object or None if the user does not exist.
    :raises: a DuplicateNameException if more than one entry exists with this name.
```

(continues on next page)

(continued from previous page)

```
def find_user_by_principal_name(self, user_name: str, attributes_to_lookup: List[str] =␣
↪None,
                                controls: List[Control] = None) -> Optional[ADUser]:
    Find a User in AD based on a specified userPrincipalName and return it along with any
    requested attributes.
    :param user_name: The userPrincipalName name of the user.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪user. Regardless of
                                 what's specified, the user's name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADUser object or None if the user does not exist.

find_user_by_sam_name(self, user_name: str, attributes_to_lookup: List[str] = None,␣
↪controls: List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_active_directory.
↪core.ad_objects.ADUser, NoneType]
    Find a User in AD based on a specified sAMAccountName name and return it along with␣
↪any
    requested attributes.
    :param user_name: The sAMAccountName name of the user.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪user. Regardless of
                                 what's specified, the user's name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADUser object or None if the user does not exist.

find_user_by_sid(self, user_sid: Union[ms_active_directory.environment.security.security_
↪config_constants.WellKnownSID, str, ms_active_directory.environment.security.security_
↪descriptor_utils.ObjectSid], attributes_to_lookup: List[str] = None, controls:␣
↪List[ldap3.protocol.rfc4511.Control] = None) -> Union[ms_active_directory.core.ad_
↪objects.ADUser, NoneType]
    Find a User in AD given its SID.
    This function takes in a user's objectSID and then looks up the user in AD using it.␣
↪SIDs are unique
    so only a single entry can be found at most.
    The user SID can be in many formats (well known SID enum, ObjectSID object,␣
↪canonical SID format,
    or bytes) and so all 4 possible formats are handled.
    :param user_sid: The user SID. This may either be a well-known SID enum, an␣
↪ObjectSID object, a string SID
                     in canonical format (e.g. S-1-1-0), object SID bytes, or the hex␣
↪representation of such bytes.
```

(continues on next page)

(continued from previous page)

```
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪user. Regardless of
                                 what's specified, the user's name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: an ADUser object or None if the user does not exist.

find_users_by_attribute(self, attribute_name: str, attribute_value, attributes_to_
↪lookup: List[str] = None, size_limit: int = 0, controls: List[ldap3.protocol.rfc4511.
↪Control] = None) -> List[ms_active_directory.core.ad_objects.ADUser]
    Find all users that possess the specified attribute with the specified value, and␣
↪return a list of ADUser
    objects.

    :param attribute_name: The LDAP name of the attribute to be used in the search.
    :param attribute_value: The value that returned groups should possess for the␣
↪attribute.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪users. Regardless of
                                 what's specified, the users' name and object class␣
↪attributes will be queried.
    :param size_limit: An integer indicating a limit to place the number of results the␣
↪search will return.
                       If not specified, defaults to 0, meaning unlimited.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :returns: a list of ADUser objects representing users with the specified value for␣
↪the specified attribute.

find_users_by_common_name(self, user_name: str, attributes_to_lookup: List[str] = None,␣
↪controls: List[ldap3.protocol.rfc4511.Control] = None) -> List[ms_active_directory.
↪core.ad_objects.ADUser]
    Find all users with a given common name and return a list of ADUser objects.
    This is particularly useful when you have multiple users with the same name in␣
↪different OUs
    as a result of a migration, and want to find them so you can combine them.

    :param user_name: The common name of the user(s) to be looked up.
    :param attributes_to_lookup: A list of additional LDAP attributes to query for the␣
↪users. Regardless of
                                 what's specified, the users' name and object class␣
↪attributes will be queried.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
```

(continues on next page)

(continued from previous page)

```
                     may succeed or fail based on their availability.
    :returns: a list of ADUser objects representing users with the specified common name.
```

Finding and Managing Group Members and Memberships

There exist functions for finding the groups for a group, user, or computer, as well as finding the members of a group. There's also functions for managing those memberships, by adding or removing members idempotently.

When looking up members or groups, you can also look up attributes of those groups or members at the same time. The library does its best to optimize these lookups.

Finding memberships and members:

```
find_groups_for_computer(self, computer: Union[str, ms_active_directory.core.ad_objects.
↪ADComputer], attributes_to_lookup: List[str] = None, controls: List[ldap3.protocol.
↪rfc4511.Control] = None, skip_validation: bool = False) -> List[ms_active_directory.
↪core.ad_objects.ADGroup]
    Find the groups that a computer belongs to, look up attributes of theirs, and return␣
↪information about them.

    :param computer: The computer to lookup group memberships for. This can either be an␣
↪ADComputer or a string
                    name of an AD computer. If it is a string, the computer will be␣
↪looked up first to get unique
                    distinguished name information about it unless it is a distinguished␣
↪name.
    :param attributes_to_lookup: A list of string LDAP attributes to look up in addition␣
↪to our basic attributes.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A list of ADGroup objects representing the groups that this user belongs␣
↪to.
    :raises: a DuplicateNameException if a computer name is specified and more than one␣
↪entry exists with the name.
    :raises: a InvalidLdapParameterException if the computer name is not a string or␣
↪ADComputer.

find_groups_for_computers(self, computers: List[Union[str, ms_active_directory.core.ad_
↪objects.ADComputer]], attributes_to_lookup: List[str] = None, controls: List[ldap3.
↪protocol.rfc4511.Control] = None, skip_validation: bool = False) -> Dict[Union[str, ms_
↪active_directory.core.ad_objects.ADComputer], List[ms_active_directory.core.ad_objects.
↪ADGroup]]
```

(continues on next page)

(continued from previous page)

```
    Find the groups that a list of computers belong to, look up attributes of theirs,␣
↪and return information
    about them.

    :param computers: The computers to lookup group memberships for. This can be a list␣
↪of either ADComputer objects
                      or string names of AD computers. If they are strings, the␣
↪computers will be looked up first
                      to get unique distinguished name information about them unless␣
↪they are distinguished names.
    :param attributes_to_lookup: A list of string LDAP attributes to look up in addition␣
↪to our basic attributes.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A dictionary mapping computers to lists of ADGroup objects representing␣
↪the groups that they belong to
    :raises: a DuplicateNameException if a computer name is specified and more than one␣
↪entry exists with the name.
    :raises: a InvalidLdapParameterException if any computers are not a string or␣
↪ADComputer.

find_groups_for_entities(self, entities: List[Union[str, ms_active_directory.core.ad_
↪objects.ADObject]], attributes_to_lookup: List[str] = None, lookup_by_name_fn: <built-
↪in function callable> = None, controls: List[ldap3.protocol.rfc4511.Control] = None,␣
↪skip_validation: bool = False) -> Dict[Union[str, ms_active_directory.core.ad_objects.
↪ADObject], List[ms_active_directory.core.ad_objects.ADGroup]]
    Find the parent groups for all of the entities in a List.
    These entities may be users, groups, or anything really because Active Directory␣
↪uses the "groupOfNames" style
    membership tracking, so all group members are just represented as distinguished␣
↪names regardless of type.
    If the elements of entities are strings and are not distinguished names, then lookup_
↪by_name_fn will be used
    to look up the appropriate ADObject for the entity and get its distinguished name.

    The parent groups of all the entities will then be queried, and the attributes␣
↪specified will be looked up
    (if any). A dictionary mapping the original entities to lists of ADGroup objects␣
↪will be returned.

    :param entities: A list of either ADObject objects or strings. These represent the␣
↪objects whose parent groups
```

(continues on next page)

(continued from previous page)

```
                     are being queried.
    :param attributes_to_lookup: A list of LDAP attributes to query about the parent␣
↪groups, in addition to the
                                 default ones queries. Optional.
    :param lookup_by_name_fn: An optional function to call to map entities to ADObjects␣
↪when the members of entities
                              are strings that are not LDAP distinguished names.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A dictionary mapping input entities to lists of ADGroup object␣
↪representing their parent groups.
    :raises: a DuplicateNameException if an entity name is specified and more than one␣
↪entry exists with the name.
    :raises: InvalidLdapParameterException if any non-string non-ADObject types are␣
↪found in entities, or if any
             non-distinguished name strings are specified.

find_groups_for_group(self, group: Union[str, ms_active_directory.core.ad_objects.
↪ADGroup], attributes_to_lookup: List[str] = None, controls: List[ldap3.protocol.
↪rfc4511.Control] = None, skip_validation: bool = False) -> List[ms_active_directory.
↪core.ad_objects.ADGroup]
    Find the groups that a group belongs to, look up attributes of theirs, and return␣
↪information about them.

    :param group: The group to lookup group memberships for. This can either be an␣
↪ADGroup or a string name of an
                  AD group. If it is a string, the group will be looked up first to get␣
↪unique distinguished name
                  information about it unless it is a distinguished name.
    :param attributes_to_lookup: A list of string LDAP attributes to look up in addition␣
↪to our basic attributes.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
```

(continues on next page)

(continued from previous page)

```
                            performs far fewer queries.
    :returns: A list of ADGroup objects representing the groups that this group belongs␣
↪to.
    :raises: a DuplicateNameException if a group name is specified and more than one␣
↪entry exists with the name.
    :raises: a InvalidLdapParameterException if the group name is not a string or␣
↪ADGroup.

find_groups_for_groups(self, groups: List[Union[str, ms_active_directory.core.ad_objects.
↪ADGroup]], attributes_to_lookup: List[str] = None, controls: List[ldap3.protocol.
↪rfc4511.Control] = None, skip_validation: bool = False) -> Dict[Union[str, ms_active_
↪directory.core.ad_objects.ADGroup], List[ms_active_directory.core.ad_objects.ADGroup]]
    Find the groups that a list of groups belong to, look up attributes of theirs, and␣
↪return information about
    them.

    :param groups: The groups to lookup group memberships for. This can be a list of␣
↪either ADGroup objects or
                   string names of AD groups. If they are strings, the groups will be␣
↪looked up first to get unique
                   distinguished name information about them unless they are␣
↪distinguished names.
    :param attributes_to_lookup: A list of string LDAP attributes to look up in addition␣
↪to our basic attributes.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A dictionary mapping groups to lists of ADGroup objects representing the␣
↪groups that they belong to.
    :raises: a DuplicateNameException if a group name is specified and more than one␣
↪entry exists with the name.
    :raises: a InvalidLdapParameterException if any groups are not a string or ADGroup.

find_groups_for_user(self, user: Union[str, ms_active_directory.core.ad_objects.ADUser],␣
↪attributes_to_lookup: List[str] = None, controls: List[ldap3.protocol.rfc4511.Control]␣
↪= None, skip_validation: bool = False) -> List[ms_active_directory.core.ad_objects.
↪ADGroup]
    Find the groups that a user belongs to, look up attributes of theirs, and return␣
↪information about them.

    :param user: The user to lookup group memberships for. This can either be an ADUser␣
↪or a string name of an
                 AD user. If it is a string, the user will be looked up first to get␣
↪unique distinguished name
```

(continues on next page)

(continued from previous page)

```
                 information about it unless it is a distinguished name.
    :param attributes_to_lookup: A list of string LDAP attributes to look up in addition␣
↪to our basic attributes.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A list of ADGroup objects representing the groups that this user belongs␣
↪to.
    :raises: a DuplicateNameException if a user name is specified and more than one␣
↪entry exists with the name.
    :raises: a InvalidLdapParameterException if the user name is not a string or ADUser.

find_groups_for_users(self, users: List[Union[str, ms_active_directory.core.ad_objects.
↪ADUser]], attributes_to_lookup: List[str] = None, controls: List[ldap3.protocol.
↪rfc4511.Control] = None, skip_validation: bool = False) -> Dict[Union[str, ms_active_
↪directory.core.ad_objects.ADUser], List[ms_active_directory.core.ad_objects.ADGroup]]
    Find the groups that a list of users belong to, look up attributes of theirs, and␣
↪return information about
    them.

    :param users: The users to lookup group memberships for. This can be a list of␣
↪either ADUser objects or
                  string names of AD users. If they are strings, the users will be␣
↪looked up first to get unique
                  distinguished name information about them unless they are␣
↪distinguished names.
    :param attributes_to_lookup: A list of string LDAP attributes to look up in addition␣
↪to our basic attributes.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A dictionary mapping users to lists of ADGroup objects representing the␣
↪groups that they belong to.
    :raises: a DuplicateNameException if a user name is specified and more than one␣
↪entry exists with the name.
```

(continues on next page)

(continued from previous page)

```
    :raises: a InvalidLdapParameterException if any users are not a string or ADUser.

find_members_of_group(self, group: Union[str, ms_active_directory.core.ad_objects.
↪ADGroup], attributes_to_lookup: List[str] = None, controls: List[ldap3.protocol.
↪rfc4511.Control] = None, skip_validation: bool = False) -> List[Union[ms_active_
↪directory.core.ad_objects.ADUser, ms_active_directory.core.ad_objects.ADComputer, ms_
↪active_directory.core.ad_objects.ADObject, ms_active_directory.core.ad_objects.
↪ADGroup]]
    Find the members of a group in the domain, along with attributes of the members.

    :param group: Either a string name of a group or ADGroup to look up the members of.
    :param attributes_to_lookup: Attributes to look up about the members of each group.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all members exist and do not raise an error␣
↪if we fail to look one up.
                            Instead, a placeholder object will be used for members that␣
↪could not be found.
                            Defaults to False.
    :return: A list of objects representing the group's members.
             The objects may be of type ADUser, ADComputer, ADGroup, etc. - this␣
↪function attempts to cast all
             member objects to the most accurate object type representing them. ADObject␣
↪will be used for members
             that do not match any of the more specific object types in the library
             (e.g. foreign security principals).
    :raises: InvalidLdapParameterException if the group is not a string or ADGroup
    :raises: ObjectNotFoundException if the group cannot be found.
    :raises: DomainSearchException if skip_validation is False and any group members␣
↪cannot be found.

find_members_of_group_recursive(self, group: Union[str, ms_active_directory.core.ad_
↪objects.ADGroup], attributes_to_lookup: List[str] = None, controls: List[ldap3.
↪protocol.rfc4511.Control] = None, skip_validation: bool = False, maximum_nesting_
↪depth: int = None, flatten: bool = False) -> List[Dict[Union[str, ms_active_directory.
↪core.ad_objects.ADGroup], List[ms_active_directory.core.ad_objects.ADGroup]]]
    Find the members of a group in the domain, along with attributes of the members.

    :param group: Either a string name of a group or ADGroup to look up the members of.
    :param attributes_to_lookup: Attributes to look up about the members of each group.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all members exist and do not raise an error␣
↪if we fail to look one up.
                            Instead, a placeholder object will be used for members that␣
↪could not be found.
```

(continues on next page)

(continued from previous page)

```
                            Defaults to False.
    :param maximum_nesting_depth: A limit to the number of levels of nesting to recurse␣
↪beyond the first lookup.
                                  A level of 0 makes this behave the same as find_
↪members_of_groups and a level of
                                  None means recurse until we've gone through all␣
↪nesting. Defaults to None.
    :param flatten: If set to True, a 1-item list of a single dictionary mapping the␣
↪input group to a list of
                    all members found recursively will be returned. This discards␣
↪information about whether
                    a member is a direct member or is a member via nesting, and what␣
↪those relationships are.
                    As an example, instead of returning [{group1 -> [group2, user1]},
↪{group2 -> [user2, user3]}],
                    we would return [{group1 -> [group2, user1, user2, user3]}]. This␣
↪makes iterating members
                    simpler, but removes the ability to use information about the␣
↪descendants of nested groups
                    as independent groups later on.
                    Defaults to False.
    :return: A list of dictionaries mapping groups to objects representing the group's␣
↪members.
             The first dictionary maps the input group to its members; the second␣
↪dictionary maps the groups that
             were members of the groups in the first dictionary to their members, and so␣
↪on and so forth.
             The objects may be of type ADUser, ADComputer, ADGroup, etc. - this␣
↪function attempts to cast all
             member objects to the most accurate object type representing them. ADObject␣
↪will be used for members
             that do not match any of the more specific object types in the library
             (e.g. foreign security principals).
    :raises: InvalidLdapParameterException if the group is not a string or ADGroup
    :raises: ObjectNotFoundException if the group cannot be found.
    :raises: DomainSearchException if skip_validation is False and any group members␣
↪cannot be found.

find_members_of_groups(self, groups: List[Union[str, ms_active_directory.core.ad_objects.
↪ADGroup]], attributes_to_lookup: List[str] = None, controls: List[ldap3.protocol.
↪rfc4511.Control] = None, skip_validation: bool = False) -> Dict[Union[str, ms_active_
↪directory.core.ad_objects.ADGroup], List[Union[ms_active_directory.core.ad_objects.
↪ADUser, ms_active_directory.core.ad_objects.ADComputer, ms_active_directory.core.ad_
↪objects.ADObject, ms_active_directory.core.ad_objects.ADGroup]]]
    Find the members of one or more groups in the domain, along with attributes of the␣
↪members.

    :param groups: A list of either strings or ADGroups to look up the members of.
    :param attributes_to_lookup: Attributes to look up about the members of each group.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
```

(continues on next page)

(continued from previous page)

```
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all members exist and do not raise an error␣
↪if we fail to look one up.
                            Instead, a placeholder object will be used for members that␣
↪could not be found.
                            Defaults to False.
    :return: A dictionary mapping groups from the input list to lists of objects␣
↪representing their members.
             The objects may be of type ADUser, ADComputer, ADGroup, etc. - this␣
↪function attempts to cast all
             member objects to the most accurate object type representing them. ADObject␣
↪will be used for members
             that do not match any of the more specific object types in the library
             (e.g. foreign security principals).
    :raises: InvalidLdapParameterException if any groups are not strings or ADGroups
    :raises: ObjectNotFoundException if any groups cannot be found.
    :raises: DomainSearchException if skip_validation is False and any group members␣
↪cannot be found.

find_members_of_groups_recursive(self, groups: List[Union[str, ms_active_directory.core.
↪ad_objects.ADGroup]], attributes_to_lookup: List[str] = None, controls: List[ldap3.
↪protocol.rfc4511.Control] = None, skip_validation: bool = False, maximum_nesting_
↪depth: int = None) -> List[Dict[Union[str, ms_active_directory.core.ad_objects.
↪ADGroup], List[ms_active_directory.core.ad_objects.ADGroup]]]
    Find the members of a group in the domain, along with attributes of the members.

