Alpine SDK Documentation

Release 1

Names

Sep 16, 2019

Contents

Welcome to the official Python library for the Alpine API. In this first release we've focused on a subset of the full API that we feel users will most frequently use.

This library can be used to automate, add, or simplify functionality of Alpine.

Documentation (and examples): http://python-alpine-api.readthedocs.io/

Source code: https://github.com/AlpineNow/python-alpine-api

Python Package Index: https://pypi.python.org/pypi/alpine

Setup:

```
pip install alpine
```

Requirements: Using this package requires access to a TIBCO Team Studio instance. For more information, see the TIBCO Team Studio homepage: https://community.tibco.com/products/tibco-data-science

License: We use the MIT license. See the LICENSE file on GitHub for details.

CHAPTER 1

Example

Running a workflow and downloading the results:

```
>>> import alpine as AlpineAPI
>>> session = AlpineAPI.APIClient(host, port, username, password)
>>> process_id = session.workfile.process.run(workfile_id)
>>> session.workfile.process.wait_until_finished(workfile_id, process_id)
>>> results = session.workfile.process.download_results(workfile_id, process_id)
```

Contents

1.1 API Reference

1.1.1 Alpine

class alpine.apiclient.**APIClient**(*host=None*, *port=None*, *username=None*, *password=None*, *is_secure=False*, *validate_certs=False*, *ca_certs=None*, *token=None*, *logging_level='WARN'*)

The main entry point for the Alpine API. Most of the functions require a logged-in user. Begin a session by creating an instance of the *APIClient* class.

Example:

```
>>> import alpine as AlpineAPI
>>> session = alpine.APIClient(host, port, username, password)
```

get_license()

Get the the current license information for Alpine.

Returns Summary of Alpine license information - expiration, user limits, add-ons, etc.

Return type dict

Example:

```
>>> license_info = session.get_license()
```

get_status()
Returns information about the currently logged-in user. Or, if no user is logged in, returns an empty dict.

Returns Logged-in user's metadata.

Return type dict

Example:

```
>>> session.get_status()
{u'admin': True,
 u'auth_method': u'internal',
 u'dept': u'Development',
 u'email': u'demoadmin@alpinenow.com',
 u'entity_type': u'user',
 u'first_name': u'Demo',
 u'id': 665,
 u'image': {u'complete_json': True,
  u'entity_type': u'image',
  u'icon': u'/users/665/image?style=icon&1483606634',
  u'original': u'/users/665/image?style=original&1483606634'},
 u'is_deleted': None,
 u'last_name': u'Admin',
 u'ldap_group_id': None,
 u'notes': u'',
 u'roles': [u'admin'],
 u'subscribed_to_emails': True,
 u'tags': [],
 u'title': u'Assistant to the Regional Manager',
 u'user_type': u'analytics_developer',
 u'username': u'demoadmin',
 u'using_default_image': True}
```

get_version()
Returns the Alpine version.

Returns Alpine version.

Return type str

Example:

```
>>> session.get_version()
'6.2.0.0.1-b8c02ca46'
```

login(*username*, *password*)
Attempts to log in to Alpine with provided username and password. Typically login is handled at session creation time.

Parameters

- **username** (*str*) – Username to log in with.
- **password** (*str*) – Password to log in with.

Returns Logged-in user's metadata.

Return type dict

Example:

```
>>> user_info = session.login(username, password)
```

logout()
Attempts to log out the current user.

Returns Request response.

Return type requests.models.Response

Example:

```
>>> session.logout()
<Response [200]>
```

1.1.2 User

class alpine.user.**User**(*base_url*, *session*, *token*)
A class for interacting with user accounts.

class AdminRole
Convenience strings for administrator roles.

class ApplicationRole
Convenience strings for application roles.

create(*username*, *password*, *first_name*, *last_name*, *email*, *title=None*, *dept=None*, *notes=None*, *admin_role=None*, *app_role=None*, *email_notification=False*)
Create a user account with the specified parameters.

Parameters

- **username** (*str*) – A unique name.
- **password** (*str*) – Password of the user.
- **first_name** (*str*) – First name of the user.
- **last_name** (*str*) – Last name of the user.
- **email** (*str*) – Email of the user.
- **title** (*str*) – Title of the user.
- **dept** (*str*) – Department of the user.
- **notes** (*str*) – Notes for the user.
- **admin_role** (*str*) – Administration role. Refer to *User.AdminRole*. By default, the user is not an Admin.
- **app_role** (*str*) – Application role. Refer to *User.ApplicationRole*. The default application role is *User.ApplicationRole.BusinessUser*.
- **email_notification** (*bool*) – Option to subscribe to email notifications.

Returns Created user information or error message.

Return type dict

Example:

```
>>> user_info = session.create(username = 'demo_user', password = 'temp_
↪password',
>>>                           first_name = 'Demo', last_name = 'User',
>>>                           email = 'demouser@alpinenow.com', title =
↪'Data Scientist',
>>>                           dept = 'Product')
```

delete (*user_id*)

Attempts to delete the given user. Will fail if the user does not exist.

Parameters **user_id** (*int*) – ID of user to be deleted.

Returns None.

Return type NoneType

Raises **UserNotFoundException** – The username does not exist.

