# Azure SDK for Python Documentation

*Release 2.0.0 RC6*

**Microsoft**

**Apr 20, 2017**

# User Documentation

The Azure SDK for Python is a set of libraries which allow you to work on Azure for your management, runtime or data needs.

For a more general view of Azure and Python, you can go on the Python Developer Center for Azure

# CHAPTER 1

## Example Usage

This example shows:

- Authentication on Azure using an AD in your subscription,
- Creation of a Resource Group and a Storage account,
- Upload a simple "Hello world" HTML page and gives you the URL to get it.

```
from azure.common.credentials import UserPassCredentials
from azure.mgmt.resource import ResourceManagementClient
from azure.mgmt.storage import StorageManagementClient
from azure.storage import CloudStorageAccount
from azure.storage.blob.models import ContentSettings, PublicAccess

credentials = UserPassCredentials('user@domain.com', 'my_smart_password')
subscription_id = '33333333-3333-3333-3333-333333333333'

resource_client = ResourceManagementClient(credentials, subscription_id)
storage_client = StorageManagementClient(credentials, subscription_id)

resource_group_name = 'my_resource_group'
storage_account_name = 'myuniquestorageaccount'

resource_client.resource_groups.create_or_update(
    resource_group_name,
    {
        'location':'westus'
    }
)

async_create = storage_client.storage_accounts.create(
    resource_group_name,
    storage_account_name,
    {
        'location':'westus',
        'kind':'storage',
```

```
        'sku': {
            'name':'standard_ragrs'
        }
    }
)
async_create.wait()

storage_keys = storage_client.storage_accounts.list_keys(resource_group_name, storage_
↪account_name)
storage_keys = {v.key_name: v.value for v in storage_keys.keys}

storage_client = CloudStorageAccount(storage_account_name, storage_keys['key1'])
blob_service = storage_client.create_block_blob_service()

blob_service.create_container(
    'mycontainername',
    public_access=PublicAccess.Blob
)

blob_service.create_blob_from_bytes(
    'mycontainername',
    'myblobname',
    b'<center><h1>Hello World!</h1></center>',
    content_settings=ContentSettings('text/html')
)

print(blob_service.make_blob_url('mycontainername', 'myblobname'))
```

CHAPTER 2

# Installation

## Overview

You can install individually each library for each Azure service:

```
$ pip install azure-batch          # Install the latest Batch runtime library
$ pip install azure-mgmt-scheduler # Install the latest Storage management library
```

Preview packages can be installed using the `--pre` flag:

```
$ pip install --pre azure-mgmt-compute # will install only the latest Compute␣
↪Management library
```

More details and information about the available libraries and their status can be found in the *Installation Page*

## The `azure` meta-package

You can also install a set of Azure libraries in a single line using the `azure` meta-package. Since not all packages in this meta-package are published as stable yet, the `azure` meta-package is still in preview. However, the core packages, from code quality/completeness perspectives can at this time be considered "stable" - it will be officially labeled as such in sync with other languages as soon as possible. We are not planning on any further major changes until then.

Since it's a preview release, you need to use the `--pre` flag:

```
$ pip install --pre azure
```

or directly

```
$ pip install azure==2.0.0rc6
```

**Important:** The azure meta-package 1.0.3 is deprecated and is not working anymore.

CHAPTER 3

# Features

## Azure Resource Management

All documentation of management libraries for Azure are on this website. This includes:

- *Authorization* : Permissions, roles and more
- *Batch* : Manage Batch accounts and applications
- *Content Delivery Network* : Profiles, endpoints creation and more
- *Cognitive Services* : Create CS accounts and more
- *Commerce - Billing API* : RateCard and Usage Billing API
- *Compute* : Create virtual machines and more
- *Data Lake Analytics* : Manage account, job, catalog and more
- *Data Lake Store* : Manage account and more
- *DevTestLabs* : Create labs and more
- *DNS* : Create DNS zone, record set and more
- *IoTHub* : Create IoTHub account and more
- *KeyVault* : Create vaults and more
- *App Service* : Create App plan, Web Apps, Logic Apps and more
- Media Services : Create account and more
- *Network* : Create virtual networks, network interfaces, public IPs and more
- *Notification Hubs* : Namespaces, hub creation/deletion and more
- *PowerBI Embedded* : Create account and more
- *Redis Cache* : Create cache and more
- *Resource Management*:

  - resources : create resource groups, register providers and more
  - features : manage features of provider and more
  - locks : manage resource group lock and more
  - subscriptions : manage subscriptions and more
- *Scheduler* : Create job collections, create job and more
- *Server Manager* : Create gateways, nodes and more
- *SQL Database* : Create servers, databases nodes and more
- *Storage* : Create storage accounts, list keys, and more
- *Traffic Manager* : Create endpoints, profiles and more

## Azure Runtime

Some documentation of data libraries are on this website. This includes:

- *Batch*
- *Key Vault*
- *Azure Monitor*
- *Azure Active Directory Graph RBAC*
- *Service Bus* using HTTP.

> **Note:** For critical performance issue, the Service Bus team currently recommends AMQP.

These Azure services have Python data libraries which are directly hosted by the service team or are extensively documented on the Azure documentation website:

- Storage
- Azure Data Lake Store Filesystem
- Azure IoT Hub service and device SDKs for Python
- SQL Azure
- DocumentDB
- Application Insight
- Redis Cache
- Write an Azure WebApp in Python

## Azure Service Management

**Note:** The Service Management SDK is deprecated and no more features will be added.

This page describes the *usage and detailed features of Azure Service Management SDK*. At a glance:

- storage accounts: create, update, delete, list, regenerate keys
- affinity groups: create, update, delete, list, get properties
- locations: list
- hosted services: create, update, delete, list, get properties
- deployment: create, get, delete, swap, change configuration, update status, upgrade, rollback
- role instance: reboot, reimage
- discover addresses and ports for the endpoints of other role instances in your service
- get configuration settings and access local resources
- get role instance information for current role and other role instances
- query and set the status of the current role

CHAPTER 4

# System Requirements:

The supported Python versions are 2.7.x, 3.3.x, 3.4.x, 3.5.x and 3.6.x To download Python, please visit https://www.python.org/download/

We recommend Python Tools for Visual Studio as a development environment for developing your applications. Please visit http://aka.ms/python for more information.

# CHAPTER 5

## Need Help?:

Be sure to check out the Microsoft Azure Developer Forums on Stack Overflow if you have trouble with the provided code.

CHAPTER 6

# Contribute Code or Provide Feedback:

If you would like to become an active contributor to this project please follow the instructions provided in Microsoft Azure Projects Contribution Guidelines.

If you encounter any bugs with the library please file an issue in the Issues section of the project.

CHAPTER 7

# Indices and tables

- genindex
- modindex
- search

## Installation

### Installation with pip

You can install individually each library for each Azure service:

```
$ pip install azure-batch          # Install the latest Batch runtime library
$ pip install azure-mgmt-scheduler # Install the latest Storage management library
```

Preview packages can be installed using the `--pre` flag:

```
$ pip install --pre azure-mgmt-compute # will install only the latest Compute␣
↪Management library
```

You can also install a set of Azure libraries in a single line using the `azure` meta-package. Since not all packages in this meta-package are published as stable yet, the `azure` meta-package is still in preview. However, the core packages, from code quality/completeness perspectives can at this time be considered "stable" - it will be officially labeled as such in sync with other languages as soon as possible. We are not planning on any further major changes until then.

Since it's a preview release, you need to use the `--pre` flag:

```
$ pip install --pre azure
```

or directly

```
$ pip install azure==2.0.0rc6
```

**Important:** The azure meta-package 1.0.3 is deprecated and is not working anymore.

## Available packages

### Stable packages

| Package name | Version |
|---|---|
| azure-batch | 2.0.1 |
| azure-mgmt-batch | 3.0.1 |
| azure-mgmt-devtestlabs | 1.0.0 |
| azure-mgmt-dns | 1.0.1 |
| azure-mgmt-logic | 2.1.0 |
| azure-mgmt-redis | 4.1.0 |
| azure-mgmt-scheduler | 1.1.2 |
| azure-mgmt-servermanager | 1.0.0 |
| azure-servicebus | 0.21.0 |
| azure-servicemanagement-legacy | 0.20.5 |
| azure-storage | 0.33.0 |

### Preview packages

| Package name | Version |
|---|---|
| azure-keyvault | 0.2.0 |
| azure-monitor | 0.3.0 |
| azure-mgmt-resource | 1.0.0rc1 |
| azure-mgmt-compute | 1.0.0rc1 |
| azure-mgmt-network | 1.0.0rc2 |
| azure-mgmt-storage | 1.0.0rc1 |
| azure-mgmt-keyvault | 0.31.0 |
| azure-graphrbac | 0.30.0 |
| azure-mgmt-authorization | 0.30.0rc6 |
| azure-mgmt-cdn | 0.30.2 |
| azure-mgmt-cognitiveservices | 0.30.0rc6 |
| azure-mgmt-containerregistry | 0.2.1 |
| azure-mgmt-commerce | 0.30.0rc6 |
| azure-mgmt-datalake-analytics | 0.1.4 |
| azure-mgmt-datalake-store | 0.1.3 |
| azure-mgmt-documentdb | 0.1.2 |
| azure-mgmt-eventhub | 0.2.0 |
| azure-mgmt-iothub | 0.2.1 |
| azure-mgmt-media | 0.1.1 |
| azure-mgmt-monitor | 0.2.0 |
| azure-mgmt-notificationhubs | 0.30.0 |
| azure-mgmt-powerbiembedded | 0.30.0rc6 |
| azure-mgmt-search | 0.1.0 |
| azure-mgmt-servicebus | 0.1.0 |
| azure-mgmt-sql | 0.5.0 |
| azure-mgmt-trafficmanager | 0.30.0 |
| azure-mgmt-web | 0.31.1 |

### Install from Github

If you want to install `azure` from source:

```
git clone git://github.com/Azure/azure-sdk-for-python.git
cd azure-sdk-for-python
python setup.py install
```

The `dev` branch contains the work in progress.