    :param groups: Either a string name of a group or ADGroup to look up the members of.
    :param attributes_to_lookup: Attributes to look up about the members of each group.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all members exist and do not raise an error␣
↪if we fail to look one up.
                            Instead, a placeholder object will be used for members that␣
↪could not be found.
                            Defaults to False.
    :param maximum_nesting_depth: A limit to the number of levels of nesting to recurse␣
↪beyond the first lookup.
                                  A level of 0 makes this behave the same as find_
↪members_of_groups and a level of
                                  None means recurse until we've gone through all␣
↪nesting. Defaults to None.
    :return: A list of dictionaries mapping groups to objects representing the group's␣
↪members.
             The first dictionary maps the input groups to members; the second␣
↪dictionary maps the groups that
             were members of the groups in the first dictionary to their members, and so␣
↪on and so forth.
             The objects may be of type ADUser, ADComputer, ADGroup, etc. - this␣
↪function attempts to cast all
```

(continues on next page)

(continued from previous page)

```
             member objects to the most accurate object type representing them. ADObject␣
↪will be used for members
             that do not match any of the more specific object types in the library
             (e.g. foreign security principals).
    :raises: InvalidLdapParameterException if the group is not a string or ADGroup
    :raises: ObjectNotFoundException if the group cannot be found.
    :raises: DomainSearchException if skip_validation is False and any group members␣
↪cannot be found.
```

Adding things to groups:

```
add_computers_to_groups(self, computers_to_add: List[Union[str, ms_active_directory.core.
↪ad_objects.ADComputer]], groups_to_add_them_to: List[Union[str, ms_active_directory.
↪core.ad_objects.ADGroup]], stop_and_rollback_on_error: bool = True, controls:␣
↪List[ldap3.protocol.rfc4511.Control] = None, skip_validation: bool = False) ->␣
↪List[Union[str, ms_active_directory.core.ad_objects.ADGroup]]
    Add one or more computers to one or more groups as members. This function attempts␣
↪to be idempotent
    and will not re-add computers that are already members.

    :param computers_to_add: A list of computers to add to other groups. These may␣
↪either be ADComputer objects or
                              string name identifiers for computers.
    :param groups_to_add_them_to: A list of groups to add members to. These may either␣
↪be ADGroup objects or string
                                   name identifiers for groups.
    :param stop_and_rollback_on_error: If true, failure to add any of the computers to␣
↪any of the groups will
                                        cause us to try and remove any computers that␣
↪have been added from any of the
                                        groups that we successfully added members to.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                      whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                      may succeed or fail based on their availability.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                             Defaults to False. This can be used to make this function␣
↪more performant when
                             the caller knows all the distinguished names being specified␣
↪are valid, as it
                             performs far fewer queries.
    :returns: A list of groups that successfully had members added. This will always be␣
↪all the groups unless
              stop_and_rollback_on_error is False.
    :raises: MembershipModificationException if we fail to add groups to any other␣
↪groups and rollback succeeds.
    :raises: MembershipModificationRollbackException if we fail to add any groups to␣
↪other groups, and then also
             fail when removing the groups that had been added successfully, leaving us␣
↪in a partially completed
```

(continues on next page)

(continued from previous page)

```
             state. This may occur if the session has permission to add members but not␣
↪to remove members.

add_groups_to_groups(self, groups_to_add: List[Union[str, ms_active_directory.core.ad_
↪objects.ADGroup]], groups_to_add_them_to: List[Union[str, ms_active_directory.core.ad_
↪objects.ADGroup]], stop_and_rollback_on_error: bool = True, controls: List[ldap3.
↪protocol.rfc4511.Control] = None, skip_validation: bool = False) -> List[Union[str, ms_
↪active_directory.core.ad_objects.ADGroup]]
    Add one or more groups to one or more other groups as members. This function␣
↪attempts to be idempotent
    and will not re-add groups that are already members.

    :param groups_to_add: A list of groups to add to other groups. These may either be␣
↪ADGroup objects or string
                          name identifiers for groups.
    :param groups_to_add_them_to: A list of groups to add members to. These may either␣
↪be ADGroup objects or string
                                  name identifiers for groups.
    :param stop_and_rollback_on_error: If true, failure to add any of the groups to any␣
↪of the other groups will
                                       cause us to try and remove any groups that have␣
↪been added from any of the
                                       groups that we successfully added members to.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A list of groups that successfully had members added. This will always be␣
↪all the groups unless
              stop_and_rollback_on_error is False.
    :raises: MembershipModificationException if any groups being added also exist in the␣
↪groups to add them to, or
             if we fail to add groups to any other groups and rollback succeeds.
    :raises: MembershipModificationRollbackException if we fail to add any groups to␣
↪other groups, and then also
             fail when removing the groups that had been added successfully, leaving us␣
↪in a partially completed
             state. This may occur if the session has permission to add members but not␣
↪to remove members.
add_users_to_groups(self, users_to_add: List[Union[str, ms_active_directory.core.ad_
↪objects.ADUser]], groups_to_add_them_to: List[Union[str, ms_active_directory.core.ad_
↪objects.ADGroup]], stop_and_rollback_on_error: bool = True, controls: List[ldap3.
↪protocol.rfc4511.Control] = None, skip_validation: bool = False) -> List[Union[str, ms_
↪active_directory.core.ad_objects.ADGroup]]
```

(continues on next page)

(continued from previous page)

```
    Add one or more users to one or more groups as members. This function attempts to be␣
↪idempotent
    and will not re-add users that are already members.

    :param users_to_add: A list of users to add to other groups. These may either be␣
↪ADUser objects or string
                         name identifiers for users.
    :param groups_to_add_them_to: A list of groups to add members to. These may either␣
↪be ADGroup objects or string
                                  name identifiers for groups.
    :param stop_and_rollback_on_error: If true, failure to add any of the users to any␣
↪of the groups will
                                       cause us to try and remove any users that have␣
↪been added from any of the
                                       groups that we successfully added members to.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A list of groups that successfully had members added. This will always be␣
↪all the groups unless
              stop_and_rollback_on_error is False.
    :raises: MembershipModificationException if we fail to add groups to any other␣
↪groups and rollback succeeds.
    :raises: MembershipModificationRollbackException if we fail to add any groups to␣
↪other groups, and then also
             fail when removing the groups that had been added successfully, leaving us␣
↪in a partially completed
             state. This may occur if the session has permission to add members but not␣
↪to remove members.
```

Removing things from groups:

```
remove_computers_from_groups(self, computers_to_remove: List[Union[str, ms_active_
↪directory.core.ad_objects.ADComputer]], groups_to_remove_them_from: List[Union[str, ms_
↪active_directory.core.ad_objects.ADGroup]], stop_and_rollback_on_error: bool = True,␣
↪controls: List[ldap3.protocol.rfc4511.Control] = None, skip_validation: bool = False) -
↪> List[Union[str, ms_active_directory.core.ad_objects.ADGroup]]
    Remove one or more computers from one or more groups as members. This function␣
↪attempts to be idempotent
    and will not remove computers that are not already members.

    :param computers_to_remove: A list of computers to remove from groups. These may␣
↪either be ADComputer objects or
```

(continues on next page)

(continued from previous page)

```
                                string name identifiers for computers.
    :param groups_to_remove_them_from: A list of groups to remove members from. These␣
↪may either be ADGroup objects
                                       or string name identifiers for groups.
    :param stop_and_rollback_on_error: If true, failure to remove any of the computers␣
↪from any of the groups
                                       will cause us to try and add any computers that␣
↪have been removed back to any
                                       of the groups that we successfully removed␣
↪members from.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A list of groups that successfully had members removed. This will always␣
↪be all the groups unless
              stop_and_rollback_on_error is False.
    :raises: MembershipModificationException if we fail to remove computers from any␣
↪groups and rollback succeeds
    :raises: MembershipModificationRollbackException if we fail to remove any computers␣
↪from groups, and then
             also fail when adding the computers that had been removed successfully,␣
↪leaving us in a partially
             completed state. This may occur if the session has permission to remove␣
↪members but not to add members.

remove_groups_from_groups(self, groups_to_remove: List[Union[str, ms_active_directory.
↪core.ad_objects.ADGroup]], groups_to_remove_them_from: List[Union[str, ms_active_
↪directory.core.ad_objects.ADGroup]], stop_and_rollback_on_error: bool = True,␣
↪controls: List[ldap3.protocol.rfc4511.Control] = None, skip_validation: bool = False) -
↪> List[Union[str, ms_active_directory.core.ad_objects.ADGroup]]
    Remove one or more groups from one or more groups as members. This function attempts␣
↪to be idempotent
    and will not remove groups that are not already members.

    :param groups_to_remove: A list of groups to remove from other groups. These may␣
↪either be ADGroup objects or
                             string name identifiers for groups.
    :param groups_to_remove_them_from: A list of groups to remove members from. These␣
↪may either be ADGroup objects
                                       or string name identifiers for groups.
    :param stop_and_rollback_on_error: If true, failure to remove any of the groups from␣
↪any of the other groups
                                       will cause us to try and add any groups that have␣
↪been removed back to any
```

(continues on next page)

(continued from previous page)

```
                                       of the groups that we successfully removed␣
↪members from.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A list of groups that successfully had members removed. This will always␣
↪be all the groups unless
              stop_and_rollback_on_error is False.
    :raises: MembershipModificationException if we fail to remove groups from any other␣
↪groups and rollback succeeds
    :raises: MembershipModificationRollbackException if we fail to remove any groups␣
↪from other groups, and then
             also fail when adding the groups that had been removed successfully,␣
↪leaving us in a partially
             completed state. This may occur if the session has permission to remove␣
↪members but not to add members.

remove_users_from_groups(self, users_to_remove: List[Union[str, ms_active_directory.core.
↪ad_objects.ADUser]], groups_to_remove_them_from: List[Union[str, ms_active_directory.
↪core.ad_objects.ADGroup]], stop_and_rollback_on_error: bool = True, controls:␣
↪List[ldap3.protocol.rfc4511.Control] = None, skip_validation: bool = False) ->␣
↪List[Union[str, ms_active_directory.core.ad_objects.ADGroup]]
    Remove one or more users from one or more groups as members. This function attempts␣
↪to be idempotent
    and will not remove users that are not already members.

    :param users_to_remove: A list of users to remove from groups. These may either be␣
↪ADUsers objects or
                            string name identifiers for users.
    :param groups_to_remove_them_from: A list of groups to remove members from. These␣
↪may either be ADGroup objects
                                       or string name identifiers for groups.
    :param stop_and_rollback_on_error: If true, failure to remove any of the users from␣
↪any of the groups
                                       will cause us to try and add any users that have␣
↪been removed back to any
                                       of the groups that we successfully removed␣
↪members from.
    :param controls: A list of LDAP controls to use when performing the search. These␣
↪can be used to specify
                     whether or not certain properties/attributes are critical, which␣
↪influences whether a search
                     may succeed or fail based on their availability.
```

(continues on next page)

(continued from previous page)

```
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A list of groups that successfully had members removed. This will always␣
↪be all the groups unless
              stop_and_rollback_on_error is False.
    :raises: MembershipModificationException if we fail to remove users from any groups␣
↪and rollback succeeds
    :raises: MembershipModificationRollbackException if we fail to remove any users from␣
↪groups, and then
             also fail when adding the users that had been removed successfully, leaving␣
↪us in a partially
             completed state. This may occur if the session has permission to remove␣
↪members but not to add members.
```

Modifying Records Within the Domain

There exist a number of functions for modifying records. For users, groups, and computers there exist functions for modifying their attributes, either by appending values to them or overwriting them. There's also functions for modifying the security descriptors of objects in order to change the permissions other principals have on them.

Appending values to user, computer, and group attributes atomically:

```
atomic_append_to_attribute_for_computer(self, computer: Union[str, ms_active_directory.
↪core.ad_objects.ADComputer], attribute: str, value, controls: List[ldap3.protocol.
↪rfc4511.Control] = None, raise_exception_on_failure: bool = True, skip_validation:␣
↪bool = False) -> bool
    Atomically append a value to an attribute for a computer in the domain.

    :param computer: Either an ADComputer object or string name referencing the computer␣
↪to be modified.
    :param attribute: A string specifying the name of the LDAP attribute to be appended␣
↪to.
    :param value: The value to append to the attribute. Value may either be a primitive,␣
↪such as a string, bytes,
                  or a number, if a single value will be appended. Value may also be an␣
↪iterable such as a set or
                  a list if a multi-valued attribute will be appended to, in order to␣
↪append multiple new values
                  to it at once.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
```

(continues on next page)

(continued from previous page)

```
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a distinguished name is specified and cannot be␣
↪found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.

atomic_append_to_attribute_for_group(self, group: Union[str, ms_active_directory.core.ad_
↪objects.ADGroup], attribute: str, value, controls: List[ldap3.protocol.rfc4511.
↪Control] = None, raise_exception_on_failure: bool = True, skip_validation: bool =␣
↪False) -> bool
    Atomically append a value to an attribute for a group in the domain.

    :param group: Either an ADGroup object or string name referencing the group to be␣
↪modified.
    :param attribute: A string specifying the name of the LDAP attribute to be appended␣
↪to.
    :param value: The value to append to the attribute. Value may either be a primitive,␣
↪such as a string, bytes,
                  or a number, if a single value will be appended. Value may also be an␣
↪iterable such as a set or
                  a list if a multi-valued attribute will be appended to, in order to␣
↪append multiple new values
                  to it at once.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a distinguished name is specified and cannot be␣
↪found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.
```

(continues on next page)

(continued from previous page)

```
atomic_append_to_attribute_for_object(self, ad_object: Union[str, ms_active_directory.
↪core.ad_objects.ADObject], attribute: str, value, controls: List[ldap3.protocol.
↪rfc4511.Control] = None, raise_exception_on_failure: bool = True, skip_validation:␣
↪bool = False) -> bool
    Atomically append a value to an attribute for an object in the domain.

    :param ad_object: Either an ADObject object or string distinguished name referencing␣
↪the object to be modified.
    :param attribute: A string specifying the name of the LDAP attribute to be appended␣
↪to.
    :param value: The value to append to the attribute. Value may either be a primitive,␣
↪such as a string, bytes,
                  or a number, if a single value will be appended. Value may also be an␣
↪iterable such as a set or
                  a list if a multi-valued attribute will be appended to, in order to␣
↪append multiple new values
                  to it at once.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a distinguished name is specified and cannot be␣
↪found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.

atomic_append_to_attribute_for_user(self, user: Union[str, ms_active_directory.core.ad_
↪objects.ADUser], attribute: str, value, controls: List[ldap3.protocol.rfc4511.Control]␣
↪= None, raise_exception_on_failure: bool = True, skip_validation: bool = False) -> bool
    Atomically append a value to an attribute for a user in the domain.

    :param user: Either an ADUser object or string name referencing the user to be␣
↪modified.
    :param attribute: A string specifying the name of the LDAP attribute to be appended␣
↪to.
    :param value: The value to append to the attribute. Value may either be a primitive,␣
↪such as a string, bytes,
                  or a number, if a single value will be appended. Value may also be an␣
↪iterable such as a set or
```

(continues on next page)

(continued from previous page)

```
                  a list if a multi-valued attribute will be appended to, in order to␣
↪append multiple new values
                  to it at once.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a distinguished name is specified and cannot be␣
↪found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.

atomic_append_to_attributes_for_computer(self, computer: Union[str, ms_active_directory.
↪core.ad_objects.ADComputer], attribute_to_value_map: dict, controls: List[ldap3.
↪protocol.rfc4511.Control] = None, raise_exception_on_failure: bool = True, skip_
↪validation: bool = False) -> bool
    Atomically append values to multiple attributes for a computer in the domain.

    :param computer: Either an ADComputer object or string name referencing the computer␣
↪to be modified.
    :param attribute_to_value_map: A dictionary mapping string LDAP attribute names to␣
↪values that will be used
                                   in the modification operation. Values may either be␣
↪primitives, such as strings,
                                   bytes, and numbers if a single value will be appended.
↪ Values may
                                   also be iterables such as sets and lists if multiple␣
↪values will be appended
                                   to the attributes.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
```

(continues on next page)

(continued from previous page)

```
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a distinguished name is specified and cannot be␣
↪found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.

atomic_append_to_attributes_for_group(self, group: Union[str, ms_active_directory.core.
↪ad_objects.ADGroup], attribute_to_value_map: dict, controls: List[ldap3.protocol.
↪rfc4511.Control] = None, raise_exception_on_failure: bool = True, skip_validation:␣
↪bool = False) -> bool
    Atomically append values to multiple attributes for a group in the domain.

    :param group: Either an ADGroup object or string name referencing the group to be␣
↪modified.
    :param attribute_to_value_map: A dictionary mapping string LDAP attribute names to␣
↪values that will be used
                                   in the modification operation. Values may either be␣
↪primitives, such as strings,
                                   bytes, and numbers if a single value will be appended.
↪ Values may
                                   also be iterables such as sets and lists if multiple␣
↪values will be appended
                                   to the attributes.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a distinguished name is specified and cannot be␣
↪found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.

atomic_append_to_attributes_for_object(self, ad_object: Union[str, ms_active_directory.
↪core.ad_objects.ADObject], attribute_to_value_map: dict, controls: List[ldap3.protocol.
↪rfc4511.Control] = None, raise_exception_on_failure: bool = True, skip_validation:␣
↪bool = False) -> bool
```

(continues on next page)

(continued from previous page)

```
    Atomically append values to multiple attributes for an object in the domain.