Example:

```
>>> session.user.delete(user_id = 51)
```

get (*user_id*)

Get a user's metadata.

Parameters **user_id** (*str*) – A unique user ID.

Returns Selected user's metadata.

Return type dict

Raises **UserNotFoundException** – The user does not exist.

Example:

```
>>> session.user.get(user_id = 51)
```

get_id (*username*)

Gets the ID of the user. Will throw an exception if the user does not exist.

Parameters **username** (*str*) – Unique user name

Returns ID of the user.

Return type int

Raises **UserNotFoundException** – The username does not exist.

Example

```
>>> user_id = session.user.get_id(username = 'demo_user')
>>> print(user_id)
51
```

get_list (*per_page=100*)

Get a list of all users' metadata.

Parameters **per_page** (*int*) – Maximum number to fetch with each API call.

Returns A list of all the users' data.

Return type list of dict

Example:

```
>>> all_users = session.user.get_list()
>>> len(all_users)
99
```

update(*user_id*, *first_name=None*, *last_name=None*, *email=None*, *title=None*, *dept=None*, *notes=None*, *admin_role=None*, *app_role=None*, *email_notification=None*)

Only included fields will be updated.

Parameters

- **user_id** (*str*) – ID of the user to update.
- **first_name** (*str*) – New first name of the user.
- **last_name** (*str*) – New last name of the user.
- **email** (*str*) – New email of the user.
- **title** (*str*) – New title of the user.
- **dept** (*str*) – New department of the user.
- **notes** (*str*) – New notes for the user.
- **admin_role** (*str*) – New Administration Role. Refer to *User.AdminRole*.
- **app_role** (*str*) – New Application Role. Refer to *User.ApplicationRole*.
- **email_notification** (*bool*) – Change option to subscribe to email notifications.

Returns Updated user information.

Return type dict

Raises **UserNotFoundException** – The user does not exist.

Example:

```
>>> updated_info = session.user.update(user_id = 51, title = "Senior Data␣
↪Scientist")
```

1.1.3 Workspace

class alpine.workspace.**Workspace**(*base_url*, *session*, *token*)

A class for interacting with workspaces. The top-level methods deal with workspace properties. The subclass *Member* can be used to interact with member lists.

class Member(*base_url*, *session*, *token*)

A class for interacting with workspace membership.

add(*workspace_id*, *user_id*, *role=None*)

Adds a new user to the workspace member list.

Parameters

- **workspace_id** (*int*) – ID of the workspace.
- **user_id** (*int*) – ID of member to add to the workspace.
- **role** (*str*) – Role for the user. Use *Workspace.MemberRole* for convenience.

Returns Updated member list.

Return type list of dict

Raises

- **WorkspaceNotFoundException** – The workspace does not exist.
- **UserNotFoundException** – The user does not exist.

Example:

```
>>> session.workspace.member.add(workspace_id = 1672, user_id = 7,
>>>                              role = session.workspace.memberRole.
↪DataScientist)
```

get_list (*workspace_id*, *per_page=100*)

Gets metadata about all the users who are members of the workspace.

Parameters

- **workspace_id** (*str*) – ID of the workspace.
- **int per_page** (*int*) – Maximum number to fetch with each API call.

Returns A list of user data.

Return type list of dict

Raises **WorkspaceNotFoundException** – The workspace does not exist.

Example:

```
>>> session.workspace.member.get_list(workspace_id = 1672)
```

remove (*workspace_id*, *user_id*)

Removes a user from the workspace member list.

Parameters

- **workspace_id** (*int*) – ID of the workspace.
- **user_id** (*int*) – ID of member to add to the workspace.

Returns Updated member list.

Return type list of dict

Raises

- **WorkspaceNotFoundException** – The workspace does not exist.
- **UserNotFoundException** – The user does not exist.

Example:

```
>>> session.workspace.member.remove(workspace_id = 1672, user_id = 7)
```

update_role (*workspace_id*, *user_id*, *new_role*)

Updates a user's role in a workspace. If the user is not a member of the workspace, then no change will be made.

Parameters

- **workspace_id** (*int*) – ID of the workspace.
- **user_id** (*int*) – ID of member to update.
- **new_role** (*str*) – New role for the user. Use *Workspace.MemberRole* for convenience.

Returns Updated member list.

Return type list of dict

Raises **WorkspaceNotFoundException** – The workspace does not exist.

Example:

```
>>> session.workspace.member.update(workspace_id = 1672, user_id = 7,
>>>                                 new_role = session.workspace.
↪memberRole.DataScientist)
```

class MemberRole

Convenience strings for user workspace member roles.

class Stage

Convenience IDs for workspace stages.

create (*workspace_name*, *public=False*, *summary=None*)

Creates a workspace. Will fail if the workspace name already exists.

Parameters

- **workspace_name** (*str*) – Unique workspace name.
- **public** (*bool*) – Allow the workspace to be viewable by non-members and non-admins.
- **summary** (*str*) – Description of new workspace.

Returns Created workspace information or error message.

Return type dict

Example:

```
>>> session.workspace.create(workspace_name = "Public Data ETL", public =␣
↪True)
```

delete (*workspace_id*)

Attempts to delete the given workspace. Will fail if the workspace does not exist.