## Resource Management Authentication

For general information on resource management, see *Resource Management*.

To be able to use use the ARM library, you need to obtain one of these instances:

- azure.common.credentials.UserPassCredentials
- azure.common.credentials.ServicePrincipalCredentials
- msrestazure.azure_active_directory.AdalAuthentication

And use it as credentials in your management configuration client. These three instances correspond to:

- OAuth authentication using Azure Active Directory user/password
- OAuth authentication using Active Directory application and service principal
- A wrapper on top of *ADAL for Python <https://github.com/AzureAD/azure-activedirectory-library-for-python>*

## Using Service Principal

There is now a detailled official tutorial to describe this: https://azure.microsoft.com/en-us/documentation/articles/resource-group-create-service-principal-portal/

At this point, you must have:

- Your client id. Found in the "client id" box in the "Configure" page of your application in the Azure portal
- Your secret key. Generated when you have created the application. You cannot show the key after creation. If you've lost the current key, you must create a new one in the "Configure" page of your application.
- You AD tenant id. It's an UUID (e.g. ABCDEFAB-1234-ABCD-1234-ABCDEFABCDEF) which point to the AD containing your application. You will found it in the URL when you are in the Azure portal in your AD, or in the "view endpoints" in any of the given url.

Then, you can create your credentials instance:

```
from azure.common.credentials import ServicePrincipalCredentials

credentials = ServicePrincipalCredentials(
    client_id = 'ABCDEFAB-1234-ABCD-1234-ABCDEFABCDEF',
    secret = 'XXXXXXXXXXXXXXXXXXXXXXXX',
    tenant = 'ABCDEFAB-1234-ABCD-1234-ABCDEFABCDEF'
)
```

## Using AD User/Password

1. Connect to the Azure Classic Portal with your admin account
2. Create a user in your default AAD

**You must NOT activate Multi-Factor Authentication**

3. Go to Settings - Administrators
4. Click on Add and enter the email of the new user. Check the checkbox of the subscription you want to test with this user.
5. Login to Azure Portal with this new user to change the temporary password to a new one. You will not be able to use the temporary password for OAuth login.

You are now able to log in Python using OAuth.

```
from azure.common.credentials import UserPassCredentials

credentials = UserPassCredentials(
    'user@domain.com',     # Your new user
    'my_smart_password',   # Your password
)
```

### Using ADAL

ADAL for Python is a library from the Azure Active Directory team, that proposes the more complex scenarios not covered by the two previous instances (like 2FA). Please refer to the ADAL website for all the available scenarios list and samples.

For example, this code from the ADAL tutorial:

```
context = adal.AuthenticationContext('https://login.microsoftonline.com/ABCDEFGH-1234-
↪1234-1234-ABCDEFGHIJKL')
RESOURCE = '00000002-0000-0000-c000-000000000000' #AAD graph resource
token = context.acquire_token_with_client_credentials(
    RESOURCE,
    "http://PythonSDK",
    "Key-Configured-In-Portal")
```

can be written here:

```
from msrestazure.azure_active_directory import AdalAuthentication

context = adal.AuthenticationContext('https://login.microsoftonline.com/ABCDEFGH-1234-
↪1234-1234-ABCDEFGHIJKL')
RESOURCE = '00000002-0000-0000-c000-000000000000' #AAD graph resource
credentials = AdalAuthentication(
    context.acquire_token_with_client_credentials,
    RESOURCE,
    "http://PythonSDK",
    "Key-Configured-In-Portal")
```

or using a lambda if you prefer:

```
from msrestazure.azure_active_directory import AdalAuthentication

context = adal.AuthenticationContext('https://login.microsoftonline.com/ABCDEFGH-1234-
↪1234-1234-ABCDEFGHIJKL')
RESOURCE = '00000002-0000-0000-c000-000000000000' #AAD graph resource
credentials = AdalAuthentication(
    lambda: context.acquire_token_with_client_credentials(
        RESOURCE,
        "http://PythonSDK",
        "Key-Configured-In-Portal"
    )
)
```

Note that the UserPassCredentials and ServicePrincipalCredentials scenarios are also covered by the ADAL library. In the near future their implementation will be rewritten using ADAL.

## Multi-cloud - use Azure on all regions

You can use the Azure SDK for Python to connect to all regions where Azure is available (list of Azure regions is available here).

## Use the SDK on a different region than US Azure

By default, the Azure SDK for Python is configured to connect to public Azure. To connect to another region, a few things have to be considered:

- What is the endpoint where to ask for a token (authentication)?
- What is the endpoint where I will use this token (usage)?

This is a generic example:

```
from azure.common.credentials import UserPassCredentials
from azure.mgmt.resource import ResourceManagementClient

# Public Azure - default values
authentication_endpoint = 'https://login.microsoftonline.com/'
azure_endpoint = 'https://management.azure.com/'

credentials = UserPassCredentials(
    'user@domain.com',
    'my_smart_password',
    auth_uri=authentication_endpoint,
    resource=azure_endpoint
)
subscription_id = '33333333-3333-3333-3333-333333333333'

resource_client = ResourceManagementClient(
    credentials,
    subscription_id,
    base_url=azure_endpoint
)
```

## Azure Government

Azure Government is currently using the same authentication endpoint that public azure. This means that we can use the default *authentication_endpoint*.

```
from azure.common.credentials import UserPassCredentials
from azure.mgmt.resource import ResourceManagementClient

azure_endpoint = 'https://management.usgovcloudapi.net/'

credentials = UserPassCredentials(
    'user@domain.com',
    'my_smart_password',
    resource=azure_endpoint
)
subscription_id = '33333333-3333-3333-3333-333333333333'

resource_client = ResourceManagementClient(
    credentials,
    subscription_id,
    base_url=azure_endpoint
)
```

### Azure Germany

```
from azure.common.credentials import UserPassCredentials
from azure.mgmt.resource import ResourceManagementClient

authentication_endpoint = 'https://login.microsoftonline.de/'
azure_endpoint = 'https://management.microsoftazure.de/'

credentials = UserPassCredentials(
    'user@domain.com',
    'my_smart_password',
    auth_uri=authentication_endpoint,
    resource=azure_endpoint
)
subscription_id = '33333333-3333-3333-3333-333333333333'

resource_client = ResourceManagementClient(
    credentials,
    subscription_id,
    base_url=azure_endpoint
)
```

### Azure China

```
from azure.common.credentials import UserPassCredentials
from azure.mgmt.resource import ResourceManagementClient

authentication_endpoint = 'https://login.chinacloudapi.cn/'
azure_endpoint = 'https://management.chinacloudapi.cn/'

credentials = UserPassCredentials(
    'user@domain.com',
    'my_smart_password',
    auth_uri=authentication_endpoint,
    resource=azure_endpoint
)
subscription_id = '33333333-3333-3333-3333-333333333333'

resource_client = ResourceManagementClient(
    credentials,
    subscription_id,
    base_url=azure_endpoint
)
```

## Exception handling

**Important:** This document concerns every package except *azure-servicebus* and *azure-servicemanagement-legacy*

This document covers the exceptions that you can get from the Azure SDK for Python, and will help you decide what is worth retrying or what is a critical failure, depending on your application.

Every exception related to the service content will be referenced in the docstring of the method. For instance, for this `create_or_update` operation you can see that you might receive a `CloudError`.

Where the service team provides no specific sub-classing, the basic exception you can expect from an operation is `CloudError`. In some situations however, the exceptions are specialized by package (see *azure-batch* package which uses `BatchErrorException`). If a specialized exception is used, it will always be a sub-class at some level of `ClientException`.

Whether you should retry the same query or not for these exceptions depends on the service and the error content and cannot be generalized on this article. For instance, an update to a SQL Database might be refused because you have another underlying operation (in this case you can retry); or a create request can be refused because your selected name does not meet the pattern enforced by the server (in this case retry will never work). Use the *error* and *response* attributes of `CloudError` to decide.

All these operations can also raise this set of generic exceptions that are defined in the `msrest.exceptions` module:

- `AuthenticationError`: password invalid/expired, etc. Retry will likely not work.
- `ClientRequestError`: *requests* library raised an exception (connection error, mostly HTTP level issues). Retry may work, though unlikely if your base_url is incorrect.
- `DeserializationError`: unexpected RestAPI answer. Retry will likely not work. Please create an issue on Github if you see this exception.
- `HttpOperationError`: bad HTTP status code from Azure. See inner exception to decide if it's worth retrying.
- `SerializationError`: unable to serialize your request. Your Python code didn't respect the expected model. Retry will never work. Please create an issue on Github if you see this exception and are 100% your parameters are correct.
- `TokenExpiredError`: please renew your credentials. Likely retry with new credentials will be ok.
- `ValidationError`: client side check of your request failed. Fix your parameters with expected value. For instance, you didn't respect the regexp for the account name. This will never work on retry.

## Asynchronous operation

An asynchronous operation is an operation that returns an `AzureOperationPoller` (like `create_or_update`). Using this kind of operation usually requires two lines:

```
async_poller = client.network_security_groups.create_or_update(myparameters)
result = async_poller.result()
```

or, if this asynchronous operation is not returning a result:

```
async_poller = client.network_security_groups.create_or_update(myparameters)
async_poller.wait()
```

Our recommendation is to surround both of the statements with the necessary try/except. More precisely, the first call might fail on the initial call and the second one might fail during polling the status of the operation

---

**Important:** Old versions of the packages never failed on the first call, but this behavior was replaced by the one described and you should follow this pattern even for old packages.

---

### Raw operation

All operations accept a *raw=True* parameter to indicate that the method must return the *requests.Response* instance directly. All the above exceptions are still applicable, except for `DeserializationError`, since the response will not be deserialized in this case.