    :param ad_object: Either an ADObject object or string distinguished name referencing␣
↪the object to be modified.
    :param attribute_to_value_map: A dictionary mapping string LDAP attribute names to␣
↪values that will be used
                                   in the modification operation. Values may either be␣
↪primitives, such as strings,
                                   bytes, and numbers if a single value will be appended.
↪ Values may
                                   also be iterables such as sets and lists if multiple␣
↪values will be appended
                                   to the attributes.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a distinguished name is specified and cannot be␣
↪found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.

atomic_append_to_attributes_for_user(self, user: Union[str, ms_active_directory.core.ad_
↪objects.ADUser], attribute_to_value_map: dict, controls: List[ldap3.protocol.rfc4511.
↪Control] = None, raise_exception_on_failure: bool = True, skip_validation: bool =␣
↪False) -> bool
    Atomically append values to multiple attributes for a user in the domain.

    :param user: Either an ADUser object or string name referencing the user to be␣
↪modified.
    :param attribute_to_value_map: A dictionary mapping string LDAP attribute names to␣
↪values that will be used
                                   in the modification operation. Values may either be␣
↪primitives, such as strings,
                                   bytes, and numbers if a single value will be appended.
↪ Values may
                                   also be iterables such as sets and lists if multiple␣
↪values will be appended
                                   to the attributes.
```

(continues on next page)

(continued from previous page)

```
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a distinguished name is specified and cannot be␣
↪found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.
```

Overwriting values for user, group, and computer attributes:

```
    overwrite_attribute_for_computer(self, computer: Union[str, ms_active_directory.core.
↪ad_objects.ADComputer], attribute: str, value, controls: List[ldap3.protocol.rfc4511.
↪Control] = None, raise_exception_on_failure: bool = True, skip_validation: bool =␣
↪False) -> bool
    Atomically overwrite the value of an attribute for a computer in the domain.

    :param computer: Either an ADComputer object or string name referencing the computer␣
↪to be modified.
    :param attribute: A string specifying the name of the LDAP attribute to be␣
↪overwritten.
    :param value: The value to set for the attribute. Value may either be a primitive,␣
↪such as a string, bytes,
                  or a number, if a single value will be set. Value may also be an␣
↪iterable such as a set or
                  a list if a multi-valued attribute will be set.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
```

(continues on next page)

(continued from previous page)

```
    :raises: ObjectNotFoundException if a distinguished name is specified and cannot be␣
↪found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.

overwrite_attribute_for_group(self, group: Union[str, ms_active_directory.core.ad_
↪objects.ADGroup], attribute: str, value, controls: List[ldap3.protocol.rfc4511.
↪Control] = None, raise_exception_on_failure: bool = True, skip_validation: bool =␣
↪False) -> bool
    Atomically overwrite the value of an attribute for a group in the domain.

    :param group: Either an ADUser object or string name referencing the group to be␣
↪modified.
    :param attribute: A string specifying the name of the LDAP attribute to be␣
↪overwritten.
    :param value: The value to set for the attribute. Value may either be a primitive,␣
↪such as a string, bytes,
                  or a number, if a single value will be set. Value may also be an␣
↪iterable such as a set or
                  a list if a multi-valued attribute will be set.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a distinguished name is specified and cannot be␣
↪found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.

overwrite_attribute_for_object(self, ad_object: Union[str, ms_active_directory.core.ad_
↪objects.ADObject], attribute: str, value, controls: List[ldap3.protocol.rfc4511.
↪Control] = None, raise_exception_on_failure: bool = True, skip_validation: bool =␣
↪False) -> bool
    Atomically overwrite the value of an attribute for an object in the domain.
```

(continues on next page)

(continued from previous page)

```
    :param ad_object: Either an ADObject object or string distinguished name referencing␣
↪the object to be modified.
    :param attribute: A string specifying the name of the LDAP attribute to be␣
↪overwritten.
    :param value: The value to set for the attribute. Value may either be a primitive,␣
↪such as a string, bytes,
                  or a number, if a single value will be set. Value may also be an␣
↪iterable such as a set or
                  a list if a multi-valued attribute will be set.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a distinguished name is specified and cannot be␣
↪found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.

overwrite_attribute_for_user(self, user: Union[str, ms_active_directory.core.ad_objects.
↪ADUser], attribute: str, value, controls: List[ldap3.protocol.rfc4511.Control] = None,␣
↪raise_exception_on_failure: bool = True, skip_validation: bool = False) -> bool
    Atomically overwrite the value of an attribute for a user in the domain.

    :param user: Either an ADUser object or string name referencing the user to be␣
↪modified.
    :param attribute: A string specifying the name of the LDAP attribute to be␣
↪overwritten.
    :param value: The value to set for the attribute. Value may either be a primitive,␣
↪such as a string, bytes,
                  or a number, if a single value will be set. Value may also be an␣
↪iterable such as a set or
                  a list if a multi-valued attribute will be set.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
```

(continues on next page)

(continued from previous page)

```
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a distinguished name is specified and cannot be␣
↪found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.

overwrite_attributes_for_computer(self, computer: Union[str, ms_active_directory.core.ad_
↪objects.ADComputer], attribute_to_value_map: dict, controls: List[ldap3.protocol.
↪rfc4511.Control] = None, raise_exception_on_failure: bool = True, skip_validation:␣
↪bool = False) -> bool
    Atomically overwrite values of multiple attributes for a computer in the domain.

    :param computer: Either an ADComputer object or string name referencing the computer␣
↪to have attributes
                     overwritten.
    :param attribute_to_value_map: A dictionary mapping string LDAP attribute names to␣
↪values that will be used
                                   in the modification operation. Values may either be␣
↪primitives, such as strings,
                                   bytes, and numbers if a single value will set. Values␣
↪may also be iterables
                                   such as sets and lists if an attribute is multi-
↪valued and multiple values will
                                   be set.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a name is specified and cannot be found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.
```

(continues on next page)

(continued from previous page)

```
overwrite_attributes_for_group(self, group: Union[str, ms_active_directory.core.ad_
↪objects.ADGroup], attribute_to_value_map: dict, controls: List[ldap3.protocol.rfc4511.
↪Control] = None, raise_exception_on_failure: bool = True, skip_validation: bool =␣
↪False) -> bool
    Atomically overwrite values of multiple attributes for a group in the domain.

    :param group: Either an ADGroup object or string name referencing the group to have␣
↪attributes overwritten.
    :param attribute_to_value_map: A dictionary mapping string LDAP attribute names to␣
↪values that will be used
                                   in the modification operation. Values may either be␣
↪primitives, such as strings,
                                   bytes, and numbers if a single value will set. Values␣
↪may also be iterables
                                   such as sets and lists if an attribute is multi-
↪valued and multiple values will
                                   be set.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a name is specified and cannot be found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.

overwrite_attributes_for_object(self, ad_object: Union[str, ms_active_directory.core.ad_
↪objects.ADObject], attribute_to_value_map: dict, controls: List[ldap3.protocol.rfc4511.
↪Control] = None, raise_exception_on_failure: bool = True, skip_validation: bool =␣
↪False) -> bool
    Atomically overwrite values of multiple attributes for an object in the domain.

    :param ad_object: Either an ADObject object or string distinguished name referencing␣
↪the object to be modified.
    :param attribute_to_value_map: A dictionary mapping string LDAP attribute names to␣
↪values that will be used
                                   in the modification operation. Values may either be␣
↪primitives, such as strings,
                                   bytes, and numbers if a single value will set. Values␣
↪may also be iterables
```

(continues on next page)

(continued from previous page)

```
                                   such as sets and lists if an attribute is multi-
↪valued and multiple values will
                                   be set.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a distinguished name is specified and cannot be␣
↪found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.

overwrite_attributes_for_user(self, user: Union[str, ms_active_directory.core.ad_objects.
↪ADUser], attribute_to_value_map: dict, controls: List[ldap3.protocol.rfc4511.Control]␣
↪= None, raise_exception_on_failure: bool = True, skip_validation: bool = False) -> bool
    Atomically overwrite values of multiple attributes for a user in the domain.

    :param user: Either an ADUser object or string name referencing the user to have␣
↪attributes overwritten.
    :param attribute_to_value_map: A dictionary mapping string LDAP attribute names to␣
↪values that will be used
                                   in the modification operation. Values may either be␣
↪primitives, such as strings,
                                   bytes, and numbers if a single value will set. Values␣
↪may also be iterables
                                   such as sets and lists if an attribute is multi-
↪valued and multiple values will
                                   be set.
    :param controls: LDAP controls to use during the modification operation.
    :param raise_exception_on_failure: If true, an exception will be raised with␣
↪additional details if the modify
                                       fails.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
```

(continues on next page)

(continued from previous page)

```
    :returns: True if the operation succeeds, False otherwise.
    :raises: InvalidLdapParameterException if any attributes or values are malformed.
    :raises: ObjectNotFoundException if a name is specified and cannot be found
    :raises: AttributeModificationException if raise_exception_on_failure is True and we␣
↪fail
    :raises: Other LDAP exceptions from the ldap3 library if the connection is␣
↪configured to raise exceptions and
             issues are seen such as determining that a value is malformed based on the␣
↪server schema.
```

Finding security descriptors:

```
find_security_descriptor_for_computer(self, computer: Union[str, ms_active_directory.
↪core.ad_objects.ADComputer], include_sacl: bool = False, skip_validation: bool =␣
↪False) -> ms_active_directory.environment.security.security_descriptor_utils.
↪SelfRelativeSecurityDescriptor
    Given a computer, find its security descriptor. The security descriptor will be␣
↪returned as a
    SelfRelativeSecurityDescriptor object.

    :param computer: The computer for which we will read the security descriptor. This␣
↪may be an ADComputer object
                     or a string name identifying the computer (in which case it will be␣
↪looked up).
    :param include_sacl: If true, we will attempt to read the System ACL for the user in␣
↪addition to the
                         Discretionary ACL and owner information when reading the␣
↪security descriptor. This is
                         more privileged than just getting the Discretionary ACL and␣
↪owner information.
                         Defaults to False.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :raises: ObjectNotFoundException if the computer cannot be found.
    :raises: InvalidLdapParameterException if the computer specified is not a string or␣
↪an ADComputer object
    :raises: SecurityDescriptorDecodeException if we fail to decode the security␣
↪descriptor.

find_security_descriptor_for_group(self, group: Union[str, ms_active_directory.core.ad_
↪objects.ADGroup], include_sacl: bool = False, skip_validation: bool = False) -> ms_
↪active_directory.environment.security.security_descriptor_utils.
↪SelfRelativeSecurityDescriptor
    Given a group, find its security descriptor. The security descriptor will be␣
↪returned as a
    SelfRelativeSecurityDescriptor object.
```

(continues on next page)

(continued from previous page)

```
    :param group: The group for which we will read the security descriptor. This may be␣
↪an ADGroup object or a
                  string name identifying the group (in which case it will be looked up).
    :param include_sacl: If true, we will attempt to read the System ACL for the group␣
↪in addition to the
                         Discretionary ACL and owner information when reading the␣
↪security descriptor. This is
                         more privileged than just getting the Discretionary ACL and␣
↪owner information.
                         Defaults to False.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :raises: ObjectNotFoundException if the group cannot be found.
    :raises: InvalidLdapParameterException if the group specified is not a string or an␣
↪ADGroup object
    :raises: SecurityDescriptorDecodeException if we fail to decode the security␣
↪descriptor.

find_security_descriptor_for_object(self, ad_object: Union[str, ms_active_directory.core.
↪ad_objects.ADObject], include_sacl: bool = False, skip_validation: bool = False) -> ms_
↪active_directory.environment.security.security_descriptor_utils.
↪SelfRelativeSecurityDescriptor
    Given an object, find its security descriptor. The security descriptor will be␣
↪returned as a
    SelfRelativeSecurityDescriptor object.

    :param ad_object: The object for which we will read the security descriptor. This␣
↪may be an ADObject object or a
                      string distinguished identifying the object.
    :param include_sacl: If true, we will attempt to read the System ACL for the object␣
↪in addition to the
                         Discretionary ACL and owner information when reading the␣
↪security descriptor. This is
                         more privileged than just getting the Discretionary ACL and␣
↪owner information.
                         Defaults to False.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :raises: ObjectNotFoundException if the object cannot be found.
    :raises: InvalidLdapParameterException if the ad_object specified is not a string DN␣
↪or an ADObject object
    :raises: SecurityDescriptorDecodeException if we fail to decode the security␣
↪descriptor.
```

(continues on next page)

(continued from previous page)

```
find_security_descriptor_for_user(self, user: Union[str, ms_active_directory.core.ad_
↪objects.ADUser], include_sacl: bool = False, skip_validation: bool = False) -> ms_
↪active_directory.environment.security.security_descriptor_utils.
↪SelfRelativeSecurityDescriptor
    Given a user, find its security descriptor. The security descriptor will be returned␣
↪as a
    SelfRelativeSecurityDescriptor object.

    :param user: The user for which we will read the security descriptor. This may be an␣
↪ADUser object or a
                 string name identifying the user (in which case it will be looked up).
    :param include_sacl: If true, we will attempt to read the System ACL for the user in␣
↪addition to the
                         Discretionary ACL and owner information when reading the␣
↪security descriptor. This is
                         more privileged than just getting the Discretionary ACL and␣
↪owner information.
                         Defaults to False.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :raises: ObjectNotFoundException if the user cannot be found.
    :raises: InvalidLdapParameterException if the user specified is not a string or an␣
↪ADUser object
    :raises: SecurityDescriptorDecodeException if we fail to decode the security␣
↪descriptor.
```

Overwriting security descriptors:

```
set_computer_security_descriptor(self, computer: Union[str, ms_active_directory.core.ad_
↪objects.ADComputer], new_sec_descriptor: ms_active_directory.environment.security.
↪security_descriptor_utils.SelfRelativeSecurityDescriptor, raise_exception_on_failure:␣
↪bool = True, skip_validation: bool = False) -> bool
    Set the security descriptor on an Active Directory computer. This can be used to␣
↪change the owner of a
    computer in AD, change its permission ACEs, etc.

    :param computer: Either an ADComputer object or string name referencing the computer␣
↪to be modified.
    :param new_sec_descriptor: The security descriptor to set on the object.
    :param raise_exception_on_failure: If true, raise an exception when modifying the␣
↪object fails instead of
                                       returning False.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
```

(continues on next page)

(continued from previous page)

```
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A boolean indicating success.
    :raises: InvalidLdapParameterException if computer is not a string or ADComputer␣
↪object
    :raises: ObjectNotFoundException if a string DN is specified and it cannot be found
    :raises: PermissionDeniedException if we fail to modify the Security Descriptor and␣
↪raise_exception_on_failure
             is true

set_group_security_descriptor(self, group: Union[str, ms_active_directory.core.ad_
↪objects.ADGroup], new_sec_descriptor: ms_active_directory.environment.security.
↪security_descriptor_utils.SelfRelativeSecurityDescriptor, raise_exception_on_failure:␣
↪bool = True, skip_validation: bool = False) -> bool
    Set the security descriptor on an Active Directory group. This can be used to change␣
↪the owner of an
    group in AD, change its permission ACEs, etc.

    :param group: Either an ADGroup object or string name referencing the group to be␣
↪modified
    :param new_sec_descriptor: The security descriptor to set on the object.
    :param raise_exception_on_failure: If true, raise an exception when modifying the␣
↪object fails instead of
                                       returning False.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A boolean indicating success.
    :raises: ObjectNotFoundException if a string DN is specified and it cannot be found
    :raises: PermissionDeniedException if we fail to modify the Security Descriptor and␣
↪raise_exception_on_failure
             is true

set_object_security_descriptor(self, ad_object: Union[str, ms_active_directory.core.ad_
↪objects.ADObject], new_sec_descriptor: ms_active_directory.environment.security.
↪security_descriptor_utils.SelfRelativeSecurityDescriptor, raise_exception_on_failure:␣
↪bool = True, skip_validation: bool = False) -> bool
    Set the security descriptor on an Active Directory object. This can be used to␣
↪change the owner of an
    object in AD, change its permission ACEs, etc.

    :param ad_object: Either an ADObject object or string distinguished name referencing␣
↪the object to be modified
    :param new_sec_descriptor: The security descriptor to set on the object.
    :param raise_exception_on_failure: If true, raise an exception when modifying the␣
↪object fails instead of
                                       returning False.
```

(continues on next page)

(continued from previous page)

```
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A boolean indicating success.
    :raises: ObjectNotFoundException if a string DN is specified and it cannot be found
    :raises: PermissionDeniedException if we fail to modify the Security Descriptor and␣
↪raise_exception_on_failure
             is true

set_user_security_descriptor(self, user: Union[str, ms_active_directory.core.ad_objects.
↪ADUser], new_sec_descriptor: ms_active_directory.environment.security.security_
↪descriptor_utils.SelfRelativeSecurityDescriptor, raise_exception_on_failure: bool =␣
↪True, skip_validation: bool = False) -> bool
    Set the security descriptor on an Active Directory object. This can be used to␣
↪change the owner of an
    user in AD, change its permission ACEs, etc.

    :param user: Either an ADUser object or string name referencing the user to be␣
↪modified.
    :param new_sec_descriptor: The security descriptor to set on the object.
    :param raise_exception_on_failure: If true, raise an exception when modifying the␣
↪object fails instead of
                                       returning False.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A boolean indicating success.
    :raises: InvalidLdapParameterException if user is not a string or ADUser object
    :raises: ObjectNotFoundException if a string DN is specified and it cannot be found
    :raises: PermissionDeniedException if we fail to modify the Security Descriptor and␣
↪raise_exception_on_failure
             is true
```

Appending permissions to security descriptors:

```
add_permission_to_computer_security_descriptor(self, computer: Union[str, ms_active_
↪directory.core.ad_objects.ADComputer], sids_to_grant_permissions_to: List[Union[str,␣
↪ms_active_directory.environment.security.security_descriptor_utils.ObjectSid, ms_
↪active_directory.environment.security.security_config_constants.WellKnownSID]], access_
↪masks_to_add: List[ms_active_directory.environment.security.security_descriptor_utils.
↪AccessMask] = None, rights_guids_to_add: List[Union[ms_active_directory.environment.
↪security.ad_security_guids.ADRightsGuid, str]] = None, read_property_guids_to_add:␣
↪List[str] = None, write_property_guids_to_add: List[str] = None, raise_exception_on_
↪failure: bool = True, skip_validation: bool = False) -> bool
```

(continues on next page)

(continued from previous page)

```
    Add specified permissions to the security descriptor on a computer for specified␣
↪SIDs.
    This can be used to grant 1 or more other users/groups/computers/etc. the right to␣
↪take broad actions or narrow
    privileged actions on the computer, via adding access masks or rights guids␣
↪respectively. It can also give
    1 or more users/groups/computers/etc. the ability to read or write specific␣
↪properties on the user by
    specifying read or write property guids to add.