Parameters **workspace_id** (*str*) – ID of the workspace to be deleted.

Returns None.

Return type NoneType

Raises **WorkspaceNotFoundException** – The workspace does not exist.

Example:

```
>>> session.workspace.delete(workspace_id = 1671)
```

get (*workspace_id*)

Gets a workspace's metadata.

Parameters **workspace_id** (*str*) – Unique workspace name.

Returns Selected workspace's data

Return type dict

Raises **WorkspaceNotFoundException** – The workspace does not exist.

Example:

```
>>> session.workspace.get(workspace_id = 1761)
```

get_id (*workspace_name*, *user_id=None*)

Get the ID of the workspace. Will throw an exception if the workspace does not exist.

Parameters

- **workspace_name** (*str*) – Unique workspace name.
- **user_id** (*int*) – ID of a user.

Returns ID of the workspace.

Return type int

Raises **WorkspaceNotFoundException** – The workspace does not exist.

Example:

```
>>> session.workspace.get_id("Public Data ETL")
1672
```

get_list(*user_id=None*, *active=None*, *per_page=50*)
Gets a list of metadata for each workspace. If a user ID is provided, only workspaces that the user is a member of will be returned.

Parameters

- **user_id** (*str*) – ID of the user.
- **active** (*bool*) – Return only active workspaces (optional). True will only return the active spaces.
- **per_page** (*int*) – Maximum number to fetch with each API call.

Returns List of workspace metadata.

Return type list of dict

Raises **UserNotFoundException** – The user does not exist.

Example:

```
>>> my_workspaces = session.workspace.get_list(user_id = my_user_id)
>>> len(my_workspaces)
8
```

update(*workspace_id*, *is_public=None*, *is_active=None*, *name=None*, *summary=None*, *stage=None*, *owner_id=None*)
Update a workspace's metadata. Only included fields will be changed.

Parameters

- **workspace_id** (*int*) – ID of the workspace.
- **is_public** (*bool*) – Allow the workspace to be viewable by non-members and non-admins.
- **is_active** (*bool*) – Set active vs. archived status.
- **name** (*str*) – New name for the workspace.
- **summary** (*str*) – New description of the workspace.
- **stage** (*int*) – Stage ID. Use the *Workspace.Stage* object for convenience.
- **owner_id** (*int*) – ID of the new workspace owner. This owner must also be a member of the workspace.

Returns Updated workspace metadata.

Return type dict

Example:

```
>>> session.workspace.update(workspace_id = 1672, summary = "New focus of␣
↪project is ML!",
>>>                          stage = session.workspace.stage.Model)
```

1.1.4 Workfile

class alpine.workfile.**Workfile**(*base_url*, *session*, *token*)
A class for interacting with workfiles. The top-level methods deal with workfile management. The subclass *Process* can be used to interact with individual workfiles, including running workflows with workflow variables.

class Process(*base_url*, *session*, *token*)

A class for interacting with workfiles.

download_results(*workflow_id*, *process_id*)

Downloads a workflow run result.

Parameters

- **workflow_id** (*int*) – ID of the workflow.
- **process_id** (*str*) – ID of a particular workflow run.

Returns JSON object of workflow results.

Return type dict

Raises

- **WorkspaceNotFoundException** – The workspace does not exist.
- **WorkfileNotFoundException** – The workflow does not exist.
- **ResultsNotFoundException** – Results not found or does not match expected structure.

Example:

```
>>> downloaded_flow_results = session.workfile.process.download_
↪results(workflow_id = 375,
>>>                                                              ␣
↪process_id = process_id)
```

static find_operator(*operator_name*, *flow_results*)

Helper method to parse a downloaded workflow result to extract data for a single operator.

Parameters

- **operator_name** (*str*) – Operator name to extract. Must be an exact match to the name in the workflow.
- **flow_results** (*dict*) – JSON object of Alpine flow results from download_results.

Returns Single operator data.

Return type dict

Raises **FlowResultsMalformedException** – Workflow result does not contain the key [‘outputs’].

Example:

```
>>> operator_data = session.workfile.process.find_operator(operator_name␣
↪= 'Row Filter',
>>>                                                        flow_results =␣
↪downloaded_flow_results)
```

static get_metadata(*flow_results*)

Returns the metadata for a particular workflow run including time, number of operators, user, and number of runtime errors.

Parameters **flow_results** (*dict*) – JSON object of Alpine flow results from download_results.

Returns Run metadata.

Return type dict

Raises **FlowResultsMalformedException** – Workflow result does not contain the key [‘flowMetaInfo’].

Example:

```
>>> session.workfile.process.get_metadata(flow_results = downloaded_flow_
↪results)
```

query_status(*process_id*)

Returns the status of a running workflow.

Parameters **process_id** (*str*) – ID of a particular workflow run.