## Resource Management

### Resource Management libraries

The azure-mgmt-resource package is splitted into several sub-librairies:

- resources : manage resources groups, template, etc (Introduction to ARM)
- features : manage features of provider (RestAPI reference)
- locks : manage resource group lock (RestAPI reference)
- subscriptions : manage subscriptions (RestAPI reference)
- policy : manage and control access to resources (RestAPI reference)

See the examples below for managing resource groups.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.resource.resources import ResourceManagementClient
    from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
            'user@domain.com',      # Your user
            'my_password',          # Your password
    )

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
```

### Usage sample for Resource Groups and Resources management

https://github.com/Azure-Samples/resource-manager-python-resources-and-groups

### Usage sample for Template deployment

https://github.com/Azure-Samples/resource-manager-python-template-deployment

## Apps Management (Web Apps, Logic Apps)

For general information on resource management, see *Resource Management*.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.logic import LogicManagementClient
from azure.mgmt.web import WebSiteManagementClient
    from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
            'user@domain.com',      # Your user
            'my_password',          # Your password
    )

logic_client = LogicManagementClient(
    credentials,
    subscription_id
)
web_client = WebSiteManagementClient(
    credentials,
    subscription_id
)
```

### Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Web')
resource_client.providers.register('Microsoft.Logic')
```

### Usage sample for Web App Management

https://github.com/Azure-Samples/app-service-web-python-manage

### Create a Logic App Workflow

The following code creates a logic app workflow.

```
from azure.mgmt.logic.models import Workflow

group_name = 'myresourcegroup'
workflow_name = '12HourHeartBeat'
logic_client.workflows.create_or_update(
    group_name,
    workflow_name,
    Workflow(
        location = 'West US',
        definition={
            "$schema": "https://schema.management.azure.com/providers/Microsoft.Logic/
↪schemas/2016-06-01/workflowdefinition.json#",
            "contentVersion": "1.0.0.0",
            "parameters": {},
            "triggers": {},
            "actions": {},
            "outputs": {}
        }
    )
)
```

## Authorization Management

For general information on resource management, see *Resource Management*.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.authorization import AuthorizationManagementClient
    from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
            'user@domain.com',      # Your user
            'my_password',          # Your password
    )
```

```
authorization_client = AuthorizationManagementClient(
    credentials,
    subscription_id
)
```

### Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Authorization')
```

### Check permissions for a resource group

The following code checks permissions in a given resource group. To create or manage resource groups, see *Resource Management*.

```
from azure.mgmt.redis.models import Sku, RedisCreateOrUpdateParameters

group_name = 'myresourcegroup'
permissions = self.authorization_client.permissions.list_for_resource_group(
    group_name
)
# permissions is a iterable of Permissions instances
```

## Batch Management

For more information on the Azure Batch service, check out the Batch Documentation. For working samples, see the Batch samples repo.

The API documentation: *API*

### Create the Batch Management client

The following code creates an instance of the management client. You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.batch import BatchManagementClient
    from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
```

```
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
            'user@domain.com',      # Your user
            'my_password',          # Your password
    )

batch_client = BatchManagementClient(
    credentials,
    subscription_id
)
```

## Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Batch')
```

## Create a Batch Account

A Batch Account will need to be created in a specified location and resource group. The default Batch Account quota is 1 per location per subscription, but can be increased to a maximum of 50. Please contact support if you require a quota increase. For more information on resource groups and resource management, see *Resource Management*.

In order to make use of Application Packages, a storage account will need to be linked to the Batch Account. A linked storage account is known as 'auto-storage'. A storage account can be created with the *Storage Resource Management Client*.

```
# Create a Resource Group (or use an existing one)
import azure.mgmt.resource

RESOURCE_GROUP = 'python_sdk'
LOCATION = 'westus'

resource_client = = azure.mgmt.storage.ResourceManagementClient(
        azure.mgmt.resources.ResourceManagementClientConfiguration(
                credentials, # See section above
                subscription_id
        )
)
group = azure.mgmt.resource.resources.models.ResourceGroup(
        name=RESOURCE_GROUP,
        location=LOCATION
)
resource_client.resource_groups.create_or_update(
        RESOURCE_GROUP,
```

```
        group,
)


# Create a storage account for 'auto-storage' (or use an existing one)
import azure.mgmt.storage
storage_client = azure.mgmt.storage.StorageManagementClient(
        azure.mgmt.storage.StorageManagementClientConfiguration(
                credentials, # See section above
                subscription_id
        )
)
storage_params = azure.mgmt.storage.models.StorageAccountCreateParameters(
        location=LOCATION,
        account_type=azure.mgmt.storage.models.AccountType.standard_lrs
)
creating = storage_client.storage_accounts.create(
        RESOURCE_GROUP,
        'pythonstorageaccount',
        storage_params
)
creating.wait()

# Create a Batch Account, specifying the storage account we want to link
storage_resource = '/subscriptions/{}/resourceGroups/{}/providers/Microsoft.Storage/
↪storageAccounts/{}'.format(
        subscription_id,
        RESOURCE_GROUP,
        'pythonstorageaccount'
)
batch_account = azure.mgmt.batch.models.BatchAccountCreateParameters(
        location=LOCATION,
        auto_storage=azure.mgmt.batch.models.AutoStorageBaseProperties(storage_
↪resource)
)
creating = batch_client.account.create('MyBatchAccount', LOCATION, batch_account)
creating.wait()
```

Account keys (used for authenticating the *Batch Client*) can be retrieved or regenerated.

```
batch_client.account.regenerate_key(
        RESOURCE_GROUP,
        'MyBatchAccount',
        'Primary'
)
accounts_keys = batch_client.account.list_keys(RESOURCE_GROUP, 'MyBatchAccount')
print('Updated primary key: {}'.format(accounts_keys.primary))
```

## Application Packages

Application packages can be configured to be used by the the *Batch Client* for running tasks. An Application can have multiple versioned packages (zipped directories containing the application to be executed on the Compute Node) associated with it. You can find an overview of this feature in this article on application deployment with Azure Batch Applications.

```
# Create Application reference
batch_client.application.add(
        RESOURCE_GROUP,
        'MyBatchAccount',
        'MyApplicationId'
        allow_updates=True,
        display_name='Test App v1'
)

# Add a new package to the application
package_ref = batch_client.application.add_application_package(
        RESOURCE_GROUP,
        'MyBatchAccount',
        'MyApplicationId',
        'v1.0'
)

# Upload a zip directory for the created package reference
import requests
with open('my_application.zip', 'rb') as app_data:
        headers = {'x-ms-blob-type': 'BlockBlob'}
        requests.put(package_ref.storage_url, headers=headers, data=app_data.read())

# In order to use the application in a job, the package must be activated
batch_client.application.activate_application_package(
        RESOURCE_GROUP,
        'MyBatchAccount',
        'MyApplicationId',
        'v1.0',
        'zip'
)
```

# Content Delivery Network Management

For general information on resource management, see *Resource Management*.

## Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.cdn import CdnManagementClient
    from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
            'user@domain.com',      # Your user
            'my_password',          # Your password
```

```
    )

cdn_client = CdnManagementClient(
    credentials,
    subscription_id
)
```

### Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Cdn')
```

### Check name availability

The following code check the name availability of a end-point.

```
output = cdn_client.check_name_availability('myendpoint')
# output is a CheckNameAvailabilityOutput instance
print(output.name_available)
```

## Commerce - Billing API

### Create the commerce client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.commerce import UsageManagementClient
from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
            'user@domain.com',      # Your user
            'my_password',          # Your password
    )
```

```
commerce_client = UsageManagementClient(
    credentials,
    subscription_id
)
```

### Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Commerce')
```

### Get rate card

```
# OfferDurableID: https://azure.microsoft.com/en-us/support/legal/offer-details/
rate = commerce_client.rate_card.get(
    "OfferDurableId eq 'MS-AZR-0062P' and Currency eq 'USD' and Locale eq 'en-US' and␣
↪RegionInfo eq 'US'"
)
```

### Get Usage

```
from datetime import date, timedelta

# Takes onky dates in full ISO8601 with 'T00:00:00Z'
usage_list = commerce_client.usage_aggregates.list(
    str(date.today() - timedelta(days=1))+'T00:00:00Z',
    str(date.today())+'T00:00:00Z'
)
```

## Compute and Network Management

For general information on resource management, see *Resource Management*.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.compute import ComputeManagementClient
from azure.mgmt.network import NetworkManagementClient
    from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
            'user@domain.com',      # Your user
            'my_password',          # Your password
    )

compute_client = ComputeManagementClient(
    credentials,
    subscription_id
)

network_client = NetworkManagementClient(
    credentials,
    subscription_id
)
```

## Registration

Some operations in the compute/network ARM APIs require a one-time registration of the storage provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Compute')
resource_client.providers.register('Microsoft.Network')
```

## Virtual Machine sample

You can get a fully functionnal Virtual Machine sample from the AzureSample Github repository: https://github.com/Azure-Samples/virtual-machines-python-manage

## Load Balancer sample

You can get a fully functionnal Load Balancer sample from the AzureSample Github repository: https://github.com/Azure-Samples/network-python-manage-loadbalancer

### List images

Use the following code to print all of the available images to use for creating virtual machines, including all skus and versions.

```
region = 'eastus2'

result_list_pub = compute_client.virtual_machine_images.list_publishers(
    region,
)

for publisher in result_list_pub:
    result_list_offers = compute_client.virtual_machine_images.list_offers(
        region,
        publisher.name,
    )

    for offer in result_list_offers:
        result_list_skus = compute_client.virtual_machine_images.list_skus(
            region,
            publisher.name,
            offer.name,
        )

        for sku in result_list_skus:
            result_list = compute_client.virtual_machine_images.list(
                region,
                publisher.name,
                offer.name,
                sku.name,
            )

            for version in result_list:
                result_get = compute_client.virtual_machine_images.get(
                    region,
                    publisher.name,
                    offer.name,
                    sku.name,
                    version.name,
                )

                print('PUBLISHER: {0}, OFFER: {1}, SKU: {2}, VERSION: {3}'.format(
                    publisher.name,
                    offer.name,
                    sku.name,
                    version.name,
                ))
```

## Notification Hubs Management

For general information on resource management, see *Resource Management*.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.notificationhubs import NotificationHubsManagementClient
    from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
            'user@domain.com',      # Your user
            'my_password',          # Your password
    )

redis_client = NotificationHubsManagementClient(
    credentials,
    subscription_id
)
```

## Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.NotificationHubs')
```

## Check namespace availability

The following code check namespace availability of a notification hub.