    This can, as an example, take a computer and give a user the right to delete it. Or␣
↪take a computer
    and give a list of computers the right to read and write the user's owner SID. Or␣
↪take a computer and let
    another user reset their password without needing the current one. Etc. Etc.

    :param computer: An ADComputer or String distinguished name, referring to the␣
↪computer that will have the
                     permissions on it modified.
    :param sids_to_grant_permissions_to: SIDs referring to the other entities that will␣
↪be given new permissions
                                         on the user. These may be ObjectSID objects,␣
↪SID strings, or
                                         WellKnownSIDs.
    :param access_masks_to_add: A list of AccessMask objects to grant to the SIDs. These␣
↪represent broad categories
                               of actions, such as GENERIC_READ and GENERIC_WRITE.
    :param rights_guids_to_add: A list of rights guids to grant to the SIDs. These may␣
↪be specified as strings or
                               as ADRightsGuid enums, and represent narrower␣
↪permissions to grant to the SIDs for
                               targeted actions such as Unexpire_Password or Apply_
↪Group_Policy. Some of these
                               do not make logical sense to use in all contexts, as␣
↪some rights guids only have
                               meaning in a self-relative context, or only have meaning␣
↪on some object types.
                               It is left up to the caller to decide what is meaningful.
    :param read_property_guids_to_add: A list of property guids that represent␣
↪properties of the computer that the
                                       SIDs will be granted the right to read. These␣
↪must be strings.
    :param write_property_guids_to_add: A list of property guids that represent␣
↪properties of the computer that the
                                        SIDs will be granted the right to write. These␣
↪must be strings.
    :param raise_exception_on_failure: A boolean indicating if an exception should be␣
↪raised if we fail to update
                                       the security descriptor, instead of returning␣
↪False. defaults to True
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
```

(continues on next page)

(continued from previous page)

```
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A boolean indicating if we succeeded in updating the security descriptor.
    :raises: InvalidLdapParameterException if any inputs are the wrong type.
    :raises: ObjectNotFoundException if the a string distinguished name is specified and␣
↪cannot be found.
    :raises: PermissionDeniedException if we fail to modify the Security Descriptor and␣
↪raise_exception_on_failure
             is true

add_permission_to_group_security_descriptor(self, group, sids_to_grant_permissions_to:␣
↪List[Union[str, ms_active_directory.environment.security.security_descriptor_utils.
↪ObjectSid, ms_active_directory.environment.security.security_config_constants.
↪WellKnownSID]], access_masks_to_add: List[ms_active_directory.environment.security.
↪security_descriptor_utils.AccessMask] = None, rights_guids_to_add: List[Union[ms_
↪active_directory.environment.security.ad_security_guids.ADRightsGuid, str]] = None,␣
↪read_property_guids_to_add: List[str] = None, write_property_guids_to_add: List[str] =␣
↪None, raise_exception_on_failure: bool = True, skip_validation: bool = False) -> bool
    Add specified permissions to the security descriptor on a group for specified SIDs.
    This can be used to grant 1 or more other users/groups/computers/etc. the right to␣
↪take broad actions or narrow
    privileged actions on the group, via adding access masks or rights guids␣
↪respectively. It can also give
    1 or more users/groups/computers/etc. the ability to read or write specific␣
↪properties on the group by
    specifying read or write property guids to add.

    This can, as an example, take a group and give another group the right to delete it.␣
↪Or take a group
    and give a list of computers the right to read the group's SID. Or take a group and␣
↪let another user
    add members to it. Etc. Etc.

    :param group: An ADGroup or String distinguished name, referring to the group that␣
↪will have the permissions on
                  it modified.
    :param sids_to_grant_permissions_to: SIDs referring to the other entities that will␣
↪be given new permissions
                                         on the group. These may be ObjectSID objects,␣
↪SID strings, or
                                         WellKnownSIDs.
    :param access_masks_to_add: A list of AccessMask objects to grant to the SIDs. These␣
↪represent broad categories
                                of actions, such as GENERIC_READ and GENERIC_WRITE.
    :param rights_guids_to_add: A list of rights guids to grant to the SIDs. These may␣
↪be specified as strings or
                                as ADRightsGuid enums, and represent narrower␣
↪permissions to grant to the SIDs for
                                targeted actions such as Unexpire_Password or Apply_
↪Group_Policy. Some of these
```

(continues on next page)

(continued from previous page)

```
                                do not make logical sense to use in all contexts, as␣
↪some rights guids only have
                                meaning in a self-relative context, or only have meaning␣
↪on some object types.
                                It is left up to the caller to decide what is meaningful.
    :param read_property_guids_to_add: A list of property guids that represent␣
↪properties of the group that the
                                       SIDs will be granted the right to read. These␣
↪must be strings.
    :param write_property_guids_to_add: A list of property guids that represent␣
↪properties of the group that the
                                        SIDs will be granted the right to write. These␣
↪must be strings.
    :param raise_exception_on_failure: A boolean indicating if an exception should be␣
↪raised if we fail to update
                                       the security descriptor, instead of returning␣
↪False. defaults to True
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A boolean indicating if we succeeded in updating the security descriptor.
    :raises: InvalidLdapParameterException if any inputs are the wrong type.
    :raises: ObjectNotFoundException if the a string distinguished name is specified and␣
↪cannot be found.
    :raises: PermissionDeniedException if we fail to modify the Security Descriptor and␣
↪raise_exception_on_failure
             is true

add_permission_to_object_security_descriptor(self, ad_object_to_modify: Union[str, ms_
↪active_directory.core.ad_objects.ADObject], sids_to_grant_permissions_to:␣
↪List[Union[str, ms_active_directory.environment.security.security_descriptor_utils.
↪ObjectSid, ms_active_directory.environment.security.security_config_constants.
↪WellKnownSID]], access_masks_to_add: List[ms_active_directory.environment.security.
↪security_descriptor_utils.AccessMask] = None, rights_guids_to_add: List[Union[ms_
↪active_directory.environment.security.ad_security_guids.ADRightsGuid, str]] = None,␣
↪read_property_guids_to_add: List[str] = None, write_property_guids_to_add: List[str] =␣
↪None, raise_exception_on_failure: bool = True, skip_validation: bool = False) -> bool
    Add specified permissions to the security descriptor on an object for specified SIDs.
    This can be used to grant 1 or more other users/groups/computers/etc. the right to␣
↪take broad actions or narrow
    privileged actions on the object, via adding access masks or rights guids␣
↪respectively. It can also give
    1 or more users/groups/computers/etc. the ability to read or write specific␣
↪properties on the object by
    specifying read or write property guids to add.

    This can, as an example, take a container object and give a user the right to delete␣
↪it. Or take a group object
```

(continues on next page)

(continued from previous page)

```
    and give a list of computers the right to read and write the group's members. Or␣
↪take a computer and let a user
    reset its password without needing the current one. Etc. Etc.

    :param ad_object_to_modify: An ADObject or String distinguished name, referring to␣
↪the object that will have
                                the permissions on it modified.
    :param sids_to_grant_permissions_to: SIDs referring to the other entities that will␣
↪be given new permissions
                                         on the object. These may be ObjectSID objects,␣
↪SID strings, or
                                         WellKnownSIDs.
    :param access_masks_to_add: A list of AccessMask objects to grant to the SIDs. These␣
↪represent broad categories
                                of actions, such as GENERIC_READ and GENERIC_WRITE.
    :param rights_guids_to_add: A list of rights guids to grant to the SIDs. These may␣
↪be specified as strings or
                                as ADRightsGuid enums, and represent narrower␣
↪permissions to grant to the SIDs for
                                targeted actions such as Unexpire_Password or Apply_
↪Group_Policy. Some of these
                                do not make logical sense to use in all contexts, as␣
↪some rights guids only have
                                meaning in a self-relative context, or only have meaning␣
↪on some object types.
                                It is left up to the caller to decide what is meaningful.
    :param read_property_guids_to_add: A list of property guids that represent␣
↪properties of the object that the
                                       SIDs will be granted the right to read. These␣
↪must be strings.
    :param write_property_guids_to_add: A list of property guids that represent␣
↪properties of the object that the
                                        SIDs will be granted the right to write. These␣
↪must be strings.
    :param raise_exception_on_failure: A boolean indicating if an exception should be␣
↪raised if we fail to update
                                       the security descriptor, instead of returning␣
↪False. defaults to True
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: A boolean indicating if we succeeded in updating the security descriptor.
    :raises: InvalidLdapParameterException if any inputs are the wrong type.
    :raises: ObjectNotFoundException if the a string distinguished name is specified and␣
↪cannot be found.
    :raises: PermissionDeniedException if we fail to modify the Security Descriptor and␣
↪raise_exception_on_failure
             is true
```

(continues on next page)

(continued from previous page)

```
add_permission_to_user_security_descriptor(self, user: Union[str, ms_active_directory.
↪core.ad_objects.ADUser], sids_to_grant_permissions_to: List[Union[str, ms_active_
↪directory.environment.security.security_descriptor_utils.ObjectSid, ms_active_
↪directory.environment.security.security_config_constants.WellKnownSID]], access_masks_
↪to_add: List[ms_active_directory.environment.security.security_descriptor_utils.
↪AccessMask] = None, rights_guids_to_add: List[Union[ms_active_directory.environment.
↪security.ad_security_guids.ADRightsGuid, str]] = None, read_property_guids_to_add:␣
↪List[str] = None, write_property_guids_to_add: List[str] = None, raise_exception_on_
↪failure: bool = True, skip_validation: bool = False) -> bool
    Add specified permissions to the security descriptor on a user for specified SIDs.
    This can be used to grant 1 or more other users/groups/computers/etc. the right to␣
↪take broad actions or narrow
    privileged actions on the user, via adding access masks or rights guids respectively.
↪ It can also give
    1 or more users/groups/computers/etc. the ability to read or write specific␣
↪properties on the user by
    specifying read or write property guids to add.

    This can, as an example, take a user and give another user the right to delete it.␣
↪Or take a user
    and give a list of computers the right to read and write the user's owner SID. Or␣
↪take a user and let another
    user reset their password without needing the current one. Etc. Etc.

    :param user: An ADUser or String distinguished name, referring to the user that will␣
↪have the permissions on it
                 modified.
    :param sids_to_grant_permissions_to: SIDs referring to the other entities that will␣
↪be given new permissions
                                         on the user. These may be ObjectSID objects,␣
↪SID strings, or
                                         WellKnownSIDs.
    :param access_masks_to_add: A list of AccessMask objects to grant to the SIDs. These␣
↪represent broad categories
                                of actions, such as GENERIC_READ and GENERIC_WRITE.
    :param rights_guids_to_add: A list of rights guids to grant to the SIDs. These may␣
↪be specified as strings or
                                as ADRightsGuid enums, and represent narrower␣
↪permissions to grant to the SIDs for
                                targeted actions such as Unexpire_Password or Apply_
↪Group_Policy. Some of these
                                do not make logical sense to use in all contexts, as␣
↪some rights guids only have
                                meaning in a self-relative context, or only have meaning␣
↪on some object types.
                                It is left up to the caller to decide what is meaningful.
    :param read_property_guids_to_add: A list of property guids that represent␣
↪properties of the user that the
                                       SIDs will be granted the right to read. These␣
↪must be strings.
    :param write_property_guids_to_add: A list of property guids that represent␣
↪properties of the user that the
```

(continues on next page)

(continued from previous page)

```
                    SIDs will be granted the right to write. These␣
↪must be strings.
  :param raise_exception_on_failure: A boolean indicating if an exception should be␣
↪raised if we fail to update
                   the security descriptor, instead of returning␣
↪False. defaults to True
  :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
              Defaults to False. This can be used to make this function␣
↪more performant when
              the caller knows all the distinguished names being specified␣
↪are valid, as it
              performs far fewer queries.
  :returns: A boolean indicating if we succeeded in updating the security descriptor.
  :raises: InvalidLdapParameterException if any inputs are the wrong type.
  :raises: ObjectNotFoundException if the a string distinguished name is specified and␣
↪cannot be found.
  :raises: PermissionDeniedException if we fail to modify the Security Descriptor and␣
↪raise_exception_on_failure
      is true
```

Creating and Taking Over Objects in the Domain

There exist functions for creating and taking over objects in the domain. Currently this is limited to computers:

```
create_computer(self, computer_name: str, computer_location: str = None, computer_
↪password: str = None, encryption_types: List[Union[str, ms_active_directory.
↪environment.security.security_config_constants.ADEncryptionType]] = None, hostnames:␣
↪List[str] = None, services: List[str] = None, supports_legacy_behavior: bool = False,␣
↪**additional_account_attributes) -> ms_active_directory.core.managed_ad_objects.
↪ManagedADComputer
  Use the session to create a computer in the domain and return a computer object.
  :param computer_name: The common name of the computer to create in the AD domain.␣
↪This
             will be used to determine the sAMAccountName, and if no␣
↪hostnames
             are specified then this will be used to determine the␣
↪hostnames for
             the computer.
  :param computer_location: The distinguished name of the location within the domain␣
↪where
               the computer will be created. It may be a relative␣
↪distinguished
               name (not including the domain component) or a full␣
↪distinguished
               name.  If not specified, defaults to CN=Computers which is
               standard for Active Directory.
  :param computer_password: The password to be set for the computer. This is␣
↪particularly
               useful to specify if the computer will be shared across␣
↪multiple
```

(continues on next page)

(continued from previous page)

```
                              applications or devices, or if pre-creating a computer for␣
↪another
                              application to use. If not specified, a random 120␣
↪character
                              password will be generated.
    :param encryption_types: The encryption types to set as supported on the computer in␣
↪AD.
                             These will also be used to generate kerberos keys for the␣
↪computer.
                             If not specified, defaults to [aes256-cts-hmac-sha1-96].
    :param hostnames: The hostnames to use for configuring the service principal names␣
↪of the
                      computer. These may be short hostnames or fully qualified domain␣
↪names.
                      If not specified, defaults to the "computer_name" as a short␣
↪hostname and
                      "computer_name.domain" as a fully qualified domain name.
    :param services: The services to enable on each hostname, which will be used with␣
↪hostnames
                     to generate the computer's service principal names. If not␣
↪specified,
                     defaults to ["HOST"] which is standard for Active Directory.
    :param supports_legacy_behavior: Does the computer being created support legacy␣
↪behavior such
                                     as NTLM authentication or UNC path addressing from␣
↪older windows
                                     clients? Defaults to False. Impacts the␣
↪restrictions on
                                     computer naming.
    :param additional_account_attributes: Additional LDAP attributes to set on the␣
↪account and their
                                          values. This is used to support power users␣
↪setting arbitrary
                                          attributes, such as "userCertificate" to set␣
↪the certificate
                                          for a computer that will use mutual TLS for␣
↪EXTERNAL SASL auth.
                                          This also allows overriding of some values␣
↪that are not explicit
                                          keyword arguments in order to avoid over-
↪complication, since most
                                          people won't set them (e.g.␣
↪userAccountControl).
    :returns: an ManagedADComputer object representing the computer.
    :raises: DomainJoinException if any of our validation of the specified attributes␣
↪fails or if anything
             specified conflicts with objects in the domain.
    :raises: ObjectCreationException if we fail to create the computer for a reason␣
↪unrelated to what we can
             easily validate in advance (e.g. permission issue)
```

(Continues on next page)

```
take_over_existing_computer(self, computer: Union[ms_active_directory.core.managed_ad_
↪objects.ManagedADComputer, ms_active_directory.core.ad_objects.ADObject, str],
↪computer_password: str = None, old_computer_password: str = None) -> ms_active_
↪directory.core.managed_ad_objects.ManagedADComputer
```

(continued from previous page)

```
    Use the session to take over a computer in the domain and return a computer object.
    This resets the computer's password so that nobody else can impersonate it, and reads
    the computer's attributes in order to create a computer object and return it.
    :param computer: This can be an ManagedADComputer or ADObject object representing␣
↪the computer that should be
                     taken over, or a string identifier for the computer.  If it is a␣
↪string, it should be
                     the common name or sAMAccountName of the computer to find in the AD␣
↪domain, or it can be
                     the distinguished name of a computer object.
                     If it appears to be a common name, not ending in $, a␣
↪sAMAccountName will
                     be derived to search for. If that cannot be found, then a search␣
↪will be
                     done for this as a common name. If no unique computer can be found␣
↪with that
                     search, then an exception will be raised.
    :param computer_password: The password to be set for the computer. This is␣
↪particularly
                              useful to specify if the computer will be shared across␣
↪multiple
                              applications or devices, or if pre-creating a computer for␣
↪another
                              application to use. If not specified, a random 120␣
↪character
                              password will be generated.
    :param old_computer_password: The current password for the computer. This is used to␣
↪reduce the level of
                                  permissions needed for the takeover operation.
    :returns: an ManagedADComputer object representing the computer.
    :raises: DomainJoinException if any of our validation of the specified attributes␣
↪fails or if anything
             specified conflicts with objects in the domain.
    :raises: ObjectNotFoundException if a computer cannot be found based on the name␣
↪specified.
```

Utility Functions For Account Management

There are a number of functions for basic account management actions. These include modifying passwords in various ways, disabling/enabling accounts, resetting lockouts, etc.