Returns State of workflow run. One of 'WORKING', 'FINISHED', or 'FAILED'.
Return type str
Raises **RunFlowFailureException** – Process ID not found.
Example:

```
>>> session.workfile.process.query_status(process_id = process_id)
```

run(*workflow_id*, *variables=None*)
Run a workflow, optionally including a list of workflow variables. Returns a process_id which is needed by other functions which query a run or download results.
Parameters
- **workflow_id** (*str*) – ID of the workflow.
- **variables** (*list*) – A list of workflow variables with the format: [{"name": "wfv_name_1", "value": "wfv_value_1"}, {"name": "wfv_name_2", "value": "wfv_value_2"}]

Returns ID for the workflow run process.
Return type str
Raises
- **WorkspaceNotFoundException** – The workspace does not exist.
- **WorkfileNotFoundException** – The workfile does not exist.

Example:

```
>>> work_flow_variables = [{"name": "@row_filter", "value": "13"}]
>>> process_id = session.workfile.process.run(workflow_id = 375,␣
↪variables = work_flow_variables)
```

stop(*process_id*)
Attempts to stop a running workflow.
Parameters **process_id** (*str*) – Process ID of the workflow.
Returns Flow status. One of 'STOPPED' or 'STOP FAILED'.
Return type str
Raises **StopFlowFailureException** – Workflow run not found.
Example:

```
>>> session.workfile.process.stop(process_id = process_id)
```

wait_until_finished(*workflow_id*, *process_id*, *verbose=False*, *query_time=10*, *timeout=3600*)
Waits for a running workflow to finish.
Parameters
- **workflow_id** (*int*) – ID of a particular workflow run.
- **process_id** (*str*) – ID of a particular workflow run.
- **verbose** (*bool*) – Optionally print approximate run time.
- **query_time** (*float*) – Number of seconds between status queries. Minimum of 1 second.
- **timeout** (*float*) – Amount of time in seconds to wait for workflow to finish. Will stop if exceeded.

Returns Workflow run status.
Return type str
Raises
- **RunFlowTimeoutException** – Workflow runtime has exceeded timeout.
- **RunFlowFailureException** – Status of FAILURE is detected.

Example:

```
>>> session.workfile.process.wait_until_finished(workflow_id = workflow_
↪id, process_id = process_id)
```

delete(*workfile_id*)

Deletes a workfile from a workspace.

Parameters **workfile_id** (*int*) – ID of workfile to delete.

Returns None.

Return type NoneType

Raises

- **WorkspaceNotFoundException** – The workspace does not exist.
- **WorkfileNotFoundException** – The workfile does not exist.

Example:

```
>>> session.workfile.delete(workflow_id = 375)
```

download(*workfile_id*)

Download an Alpine workfile. Will not download Alpine workflows.

Parameters **workfile_id** (*int*) – ID of the workfile to download.

Returns file

Return type file

Raises **WorkfileNotFoundException** – Workfile ID does not exist or is an Alpine workflow.

Example:

```
>>> operator_data = session.workfile.download(workfile_id = 1351)
```

get(*workfile_id*)

Returns metadata for a workfile.

Parameters **workfile_id** (*str*) – ID of workfile.

Returns Selected workfile's metadata.

Return type dict

Raises

- **WorkspaceNotFoundException** – The workspace does not exist.
- **WorkfileNotFoundException** – The workfile does not exist.

Example:

```
>>> session.workfile.get(workflow_id = 375)
```

get_id(*workfile_name*, *workspace_id*)

Returns the ID of a workfile in a workspace.

Parameters

- **workfile_name** (*str*) – Name of the workfile.
- **workspace_id** (*int*) – ID of the workspace that contains the workfile.

Returns ID of the workfile.

Return type int

Raises

- **WorkspaceNotFoundException** – The workspace does not exist.
- **WorkfileNotFoundException** – The workfile does not exist.

Example:

```
>>> session.workspace.get_id(workfile_name = "WineData", workspace_id =
↪"APITests")
```

get_list (*workspace_id*, *per_page=100*)

Returns all workfiles in a workspace.

Parameters

- **workspace_id** (*int*) – ID of the workspace.
- **per_page** (*int*) – Maximum number to fetch with each API call.

Returns List of workfiles' metadata.

Return type list of dict

Raises **WorkspaceNotFoundException** – The workspace does not exist.

Example:

```
>>> session.workfile.get_list(workspace_id = 1672)
```

upload (*workspace_id*, *afm_file*, *data_sources_list*)

Uploads an Alpine workfile file (.afm format). Will alter the workfile to use the data source(s) chosen. Operators within a workflow must remain consistent with type of data source. A workflow built on a Hadoop data source can be converted to use a different Hadoop data source, but not to use a database.

Parameters

- **workspace_id** (*int*) – ID of workspace.
- **afm_file** (*str*) – Local path to the Alpine workfile (.afm).
- **data_sources_list** (*list*) – A list of data source information with the following format: datasource_info = [{"data_source_type": DataSource.dsType.HadoopCluster, "data_source_id": "1", "database_id":""}, {"data_source_type": DataSource.dsType.JDBCDataSource, "data_source_id": "421", "database_id": ""}, {"data_source_type": DataSource.dsType.GreenplumDatabase, "data_source_id": "1", "database_id": "42"}]

Returns Selected workfile's metadata.