```
from azure.mgmt.notificationhubs.models import CheckAvailabilityParameters

account_name = 'mynotificationhub'
output = notificationhubs_client.namespaces.check_availability(
    azure.mgmt.notificationhubs.models.CheckAvailabilityParameters(
        name = account_name
    )
)
# output is a CheckAvailibilityResource instance
print(output.is_availiable) # Yes, it's 'availiable', it's a typo in the REST API
```

# Redis Cache Management

For general information on resource management, see *Resource Management*.

## Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.redis import RedisManagementClient
    from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
            'user@domain.com',      # Your user
            'my_password',          # Your password
    )

redis_client = RedisManagementClient(
    credentials,
    subscription_id
)
```

## Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Cache')
```

## Create Redis Cache

The following code creates a new redis cache under an existing resource group. To create or manage resource groups, see *Resource Management*.

```
from azure.mgmt.redis.models import Sku, RedisCreateOrUpdateParameters

group_name = 'myresourcegroup'
cache_name = 'mycachename'
redis_cache = redis_client.redis.create_or_update(
    group_name,
    cache_name,
    RedisCreateOrUpdateParameters(
        sku = Sku(name = 'Basic', family = 'C', capacity = '1'),
        location = "West US"
    )
```

```
)
# redis_cache is a RedisResourceWithAccessKey instance
```

# Scheduler Management

For general information on resource management, see *Resource Management*.

## Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.scheduler import SchedulerManagementClient
    from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
            'user@domain.com',      # Your user
            'my_password',          # Your password
    )

scheduler_client = SchedulerManagementClient(
    credentials,
    subscription_id
)
```

## Registration

Some operations in the storage ARM APIs require a one-time registration of the storage provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Scheduler')
```

## Create a job collection

The following code creates a job collection under an existing resource group. To create or manage resource groups, see *Resource Management*.

```
from azure.mgmt.scheduler.models import JobCollectionDefinition,␣
↪JobCollectionProperties, Sku

group_name = 'myresourcegroup'
job_collection_name = "myjobcollection"
scheduler_client.job_collections.create_or_update(
    group_name,
    job_collection_name,
    JobCollectionDefinition(
        location = "West US",
        properties = JobCollectionProperties(
            sku = Sku(
                name="Free"
            )
        )
    )
)
# scheduler_client is a JobCollectionDefinition instance
```

# Storage Management

For general information on resource management, see *Resource Management*.

## Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.storage import StorageManagementClient
    from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
            'user@domain.com',      # Your user
            'my_password',          # Your password
    )

storage_client = StorageManagementClient(
    credentials,
    subscription_id
)
```

## Registration

Some operations in the storage ARM APIs require a one-time registration of the storage provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Storage')
```

### Usage sample

https://github.com/Azure-Samples/storage-python-manage

## Cognitive Services Management

For general information on resource management, see *Resource Management*.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.cognitiveservices import CognitiveServicesManagementClient
from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
    'user@domain.com',  # Your user
    'my_password',      # Your password
)

cognitiveservices_client = CognitiveServicesManagementClient(
    credentials,
    subscription_id
)
```

### Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
```

```
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.CognitiveServices')
```

# Azure Data Lake Management Analytics

For general information on resource management, see *Resource Management*.

## Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.datalake.analytics import (
            DataLakeAnalyticsAccountManagementClient,
            DataLakeAnalyticsCatalogManagementClient,
            DataLakeAnalyticsJobManagementClient
    )
from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
    'user@domain.com',  # Your user
    'my_password',      # Your password
)

account_client = DataLakeAnalyticsAccountManagementClient(
    credentials,
    subscription_id
)
catalog_client = DataLakeAnalyticsCatalogManagementClient(
    credentials,
    subscription_id
)
job_client = DataLakeAnalyticsJobManagementClient(
    credentials,
    subscription_id
)
```

## Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.DataLakeAnalytics')
```

### Samples

Samples writing in progress! In the meantime, you should look at the unit tests of this package:

https://github.com/Azure/azure-sdk-for-python/blob/master/azure-mgmt/tests/test_mgmt_datalake_analytics.py

## Azure Data Lake Management Store

For general information on resource management, see *Resource Management*.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.datalake.store import DataLakeStoreAccountManagementClient
from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
    'user@domain.com',  # Your user
    'my_password',      # Your password
)

client = DataLakeStoreAccountManagementClient(
    credentials,
    subscription_id
)
```

### Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.DataLakeStore')
```

### Samples

Samples writing in progress! In the meantime, you should look at the unit tests of this package:

https://github.com/Azure/azure-sdk-for-python/blob/master/azure-mgmt/tests/test_mgmt_datalake_store.py

## DevTestLabs Management

For general information on resource management, see *Resource Management*.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.devtestlabs import DevTestLabsClient
from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
    'user@domain.com',  # Your user
    'my_password',      # Your password
)

devtestlabs_client = DevTestLabsClient(
    credentials,
    subscription_id
)
```

### Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.DevTestLab')
```

### Create lab

```
async_lab = self.client.lab.create_or_update_resource(
    'MyResourceGroup',
    'MyLab',
    {'location': 'westus'}
)
lab = async_lab.result() # Blocking wait
```

## DNS Management

For general information on resource management, see *Resource Management*.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.dns import DnsManagementClient
from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
    'user@domain.com',  # Your user
    'my_password',      # Your password
)

dns_client = DnsManagementClient(
    credentials,
    subscription_id
)
```

### Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Network')
```

### Create DNS zone

```
# The only valid value is 'global', otherwise you will get a:
# The subscription is not registered for the resource type 'dnszones' in the location
↪'westus'.
zone = dns_client.zones.create_or_update(
        'MyResourceGroup',
        'pydns.com',
        {
                'location': 'global'
        }
)
```

### Create a Record Set

```
record_set = dns_client.record_sets.create_or_update(
        'MyResourceGroup',
        'pydns.com',
        'MyRecordSet',
        'A',
        {
                "ttl": 300,
                "arecords": [
                        {
                                "ipv4_address": "1.2.3.4"
                        },
                        {
                                "ipv4_address": "1.2.3.5"
                        }
                ]
        }
)
```

## IoTHub Management

For general information on resource management, see *Resource Management*.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.iothub import IotHubClient
from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
    'user@domain.com',  # Your user
    'my_password',      # Your password
)

iothub_client = IotHubClient(
    credentials,
    subscription_id
)
```

## Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Devices')
```

## Create an IoTHub

```
async_iot_hub = iothub_client.iot_hub_resource.create_or_update(
        'MyResourceGroup',
        'MyIoTHubAccount',
        {
                'location': 'westus',
                'subscriptionid': subscription_id,
                'resourcegroup': 'MyResourceGroup',
                'sku': {
                  'name': 'S1',
                  'capacity': 2
                },
                'properties': {
                  'enable_file_upload_notifications': False,
                  'operations_monitoring_properties': {
                        'events': {
                          "C2DCommands": "Error",
                          "DeviceTelemetry": "Error",
                          "DeviceIdentityOperations": "Error",
```

```
                            "Connections": "Information"
                        }
                    },
                    "features": "None",
                }
        }
)
iothub = async_iot_hub.result() # Blocking wait for creation
```

# KeyVault Management

For general information on resource management, see *Resource Management*.

## Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.keyvault import KeyVaultManagementClient
from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
    'user@domain.com',  # Your user
    'my_password',      # Your password
)

kv_client = KeyVaultManagementClient(
    credentials,
    subscription_id
)
```

## Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.KeyVault')
```

### Samples usage

https://github.com/Azure-Samples/key-vault-python-manage

## Managed Disks

Azure Managed Disks and 1000 VMs in a Scale Set are now generally available. Azure Managed Disks provide a simplified disk Management, enhanced Scalability, better Security and Scale. It takes away the notion of storage account for disks, enabling customers to scale without worrying about the limitations associated with storage accounts. This post provides a quick introduction and reference on consuming the service from Python.

From a developer perspective, the Managed Disks experience in Azure CLI is idomatic to the CLI experience in other cross-platform tools. You can use the Azure Python SDK and the azure-mgmt-compute package 0.33.0 to administer Managed Disks. You can create a compute client using this tutorial. The complete API documentation is available on ReadTheDocs.

## Standalone Managed Disks

You can easily create standalone Managed Disks in a variety of ways.

### Create an empty Managed Disk.

```
from azure.mgmt.compute.models import DiskCreateOption

async_creation = compute_client.disks.create_or_update(
    'my_resource_group',
    'my_disk_name',
    {
        'location': 'westus',
        'disk_size_gb': 20,
        'creation_data': {
            'create_option': DiskCreateOption.empty
        }
    }
)
disk_resource = async_creation.result()
```

### Create a Managed Disk from Blob Storage.

```
from azure.mgmt.compute.models import DiskCreateOption

async_creation = compute_client.disks.create_or_update(
    'my_resource_group',
    'my_disk_name',
    {
        'location': 'westus',
        'creation_data': {
            'create_option': DiskCreateOption.import_enum,
            'source_uri': 'https://bg09.blob.core.windows.net/vm-images/non-existent.
↪vhd'
```

```
        }
    }
)
disk_resource = async_creation.result()
```

### Create a Managed Disk from your own Image

```
from azure.mgmt.compute.models import DiskCreateOption

# If you don't know the id, do a 'get' like this to obtain it
managed_disk = compute_client.disks.get(self.group_name, 'myImageDisk')
async_creation = compute_client.disks.create_or_update(
    'my_resource_group',
    'my_disk_name',
    {
        'location': 'westus',
        'creation_data': {
            'create_option': DiskCreateOption.copy,
            'source_resource_id': managed_disk.id
        }
    }
)

disk_resource = async_creation.result()
```

## Virtual Machine with Managed Disks

You can create a Virtual Machine with an implicit Managed Disk for a specific disk image. Creation is simplified with implicit creation of managed disks without specifying all the disk details. You do not have to worry about creating and managing Storage Accounts.

A Managed Disk is created implicitly when creating VM from an OS image in Azure. In the `storage_profile` parameter, `os_disk` is now optional and you don't have to create a storage account as required precondition to create a Virtual Machine.