```
change_password_for_account(self, account: Union[str, ms_active_directory.core.ad_
↪objects.ADUser, ms_active_directory.core.ad_objects.ADComputer], new_password: str,␣
↪current_password: str, skip_validation: bool = False) -> bool
    Change a password for a user (includes computers) given the new desired password and␣
↪old desired password.
    When a password is changed, the old password is provided along with the new one, and␣
↪this significantly reduces
    the permissions needed in order to perform the operation. By default, any user can␣
↪perform CHANGE_PASSWORD for
    any other user.
```

(continues on next page)

(continued from previous page)

```
    This also avoids invalidating kerberos keys generated by the old password. Their␣
↪validity will depend on the
    domain's policy regarding old passwords/keys and their allowable use period after␣
↪change.

    :param account: The account whose password is being changed. This may either be a␣
↪string account name, to be
                    looked up, or an ADObject object.
    :param current_password: The current password for the account.
    :param new_password: The new password for the account. Technically, if None is␣
↪specified, then this behaves
                         as a RESET_PASSWORD operation.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds. If the operation fails, either an␣
↪exception will be raised or False
              will be returned depending on whether the ldap connection for this session␣
↪has "raise_exceptions"
              set to True or not.

disable_account(self, account: Union[str, ms_active_directory.core.ad_objects.ADUser, ms_
↪active_directory.core.ad_objects.ADComputer]) -> bool
    Disable a user account.
    :param account: The string name of the user/computer account to disable. This may␣
↪either be a
                    sAMAccountName, a distinguished name, or a unique common name. This␣
↪can also be an ADObject,
                    and the distinguished name will be extracted from it.
    :returns: True if the operation succeeds. If the operation fails, either an␣
↪exception will be raised or False
              will be returned depending on whether the ldap connection for this session␣
↪has "raise_exceptions"
              set to True or not.

enable_account(self, account: Union[str, ms_active_directory.core.ad_objects.ADComputer,␣
↪ms_active_directory.core.ad_objects.ADUser]) -> bool
    Enable a user account.
    :param account: The string name of the user/computer account to enable. This may␣
↪either be a
                    sAMAccountName, a distinguished name, or a unique common name. This␣
↪can also be an ADObject,
                    and the distinguished name will be extracted from it.
    :returns: True if the operation succeeds. If the operation fails, either an␣
↪exception will be raised or False
              will be returned depending on whether the ldap connection for this session␣
↪has "raise_exceptions"
              set to True or not.
```

(continues on next page)

(continued from previous page)

```
reset_password_for_account(self, account: Union[str, ms_active_directory.core.ad_objects.
↪ADUser, ms_active_directory.core.ad_objects.ADComputer], new_password: str, skip_
↪validation: bool = False) -> bool
    Resets a password for a user (includes computers) to a new desired password.
    To reset a password, a new password is provided to replace the current one without␣
↪providing the current
    password. This is a privileged operation and maps to the RESET_PASSWORD permission␣
↪in AD.

    :param account: The account whose password is being changed. This may either be a␣
↪string account name, to be
                    looked up, or an ADObject object.
    :param new_password: The new password for the account.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds. If the operation fails, either an␣
↪exception will be raised or False
              will be returned depending on whether the ldap connection for this session␣
↪has "raise_exceptions"
              set to True or not.

unlock_account(self, account: Union[str, ms_active_directory.core.ad_objects.ADComputer,␣
↪ms_active_directory.core.ad_objects.ADUser], skip_validation: bool = False) -> bool
    Unlock a user who's been locked out for some period of time.
    :param account: The string name of the user/computer account that has been locked␣
↪out. This may either be a
                    sAMAccountName, a distinguished name, or a unique common name. This␣
↪can also be an ADObject,
                    and the distinguished name will be extracted from it.
    :param skip_validation: If true, assume all distinguished names exist and do not␣
↪look them up.
                            Defaults to False. This can be used to make this function␣
↪more performant when
                            the caller knows all the distinguished names being specified␣
↪are valid, as it
                            performs far fewer queries.
    :returns: True if the operation succeeds. If the operation fails, either an␣
↪exception will be raised or False
              will be returned depending on whether the ldap connection for this session␣
↪has "raise_exceptions"
              set to True or not.
```

Working With Trusted Domains

There exist functions for finding trusted domains as well as transferring authentication sessions to them:

```
create_transfer_sessions_to_all_trusted_domains(self, ignore_and_remove_failed_
↪transfers=False) -> List[ForwardRef('ADSession')]
    Create transfer sessions to all of the different active directory domains that trust␣
↪the domain used for
    this session.

    :param ignore_and_remove_failed_transfers: If true, failures to transfer the session␣
↪to a trusted domain will
                                                be ignored, and will be excluded from␣
↪results. If false, errors will
                                                be raised by failed transfers. Defaults␣
↪to false.
    :returns: A list of ADSession objects representing the transferred authentication to␣
↪the trusted domains.
    :raises: Other LDAP exceptions if the attempt to bind the transfer session in the␣
↪trusted domain fails due to
             authentication issues (e.g. trying to use a non-transitive trust when␣
↪transferring a user that is
             not from the primary domain, transferring across a one-way trust when␣
↪skipping validation,
             transferring to a domain using SID filtering to restrict cross-domain users)

find_trusted_domains_for_domain(self, force_cache_refresh=False) -> List[ForwardRef(
↪'ADTrustedDomain')]
    Find the trusted domains for this domain.
    If we have cached trusted domains for this session's domain, and the cache is still␣
↪valid based on our
    cache lifetime, return that.

    :param force_cache_refresh: If true, don't use our cached trusted domains even if␣
↪the cache is valid.
                                 Defaults to false.
    :returns: A list of ADTrustedDomain objects
```

Other Utility Functions

There are other miscellaneous functions for various utility actions, like checking the name of the user/computer the current session has been established for, checking if distinguished names exist, checking the URI of the current server a session is communicating with, etc.

```
dn_exists_in_domain(self, distinguished_name: str) -> bool
    Check if a distinguished name exists within the domain, regardless of what it is.
    :param distinguished_name: Either a relative distinguished name or full␣
↪distinguished name
                                to search for within the domain.
    :returns: True if the distinguished name exists within the domain.

get_current_server_uri(self) -> str
```

(continues on next page)

(continued from previous page)

```
    Returns the URI of the server that this session is currently communicating with

get_domain(self) -> 'ADDomain'
    Returns the domain that this session is connected to

get_domain_dns_name(self) -> str
    Returns the domain that this session is connected to

get_domain_search_base(self) -> str
    Returns the LDAP search base used for all 'find' functions as the search base

get_ldap_connection(self) -> ldap3.core.connection.Connection
    Returns the LDAP connection that this session uses for communication.
    This is particularly useful if a user wants to make complex LDAP queries or perform
    operations that are not supported by the ADSession object, and is willing to craft
    them and parse results themselves.

get_search_paging_size(self) -> int

get_trusted_domain_cache_lifetime_seconds(self) -> int

is_authenticated(self) -> bool
    Returns if the session is currently authenticated

is_encrypted(self) -> bool
    Returns if the session's connection is encrypted

is_open(self) -> bool
    Returns if the session's connection is currently open

is_session_user_from_domain(self) -> bool
    Return a boolean indicating whether or not the session's user is a member of the␣
↪domain that we're
    communicating with, or is trusted from another domain.
    :returns: True if the user is from the domain we're communicating with, False␣
↪otherwise.

is_thread_safe(self) -> bool
    Returns if the session's connection is thread-safe

object_exists_in_domain_with_attribute(self, attr: str, unescaped_value: str) -> bool
    Check if any objects exist in the domain with a given attribute. Returns True if so,␣
↪False otherwise.
    :param attr: The LDAP attribute to examine in the search.
    :param unescaped_value: The value of the attribute that we're looking for, in its␣
↪raw form.
    :returns: True if any objects exist in the domain with the attribute specified equal␣
↪to the value.

who_am_i(self) -> str
    Return the authorization identity of the session's user as recognized by the server.
    This can be helpful when a script is provided with an identity in one form that is␣
↪used to start a session
```

(continues on next page)

(continued from previous page)

```
    (e.g. a distinguished name, or a pre-populated kerberos cache) and then it wants to␣
↪determine its identity
    that the server actually sees.
    This just calls the LDAP connection function, as it's suitable for AD as well.
    :returns: A string indicating the authorization identity of the session's user as␣
↪recognized by the server.
```

Help on class ManagedADComputer in module ms_active_directory.core.managed_ad_objects:

class ManagedADComputer(ManagedADObject)

ManagedADComputer(samaccount_name: str, domain: 'ADDomain', location: str = None, password: str = None, service_principal_names: List[str] = None, encryption_types: List[ms_active_directory.environment.security.security_config_constants.ADEncryptionType] = None, kvno: int = None)

Method resolution order:

ManagedADComputer
ManagedADObject
builtins.object

Methods defined here:

__init__(self, samaccount_name: str, domain: 'ADDomain', location: str = None, password: str = None, service_principal_names: List[str] = None, encryption_types: List[ms_active_directory.environment.security.security_config_constants.ADEncryptionType] = None, kvno: int = None)

Initialize self. See help(type(self)) for accurate signature.

add_encryption_type_locally(self, encryption_type: ms_active_directory.environment.security.security_config_constants.ADEncryptionType)

Adds an encryption type to the computer locally. This will generate new kerberos keys
for the computer as a user and for all of the computer's service principal names using the
new encryption type.
This function does nothing if the encryption type is already on the computer.
This function raises an exception if the computer's password is not set, as the password is
needed to generate new kerberos keys.
:param encryption_type: The encryption type to add to the computer.

add_service_principal_name_locally(self, service_principal_name: str)

Adds a service principal name to the computer locally. This will generate new kerberos keys
for the computer to use to accept security contexts for the service principal name using
all raw kerberos keys that the account has (and therefore all encryption types it has).
This function does nothing if the service principal name is already on the computer.

:param service_principal_name: The service principal name to add to the computer.

get_computer_distinguished_name(self) -> str

Get the LDAP distinguished name for the computer. This raises an exception if location is not set for the computer.

get_computer_name(self) -> str

get_encryption_types(self) -> List[ms_active_directory.environment.security.security_config_constants.ADEncryptionType]

get_full_keytab_file_bytes_for_computer(self) -> bytes

Get the raw bytes that would comprise a complete keytab file for this computer. The resultant bytes form a file that can be used to either accept GSS security contexts as a server for any protocol and hostname combinations defined in the service principal names, or initiate them as the computer with the computer's user principal name being the sAMAccountName.

get_name(self) -> str

get_server_kerberos_keys(self) -> List[ms_active_directory.core.ad_kerberos_keys.GssKerberosKey]

get_server_keytab_file_bytes_for_computer(self) -> bytes

Get the raw bytes that would comprise a server keytab file for this computer. The resultant bytes form a file that can be used to accept GSS security contexts as a server for any protocol and hostname combinations defined in the service principal names.

get_service_principal_names(self) -> List[str]

get_user_kerberos_keys(self) -> List[ms_active_directory.core.ad_kerberos_keys.GssKerberosKey]

get_user_keytab_file_bytes_for_computer(self) -> bytes

Get the raw bytes that would comprise a server keytab file for this computer. The resultant bytes form a file that can be used to initiate GSS security contexts as the computer with the computer's user principal name being the sAMAccountName.

get_user_principal_name(self) -> str

Gets the user principal name for the computer, to be used in initiating GSS security contexts

set_encryption_types_locally(self, encryption_types: List[ms_active_directory.environment.security.security_config_constants.ADEncryptionType])

Sets the encryption types of the computer locally. This will generate new kerberos keys for the computer as a user and for all of the computer's service principal names using the new encryption type.

This function raises an exception if the computer's password is not set, as the password is needed to generate new kerberos keys.

:param encryption_types: The list of AD encryption types to set on the computer.

set_password_locally(self, password: str)

Sets the password on the AD computer locally. This will regenerate server and user kerberos keys for all of the encryption types on the computer.

This function is meant to be used when the password was not set locally or was incorrectly set. This function WILL NOT update the key version number of the kerberos keys; if a computer's password is actually changed, then update_password_locally should be used as that will update the key version number properly and ensure the resultant kerberos keys can be properly used for initiating and accepting security contexts.
:param password: The string password to set for the computer.

set_service_principal_names_locally(self, service_principal_names: List[str])
Sets the service principal names for the computer, and regenerates new server kerberos keys for all of the newly set service principal names.
:param service_principal_names: A list of string service principal names to set for the computer.

update_password_locally(self, password: str)
Update the password for the computer locally and generate new kerberos keys for the new password.
:param password: The string password to set for the computer.

write_full_keytab_file_for_computer(self, file_path: str, merge_with_existing_file: bool = True)
Write all of the keytabs for this computer to a file, regardless of whether they represent keys for the computer to authenticate with other servers as a client, or keys to authenticate clients when acting as a server.

:param file_path: The path to the file where the keytabs will be written. If it does not exist, it will be created.
:param merge_with_existing_file: If True, the computers keytabs will be added into the keytab file at *file_path* if one exists. If False, the file at *file_path* will be overwritten if it exists. If the file does not exist, this does nothing. Defaults to True.

write_server_keytab_file_for_computer(self, file_path: str, merge_with_existing_file: bool = True)
Write all of the server keytabs for this computer to a file, which are the keys used to authenticate clients when acting as a server.

:param file_path: The path to the file where the keytabs will be written. If it does not exist, it will be created.
:param merge_with_existing_file: If True, the computers keytabs will be added into the keytab file at *file_path* if one exists. If False, the file at *file_path* will be overwritten if it exists. If the file does not exist, this does nothing. Defaults to True.

write_user_keytab_file_for_computer(self, file_path: str, merge_with_existing_file: bool = True)
Write all of the user keytabs for this computer to a file, which are the keys used to authenticate with other servers when acting as a client.

:param file_path: The path to the file where the keytabs will be written. If it does not exist, it will be created.
:param merge_with_existing_file: If True, the computers keytabs will be added into the keytab file at

file_path if one exists. If False, the file at *file_path* will be overwritten if it exists. If the file does not exist, this does nothing. Defaults to True.

—————————————————————————-

Methods inherited from ManagedADObject:

get_domain(self) -> 'ADDomain'

get_domain_dns_name(self) -> str

get_samaccount_name(self) -> str

—————————————————————————-

Data descriptors inherited from ManagedADObject:

__dict__
    dictionary for instance variables (if defined)

__weakref__
    list of weak references to the object (if defined)

Secondary Objects in ms_active_directory

The following are objects that you may interact with in order to construct other objects, or which may be returned by function calls. These often contain information describing an entity within a domain or describing an aspect of an entity.

GssKerberosKey Objects

```
class GssKerberosKey(builtins.object)
    GssKerberosKey(principal: str, realm: str, raw_key: ms_active_directory.core.ad_
↪kerberos_keys.RawKerberosKey, kvno: int, flags: int = None, timestamp: int = None, gss_
↪name_type: int = 0, format_version: int = 2)

    A kerberos key that can actually be used in kerberos negotiation (as either a user␣
↪or a server).
    This is a raw key properly wrapped with encoded additional information about the␣
↪principal, kvno,
    encryption type, etc.

    Methods defined here:

    __init__(self, principal: str, realm: str, raw_key: ms_active_directory.core.ad_
↪kerberos_keys.RawKerberosKey, kvno: int, flags: int = None, timestamp: int = None, gss_
↪name_type: int = 0, format_version: int = 2)
        Initialize self.  See help(type(self)) for accurate signature.
```

(continues on next page)

(continued from previous page)

```
    get_complete_keytab_bytes(self, format_version: int = None, use_current_time: bool =␣
↪None)
        Get this key object encoded as the bytes of a complete, usable keytab that can␣
↪be written
        to a file and used for kerberos authentication (initiating or accepting␣
↪contexts).
        :param format_version: An keytab format version. If not specified, defaults to␣
↪the format version
                              in the object. If the object's format version is null,␣
↪defaults to 2.
        :param use_current_time: Whether or not the current time should be used as the␣
↪timestamp in the
                                keytab produced, overwriting the time in the object. If␣
↪no timestamp is
                                in the object, the current time is used. Defaults to␣
↪False if not specified.

    get_raw_key_bytes(self)

    set_flags(self, flags: int)
        Sets the flags and clears complete_gss_keytab_bytes so we re-compute it

    set_format_version(self, format_version: int)
        Sets the keytab format version and clears complete_gss_keytab_bytes so we re-
↪compute it

    set_gss_name_type(self, name_type: int)
        Sets the gss name type and friendly name type and clears complete_gss_keytab_
↪bytes so we re-compute it

    set_kvno(self, kvno: int)
        Sets the kvno and clears complete_gss_keytab_bytes so we re-compute it

    set_principal(self, principal: str)
        Sets the principal and clears complete_gss_keytab_bytes so we re-compute it

    set_raw_key(self, raw_key: ms_active_directory.core.ad_kerberos_keys.RawKerberosKey)
        Sets the raw key, updates our encryption type and clears complete_gss_keytab_
↪bytes so we re-compute it.
        The encryption type is directly tied to our raw key and vice versa, so setting␣
↪one without the other makes no
        sense.

    set_realm(self, realm: str)
        Sets the realm and clears complete_gss_keytab_bytes so we re-compute it

    set_timestamp(self, timestamp: int)
        Sets the timestamp and clears complete_gss_keytab_bytes so we re-compute it

    uses_active_directory_supported_encryption_type(self)
```

RawKerberosKey Objects

```
class RawKerberosKey(builtins.object)
    RawKerberosKey(enc_type: Union[ms_active_directory.environment.security.security_
↪config_constants.ADEncryptionType, str], key_bytes: bytes)

    A raw kerberos key - containing only the generated shared secret and the encryption␣
↪type.
    This does not contain any information about who's using it, its purpose, etc. and is␣
↪tied
    only to the password used, the salt, and the encryption type. It can therefore be␣
↪used to
    generate usable kerberos keys for either accepting or initiating GSS authentication.