Return type dict

Example:

```
>>> base_dir = os.getcwd()
>>> afm_path = "{0}/afm/test.afm".format(base_dir)
>>> datasource_info = [{"data_source_type": session.datasource.dsType.
↪GreenplumDatabase,
>>>                     "data_source_id": 1,
```

(continues on next page)

(continued from previous page)

```
>>>                     "database_id": 42}]
>>> workfile_info = session.workfile.upload(workspace_id, afm_path,␣
↪datasource_info)
```

1.1.5 Job

class alpine.job.**Job**(*base_url*, *session*, *token*)

A class for interacting with jobs. The top-level methods deal with jobs. The subclass *Task* can be used to interact with individual tasks within a job.

class ScheduleType

Convenience strings for schedule types.

class Task(*base_url*, *session*, *token*)

A class for interacting with job tasks.

create(*workspace_id*, *job_id*, *workfile_id*, *task_type=None*)

Add a new task to an existing job using an existing workfile.

Parameters

- **workspace_id** (*int*) – ID of the workspace.
- **job_id** (*int*) – ID of the job to which the task is to be added.
- **workfile_id** (*int*) – ID of the workfile to be added as a task.
- **task_type** (*str*) – Task type. Use the *Workspace.Stage* object for convenience. The default is "run_work_flow".

Returns Metadata of the new task.

Return type dict

Example:

```
>>> session.job.task.create(workspace_id = 1672, job_id = 675, workfile_
↪id = 823)
```

delete(*workspace_id*, *job_id*, *task_id*)

Delete a task from a job.

Parameters

- **workspace_id** (*int*) – ID of the workspace.
- **job_id** (*int*) – ID of the job that has the task to be deleted.
- **task_id** (*int*) – ID of the task.

Returns None

Return type NoneType

Raises

- **TaskNotFoundException** – The job does not exist.
- **InvalidResponseCodeException** – The request got an unexpected HTTP status code in response (not 200 OK).

Example:

```
>>> session.job.task.delete(workspace_id = 1672, job_id = 675, task_id =␣
↪344)
```

get(*workspace_id*, *job_id*, *task_id*)

Return metadata of one task.

Parameters

- **workspace_id** (*int*) – ID of the workspace.
- **job_id** (*int*) – ID of the job.
- **task_id** (*int*) – ID of the task.

Returns Selected task's metadata.
Return type dict

Example:

```
>>> session.job.task.get(workspace_id = 1672, job_id = 675, task_id = 344)
```

get_id(*workspace_id*, *job_id*, *task_name*)

Return the ID of a task.

Parameters

- **workspace_id** (*int*) – ID of the workspace.
- **job_id** (*int*) – ID of the job.
- **task_name** (*str*) – Name of the task.

Returns ID of the task.
Return type int

Example:

```
>>> session.job.task.get_id(workspace_id = 1672, job_id = 675, task_name␣
↪= "Run test2")
344
```

get_list(*workspace_id*, *job_id*)

Get a list of all tasks in a job.

Parameters

- **workspace_id** (*int*) – ID of the workspace.
- **job_id** (*int*) – ID of the job.

Returns List of all tasks in the job.
Return type list of dict

Example:

```
>>> session.job.task.get_list(workspace_id = 1672, job_id = 675);
```

class TaskType

Convenience strings for task types.

create(*workspace_id*, *job_name*, *schedule_type=None*, *interval_value=0*, *next_run=None*, *time_zone=None*)

Create a new job in a workspace with specified configuration.

Parameters

- **workspace_id** (*int*) – ID of the workspace where the job is to be created.
- **job_name** (*str*) – Name of the job to be created.
- **schedule_type** (*str*) – Job run interval time unit. Use the *Job.ScheduleType* object for convenience. The default value is "on_demand".
- **interval_value** (*int*) – Job run interval value. If you choose 'Job.ScheduleType.Weekly' for *schedule_type* and '2' for *interval_value*, then it will run every 2 weeks.
- **next_run** (*datetime*) – When the next run should happen.
- **time_zone** (*timezone*) – Time zone info. If no time zone is provided, we use UTC.

Returns Created job metadata.

Return type dict

Example:

```
>>> session.job.create(workspace_id = 1672, job_name = "APICreatedJob",
>>>                    schedule_type = Job.ScheduleType.Weekly, interval_
↪value = 2,
>>>                    next_run = datetime.today().now(pytz.timezone('US/
↪Pacific')) + timedelta(hours=1),
>>>                    time_zone =pytz.timezone('US/Pacific')
>>>                  )
```

delete (*workspace_id*, *job_id*)
Delete a job from a workspace.

Parameters

- **workspace_id** (*int*) – ID of the workspace that contains the job.
- **job_id** (*str*) – ID of the job to delete.

Returns None.

Return type NoneType

Raises

- **JobNotFoundException** – The job does not exist.
- **InvalidResponseCodeException** – The request got an unexpected HTTP status code in response (not 200 OK).

Example:

```
>>> session.job.delete(workspace_id = 1672, job_id = 675)
```

get (*workspace_id*, *job_id*)
Get one job's metadata.

Parameters

- **workspace_id** (*int*) – ID of the workspace that contains the job.
- **job_id** (*str*) – ID of the job.

Returns Selected job's metadata.

Return type dict

Example:

```
>>> job_info = session.job.get(workspace_id = 1672, job_id = 675)
```

get_id (*workspace_id*, *job_name*)
Gets the job ID.