```
storage_profile = azure.mgmt.compute.models.StorageProfile(
     image_reference = azure.mgmt.compute.models.ImageReference(
         publisher='Canonical',
         offer='UbuntuServer',
         sku='16.04-LTS',
         version='latest'
     )
 )
```

This `storage_profile` parameter is now valid. To get a complete example on how to create a VM in Python (including network, etc), check the full VM tutorial in Python here.

You can easily attach a previously provisioned Managed Disk.

```
vm = compute.virtual_machines.get(
    'my_resource_group',
    'my_vm'
)
managed_disk = compute_client.disks.get('my_resource_group', 'myDisk')
```

```
vm.storage_profile.data_disks.append({
    'lun': 12, # You choose the value, depending of what is available for you
    'name': managed_disk.name,
    'create_option': DiskCreateOption.attach,
    'managed_disk': {
        'id': managed_disk.id
    }
})
async_update = compute_client.virtual_machines.create_or_update(
    'my_resource_group',
    vm.name,
    vm,
)
async_update.wait()
```

## Virtual Machine Scale Sets with Managed Disks

Before Managed Disks, you needed to create a storage account manually for all the VMs you wanted inside your Scale Set, and then use the list parameter `vhd_containers` to provide all the storage account name to the Scale Set RestAPI. The official transition guide is available in this article.

Now with Managed Disk, you don't have to manage any storage account at all. If you're are used to the VMSS Python SDK, your `storage_profile` can now be exactly the same as the one used in VM creation:

```
'storage_profile': {
    'image_reference': {
        "publisher": "Canonical",
        "offer": "UbuntuServer",
        "sku": "16.04-LTS",
        "version": "latest"
    }
},
```

The full sample being:

```
naming_infix = "PyTestInfix"
vmss_parameters = {
    'location': self.region,
    "overprovision": True,
    "upgrade_policy": {
        "mode": "Manual"
    },
    'sku': {
        'name': 'Standard_A1',
        'tier': 'Standard',
        'capacity': 5
    },
    'virtual_machine_profile': {
        'storage_profile': {
            'image_reference': {
                "publisher": "Canonical",
                "offer": "UbuntuServer",
                "sku": "16.04-LTS",
                "version": "latest"
            }
```

```
        },
        'os_profile': {
            'computer_name_prefix': naming_infix,
            'admin_username': 'Foo12',
            'admin_password': 'BaR@123!!!!',
        },
        'network_profile': {
            'network_interface_configurations' : [{
                'name': naming_infix + 'nic',
                "primary": True,
                'ip_configurations': [{
                    'name': naming_infix + 'ipconfig',
                    'subnet': {
                        'id': subnet.id
                    }
                }]
            }]
        }
    }
}

# Create VMSS test
result_create = compute_client.virtual_machine_scale_sets.create_or_update(
    'my_resource_group',
    'my_scale_set',
    vmss_parameters,
)
vmss_result = result_create.result()
```

# Other Operations with Managed Disks

## Resizing a managed disk.

```
managed_disk = compute_client.disks.get('my_resource_group', 'myDisk')
managed_disk.disk_size_gb = 25
async_update = self.compute_client.disks.create_or_update(
    'my_resource_group',
    'myDisk',
    managed_disk
)
async_update.wait()
```

## Update the Storage Account type of the Managed Disks.

```
from azure.mgmt.compute.models import StorageAccountTypes

managed_disk = compute_client.disks.get('my_resource_group', 'myDisk')
managed_disk.account_type = StorageAccountTypes.standard_lrs
async_update = self.compute_client.disks.create_or_update(
    'my_resource_group',
    'myDisk',
    managed_disk
```

```
)
async_update.wait()
```

### Create an image from Blob Storage.

```
async_create_image = compute_client.images.create_or_update(
    'my_resource_group',
    'myImage',
    {
        'location': 'westus',
        'storage_profile': {
            'os_disk': {
                'os_type': 'Linux',
                'os_state': "Generalized",
                'blob_uri': 'https://bg09.blob.core.windows.net/vm-images/non-
↪existent.vhd',
                'caching': "ReadWrite",
            }
        }
    }
)
image = async_create_image.result()
```

### Create a snapshot of a Managed Disk that is currently attached to a Virtual Machine.

```
managed_disk = compute_client.disks.get('my_resource_group', 'myDisk')
async_snapshot_creation = self.compute_client.snapshots.create_or_update(
        'my_resource_group',
        'mySnapshot',
        {
            'location': 'westus',
            'creation_data': {
                'create_option': 'Copy',
                'source_uri': managed_disk.id
            }
        }
    )
snapshot = async_snapshot_creation.result()
```

## PowerBI Embedded Management

For general information on resource management, see *Resource Management*.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.powerbiembedded import PowerBIEmbeddedManagementClient
from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
    'user@domain.com',  # Your user
    'my_password',      # Your password
)

pbi_client = PowerBIEmbeddedManagementClient(
    credentials,
    subscription_id
)
```

### Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.PowerBI')
```

## ServerManager Management

For general information on resource management, see *Resource Management*.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.servermanager import ServerManagement
from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
    'user@domain.com',  # Your user
```

```
    'my_password',      # Your password
)

servermanager_client = ServerManagement(
    credentials,
    subscription_id
)
```

### Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.ServerManagement')
```

### Create gateway

```
gateway_async = servermanager_client.gateway.create(
    'MyResourceGroup',
    'MyGateway',
    'centralus'
)
gateway = gateway_async.result() # Blocking wait
```

## SQL Database Management

For general information on resource management, see *Resource Management*.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.sql import SqlManagementClient
from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
```

```
credentials = UserPassCredentials(
    'user@domain.com',  # Your user
    'my_password',      # Your password
)

sql_client = SqlManagementClient(
    credentials,
    subscription_id
)
```

### Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Sql')
```

### Servers/Databases sample

You can get a fully functional SQL servers/databases sample from the AzureSample Github repository: https://github.com/Azure-Samples/sql-database-python-manage

## Traffic Manager Management

For general information on resource management, see *Resource Management*.

### Create the management client

The following code creates an instance of the management client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.mgmt.trafficmanager import TrafficManagerManagementClient
from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
    'user@domain.com',  # Your user
    'my_password',      # Your password
```

```
)

tf_client = TrafficManagerManagementClient(
    credentials,
    subscription_id
)
```

### Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Network')
```

## Service Management

### Usage

#### Set-up certificates

You will need two certificates, one for the server (a .cer file) and one for the client (a .pem file).

#### Using the Azure .PublishSettings certificate

You can download your Azure publish settings file and use the certificate that is embedded in that file to create the client certificate. The server certificate already exists, so you won't need to upload one.

To do this, download your publish settings then use this code to create the .pem file.

```
from azure.servicemanagement import get_certificate_from_publish_settings

subscription_id = get_certificate_from_publish_settings(
    publish_settings_path='MyAccount.PublishSettings',
    path_to_write_certificate='mycert.pem',
    subscription_id='00000000-0000-0000-0000-000000000000',
)
```

The subscription id parameter is optional. If there are more than one subscription in the publish settings, the first one will be used.

#### Creating and uploading new certificate with OpenSSL

To create the .pem file using OpenSSL, execute this:

```
openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout mycert.pem -out mycert.pem
```

To create the .cer certificate, execute this:

```
openssl x509 -inform pem -in mycert.pem -outform der -out mycert.cer
```

After you have created the certificate, you will need to upload the .cer file to Microsoft Azure via the "Upload" action of the "Settings" tab of the management portal.

### Initialization

To initialize the management service, pass in your subscription id and the path to the .pem file.

```
from azure.servicemanagement import ServiceManagementService
subscription_id = '00000000-0000-0000-0000-000000000000'
cert_file = 'mycert.pem'
sms = ServiceManagementService(subscription_id, cert_file)
```

### List Available Locations

```
locations = sms.list_locations()
for location in locations:
    print(location.name)
```

### Create a Storage Service

To create a storage service, you need a name for the service (between 3 and 24 lowercase characters and unique within Microsoft Azure), a label (up to 100 characters, automatically encoded to base-64), and either a location or an affinity group.

```
name = "mystorageservice"
desc = name
label = name
location = 'West US'

result = sms.create_storage_account(name, desc, label, location=location)
```

### Create a Cloud Service

A cloud service is also known as a hosted service (from earlier versions of Microsoft Azure). The **create_hosted_service** method allows you to create a new hosted service by providing a hosted service name (which must be unique in Microsoft Azure), a label (automatically encoded to base-64), and the location *or* the affinity group for your service.

```
name = "myhostedservice"
desc = name
label = name
location = 'West US'

result = sms.create_hosted_service(name, label, desc, location=location)
```

### Create a Deployment

To make a new deployment to Azure you must store the package file in a Microsoft Azure Blob Storage account under the same subscription as the hosted service to which the package is being uploaded. You can create a deployment package with the Microsoft Azure PowerShell cmdlets, or with the cspack commandline tool.

```
service_name = "myhostedservice"
deployment_name = "v1"
slot = 'Production'
package_url = "URL_for_.cspkg_file"
configuration = base64.b64encode(open(file_path, 'rb').read('path_to_.cscfg_file'))
label = service_name

result = sms.create_deployment(service_name,
                      slot,
                      deployment_name,
                      package_url,
                      label,
                      configuration)

operation = sms.get_operation_status(result.request_id)
print('Operation status: ' + operation.status)
```

## Batch

For more information on the Azure Batch service, check out the Batch Documentation. For working samples, see the Batch samples repo.

### Create the Batch client

The following code creates an instance of the Batch client. The Batch client provides access to create pools, manage and schedule jobs, and access compute nodes.

A Batch Account that allows the Batch Service to allocate pools can be authenticated either via Shared Key authentication, or an Azure Active Directory token. Batch Accounts configured to allocate pools into the users subscription must be authenticated with an Azure Active Directory token.

For more information on creating and managing Batch accounts, including retrieving account URL and keys, and pool allocation modes, see the *Batch Management Client*.

#### Shared Key Authentication

```
from azure.batch import BatchServiceClient
from azure.batch.batch_auth import SharedKeyCredentials

credentials = SharedKeyCredentials(BATCH_ACCOUNT_NAME, BATCH_ACCOUNT_KEY)
batch_client = BatchServiceClient(
    credentials,
    base_url=BATCH_ACCOUNT_URL
)
```

### Azure Active Directory Authentication

```
from azure.batch import BatchServiceClient
from azure.common.credentials import ServicePrincipalCredentials

credentials = ServicePrincipalCredentials(
    client_id=CLIENT_ID,
    secret=SECRET,
    tenant=TENANT_ID,
    resource="https://batch.core.windows.net/"
)
    batch_client = BatchServiceClient(
    credentials,
    base_url=BATCH_ACCOUNT_URL
)
```

## Manage Pools and Nodes

The Batch Client allows you to create, modify, and delete Batch Pools. You can find more information on pools in this overview of Azure Batch features.