    Methods defined here:

    __init__(self, enc_type: Union[ms_active_directory.environment.security.security_
↪config_constants.ADEncryptionType, str], key_bytes: bytes)
        Initialize self.  See help(type(self)) for accurate signature.

    get_hex_encoded_key(self)

    get_key_bytes(self)

    get_raw_hex_encoded_key(self)

    uses_active_directory_supported_encryption_type(self)
```

Examples of Using Key Features

The following are the key features of the library that have examples.

Discovering a Domain

The library supports discovering LDAP and Kerberos servers within a domain using special DNS entries defined for Active Directory.

Smart Defaults

By default, it will use the system DNS configuration, find LDAP servers that support TLS, and sort LDAP and Kerberos servers by the RTT to communicate with them.

Here's an example of creating a simple configuration and working with server discovery.

```
from ms_active_directory import ADDomain

example_domain_dns_name = 'example.com'
domain = ADDomain(example_domain_dns_name)
ldap_servers = domain.get_ldap_uris()
kerberos_servers = domain.get_kerberos_uris()
```

(continues on next page)

(continued from previous page)

```
# re-discover servers in dns and sort them by RTT again at a later time to pick up␣
↪changes
domain.refresh_ldap_server_discovery()
domain.refresh_kerberos_server_discovery()
```

Site Awareness and Flexible DNS

The library also supports site awareness, which will result in only discovering servers within a specified Active Directory Site. You can also specify alternative DNS nameservers to use instead of the system ones.

Here's an example of specifying an AD site and alternative DNS server.

```
from ms_active_directory import ADDomain

example_domain_dns_name = 'example.com'
site_name = 'us-eastern-datacenter'
domain = ADDomain(example_domain_dns_name, site=site_name,
                  dns_nameservers=['eastern-private-dns-01.local'])
```

Network Multi-Tenancy and Security Support

You can also specify exactly which LDAP or Kerberos servers should be used, and skip discovery. Additional configurations are available such as configuring the CA file path to use for trust, and the source IP to use for outbound traffic to the domain, which is helpful when there are firewall rules in place, or when a machine has both private and public IP addresses.

Here's an example of specifying which servers to communicate with, and CA certs to secure that communication.

```
from ms_active_directory import ADDomain

example_domain_dns_name = 'example.com'
local_machine_ip = '10.251.12.1'
local_ldap_ip = '10.251.12.30'
public_machine_ip = '194.32.21.30'
# the servers that live on the public internet use well-known public
# CAs for trust, but we have a local CA for the private network servers
private_securing_cas = '/etc/internal-ca.cert'

# set up an object for the local domain in the same network as this machine,
# but also have an instance that can be used to make instances to reach out
# to the rest of the domain outside of the local private network
local_domain = ADDomain(example_domain_dns_name, ldap_servers_or_uris=[local_ldap_ip],
                        source_ip=local_ldap_ip, ca_certificates_file_path=private_
↪securing_cas)
global_domain = ADDomain(example_domain_dns_name, source_ip=public_machine_ip)
```

Local System Configuration

By default, you'll need to configure your local system files to enable kerberos authentication to work properly. However, you can also automatically set up the krb5 configuration when creating a domain object.

```
from ms_active_directory import ADDomain

example_domain_dns_name = 'example.com'
# set up the local system krb5 config based on discovered kerberos uris
domain = ADDomain(example_domain_dns_name,
                  auto_configure_kerberos_client=True)
```

The file configured will be `/etc/krb5.conf` on posix systems (e.g. macOS, Ubuntu), and on windows both `/winnt/krb5.ini` and `/windows/krb5.ini` will be configured for backwards compatibility. By default, a new kerberos realm configuration will be merged into these files if they exist, or new files will be created if none exists.

If you want to update a different configuration file, or if you want to overwrite the file instead of updating it, or if you want to set things like a default realm, you can also directly call the function for configuring the local system.

```
from ms_active_directory.environment.kerberos.kerberos_client_configurer import update_
↪system_kerberos_configuration_for_domains
from ms_active_directory import ADDomain

example_domain_dns_name = 'example.com'
domain = ADDomain(example_domain_dns_name)

# overwrite the existing file instead of updating it
update_system_kerberos_configuration_for_domains([domain], merge_with_existing_
↪file=False)
# update a file in a different location
update_system_kerberos_configuration_for_domains([domain], krb5_location='/etc/user_100/
↪krb5.conf')
# set a default authentication realm
update_system_kerberos_configuration_for_domains([domain], default_domain=domain)
```

Note: if multiple `ADDomain` objects all attempt to configure the local system kerberos file, only one will "win". This means that if they have different sites specified, or used different source addresses on a network where kdc reachability is reliant on that source address, having a single `ADDomain` object automatically configure the krb5 configuration file can be risky.

In these scenarios, it's recommended that you manually write the krb5 configuration or that you set up an `ADDomain` object with kerberos uris for the entire domain and use that to initiate the auto-configuration.

Discovering Additional Domain Resources

The library supports discovering a wide variety of information about the domain beyond the basics needed to communicate with it. This discovery doesn't require you to know any niche information about Active Directory.

Discoverable resources include but are not limited to:

1. Supported SASL mechanisms, which is important for authentication
2. The current domain time, which is important for NTP synchronization
3. Domain Functional Level, which governs things like support encryption types
4. DNS servers

5. Issuing certificates for CAs in the domain

Finding supported SASL mechanisms

Discovering SASL mechanisms can be done without needing to create a session with a domain, as it's needed before authentication in many cases.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')

# might print "['EXTERNAL', 'DIGEST-MD5']"
print(domain.find_supported_sasl_mechanisms())
```

Finding the current domain time

Discovering the domain time can be done without needing to create a session with a domain, as time synchronization is necessary for kerberos authentication to succeed and can impact TLS negotiation as well.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')

# returns a python datetime object in utc time
curr_time = domain.find_current_time()

# allowed drift defaults to 5 minutes which is the kerberos standard,
# but we can use a shorter window to detect drift before it causes an
# outage. this returns a boolean
synced = domain.is_close_in_time_to_localhost(allowed_drift_seconds=60)
```

Finding the domain functional level

Discovering the domain time can be done without needing to create a session with a domain, as it can inform us as to what encryption types and TLS versions/ciphers will be supported by the domain.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')

# find_functional_level returns an enum indicating the level.
# decision making based on level should be done based on the
# needs of your application
print(domain.find_functional_level())
```

Finding DNS servers

Discovering DNS servers requires an authenticated session with the domain, as searching the records within the domain for computers that run a DNS service is privileged.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')

session = domain.create_session_as_user('username@example.com', 'password')
# returns a map that maps server hostnames -> ip addresses, where
# the hostnames are computers running dns services
dns_map = session.find_dns_servers_for_domain()
ip_addrs = dns_map.values()
hostnames = dns_map.keys()
```

Finding CA certificates

Discovering DNS servers requires an authenticated session with the domain, as searching the records within the domain for records that are indicated as being certificate authorities is privileged.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')

session = domain.create_session_as_user('username@example.com', 'password')
# returns a list of PEM-formatted strings representing the signing certificates
# of all certificate authorities in the domain
pem_certs = session.find_certificate_authorities_for_domain()

# you can also get the certificates in DER format, which might be
# preferred on windows
der_certs = session.find_certificate_authorities_for_domain(pem_format=False)
```

Creating a Session With a Domain

You can establish a session with the AD Domain on behalf of either a user or computer.

Broadly, any keyword arguments that would normally be supported when creating a `Connection` with the `ldap3` library are supported when creating a session, allowing for flexibility while still providing an "it just works" option for most users.

Support for Computer Authentication

Computers default to using Kerberos SASL authentication, as SIMPLE authentication is not support for computers with Active Directory. To use kerberos, either `gssapi` or `winkerberos` must be installed.

Here's an example of authenticating as a computer

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')

```

(continues on next page)

(continued from previous page)

```
# when using kerberos auth, the default is to use the kerberos
# credential cache on the machine, so no password is needed
computer_name = 'machine01'
session1 = domain.create_session_as_computer(computer_name)

# but you can pass sasl credentials, and if you use gssapi you can
# specify a username and password
# see the ldap3 documentation for details on SASL credentials and other
# connection options
other_name = 'other-machine-identity'
password = 'password01'
session2 = domain.create_session_as_computer(other_name, sasl_credentials=('', other_
↪name, password))
```

You can also use other authentication mechanisms like NTLM.:

```
from ldap3 import NTLM
from ms_active_directory import ADDomain
domain = ADDomain('example.com')

ntlm_name = 'EXAMPLE.COM\\computer01'
password = 'password1'
session = domain.create_session_as_computer(ntlm_name, password, authentication_
↪mechanism=NTLM)
```

Support for User Authentication

You can authenticate as a user by using simple binds, or by using SASL mechanisms or NTLM as computers do. The default for users is simple binds.

Here's an example of using some different authentication mechanisms for the same user:

```
from ldap3 import NTLM
from ms_active_directory import ADDomain
domain = ADDomain('example.com')

session = domain.create_session_as_user('username@example.com', 'password')
ntlm_session =  domain.create_session_as_user('username@example.com', 'password',␣
↪authentication_mechanism=NTLM)
```

Joining an Active Directory Domain

The action of joining a computer to a domain is not a well-defined operation, and so the exact mechanics of how you utilize the domain joining functionality and how its outputs are integrated with the rest of your system will vary depending on your use case.

This will try to cover some common examples.

Join the domain with default configurations for everything

The default behavior requires only the domain name and the credentials of a user with sufficient administrative rights to create computers within the domain.

```
from ms_active_directory import join_ad_domain

comp = join_ad_domain('example.com', 'Administrator@example.com', 'example-password')
```

The `join_ad_domain` function returns a `ManagedADComputer` object with many helpful functions describing properties of the created computer.

This will use the local hostname of the machine running this code as the computer name. It will create the computer in AD's default `Computers` container.

It enables `AES256-SHA1` as an encryption type for both receiving and initiating kerberos contexts, and it configures `<local hostname>.<domain dns name>` as the hostname of the computer in AD and registers the default `HOST` service.

It then writes kerberos keys for the new computer account to `/etc/krb5.keytab`, which is the default location for kerberos keytabs.

This all enables the account to be used for authenticating with other domain resources as a client over protocols like SMB and LDAP using kerberos, as well as receiving incoming kerberos authentication as a server for things like SSH. This is because the `HOST` service encapsulates many standard services in the domain.

However, it is still up to the caller to do things like configure sshd to utilize the keytab.

Join the domain with customization of the account for security reasons

A number of customizations exist for security reasons.

You can change things like the encryption types enabled on the account to support older clients. You can also change location where the account is created when joining a domain in order to use a less privileged user for the act of joining. Locations can be LDAP distinguished names or windows path style canonical names.

You can also set the computer name if you have a desired naming scheme. This will impact the hostnames configured in the domain for the computer.

```
from ms_active_directory import join_ad_domain, ADEncryptionType

domain = 'example.com'
less_privileged_user = 'ops-manager@example.com'
password = 'password2'
# ldap-style relative distinguished name of a location
less_privileged_loc = 'OU=service-machines,OU=ops'
computer_name = 'workstation10'

legacy_enc_type = ADEncryptionType.RC4_HMAC
new_enc_type = ADEncryptionType.AES256_CTS_HMAC_SHA1_96

comp = join_ad_domain(domain, less_privileged_user, password, computer_name=computer_
↪name,
                      computer_location=less_privileged_loc, computer_encryption_
↪types=[legacy_enc_type, new_enc_type])
```

(continues on next page)

(continued from previous page)

```
alt_format_loc = '/ops/service-machines'
comp = join_ad_domain(domain, less_privileged_user, password, computer_name=computer_
↪name,
                      computer_location=alt_format_loc, computer_encryption_
↪types=[legacy_enc_type, new_enc_type])
```

You can also manually set the computer password. The default is to generate a random 120 character password, but if you want to share this computer across services, and some cannot interact with the generated kerberos keys, then you may wish to set a password manually.

You can also change where the kerberos keys are written to.

```
from ms_active_directory import join_ad_domain

domain = 'example.com'
user = 'ops-manager@example.com'
password = 'password2'
kerberos_key_location = '/usr/shared/keys/workstation-key.keytab'
computer_name = 'workstation10'
computer_password = 'workstation-shared-pw'

comp = join_ad_domain(domain, user, password, computer_key_file_path=kerberos_key_
↪location,
                      computer_name=computer_name, computer_password=computer_password)
```

Join the domain with different network or service settings

You can configure different hostnames for your computer when joining the domain. This is useful when you have multiple different hostnames for a single device, or want to use a computer name that doesn't match your network name.

You can also configure services in order to restrict or broaden what is supported by the computer when acting as a server (e.g. you can add *nfs* if the machine will be an nfs server).

Joining will fail if another computer in the domain is using the services you specify on any of the hostnames you specify in order to avoid conflicts that cause undefined behavior.

```
from ms_active_directory import join_ad_domain

domain = 'example.com'
user = 'ops-manager@example.com'
password = 'password2'

services = ['HOST', 'nfs', 'cifs', 'HTTP']
computer_name = 'workstation10'
computer_host1 = 'central-mount-point.example.com'
computer_host2 = 'example-web-server.example.com'
comp = join_ad_domain(domain, user, password, computer_name=computer_name,
                      computer_hostnames=[computer_host1, computer_host2],
                      computer_services=services)
```

Join using a domain object

You can use an ADDomain object to join the domain as well, using a join function. This allows you to combine all of the functionality mentioned earlier around site-awareness, server preferences, TLS settings, and network multi-tenancy with the domain joining functionality mentioned in this section.

The parameters are all the same, except the domain need not be provided when using an ADDomain object, so it just adds more functionality in exchange for a slightly less simple workflow.

```
from ms_active_directory import ADDomain

domain = ADDomain('example.com', site='us-eastern-dc',
                  source_ip='10.25.21.30', dns_nameservers=['10.25.21.20'])

user = 'ops-manager@example.com'
password = 'password2'
less_privileged_loc = 'OU=service-machines,OU=ops'
services = ['HOST', 'nfs', 'cifs', 'HTTP']
computer_name = 'workstation10'

comp = domain.join(user, password, computer_hostnames=[computer_host1, computer_host2],
                   computer_services=services, computer_location=less_privileged_loc)
```

Join the domain by taking over an existing account

For some setups, accounts may be pre-created and then taken over by the computers that will use them.

This can be done in order to greatly restrict the permissions of the user that is used for joining, as they only need RESET PASSWORD permissions on the computer account, or CHANGE PASSWORD if the current computer password is provided.

```
from ms_active_directory import ADDomain, join_ad_domain_by_taking_over_existing_computer

domain_dns_name = 'example.com'
site = 'us-eastern-dc'
existing_computer_name = 'precreated-comp'
user = 'single-account-admin@example.com'
password = 'password2'

computer_obj = join_ad_domain_by_taking_over_existing_computer(domain_dns_name, user,␣
↪password,
                                                               ad_site=site, computer_
↪name=existing_computer_name)

# or use a domain object to use various power-user domain features
domain = ADDomain(domain_dns_name, site=site,
                  source_ip='10.25.21.30', dns_nameservers=['10.25.21.20'])
domain.join_by_taking_over_existing_computer(user, password, computer_name=existing_
↪computer_name)
```

Finding and Working With Trusted Domains

You can discover trusted domains using a session, and check properties about them.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

trusted_domains = session.find_trusted_domains_for_domain()

# split domains up based on trust type
trusted_mit_domains = [dom for dom in trusted_domains if dom.is_mit_trust()]
trusted_ad_domains = [dom for dom in trusted_domains if dom.is_active_directory_domain_
↪trust()]

# print a few attributes that may be relevant
for ad_dom in trusted_ad_domains:
    print('FQDN: {}'.format(ad_dom.get_netbios_name()))
    print('Netbios name: {}'.format(ad_dom.get_netbios_name()))
    print('Disabled: {}'.format(ad_dom.is_disabled())
    print('Bi-directional: {}'.format(ad_dom.is_bidirectional_trust())
    print('Transitive: {}'.format(ad_dom.is_transitive_trust())
```

Turning Trusted Domains into ADDomains

You can also convert AD domains that are trusted into fully usable ADDomain objects for the purpose of creating sessions and looking up information there.

```
from ms_active_directory import ADDomain
from ldap3 import NTLM
domain = ADDomain('example.com')
widely_trusted_user = 'example.com\\org-admin'
password = 'password'

primary_session = domain.create_session_as_user(widely_trusted_user, password,
                                                authentication_mechanism=NTLM)

# get our trusted AD domains
trusted_domains = session.find_trusted_domains_for_domain()
trusted_ad_domains = [dom for dom in trusted_domains if dom.is_active_directory_domain_
↪trust()]

# convert them into domains where our user should be trusted
domains_our_user_can_auth_with = []
for trusted_dom in trusted_ad_domains:
    if trusted_dom.trusts_primary_domain() and not trusted_dom.is_disabled():
        full_domain = trusted_dom.convert_to_ad_domain()
        domains_our_user_can_auth_with.append(full_domain)

# create sessions so we can search across many domains
all_user_sessions = [primary_session]
for dom in domains_our_user_can_auth_with:
```

(continues on next page)

(continued from previous page)

```
    # SASL is needed for cross-domain authentication in general
    session = dom.create_session_as_user(widely_trusted_user, password,
                                         authentication_mechanism=NTLM)
    all_user_sessions.append(session)
```

Transferring Sessions Across Domains

You can convert an existing authenticated session with one domain into an authenticated session with a trusted AD domain that trusts the first domain.

```
from ms_active_directory import ADDomain
from ldap3 import NTLM
domain = ADDomain('example.com')
widely_trusted_user = 'example.com\\org-admin'
password = 'password'

primary_session = domain.create_session_as_user(widely_trusted_user, password,
                                                authentication_mechanism=NTLM)

# get our trusted AD domains
trusted_domains = session.find_trusted_domains_for_domain()
# filter for a domain being AD and it trusting the primary domain
trusted_ad_domains = [dom for dom in trusted_domains if dom.is_active_directory_domain_
↪trust()
                      and dom.trusts_primary_domain()]

# create a new session with the trusted domain using our existing primary domain session,
# and use it to look up users/groups/etc. in the other domain
transferred_session = trusted_ad_domains[0].create_transfer_session_to_trusted_
↪domain(primary_session)
transferred_session.find_user_by_name('other-domain-user')
```

Expanding A Session To All Its Trusted Domains

You can also automatically have a session create sessions for all its trusted domains that trust the session's domain.

```
from ms_active_directory import ADDomain
from ldap3 import NTLM
domain = ADDomain('example.com')
widely_trusted_user = 'example.com\\org-admin'
password = 'password'

primary_session = domain.create_session_as_user(widely_trusted_user, password,
                                                authentication_mechanism=NTLM)

# find a user that we know exists somewhere, but not the primary domain
user_to_find = 'some-lost-user'
# by default this filters to AD domains, and further filters to domains that trust the␣
↪session's domain
```

(continues on next page)

(continued from previous page)

```
# if the user used for the session is from the session's domain (which they are in this
# example)
trust_sessions = primary_session.create_transfer_sessions_to_all_trusted_domains()
user = None
for session in trust_sessions:
    user = session.find_user_by_name(user_to_find)
    if user is not None:
        print('Found user in {}'.format(session.get_domain_dns_name()))
        break
```

Finding users, computers, and groups

The library provides a number of different functions for finding users, computers, and groups by different identifiers, and for querying information about them.