Parameters

- **workspace_id** (*int*) – ID of the workspace the job is in.
- **job_name** (*str*) – Name of the job.

Returns ID of the job.

Return type int

Example:

```
>>> job_id = session.job.get_id(workspace_id = 1672, job_name = "DemoJob")
>>> print(job_id)
675
```

get_list (*workspace_id*, *per_page=50*)
Get a list of all jobs in a workspace.

Parameters

- **workspace_id** (*int*) – ID of the workspace.
- **per_page** (*int*) – Maximum number to fetch with each API call.

Returns List of jobs' metadata.

Return type list of dict

Example:

```
>>> all_jobs = session.job.get_list(workspace_id = 1672)
```

run (*job_id*)
Run a job.

Parameters **job_id** (*int*) – ID of the job.

Returns HTTP response.

Return type response

Example:

```
>>> session.job.run(job_id = 675)
```

1.1.6 Data Source

class alpine.datasource.**DataSource** (*base_url=None*, *session=None*, *token=None*)
A class for interacting with data sources. These methods may require a login as a user with admin privileges.

class DSType
Convenience strings for data source types.

get (*ds_id*, *type*)
Get one data source's metadata.

Parameters

- **ds_id** (*int*) – A unique ID of the data source.
- **type** (*str*) – Data source type. Either "Database" or "Hadoop".

Returns Selected data source's metadata.

Return type dict

Raises **DataSourceNotFoundException** – the data source does not exist.

Example:

```
>>> session.datasource.get(ds_id = 1, type = "Database")
```

get_database_list (*data_source_id*, *per_page=100*)
Return a list of metadata for all databases in a data source.

Parameters

- **data_source_id** (*int*) – ID of the data source.
- **per_page** (*int*) – Maximum number to fetch with each API call.

Returns List of database metadata.

Return type list of dict

Example:

```
>>> database_list = session.datasource.get_database_list(data_source_id = 1)
>>> len(database_list)
3
```

get_id(*name*, *type=None*)

Gets the ID of the data source. Will throw an exception if the data source does not exist.

Parameters

- **name** (*str*) – Data source name.
- **type** (*str*) – Data source type. Choose to search by "Database" or "Hadoop." Entering None searches both types.

Returns ID of the data source.

Return type int

Raises **DataSourceNotFoundException** – The data source does not exist.

Example:

```
>>> data_source_id = session.datasource.get_id(name = "Demo_GP", type =
↪"Database")
>>> print(data_source_id)
786
```

get_list(*type=None*, *per_page=100*)

Get a list of metadata for all data sources.

Parameters

- **type** (*str*) – Type of the data source. Select "Database", "Hadoop", or None for both types.
- **per_page** (*int*) – Maximum number to fetch with each API call.

Returns List of data source's metadata.

Return type list of dict

Example:

```
>>> all_datasources = session.datasource.get_list()
>>> all_database_datasources = session.datasource.get_list(type = "Database")
>>> all_hadoop_datasources = session.datasource.get_list(type = "Hadoop")
>>> len(all_datasources)
20
```

1.2 Examples

1.2.1 Introduction

Let's start with a example of an Alpine API session.

1. Initialize a session.
2. Take a tour of some commands.
3. Run a workflow and download the results.

Import the Python Alpine API and some other useful packages.

[1]:
```
import alpine as AlpineAPI
```

[2]:
```
from pprint import pprint
import json
```

Setup

Have access to a workflow on your Alpine instance that you can run. You'll need a few pieces of information in order to log in and run the workflow. First, find the URL of the open workflow. It should look something like:

`https://<AlpineHost>:<PortNum>/#workflows/<WorkflowID>`

You'll also need your Alpine username and password.

I've stored my connection information in a configuration file named `alpine_login.conf` that looks something like this:

```
{
    "host": "AlpineHost",
    "port": "PortNum",
    "username": "fakename",
    "password": "12345"
}
```

[3]:
```
filename = "alpine_login.conf"

with open(filename, "r") as f:
    data = f.read()

conn_info = json.loads(data)

host = conn_info["host"]
port = conn_info["port"]
username = conn_info["username"]
password = conn_info["password"]
```

Here are the names of a workspace and a workflow within it that we want to run.

[4]:
```
test_workspace_name = "API Sample Workspace"
test_workflow_name = "Data ETL"
```

Create a session and log in the user.

[5]:
```
session = AlpineAPI.APIClient(host, port, username, password)
```

Use the API

Get information about the Alpine instance.