```
    # Create a new pool of Windows nodes from Cloud Services
    pool_config = batch.models.CloudServiceConfiguration(os_family='4')
    new_pool = batch.models.PoolAddParameter(
            'my_pool',
            'small',
            cloud_service_configuration=pool_config,
            target_dedicated=3
    )

    batch_client.pool.add(new_pool)

    # Create a new pool with Linux nodes using Azure Virtual Machines
    # Marketplace images. For full example, see the Batch samples repo.
    pool = batch.models.PoolAddParameter(
    id='my_pool',
    enable_inter_node_communication=True,
    virtual_machine_configuration=batch.models.VirtualMachineConfiguration(
        image_reference=MY_IMAGE_REF,
        node_agent_sku_id=MY_NODE_AGENT),
    vm_size='small',
    target_dedicated=6,
    start_task=batch.models.StartTask(
        command_line=STARTTASK_RESOURCE_FILE,
        run_elevated=True,
        wait_for_success=True,
        resource_files=[
            batch.models.ResourceFile(
                file_path=STARTTASK_RESOURCE_FILE, blob_source=SAS_URL)
        ]),
)
    batch_client.pool.add(new_pool)
```

Existing pools can be upgraded, patched, and resized. You can change the size of a pool either explicitly, or via an auto-scaling formula. For more information, see this article on automatically scaling nodes in a Batch pool.

```
# Resize an existing pool to a specific number of VMs
resize = batch.models.PoolResizeParameter(target_dedicated=5)
batch_client.pool.resize('my_pool', resize)

# Or set a formula to allow the pool to auto-scale
autoscale_interval = datetime.timedelta(minutes=10)
batch_client.pool.enable_auto_scale(
        'my_pool',
        auto_scale_formula='$TargetDedicated = (time().weekday==1?5:1);'
        auto_scale_evaluation_interval=autoscale_interval
)

# Update or patch a pool. Note that when updating, all pool parameters must be␣
↪updated,
# but when patching, individual parameters can be selectively updated.
updated_info=batch.models.PoolPatchPropertiesParameter(
        metadata=[batch.models.MetadataItem('foo', 'bar')]
)
batch_client.pool.patch('my_pool', updated_info)

# Upgrade pool OS
batch_client.pool.upgrade_os('my_pool', 'WA-GUEST-OS-4.28_201601-01')
```

You can monitor pools by retrieving data individually, or grouped using OData filters. You can learn more about filters with this article on querying the Batch service efficiently. You can also retrieve statistics on the usage of a specific pool, or all the pools in the lifetime of your Batch account.

```
if batch_client.pool.exists('my_pool'):
        my_pool = batch_client.pool.get('my_pool')
        print("Current state: {}".format(my_pool.allocation_state))

# List all pools in the Batch account
pools = batch_client.pool.list()
all_pools = [p.id for p in pools]

# Or retrieve just a selection of pools
options = batch.models.PoolListOptions(filter='startswith(id,\'my_\')')
my_pools = batch_client.pool.list(options)
only_my_pools = [p.id for p in my_pools]

stats = batch_client.pool.get_all_pools_lifetime_statistics()
print("Average CPU usage across pools: {}%".format(stats.resource_stats.avg_cpu_
↪percentage))
```

The Batch client also allows you to access individual nodes within a pool.

```
# List compute nodes in a pool, then remove any erroring ones
nodes = list(batch_client.compute_node.list())
errored = [n.id for n in nodes if n.state == batch.models.ComputeNodeState.unusable]
working_nodes = [n.id for n in nodes if n not in errored]
batch_client.pool.remove_nodes('my_pool', batch.models.NodeRemoveParameter(errored))

# Add a user account to a Windows Cloud Services node and retrieve an RDP file
user = batch.models.ComputeNodeUser('MyTestUser', password='kt#_gahr!@aGERDXA')
batch_client.compute_node.add_user('my_pool', working_nodes[0], user)
with open('node.rdp', 'w') as rdp_file:
        data = batch_client.compute_node.get_remote_desktop('my_pool', working_
↪nodes[0])
```

```
        for chunk in data:
                rdp_file.write(chunk)

# Add a user to a Linux node and retrieve login settings
# For full sample see the Batch samples repo
batch_client.compute_node.add_user(
        'my_pool',
        working_nodes[0],
        batch.models.ComputeNodeUser(
                'MyTestUser',
                is_admin=True,
                password=None,
                ssh_public_key=SSH_PUBLIC_KEY
)
login_details = batch_client.compute_node.get_remote_login_settings(
        'my_pool',
        working_nodes[0]
)
print("Remote IP: {}".format(login_details.remote_login_ip_address))
print("SSH Port: {}".format(login_details.remote_login_port))

# Reboot or reimage a node
batch_client.compute_node.reimage('my_pool', working_nodes[1])
batch_client.compute_node.reboot('my_pool', working_nodes[2])
```

## Manage Jobs and Tasks

You can create new jobs and add tasks, monitor existing jobs and download outputs. You can also set up job schedules for future or recurring jobs.

```
# Create Job
job = batch.models.JobAddParameter(
        'python_test_job',
        batch.models.PoolInformation(pool_id='my_pool')
)
batch_client.job.add(job)

# Add a task
task = batch.models.TaskAddParameter(
        'python_task_1',
        'cmd /c echo hello world'
)
batch_client.task.add('python_test_job', task)

# Add lots of tasks (up to 100 per call)
tasks = []
for i in range(2, 50):
        tasks.append(batch.models.TaskAddParameter(
                'python_task_{}'.format(i),
                'cmd /c echo hello world {}'.format(i))
        )
batch_client.task.add_collection('python_test_job', tasks)

# Download task output
with open('task_output.txt', 'w') as file_output:
        output = batch_client.file.get_from_task(
```

```
                'python_test_job',
                'python_task_1',
                'stdout.txt'
        )
        for data in output:
                file_output.write(data)

# Set up a schedule for a recurring job
job_spec = batch.models.JobSpecification(
        pool_info=batch.models.PoolInformation(pool_id='my_pool')
)
schedule = batch.models.Schedule(
        start_window=datetime.timedelta(hours=1),
        recurrance_interval=datetime.timedelta(days=1)
)
setup = batch.models.JobScheduleAddParameter(
        'python_test_schedule',
        schedule,
        job_spec
)
batch_client.job_schedule.add(setup)
```

# Azure Monitor

For general information on resource management, see *Resource Management*.

## Create the Monitor client

The following code creates an instance of the client.

You will need to provide your `subscription_id` which can be retrieved from your subscription list.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

```
from azure.monitor import MonitorClient
from azure.common.credentials import UserPassCredentials

# Replace this with your subscription id
subscription_id = '33333333-3333-3333-3333-333333333333'

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
    'user@domain.com',  # Your user
    'my_password',      # Your password
)

client = MonitorClient(
    credentials,
    subscription_id
)
```

## Registration

Some operations in the ARM APIs require a one-time registration of the provider with your subscription.

Use the following code to do the registration. You can use the same credentials you created in the previous section.

```
from azure.mgmt.resource.resources import ResourceManagementClient

resource_client = ResourceManagementClient(
    credentials,
    subscription_id
)
resource_client.providers.register('Microsoft.Insights')
```

You also might need to add the "Monitoring Contributor Service Role" role to your credentials. See here to do it using the Python CLI: https://docs.microsoft.com/cli/azure/role/assignment

## Get the Activity Log

This sample gets the logs from the ActivityLog of today for resource group ResourceGroupName, filtering the attributes to get only the "eventName" and "operationName".

A complete list of available keywords for filters and available attributes is available here: https://msdn.microsoft.com/library/azure/dn931934.aspx

```
import datetime

today = datetime.datetime.now().date()
filter = " and ".join([
    "eventTimestamp ge {}".format(today),
    "resourceGroupName eq 'ResourceGroupName'"
])
select = ",".join([
    "eventName",
    "operationName"
])

activity_logs = client.activity_logs.list(
    filter=filter,
    select=select
)
for log in activity_logs:
    # assert isinstance(log, azure.monitor.models.EventData)
    print(" ".join([
        log.event_name.localized_value,
        log.operation_name.localized_value
    ]))
```

## Metrics

This sample get the metrics of a resource on Azure (VMs, etc.).

A complete list of available keywords for filters is available here: https://msdn.microsoft.com/en-us/library/azure/mt743622.aspx

```
import datetime

# Get the ARM id of your resource. You might chose to do a "get"
# using the according management or to build the URL directly
# Example for a ARM VM
resource_id = (
    "subscriptions/{}/"
    "resourceGroups/{}/"
    "providers/Microsoft.Compute/virtualMachines/{}"
).format(subscription_id, resource_group_name, vm_name)

# You can get the available metrics of this specific resource
for metric in client.metric_definitions.list(resource_id):
    # azure.monitor.models.MetricDefinition
    print("{}: id={}, unit={}".format(
        metric.name.localized_value,
        metric.name.value,
        metric.unit
    ))

# Example of result for a VM:
# Percentage CPU: id=Percentage CPU, unit=Unit.percent
# Network In: id=Network In, unit=Unit.bytes
# Network Out: id=Network Out, unit=Unit.bytes
# Disk Read Bytes: id=Disk Read Bytes, unit=Unit.bytes
# Disk Write Bytes: id=Disk Write Bytes, unit=Unit.bytes
# Disk Read Operations/Sec: id=Disk Read Operations/Sec, unit=Unit.count_per_second
# Disk Write Operations/Sec: id=Disk Write Operations/Sec, unit=Unit.count_per_second