Looking up users, computers, groups, and information about them

Users, computers, and groups can both be looked up by one of:

- sAMAccountName
- distinguished name
- common name
- a generic “name” that will attempt the above 3
- an attribute

Look up by sAMAccountName

A `sAMAccountName` is unique within a domain, and so looking up users or groups by `sAMAccountName` returns a single result. `sAMAccountName` was a user’s windows logon name in older versions of windows, and may be referred to as such in some documentation.

For computers, the standard convention is for their `sAMAccountName` to end with a `$`, but many tools/docs leave that out. So if a `sAMAccountName` is specified that does not end with a `$` and cannot be found, a lookup will also be attempted after adding a `$` to the end.

When looking up users, computers, and groups, you can also query for additional information about them by specifying a list of LDAP attributes.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

user = session.find_user_by_sam_name('user1', ['employeeID'])
group = session.find_group_by_sam_name('group1', ['gidNumber'])
# users and groups support a generic "get" for any attributes queried
print(user.get('employeeID'))
print(group.get('gidNumber'))
```

Look up by distinguished name

A distinguished name is unique within a forest, and so looking up users or groups by it returns a single result. A distinguished name should not be escaped when provided to the search function.

When looking up users, computers, and groups, you can also query for additional information about them by specifying a list of LDAP attributes.

```python
from ms_active_directory import ADDomain
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

user_dn = 'CN=user one,CN=Users,DC=example,DC=com'
user = session.find_user_by_distinguished_name(user_dn, ['employeeID'])
group_dn = 'CN=group one,OU=employee-groups,DC=example,DC=com'
group = session.find_group_by_distinguished_name(group_dn, ['gidNumber'])
# users and groups support a generic "get" for any attributes queried
print(user.get('employeeID'))
print(group.get('gidNumber'))
```

Look up by common name

A common name is not unique within a domain, and so looking up users or groups by it returns a list of results, which may have 0 or more entries.

When looking up users, computers, and groups, you can also query for additional information about them by specifying a list of LDAP attributes.

```python
from ms_active_directory import ADDomain
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

user_cn = 'John Doe'
users = session.find_users_by_common_name(user_cn, ['employeeID'])
group_dn = 'operations managers'
groups = session.find_groups_by_common_name(group_dn, ['gidNumber'])
# users and groups support a generic "get" for any attributes queried
for user in users:
    print(user.get('employeeID'))
for group in groups:
    print(group.get('gidNumber'))
```

Look up by generic name

You can also query by a generic “name”, and the library will attempt to find a unique user or group with that name. The library will either lookup by DN or will attempt `sAMAccountName` and common name lookups depending on the name format.

If more than one result is found by common name and no result is found by `sAMAccountName` then this will produce an error.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

user_name = 'John Doe'
user = session.find_user_by_name(user_name, ['employeeID'])
group_name = 'operations managers'
groups = session.find_groups_by_name(group_name, ['gidNumber'])
# users and groups support a generic "get" for any attributes queried
print(user.get('employeeID'))
print(group.get('gidNumber'))
```

Look up by attribute

You can also query for users, computers, or groups that possess a certain value for a specified attribute. This can produce any number of results, so a list is returned.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

desired_employee_type = 'temporary'
users = session.find_users_by_attribute('employeeType', desired_employee_type, [
↪'employeeID'])
desired_group_manager = 'Alice P Hacker'
groups = session.find_groups_by_attribute('managedBy', desired_group_manager, ['gidNumber
↪'])

# users and groups support a generic "get" for any attributes queried
for user in users:
    print(user.distinguished_name)
    print(user.get('employeeID'))
for group in groups:
    print(group.distinguished_name)
    print(group.get('gidNumber'))
```

Updating user, computer, or group attributes.

You can use this library to modify the values of various LDAP attributes on users, computers, groups, or generic objects.

Users, computers, and groups provide the convenient name lookup functionality mentioned above, while for generic objects you either need to pass an `ADObject` or a distinguished name.

Appending to one or more attributes

You can atomically append values to multi-valued attributes, such as accountNameHistory. This allows you to update their values without needing to know the current value or worry about race conditions, as it's handled server-side.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

user_name = 'sarah1'
previous_account_name = 'sarah'
success = session.atomic_append_to_attribute_for_user(user_name, 'accountNameHistory',
                                                      previous_account_name)

# you can also append multiple values at once, or append to multiple
# attributes at once
user_name = 'monica pham-chen'
previous_account_names = ['monica pham', 'monica chen']
previous_uid = 'mpham'
update_map = {
    'accountNameHistory': previous_account_names,
    'uid': previous_uid
}
success = session.atomic_append_to_attributes_for_user(user_name, update_map)
```

You can also perform these actions on groups and objects using the similarly named functions atomic_append_to_attribute_for_group, atomic_append_to_attributes_for_group, atomic_append_to_attribute_for_computer, atomic_append_to_attributes_for_computer, atomic_append_to_attribute_for_object, and atomic_append_to_attributes_for_object.

Overwriting one or more attributes

If you want to totally replace the value of an attribute, that's supported as well. This can be done for single-valued or multi-valued attributes.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

user_name = 'arjun'
new_uid_number = 1093
success = session.overwrite_attribute_for_user(user_name, 'uidNumber',
                                               new_uid_number)

# just like appending, we can do multiple attributes at once atomically
user_name = 'nikita'
new_employee_type = 'Director'
new_gid = 0
new_addresses = [
   '123 mulberry lane',
   '456 vacation home drive'
]
```

(continues on next page)

(continued from previous page)

```
new_value_map = {
   'employeeType': new_employee_type,
   'gidNumber': new_gid,
   'postalAddress': new_addresses
}
success = session.overwrite_attributes_for_user(user_name, new_value_map)
```

You can also perform these actions on groups and objects using the similarly named functions, just like with appending.

Managing User, Computer, and Group Membership

You can look up the groups that a user belongs to, the groups that a computer belongs to, or the groups that a group belongs to. Active Directory supports nested groups, which is why there's `user->groups`, `computer->groups`, and `group->groups` mapping capability.

When querying the membership information for users or groups, the input type for any user or group must either be a string name identifying the user, computer, or group as described in the prior section, or must be an `ADUser`, `ADComputer`, or `ADGroup` object returned by one of the functions described in the prior section.

Similarly to looking up users, computers, and groups, you can query for attributes of the parent groups by providing a list of LDAP attributes to look up for them.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

user_sam_account_name = 'user-sam-1'
user_dn = 'CN=user sam 1,CN=users,DC=example,DC=com'
user_cn = 'user same 1'

desired_group_attrs = ['gidNumber', 'managedBy']
# all 3 of these do the same thing, and internally map the different
# name types to a user object
groups_res1 = session.find_groups_for_user(user_sam_account_name, desired_group_attrs)
groups_res2 = session.find_groups_for_user(user_dn, desired_group_attrs)
groups_res3 = session.find_groups_for_user(user_cn, desired_group_attrs)

# you can also directly use a user object to query groups
user_obj = session.find_user_by_name(user_sam_account_name)
groups_res4 = session.find_groups_for_user(user_obj, desired_group_attrs)

# you can also look up the parents of groups in the same way
example_group_obj = groups_res4[0]
example_group_dn = example_group_obj.distinguished_name

# these both work. sAMAccountName could also be used, etc.
second_level_groups_res1 = session.find_groups_for_group(example_group_obj, desired_
↪group_attrs)
second_level_groups_res2 = session.find_groups_for_group(example_group_dn, desired_group_
↪attrs)
```

You can also query `users->groups`, `computers->groups`, and `groups->groups` to find the memberships of multiple users, computers, and groups, and the library will make a minimal number of queries to determine membership;

it will be more efficient that doing a `user->groups` for each user (or similar for computers and groups). The result will be a map that maps the input users or groups to lists of parent groups.

The input lists' elements must be the same format as what's provided when looking up group memberships for a single user or group.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

user1_name = 'user1'
user2_name = 'user2'
users = [user1_name, user2_name]
desired_group_attrs = ['gidNumber', 'managedBy']

user_group_map = session.find_groups_for_users(users, desired_group_attrs)
# the dictionary result keys are the users from the input
user1_groups = user_group_map[user1_name]
user2_groups = user_group_map[user2_name]

# you can use the groups->groups mapping functionality to enumerate the
# full tree of a users' group memberships (or a groups' group memberships)
user1_second_level_groups_map = session.find_groups_for_groups(user1_groups, desired_
↪group_attrs)
all_second_level_groups = []
for group_list in user1_second_level_groups_map.values():
    for group in group_list:
        if group not in all_second_level_groups:
            all_second_level_groups.append(group)
all_user1_groups_in_2_levels = user1_groups + all_second_level_groups
```

Finding the members of groups

You can look up the members of one or more groups and get attributes about those members.

```
from ms_active_directory import ADDomain, ADUser, ADGroup
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

# get emails of users and groups that are members
desired_attrs = ['mail']

# look up members of a single group
single_group_member_list = session.find_members_of_group('group1', desired_attrs)

# look up members of multiple groups at once
groups = ['group1', 'group2']
group_to_member_list_map = session.find_members_of_groups(groups, desired_attrs)
group2_member_list = group_to_member_list_map['group2']
group2_user_members = [mem for mem in group2_member_list if isintance(mem, ADUser)]
group2_group_members = [mem for mem in group2_member_list if isintance(mem, ADGroup)]
```

You can also look up members recursively to handle nesting. A maximum depth for lookups may be specified, but by default all nesting will be enumerated.

```
from ms_active_directory import ADDomain, ADUser, ADGroup
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

# get emails of users and groups that are members
desired_attrs = ['mail']
group_name = 'has-groups-as-members'
groups_to_member_lists_maps = session.find_members_of_groups_recursive(group_name,␣
↪desired_attrs)
```

Adding users to groups

You can add users to groups by specifying a list of ADUser objects or string names of AD users to be added to the groups, and a list of ADGroup objects or string names of AD groups to add the users to.

If string names are specified, they'll be mapped to users/groups using the functions discussed in the prior sections.

If a user is already in a group, this is idempotent and will not re-add them.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

user1_name = 'user1'
user2_name = 'user2'
group1_name = 'target-group1'
group2_name = 'target-group2'

session.add_users_to_groups([user1_name, user2_name],
                            [group1_name, group2_name])
```

By default, if we fail to add users to one of the groups specified, we'll attempt to rollback and remove users from any groups they were added to. You can choose to forgo this and a list of groups that users were successfully added to will be returned instead.

```
from ms_active_directory import ADDomain
domain = ADDomain('example.com')
session = domain.create_session_as_user('username@example.com', 'password')

user1_name = 'user1'
user2_name = 'user2'
group1_name = 'target-group1'
group2_name = 'target-group2'
privileged_group = 'group-that-will-fail'

succeeeded = session.add_users_to_groups([user1_name, user2_name],
                                         [group1_name, group2_name, privileged_group],
                                         stop_and_rollback_on_error=False)
# this will print "['target-group1', 'target-group2']" assuming that
# adding users to 'group-that-will-fail' failed
print(succeeeded)
```

Adding groups to groups

Adding groups to other groups works exactly the same way as adding users to groups, but the function is called `add_groups_to_groups` and both inputs are lists of groups.

Adding computers to groups

Adding computers to groups works exactly the same way as adding users to groups, but the function is called `add_computers_to_groups` and the first input is a list of computers.

Removing users, computers, or groups from groups

Removing users, computers, or groups from groups works identically to adding users, computers, or groups to groups, including input format, idempotency, and rollback functionality. The only difference is that the functions are called `remove_users_from_groups`, `remove_computers_from_groups`, and `remove_groups_from_groups` instead.

Feel free to contribute more! See the main page for information on how to contribute.

Exceptions

The following exception types have been created for this library. If you wish to create a catch-all try/except then you can use **MsActiveDirectoryException**, as it's the parent exception for all others.

`class MsActiveDirectoryException(Exception):` “”” A parent class for all other exceptions so that users can have a catch-all exception for functional issues that still doesn't blind them to things like accidentally providing a string where a number is needed. “””

`class AttributeModificationException(MsActiveDirectoryException):` “”” An exception raised when an error is encountered modifying attributes of users, groups, etc. “””

`class DomainConnectException(MsActiveDirectoryException):` “”” An exception raised when an error is encountered connecting to an AD Domain “””

`class DomainJoinException(MsActiveDirectoryException):` “”” An exception raised when an error is encountered joining to an AD Domain or validating the join “””

`class DomainSearchException(MsActiveDirectoryException):` “”” An exception raised when an error is encountered searching an AD Domain “””

`class DuplicateNameException(MsActiveDirectoryException):` “”” An exception raised when multiple records are found during an operation that expects to operate on a unique object “””

`class InvalidComputerParameterException(MsActiveDirectoryException):` “”” An exception raised when functions are called on a ManagedADComputer object with invalid parameters or that rely on unpopulated attributes. “””

`class InvalidDomainParameterException(MsActiveDirectoryException):` “”” An exception raised when invalid parameters are used for creating a domain object or establishing a connection with a domain. “””

`class InvalidLdapParameterException(MsActiveDirectoryException):` “”” An exception raised when a parameter specified is not of a proper type or format to convert to an LDAP attribute as needed for a function. “””

`class KeytabEncodingException(MsActiveDirectoryException):` “”” An exception raised when a keytab is read in from a file but the encoding is invalid “””

class LdapResponseDecodeException(MsActiveDirectoryException): “”” An exception raised when an LDAP response cannot be parsed properly “””

class MembershipModificationException(MsActiveDirectoryException): “”” An exception raised when an error is encountered modifying group memberships, and rollback of the incomplete changes was successful. “””

class MembershipModificationRollbackException(MsActiveDirectoryException): “”” An exception raised when an error is encountered modifying group memberships, but rollback of the incomplete changes was unsuccessful. “””

class ObjectCreationException(MsActiveDirectoryException): “”” An exception raised when an error is encountered creating an object “””

class ObjectNotFoundException(MsActiveDirectoryException): “”” An exception raised when an object cannot be found when performing validation that an object exists as part of a function. “””

class PermissionDeniedException(MsActiveDirectoryException): “”” An exception raised when permission errors occur operating within AD “””

class SecurityDescriptorDecodeException(MsActiveDirectoryException): “”” An exception raised when errors occur decoding a security descriptor “””

class SecurityDescriptorEncodeException(MsActiveDirectoryException): “”” An exception raised when errors occur encoding a security descriptor “””

class SessionTransferException(MsActiveDirectoryException): “”” An exception raised when errors occur transferring an authentication session from one domain to another “””

class TrustedDomainConversionException(MsActiveDirectoryException): “”” An exception raised when trying to convert a trusted domain that has a non-AD type to an ADDomain “””

Generating Kerberos Keys From AD Passwords

```
ad_password_string_to_key(ad_encryption_type: ms_active_directory.environment.security.
↪security_config_constants.ADEncryptionType, ad_computer_name: str, ad_password: str,␣
↪ad_domain_dns_name: str, ad_auth_realm: str = None) -> ms_active_directory.core.ad_
↪kerberos_keys.RawKerberosKey
    Given an encryption type, a computer name, a password, and a domain, generate the␣
↪raw kerberos key for an AD
    account. Optionally, a realm may be specified if the kerberos realm for the domain␣
↪is not the domain itself
    (this may be the case for subdomains or when AD is not the central authentication␣
↪for an environment).
    :param ad_encryption_type: The kerberos encryption type to use for generating the␣
↪key.
    :param ad_computer_name: The name of the computer in AD. This is the sAMAccountName␣
↪without the trailing $.
    :param ad_password: The password of the computer.
    :param ad_domain_dns_name: The DNS name of the AD domain where the computer exists.
    :param ad_auth_realm: The realm used by the domain for authentication. If not␣
↪specified, defaults to the domain
                          in all captial letters.
```

Joining an AD Domain

To join the local machine to an AD Domain, you can use an `ADDomain` object and use its function, but there’s also a standalone function that can be imported from the library directly as:

```
>>> from ms_active_directory import join_ad_domain
```

This function can be used to have a 1-line call to join the machine to the domain by creating a new computer to represent it.

You can specify a lot of properties about the computer to be created, but by default it will be named after the local machine’s hostname (if it’s a valid AD name) and created in AD’s default Computers container. A strong password is set for the computer that is 120 characters long and random, strong encryption types are enabled, and Kerberos keys will be generated for the computer and written to the standard default system location (`/etc/krb5.keytab`).

A `ManagedADComputer` object is returned which has many helper functions for reading information about the created computer and managing its keys.