[6]:
```
pprint(session.get_license())
```

```
{u'admins': 400,
 u'advisor_now_enabled': True,
 u'analytics_developer': 100,
 u'analytics_developer_limit_reached': False,
 u'branding': u'alpine',
 u'business_user': 100,
 u'business_user_limit_reached': False,
 u'client_name': None,
 u'collaborator': 100,
 u'collaborator_limit_reached': False,
 u'correct_mac_address': True,
 u'data_analyst': 100,
 u'data_analyst_limit_reached': False,
 u'expired': False,
 u'expires': u'2099-12-31',
 u'is_enabled_api': True,
 u'is_enabled_custom': True,
 u'is_enabled_jobs': True,
 u'is_enabled_milestones': True,
 u'is_enabled_modeling': True,
 u'is_enabled_touchpoints': True,
 u'level': None,
 u'limit_api': False,
 u'limit_custom': False,
 u'limit_jobs': False,
 u'limit_milestones': False,
 u'limit_modeling': False,
 u'limit_touchpoints': False,
 u'mac_address': u'!!!!!!!!!!',
 u'organization_uuid': None,
 u'users_license_limit_reached': False,
 u'vendor': u'alpine',
 u'version': u'6.3.0.0.5410-4ef43d3c9\n',
 u'workflow_enabled': True}
```

[7]:
```
pprint(session.get_version())
```

```
u'6.3.0.0.5410-4ef43d3c9'
```

Find information about the logged-in user.

[8]:
```
pprint(session.get_status())
```

```
{u'admin': True,
 u'auth_method': u'internal',
 u'dept': u'',
 u'email': u'tjbay@alpinenow.com',
 u'entity_type': u'user',
 u'first_name': u'T.J.',
```

(continues on next page)

(continued from previous page)

```
 u'id': 7,
 u'image': {u'complete_json': True,
            u'entity_type': u'image',
            u'icon': u'/users/7/image?style=icon&1482194432',
            u'original': u'/users/7/image?style=original&1482194432'},
 u'is_deleted': None,
 u'last_name': u'Bay',
 u'ldap_group_id': None,
 u'notes': u'',
 u'roles': [u'admin'],
 u'subscribed_to_emails': False,
 u'tags': [],
 u'title': u'Assistant Regional Manager',
 u'user_type': u'analytics_developer',
 u'username': u'tjbay',
 u'using_default_image': True}
```

Find information on all users.

[9]:
```
len(session.user.get_list())
```

[9]:
```
103
```

Find your user ID and then use it to update your user data.

[10]:
```
user_id = session.user.get_id(username)
```

[11]:
```
pprint(session.user.update(user_id, title = "Assistant to the Regional Manager"))
```

```
{u'admin': True,
 u'auth_method': u'internal',
 u'complete_json': True,
 u'dept': u'',
 u'email': u'tjbay@alpinenow.com',
 u'entity_type': u'user',
 u'first_name': u'T.J.',
 u'id': 7,
 u'image': {u'complete_json': True,
            u'entity_type': u'image',
            u'icon': u'/users/7/image?style=icon&1482194432',
            u'original': u'/users/7/image?style=original&1482194432'},
 u'is_deleted': None,
 u'last_name': u'Bay',
 u'ldap_group_id': None,
 u'notes': u'',
 u'roles': [u'admin'],
 u'subscribed_to_emails': False,
 u'tags': [],
 u'title': u'Assistant to the Regional Manager',
 u'user_type': u'analytics_developer',
 u'username': u'tjbay',
 u'using_default_image': True}
```

A similar set of commands can be used to create and update workspaces and the membership of each workspace.

[12]:
```
test_workspace_id = session.workspace.get_id(test_workspace_name)
session.workspace.member.add(test_workspace_id, user_id);
```

Run a workflow

To run a workflow use the Process subclass of the Workfile class. The `wait_until_finished` method will periodically query the status of the running workflow and returns control to the user when the workflow has completed.

[13]:
```
workflow_id = session.workfile.get_id(workfile_name = "Data ETL",
                                      workspace_id = test_workspace_id)

process_id = session.workfile.process.run(workflow_id)

session.workfile.process.wait_until_finished(workflow_id = workflow_id,
                                             process_id = process_id,
                                             verbose = True,
                                             query_time = 5)
```

```
Workflow in progress for ~88.1 seconds.
```

[13]:
```
u'SUCCESS'
```

We can download results using the `download_results` method. The workflow results contain a summary of the output of each operator as well as metadata about the workflow run.

[14]:
```
flow_results = session.workfile.process.download_results(workflow_id, process_id)
pprint(flow_results, depth=2)
```

```
{u'flowMetaInfo': {u'endTime': u'2017-04-25T15:00:35.178-0700',
                   u'executeUser': u'7',
                   u'noOfError': 0,
                   u'noOfNodesProcessed': 3,
                   u'processId': u'4b820a92-08d1-49f7-9aac-ce07eef4dd3d',
                   u'startTime': u'2017-04-25T14:59:03.259-0700',
                   u'status': u'SUCCESS',
                   u'workflowId': u'1269',
                   u'workflowName': u'Data ETL'},
 u'logs': [{...}, {...}, {...}, {...}, {...}, {...}, {...}],
 u'outputs': [{...}, {...}, {...}]}
```

[ ]:
```

```

1.2.2 Hyperparameter search

We can use the Python API to iteratively run a workflow with different injected variables.