# Get CPU total of yesterday for this VM, by hour

today = datetime.datetime.now().date()
yesterday = today - datetime.timedelta(days=1)

filter = " and ".join([
    "name.value eq 'Percentage CPU'",
    "aggregationType eq 'Total'",
    "startTime eq {}".format(yesterday),
    "endTime eq {}".format(today),
    "timeGrain eq duration'PT1H'"
])

metrics_data = client.metrics.list(
    resource_id,
    filter=filter
)

for item in metrics_data:
    # azure.monitor.models.Metric
    print("{} ({})".format(item.name.localized_value, item.unit.name))
    for data in item.data:
        # azure.monitor.models.MetricData
        print("{}: {}".format(data.time_stamp, data.total))

# Example of result:
# Percentage CPU (percent)
# 2016-11-16 00:00:00+00:00: 72.0
# 2016-11-16 01:00:00+00:00: 90.59
```

```
# 2016-11-16 02:00:00+00:00: 60.58
# 2016-11-16 03:00:00+00:00: 65.78
# 2016-11-16 04:00:00+00:00: 43.96
# 2016-11-16 05:00:00+00:00: 43.96
# 2016-11-16 06:00:00+00:00: 114.9
# 2016-11-16 07:00:00+00:00: 45.4
# 2016-11-16 08:00:00+00:00: 57.59
# 2016-11-16 09:00:00+00:00: 67.85
# 2016-11-16 10:00:00+00:00: 76.36
# 2016-11-16 11:00:00+00:00: 87.41
# 2016-11-16 12:00:00+00:00: 67.53
# 2016-11-16 13:00:00+00:00: 64.78
# 2016-11-16 14:00:00+00:00: 66.55
# 2016-11-16 15:00:00+00:00: 69.82
# 2016-11-16 16:00:00+00:00: 96.02
# 2016-11-16 17:00:00+00:00: 272.52
# 2016-11-16 18:00:00+00:00: 96.41
# 2016-11-16 19:00:00+00:00: 83.92
# 2016-11-16 20:00:00+00:00: 95.57
# 2016-11-16 21:00:00+00:00: 146.73
# 2016-11-16 22:00:00+00:00: 73.86
# 2016-11-16 23:00:00+00:00: 84.7
```

# KeyVault

For general information on resource management, see *Resource Management*.

## Create the client

The following code creates an instance of the client.

See *Resource Management Authentication* for details on handling Azure Active Directory authentication with the Python SDK, and creating a `Credentials` instance.

**Important:** You must specify *resource="https://vault.azure.net"* while authenticating to get a valid token

```
from azure.keyvault import KeyVaultClient
from azure.common.credentials import UserPassCredentials

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
    'user@domain.com',  # Your user
    'my_password',      # Your password
    resource='https://vault.azure.net'
)

client = KeyVaultClient(
    credentials
)
```

## Access policies

Some operations require the correct access policies for your credentials.

If you get an "Unauthorized" error, please add the correct access policies to this credentials using the Azure Portal, the Azure CLI or the *Key Vault Management SDK itself*

## Example

`KEY_VAULT_URI` is the base url of your keyvault. Eg. *https://myvault.vault.azure.net*

```
# Create a key
key_bundle = client.create_key(KEY_VAULT_URI, 'FirstKey', 'RSA')
key_id = key_vault_id.parse_key_id(key_bundle.key.kid)

# Update a key without version
client.update_key(key_id.base_id, key_attributes={'enabled': False})

# Update a key with version
client.update_key(key_id.id, key_attributes={'enabled': False})

# Print a list of versions for a key
versions = client.get_key_versions(KEY_VAULT_URI, 'FirstKey')
for version in versions:
    print(version.kid)  # https://myvault.vault.azure.net/keys/FirstKey/
↪00010203040506070809 0a0b0c0d0e0f

# Read a key without version
client.get_key(key_id.base_id)

# Read a key with version
client.get_key(key_id.id)

# Delete a key
client.delete_key(KEY_VAULT_URI, 'FirstKey')


# Create a secret
secret_bundle = client.set_secret(KEY_VAULT_URI, 'FirstSecret', 'Hush, that is secret!
↪!')
secret_id = key_vault_id.parse_secret_id(secret_bundle.id)

# Update a secret without version
client.update_key(secret_id.base_id, secret_attributes={'enabled': False})

# Update a secret with version
client.update_key(secret_id.id, secret_attributes={'enabled': False})

# Print a list of versions for a secret
versions = client.get_secret_versions(KEY_VAULT_URI, 'FirstSecret')
for version in versions:
    print(version.id)  # https://myvault.vault.azure.net/secrets/FirstSecret/
↪00010203040506070809 0a0b0c0d0e0f

# Read a secret without version
client.get_secret(secret_id.base_id)
```

```
# Read a secret with version
client.get_secret(secret_id.id)

# Delete a secret
client.delete_secret(KEY_VAULT_URI, 'FirstSecret')
```

# Azure Active Directory Graph Rbac API

## Create the client

The following code creates an instance of the client.

See *Resource Management Authentication* for details on getting a `Credentials` instance.

You will also need the tenant id of the AD you want to manage. Could be the AD UUID or domain name. You can follow this documentation to get it.

**Note:** You need to change the *resource* parameter to *https://graph.windows.net* while creating the credentials instance

```
from azure.graphrbac import GraphRbacManagementClient
    from azure.common.credentials import UserPassCredentials

# See above for details on creating different types of AAD credentials
credentials = UserPassCredentials(
            'user@domain.com',      # Your user
            'my_password',          # Your password
            resource="https://graph.windows.net"
    )

tenant_id = "myad.onmicrosoft.com"

graphrbac_client = GraphRbacManagementClient(
    credentials,
    tenant_id
)
```

## Manage users

The following code creates a user, get it directly and by list filtering, and then delete it. Filter syntax can be found here.

```
from azure.graphrbac.models import UserCreateParameters,␣
↪UserCreateParametersPasswordProfile

user = graphrbac_client.user_operations.create(
    UserCreateParameters(
        user_principal_name="testbuddy@{}".format(MY_AD_DOMAIN),
        account_enabled=False,
        display_name='Test Buddy',
        mail_nickname='testbuddy',
        password_profile=UserCreateParametersPasswordProfile(
            password='MyStr0ngP4ssword',
```

```
            force_change_password_next_login=True
        )
    )
)
# user is a User instance
self.assertEqual(user.display_name, 'Test Buddy')

user = graphrbac_client.user.get(user.object_id)
self.assertEqual(user.display_name, 'Test Buddy')

for user in graphrbac_client.user_operations.list(filter="displayName eq 'Test Buddy'
↪"):
    self.assertEqual(user.display_name, 'Test Buddy')

graphrbac_client.user_operations.delete(user.object_id)
```

# Service Bus

## ServiceBus Queues

ServiceBus Queues are an alternative to Storage Queues that might be useful in scenarios where more advanced messaging features are needed (larger message sizes, message ordering, single-operaiton destructive reads, scheduled delivery) using push-style delivery (using long polling).

The service can use Shared Access Signature authentication, or ACS authentication.

Service bus namespaces created using the Azure portal after August 2014 no longer support ACS authentication. You can create ACS compatible namespaces with the Azure SDK.

### Shared Access Signature Authentication

To use Shared Access Signature authentication, create the service bus service with:

```
from azure.servicebus import ServiceBusService

key_name = 'RootManageSharedAccessKey' # SharedAccessKeyName from Azure portal
key_value = '' # SharedAccessKey from Azure portal
sbs = ServiceBusService(service_namespace,
                        shared_access_key_name=key_name,
                        shared_access_key_value=key_value)
```

### ACS Authentication

To use ACS authentication, create the service bus service with:

```
from azure.servicebus import ServiceBusService

account_key = '' # DEFAULT KEY from Azure portal
issuer = 'owner' # DEFAULT ISSUER from Azure portal
sbs = ServiceBusService(service_namespace,
                        account_key=account_key,
                        issuer=issuer)
```

### Sending and Receiving Messages

The **create_queue** method can be used to ensure a queue exists:

```
sbs.create_queue('taskqueue')
```

The **send_queue_message** method can then be called to insert the message into the queue:

```
from azure.servicebus import Message

msg = Message('Hello World!')
sbs.send_queue_message('taskqueue', msg)
```

The **send_queue_message_batch** method can then be called to send several messages at once:

```
from azure.servicebus import Message

msg1 = Message('Hello World!')
msg2 = Message('Hello World again!')
sbs.send_queue_message_batch('taskqueue', [msg1, msg2])
```

It is then possible to call the **receive_queue_message** method to dequeue the message.

```
msg = sbs.receive_queue_message('taskqueue')
```

## ServiceBus Topics

ServiceBus topics are an abstraction on top of ServiceBus Queues that make pub/sub scenarios easy to implement.

The **create_topic** method can be used to create a server-side topic:

```
sbs.create_topic('taskdiscussion')
```

The **send_topic_message** method can be used to send a message to a topic:

```
from azure.servicebus import Message

msg = Message(b'Hello World!')
sbs.send_topic_message('taskdiscussion', msg)
```

The **send_topic_message_batch** method can be used to send several messages at once:

```
from azure.servicebus import Message

msg1 = Message(b'Hello World!')
msg2 = Message(b'Hello World again!')
sbs.send_topic_message_batch('taskdiscussion', [msg1, msg2])
```

Please consider that in Python 3 a str message will be utf-8 encoded and you should have to manage your encoding yourself in Python 2.

A client can then create a subscription and start consuming messages by calling the **create_subscription** method followed by the **receive_subscription_message** method. Please note that any messages sent before the subscription is created will not be received.