To join a domain and create a new computer, use the following function:

```
join_ad_domain(domain_dns_name: str, admin_username: str, admin_password: str,␣
↪authentication_mechanism: str = 'SIMPLE',
               ad_site: str = None, computer_name: str = None, computer_location: str =␣
↪None, computer_password: str = None,
               computer_encryption_types: List[Union[str, ms_active_directory.
↪environment.security.security_config_constants.ADEncryptionType]] = None,
               computer_hostnames: List[str] = None, computer_services: List[str] = None,
               supports_legacy_behavior: bool = False, computer_key_file_path: str = '/
↪etc/krb5.keytab',
               **additional_account_attributes) -> ms_active_directory.core.managed_ad_
↪objects.ManagedADComputer

    A super simple 'join a domain' function that requires minimal input - the domain dns␣
↪name and admin credentials
    to use in the join process.
    Given those basic inputs, the domain's nearest controllers are automatically␣
↪discovered and an account is made
    with strong security settings. The account's attributes follow AD naming conventions␣
↪based on the computer's
    hostname by default.
    :param domain_dns_name: The DNS name of the domain being joined.
    :param admin_username: The username of a user or computer with the rights to create␣
↪the computer.
                           This username should be formatted based on the authentication␣
↪protocol being used.
                           For example, DOMAIN\username for NTLM as opposed to␣
↪username@DOMAIN for GSSAPI, or
                           a distinguished name for SIMPLE.
                           If `old_computer_password` is specified, then this account␣
↪only needs permission to
                           change the password of the computer being taken over, which␣
↪is different from the reset
                           password permission.
    :param admin_password: The password for the user. Optional, as SASL authentication␣
↪mechanisms can use
```

(continues on next page)

(continued from previous page)

```
                           `sasl_credentials` specified as a keyword argument, and␣
↪things like KERBEROS will use
                           default system kerberos credentials if they're available.
    :param authentication_mechanism: An LDAP authentication mechanism or SASL mechanism.␣
↪If 'SASL' is specified,
                                     then the keyword argument `sasl_mechanism` must␣
↪also be specified. Valid values
                                     include all authentication mechanisms and SASL␣
↪mechanisms from the ldap3
                                     library, such as SIMPLE, NTLM, KERBEROS, etc.
    :param ad_site: Optional. The site within the active directory domain where our␣
↪communication should be confined.
    :param computer_name: The name of the computer to take over in the domain. This␣
↪should be the sAMAccountName
                          of the computer, though if computer has a trailing $ in its␣
↪sAMAccountName and that is
                          omitted, that's ok. If not specified, we will attempt to find␣
↪a computer with a name
                          matching the local system's hostname.
    :param computer_location: The location in which to create the computer. This may be␣
↪specified as an LDAP-style
                              relative distinguished name (e.g. OU=ServiceMachines,
↪OU=Machines) or a windows path
                              style canonical name (e.g. example.com/Machines/
↪ServiceMachines).
                              If not specified, defaults to CN=Computers which is the␣
↪standard default for AD.
    :param computer_password: The password to set for the computer when taking it over.␣
↪If not specified, a random
                              120 character password will be generated and set.
    :param computer_encryption_types: A list of encryption types, based on the␣
↪ADEncryptionType enum, to enable on
                                      the account created. These may be strings or enums;
↪ if they are strings,
                                      they should be strings of the encryption types as␣
↪written in kerberos
                                      RFCs or in AD management tools, and we will try to␣
↪map them to enums and
                                      raise an error if they don't match any supported␣
↪values.
                                      AES256-SHA1, AES128-SHA1, and RC4-HMAC encryption␣
↪types are supported. DES
                                      encryption types aren not.
                                      If not specified, defaults to [AES256-SHA1].
    :param computer_hostnames: Hostnames to set for the computer. These will be used to␣
↪set the dns hostname
                               attribute in AD. If not specified, the computer hostnames␣
↪will default to
                               [`computer_name`, `computer_name`.`domain`] which is the␣
↪AD standard default.
    :param computer_services: Services to enable on the computers hostnames. These␣
↪services dictate what clients
```

(continues on next page)

(continued from previous page)

```
                              can get kerberos tickets for when communicating with this␣
↪computer, and this property
                              is used with `computer_hostnames` to set the service␣
↪principal names for the computer.
                              For example, having `nfs` specified as a service principal␣
↪is necessary if you want
                              to run an NFS server on this computer and have clients get␣
↪kerberos tickets for
                              mounting shares; having `ssh` specified as a service␣
↪principal is necessary for
                              clients to request kerberos tickets for sshing to the␣
↪computer.
                              If not specified, defaults to `HOST` which is the standard␣
↪AD default service.
                              `HOST` covers a wide variety of services, including `cifs`,
↪ `ssh`, and many others
                              depending on your domain. Determining exactly what␣
↪services are covered by `HOST`
                              in your domain requires checking the aliases set on a␣
↪domain controller.
    :param supports_legacy_behavior: If `True`, then an error will be raised if the␣
↪computer name is longer than
                                     15 characters (not including the trailing $). This␣
↪is because various older
                                     systems such as NTLM, certain UNC path applications,
↪ Netbios, etc. cannot
                                     use names longer than 15 characters. This name␣
↪cannot be changed after
                                     creation, so this is important to control at␣
↪creation time.
                                     If not specified, defaults to `False`.
    :param computer_key_file_path: The path of where to write the keytab file for the␣
↪computer after taking it over.
                                   This will include keys for both user and server keys␣
↪for the computer.
                                   If not specified, defaults to /etc/krb5.keytab
    :param additional_account_attributes: Additional keyword argument can be specified␣
↪to set other LDAP attributes
                                          of the computer that are not covered above, or␣
↪where the above controls
                                          are not sufficiently granular. For example,␣
↪`userAccountControl` could
                                          be used to set the user account control values␣
↪for the computer if it's
                                          desired to set it differently from the default␣
↪(e.g. create a computer
                                          in a disabled state and enable it later).
    :returns: A ManagedADComputer object representing the computer created.
```

Joining an AD Domain by taking over an existing computer

To join the local machine to an AD Domain, you can use an `ADDomain` object and use its function, but there's also a standalone function that can be imported from the library directly as:

```
>>> from ms_active_directory import join_ad_domain_by_taking_over_existing_computer
```

This function can be used to have a 1-line call to join the machine to the domain by taking over a pre-created computer account. This is convenient for setups where the computer is pre-created with a lot of settings so that the machines joining don't need to know what attribute values to set.

Taking over an existing computer returns the a `ManagedADComputer` object, and writes kerberos keys to the local file system and such, but there's no option to specify things like services and dns hostnames as those are read from the existing computer.

To take over a computer in this way, use the following function:

```
join_ad_domain_by_taking_over_existing_computer(domain_dns_name: str, admin_username:␣
↪str, admin_password: str,
                                                authentication_mechanism: str = 'SIMPLE',
↪ ad_site: str = None,
                                                computer_name: str = None, computer_
↪password: str = None,
                                                old_computer_password: str = None,
                                                computer_key_file_path: str = '/etc/krb5.
↪keytab',
                                                **additional_connection_attributes) ->␣
↪ms_active_directory.core.managed_ad_objects.ManagedADComputer

    A super simple 'join a domain' function using pre-created computer accounts, which␣
↪requires minimal input -
    the domain dns name and admin credentials to use in the join process.
    Specifying a computer name explicitly for the account to take over is also highly␣
↪recommended.

    Given those basic inputs, the domain's nearest controllers are automatically␣
↪discovered and the computer account
    with the specified computer name is found and taken over so it can represent the␣
↪local system in the domain,
    and the local system can act as it.
    :param domain_dns_name: The DNS name of the domain being joined.
    :param admin_username: The username of a user or computer with the rights to reset␣
↪the password of the computer
                           being taken over.
                           This username should be formatted based on the authentication␣
↪protocol being used.
                           For example, DOMAIN\username for NTLM as opposed to␣
↪username@DOMAIN for GSSAPI, or
                           a distinguished name for SIMPLE.
                           If `old_computer_password` is specified, then this account␣
↪only needs permission to
                           change the password of the computer being taken over, which␣
↪is different from the reset
                           password permission.
    :param admin_password: The password for the user. Optional, as SASL authentication␣
↪mechanisms can use
```

(continues on next page)

(continued from previous page)

```
                           `sasl_credentials` specified as a keyword argument, and␣
↪things like KERBEROS will use
                           default system kerberos credentials if they're available.
    :param authentication_mechanism: An LDAP authentication mechanism or SASL mechanism.␣
↪If 'SASL' is specified,
                                     then the keyword argument `sasl_mechanism` must␣
↪also be specified. Valid values
                                     include all authentication mechanisms and SASL␣
↪mechanisms from the ldap3
                                     library, such as SIMPLE, NTLM, KERBEROS, etc.
    :param ad_site: Optional. The site within the active directory domain where our␣
↪communication should be confined.
    :param computer_name: The name of the computer to take over in the domain. This␣
↪should be the sAMAccountName
                          of the computer, though if computer has a trailing $ in its␣
↪sAMAccountName and that is
                          omitted, that's ok. If not specified, we will attempt to find␣
↪a computer with a name
                          matching the local system's hostname.
    :param computer_password: The password to set for the computer when taking it over.␣
↪If not specified, a random
                              120 character password will be generated and set.
    :param old_computer_password: The current password of the computer being taken over.␣
↪If specified, the action
                                  of taking over the computer will use a "change password
↪" operation, which is less
                                  privileged than a "reset password" operation. So␣
↪specifying this reduces the
                                  permissions needed by the user specified.
    :param computer_key_file_path: The path of where to write the keytab file for the␣
↪computer after taking it over.
                                   This will include keys for both user and server keys␣
↪for the computer.
                                   If not specified, defaults to /etc/krb5.keytab
    :param additional_connection_attributes: Additional keyword arguments may be␣
↪specified for any properties of
                                             the `Connection` object from the `ldap3`␣
↪library that is desired to
                                             be set on the connection used in the␣
↪session created for taking over
                                             the computer. Examples include `sasl_
↪credentials`, `client_strategy`,
                                             `cred_store`, and `pool_lifetime`.
    :returns: A ManagedADComputer object representing the computer taken over.
```

Joining an AD Domain Using an Existing Session

To join the local machine to an AD Domain, you can use an `ADDomain` object and use its function, but if you already have an existing session that you want to leverage, there's a standalone function that can be imported from the library directly as:

```
>>> from ms_active_directory import join_ad_domain_using_session
```

This function can be used to have a 1-line call to join the machine to the domain by creating a new computer to represent it, using a session you got from elsewhere.

You can specify a lot of properties about the computer to be created, but by default it will be named after the local machine's hostname (if it's a valid AD name) and created in AD's default Computers container. A strong password is set for the computer that is 120 characters long and random, strong encryption types are enabled, and Kerberos keys will be generated for the computer and written to the standard default system location (`/etc/krb5.keytab`).

A `ManagedADComputer` object is returned which has many helper functions for reading information about the created computer and managing its keys.

To join a domain using a pre-existing session and create a new computer, use the following function:

```
join_ad_domain_using_session(ad_session: ms_active_directory.core.ad_session.ADSession,␣
↪computer_name=None, computer_location=None,
                             computer_password=None, computer_encryption_types=None,␣
↪computer_hostnames=None,
                             computer_services=None, supports_legacy_behavior=False,␣
↪computer_key_file_path='/etc/krb5.keytab',
                             **additional_account_attributes) -> ms_active_directory.
↪core.managed_ad_objects.ManagedADComputer
    A fairly simple 'join a domain' function that requires minimal input - an AD session.
    Given those basic inputs, the domain's nearest controllers are automatically␣
↪discovered and an account is made
    with strong security settings. The account's attributes follow AD naming conventions␣
↪based on the computer's
    hostname by default.
    By providing an AD session, one can build a connection to the domain however they so␣
↪choose and then use it to
    join this computer, so you don't even need to necessarily use user credentials.
    :param ad_session: The ADSession object representing a connection with the domain to␣
↪be joined.
    :param computer_name: The name of the computer to take over in the domain. This␣
↪should be the sAMAccountName
                          of the computer, though if computer has a trailing $ in its␣
↪sAMAccountName and that is
                          omitted, that's ok. If not specified, we will attempt to find␣
↪a computer with a name
                          matching the local system's hostname.
    :param computer_location: The location in which to create the computer. This may be␣
↪specified as an LDAP-style
                              relative distinguished name (e.g. OU=ServiceMachines,
↪OU=Machines) or a windows path
                              style canonical name (e.g. example.com/Machines/
↪ServiceMachines).
                              If not specified, defaults to CN=Computers which is the␣
↪standard default for AD.
```

(continues on next page)

(continued from previous page)

```
    :param computer_password: The password to set for the computer when taking it over.␣
↪If not specified, a random
                              120 character password will be generated and set.
    :param computer_encryption_types: A list of encryption types, based on the␣
↪ADEncryptionType enum, to enable on
                                      the account created. These may be strings or enums;
↪ if they are strings,
                                      they should be strings of the encryption types as␣
↪written in kerberos
                                      RFCs or in AD management tools, and we will try to␣
↪map them to enums and
                                      raise an error if they don't match any supported␣
↪values.
                                      AES256-SHA1, AES128-SHA1, and RC4-HMAC encryption␣
↪types are supported. DES
                                      encryption types aren not.
                                      If not specified, defaults to [AES256-SHA1].
    :param computer_hostnames: Hostnames to set for the computer. These will be used to␣
↪set the dns hostname
                               attribute in AD. If not specified, the computer hostnames␣
↪will default to
                               [`computer_name`, `computer_name`.`domain`] which is the␣
↪AD standard default.
    :param computer_services: Services to enable on the computers hostnames. These␣
↪services dictate what clients
                              can get kerberos tickets for when communicating with this␣
↪computer, and this property
                              is used with `computer_hostnames` to set the service␣
↪principal names for the computer.
                              For example, having `nfs` specified as a service principal␣
↪is necessary if you want
                              to run an NFS server on this computer and have clients get␣
↪kerberos tickets for
                              mounting shares; having `ssh` specified as a service␣
↪principal is necessary for
                              clients to request kerberos tickets for sshing to the␣
↪computer.
                              If not specified, defaults to `HOST` which is the standard␣
↪AD default service.
                              `HOST` covers a wide variety of services, including `cifs`,
↪ `ssh`, and many others
                              depending on your domain. Determining exactly what␣
↪services are covered by `HOST`
                              in your domain requires checking the aliases set on a␣
↪domain controller.
    :param supports_legacy_behavior: If `True`, then an error will be raised if the␣
↪computer name is longer than
                                     15 characters (not including the trailing $). This␣
↪is because various older
                                     systems such as NTLM, certain UNC path applications,
↪ Netbios, etc. cannot
                                     use names longer than 15 characters. This name␣
↪cannot be changed after
```

(continues on next page)

(continued from previous page)

```
                                     creation, so this is important to control at␣
↪creation time.
                                     If not specified, defaults to `False`.
    :param computer_key_file_path: The path of where to write the keytab file for the␣
↪computer after taking it over.
                                   This will include keys for both user and server keys␣
↪for the computer.
                                   If not specified, defaults to /etc/krb5.keytab
    :param additional_account_attributes: Additional keyword argument can be specified␣
↪to set other LDAP attributes
                                          of the computer that are not covered above, or␣
↪where the above controls
                                          are not sufficiently granular. For example,␣
↪`userAccountControl` could
                                          be used to set the user account control values␣
↪for the computer if it's
                                          desired to set it differently from the default␣
↪(e.g. create a computer
                                          in a disabled state and enable it later).
    :returns: A ManagedADComputer object representing the computer created.
```

Joining an AD Domain by taking over an existing computer using an existing session

To join the local machine to an AD Domain, you can use an ADDomain object and use its function, but if you already have a pre-existing session from elsewhere, there's also a standalone function that can be imported from the library directly as:

```
>>> from ms_active_directory import join_ad_domain_by_taking_over_existing_computer_
↪using_session
```

This function can be used to have a 1-line call to join the machine to the domain by taking over a pre-created computer account. This is convenient for setups where the computer is pre-created with a lot of settings so that the machines joining don't need to know what attribute values to set.

Taking over an existing computer returns the a ManagedADComputer object, and writes kerberos keys to the local file system and such, but there's no option to specify things like services and dns hostnames as those are read from the existing computer.

To take over a computer in this way, use the following function:

```
join_ad_domain_by_taking_over_existing_computer_using_session(ad_session: ms_active_
↪directory.core.ad_session.ADSession,
                                                              computer_name=None,␣
↪computer_password=None, old_computer_password=None,
                                                              computer_key_file_path='/
↪etc/krb5.keytab') -> ms_active_directory.core.managed_ad_objects.ManagedADComputer
    A fairly simple 'join a domain' function using pre-created accounts, which requires␣
↪minimal input - an AD
    session. Specifying the name of the computer to takeover explicitly is also␣
↪encouraged.

    Given those basic inputs, the domain's nearest controllers are automatically␣
↪discovered and an account is found
```

(continues on next page)

(continued from previous page)

```
  with the computer name specified.
  That account is then taken over so that it can be controlled by the local system,␣
↪and kerberos keys and such are
  generated for it.

  By providing an AD session, one can build a connection to the domain however they so␣
↪choose and then use it to
  join this computer, so you don't even need to necessarily use user credentials.
  :param ad_session: The ADSession object representing a connection with the domain to␣
↪be joined.
  :param computer_name: The name of the computer to take over in the domain. This␣
↪should be the sAMAccountName
                        of the computer, though if computer has a trailing $ in its␣
↪sAMAccountName and that is
                        omitted, that's ok. If not specified, we will attempt to find␣
↪a computer with a name
                        matching the local system's hostname.
  :param computer_password: The password to set for the computer when taking it over.␣
↪If not specified, a random
                            120 character password will be generated and set.
  :param old_computer_password: The current password of the computer being taken over.␣
↪If specified, the action
                                of taking over the computer will use a "change password
↪" operation, which is less
                                privileged than a "reset password" operation. So␣
↪specifying this reduces the
                                permissions needed by the user specified.
  :param computer_key_file_path: The path of where to write the keytab file for the␣
↪computer after taking it over.
                                 This will include keys for both user and server keys␣
↪for the computer.
                                 If not specified, defaults to /etc/krb5.keytab
  :returns: A ManagedADComputer object representing the computer taken over.
```