```
from azure.servicebus import Message

sbs.create_subscription('taskdiscussion', 'client1')
msg = Message('Hello World!')
sbs.send_topic_message('taskdiscussion', msg)
msg = sbs.receive_subscription_message('taskdiscussion', 'client1')
```

## Event Hub

Event Hubs enable the collection of event streams at high throughput, from a diverse set of devices and services.

The **create_event_hub** method can be used to create an event hub:

```
sbs.create_event_hub('myhub')
```

To send an event:

```
sbs.send_event('myhub', '{ "DeviceId":"dev-01", "Temperature":"37.0" }')
```

The event content is the event message or JSON-encoded string that contains multiple messages.

## Advanced features

### Broker Properties and User Properties

This section describes how to use Broker and User properties defined here: https://docs.microsoft.com/rest/api/servicebus/message-headers-and-properties

```
sent_msg = Message(b'This is the third message',
                   broker_properties={'Label': 'M3'},
                   custom_properties={'Priority': 'Medium',
                                      'Customer': 'ABC'}
           )
```

You can use datetime, int, float or boolean

```
props = {'hello': 'world',
         'number': 42,
         'active': True,
         'deceased': False,
         'large': 8555111000,
         'floating': 3.14,
         'dob': datetime(2011, 12, 14),
         'double_quote_message': 'This "should" work fine',
         'quote_message': "This 'should' work fine"}
sent_msg = Message(b'message with properties', custom_properties=props)
```

For compatibility reason with old version of this library, *broker_properties* could also be defined as a JSON string. If this situation, you're responsible to write a valid JSON string, no check will be made by Python before sending to the RestAPI.

```
broker_properties = '{"ForcePersistence": false, "Label": "My label"}'
sent_msg = Message(b'receive message',
                   broker_properties = broker_properties
)
```

# azure.common package

## Submodules

## Module contents

# azure.batch package

## Submodules

### azure.batch.models module

### azure.batch.operations module

## Module contents

# azure.graphrbac package

## Submodules

### azure.graphrbac.models module

### azure.graphrbac.operations module

## Module contents

# azure.keyvault package

## Submodules

### azure.keyvault.key_vault_id module

### Module contents

### azure.keyvault.generated package

### Submodules

### azure.keyvault.generated.models module

### Module contents

## azure.mgmt.authorization package

### Submodules



### Module contents

## azure.mgmt.batch package

### Submodules



### Module contents

## azure.mgmt.cdn package

### Submodules



### Module contents

## azure.mgmt.cognitiveservices package

### Submodules

#### azure.mgmt.cognitiveservices.operations module

### Module contents

## azure.mgmt.commerce package

### Submodules

#### azure.mgmt.commerce.models module

#### azure.mgmt.commerce.operations module

### Module contents

## azure.mgmt.compute.compute package

### Module contents

### Submodules

#### azure.mgmt.compute.compute.v2016_04_30_preview package

##### Submodules

#### azure.mgmt.compute.compute.v2016_04_30_preview.models module

#### azure.mgmt.compute.compute.v2016_04_30_preview.operations module

##### Module contents

#### azure.mgmt.compute.compute.v2015_06_15 package

##### Submodules

#### azure.mgmt.compute.compute.v2015_06_15.models module

#### azure.mgmt.compute.compute.v2015_06_15.operations module

##### Module contents

## azure.mgmt.compute.containerservice package

### Module contents

### Submodules

# azure.mgmt.containerregistry package

## Submodules



## Module contents

# azure.mgmt.datalake.analytics.account package

## Submodules



## Module contents

# azure.mgmt.datalake.analytics.catalog package

## Submodules



## Module contents

# azure.mgmt.datalake.analytics.job package

## Submodules

azure.mgmt.datalake.analytics.job.operations module

### Module contents

## azure.mgmt.datalake.store package

### Submodules

azure.mgmt.datalake.store.models module

azure.mgmt.datalake.store.operations module

### Module contents

## azure.mgmt.devtestlabs package

### Submodules

azure.mgmt.devtestlabs.models module

azure.mgmt.devtestlabs.operations module

### Module contents

## azure.mgmt.dns package

### Submodules

azure.mgmt.dns.models module

azure.mgmt.dns.operations module

### Module contents

## azure.mgmt.documentdb package

### Submodules

azure.mgmt.documentdb.models module

azure.mgmt.documentdb.operations module

### Module contents

## azure.mgmt.eventhub package

### Submodules

azure.mgmt.eventhub.models module

azure.mgmt.eventhub.operations module

### Module contents

## azure.mgmt.iothub package

### Submodules

azure.mgmt.iothub.models module

azure.mgmt.iothub.operations module

### Module contents

## azure.mgmt.keyvault package

### Submodules

azure.mgmt.keyvault.models module

azure.mgmt.keyvault.operations module

### Module contents

## azure.mgmt.logic package

### Submodules

azure.mgmt.logic.models module

#### azure.mgmt.logic.operations module

### Module contents

## azure.mgmt.media package

### Submodules

#### azure.mgmt.media.models module

#### azure.mgmt.media.operations module

### Module contents

## azure.mgmt.monitor package

### Submodules

#### azure.mgmt.monitor.models module

#### azure.mgmt.monitor.operations module

### Module contents

## azure.mgmt.network package

### Module contents

### Submodules

#### azure.mgmt.network.v2017_03_01 package

##### Submodules

#### azure.mgmt.network.v2017_03_01.models module

#### azure.mgmt.network.v2017_03_01.operations module

##### Module contents

#### azure.mgmt.network.v2016_12_01 package

##### Submodules

#### azure.mgmt.network.v2016_12_01.models module

## azure.mgmt.notificationhubs package

### Submodules



### Module contents

## azure.mgmt.powerbiembedded package

### Submodules

#### azure.mgmt.powerbiembedded.operations module

### Module contents

## azure.mgmt.redis package

### Submodules

#### azure.mgmt.redis.models module

#### azure.mgmt.redis.operations module

### Module contents

## azure.mgmt.resource.features package

### Module contents

### Submodules

#### azure.mgmt.resource.features.v2015_12_01 package

##### Submodules

#### azure.mgmt.resource.features.v2015_12_01.models module

#### azure.mgmt.resource.features.v2015_12_01.operations module

##### Module contents

## azure.mgmt.resource.links package

### Module contents

### Submodules

#### azure.mgmt.resource.links.v2016_09_01 package

##### Submodules

#### azure.mgmt.resource.links.v2016_09_01.models module

#### azure.mgmt.resource.links.v2016_09_01.operations module

#### Module contents

## azure.mgmt.resource.locks package

### Module contents

### Submodules

#### azure.mgmt.resource.locks.v2016_09_01 package

#### Submodules

#### azure.mgmt.resource.locks.v2016_09_01.models module

#### azure.mgmt.resource.locks.v2016_09_01.operations module

#### Module contents

#### azure.mgmt.resource.locks.v2015_01_01 package

#### Submodules

#### azure.mgmt.resource.locks.v2015_01_01.models module

#### azure.mgmt.resource.locks.v2015_01_01.operations module

#### Module contents

## azure.mgmt.resource.policy package

### Module contents

### Submodules

#### azure.mgmt.resource.policy.v2016_12_01 package

#### Submodules

#### azure.mgmt.resource.policy.v2016_12_01.models module

#### azure.mgmt.resource.policy.v2016_12_01.operations module

#### Module contents

#### azure.mgmt.resource.policy.v2016_04_01 package

#### Submodules

#### azure.mgmt.resource.policy.v2016_04_01.models module

#### azure.mgmt.resource.policy.v2016_04_01.operations module

#### Module contents

## azure.mgmt.resource.resources package

### Module contents

### Submodules

#### azure.mgmt.resource.resources.v2016_09_01 package

#### Submodules

#### azure.mgmt.resource.resources.v2016_09_01.models module

#### azure.mgmt.resource.resources.v2016_09_01.operations module

#### Module contents

#### azure.mgmt.resource.resources.v2016_02_01 package

#### Submodules

#### azure.mgmt.resource.resources.v2016_02_01.models module

#### azure.mgmt.resource.resources.v2016_02_01.operations module

#### Module contents

## azure.mgmt.resource.subscriptions package

### Module contents

### Submodules

#### azure.mgmt.resource.subscriptions.v2016_06_01 package

#### Submodules

#### azure.mgmt.resource.subscriptions.v2016_06_01.models module

azure.mgmt.resource.subscriptions.v2016_06_01.operations module

Module contents

# azure.mgmt.scheduler package

## Submodules

azure.mgmt.scheduler.models module

azure.mgmt.scheduler.operations module

## Module contents

# azure.mgmt.search package

## Submodules

azure.mgmt.search.models module

azure.mgmt.search.operations module

## Module contents

# azure.mgmt.servermanager package

## Submodules

azure.mgmt.servermanager.models module

azure.mgmt.servermanager.operations module

## Module contents

# azure.mgmt.servicebus package

## Submodules

azure.mgmt.servicebus.models module

#### azure.mgmt.servicebus.operations module

### Module contents

## azure.mgmt.sql package

### Submodules

#### azure.mgmt.sql.models module

#### azure.mgmt.sql.operations module

### Module contents

## azure.mgmt.storage package

### Module contents

### Submodules

#### azure.mgmt.storage.v2016_12_01 package

##### Submodules

#### azure.mgmt.storage.v2016_12_01.models module

#### azure.mgmt.storage.v2016_12_01.operations module

##### Module contents

#### azure.mgmt.storage.v2015_06_15 package

##### Submodules

#### azure.mgmt.storage.v2015_06_15.models module

#### azure.mgmt.storage.v2015_06_15.operations module

##### Module contents

## azure.mgmt.trafficmanager package

### Submodules

#### azure.mgmt.trafficmanager.models module

azure.mgmt.trafficmanager.operations module

### Module contents

## azure.mgmt.web package

### Submodules

azure.mgmt.web.models module

azure.mgmt.web.operations module

### Module contents

## azure.monitor package

### Submodules

azure.monitor.models module

azure.monitor.operations module

### Module contents

## azure.servicebus package

### Submodules

azure.servicebus.constants module

azure.servicebus.models module

azure.servicebus.servicebusservice module

### Module contents

## azure.servicemanagement package

### Submodules

azure.servicemanagement.constants module

azure.servicemanagement.models module

azure.servicemanagement.publishsettings module

## Module contents