[1]:
```
import alpine as AlpineAPI
```

[2]:
```
from pprint import pprint
import json
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

Connection information

[3]:
```
filename = "alpine_login.conf"

with open(filename, "r") as f:
    data = f.read()

conn_info = json.loads(data)

host = conn_info["host"]
port = conn_info["port"]
username = conn_info["username"]
password = conn_info["password"]
```

Create a session and log in

[4]:
```
session = AlpineAPI.APIClient(host, port, username, password)
```

[5]:
```
pprint(session.workfile.get(701), depth = 2)
```

```
{u'response': {u'associated_worklets': [],
               u'comment_count': 0,
               u'complete_json': True,
               u'dataset_ids': [],
               u'description': u'',
               u'entity_subtype': u'alpine',
               u'entity_type': u'workfile',
               u'execution_locations': [...],
               u'file_name': u'Random Forest Parameter Search',
               u'file_type': u'work_flow',
               u'id': 701,
               u'is_deleted': None,
               u'latest_version_id': None,
               u'link_url': None,
               u'owner': {...},
               u'recent_comments': [],
               u'status': u'idle',
               u'tags': [],
               u'user_modified_at': u'2017-02-28T21:35:54Z',
               u'version_info': {...},
               u'workspace': {...}}}
```

Running a workflow and downloading the results

[6]:
```
workflow_id = 701
```

[7]:
```
process_id = session.workfile.process.run(workflow_id)

session.workfile.process.wait_until_finished(workflow_id,
                                             process_id,
                                             verbose=True,
                                             query_time=5,
                                             timeout=1000)
```

```
Workflow in progress for ~242.5 seconds.
```

[7]:
```
u'SUCCESS'
```

[8]:
```
flow_results = session.workfile.process.download_results(workflow_id, process_id)
pprint(flow_results['outputs'], depth = 2)
```

```
[{u'isGenerateReport': True,
  u'node_meta_info': [...],
  u'out_id': 1493155451032.9402,
  u'out_title': u'magic04.csv',
  u'visualData': {...},
  u'visualType': 0},
 {u'isGenerateReport': True,
  u'node_meta_info': [...],
  u'out_id': 1493155489783.9727,
  u'out_title': u'Random Sampling',
  u'visualData': [...],
  u'visualType': 6},
 {u'isGenerateReport': True,
  u'node_meta_info': [...],
  u'out_id': 1493155499187.6677,
  u'out_title': u'Test Set',
  u'visualData': {...},
  u'visualType': 0},
 {u'isGenerateReport': True,
  u'node_meta_info': [...],
  u'out_id': 1493155506846.1965,
  u'out_title': u'Train Set',
  u'visualData': {...},
  u'visualType': 0},
 {u'isGenerateReport': True,
  u'node_meta_info': [...],
  u'out_id': 1493155616276.4927,
  u'out_title': u'Alpine Forest Classification',
  u'visualData': [...],
  u'visualType': 6},
 {u'isGenerateReport': True,
  u'out_id': 1493155680415.3167,
  u'out_title': u'Confusion Matrix',
  u'visualData': [...],
  u'visualType': 6}]
```

The downloaded results file is a summary of all the operator output in the workflow. In particular, it is a JSON file that we can manipulate or save to disk.

[9]:
```
outfile = "Results_File_N_Trees_{}.fr".format(str(50))

with open(outfile, "w") as f:
    json.dump(flow_results, f)
```

Parsing workflow results

When we convert the downloaded results to a Python object we get a nested dictionary/list object. Here we're pulling two values out of the results:

1. The overall prediction accuracy. This comes from the *Confusion Matrix* operator.
2. The number of trees. This comes from the *Alpine Forest Classification* operator.

This function parses the file to return those two values.

[10]:
```
def parse_flow_results(workflow_id, process_id):
    flow_results = session.workfile.process.download_results(workflow_id, process_id)

    # Get accuracy from the confusion matrix
    conf_matrix_data = session.workfile.process.find_operator('Confusion Matrix',␣
↪flow_results)
    acc = float(conf_matrix_data['visualData'][0]['visualData']['items'][2]['Class␣
↪Recall'].split()[1])

    # Get number of trees from the Alpine Forest
    alpine_forest_data = session.workfile.process.find_operator('Alpine Forest␣
↪Classification', flow_results)
    N = int(alpine_forest_data['visualData'][2]['visualData']['items'][0]['Average␣
↪over All Trees'])

    return (N, acc)
```

Workflow variables

Variables with different values can be inserted into workflows. They have to be formatted as below and are passed as an optioned argument to the workflow run method.

[11]:
```
ntrees = [5,10,25,50,75]
variables = [[{"name": "@n_trees", "value": str(N)}] for N in ntrees]
```

[12]:
```
variables
```

[12]:
```
[[{'name': '@n_trees', 'value': '5'}],
 [{'name': '@n_trees', 'value': '10'}],
 [{'name': '@n_trees', 'value': '25'}],
 [{'name': '@n_trees', 'value': '50'}],
 [{'name': '@n_trees', 'value': '75'}]]
```

Run the workflow in a loop, extracting test accuracy

[15]:
```
test_acc = []

for variable in variables:
    print("Running with workflow variable: {}".format(variable))
    process_id = session.workfile.process.run(workflow_id, variables=variable)
    session.workfile.process.wait_until_finished(workflow_id, process_id,␣
↪verbose=True, query_time=5, timeout=1000)

    (N, acc) = parse_flow_results(workflow_id, process_id)
    test_acc.append(acc)

    print("For {} trees, test accuracy is {}".format(N, acc))
    print("")
```

```
Running with workflow variable: [{'name': '@n_trees', 'value': '5'}]
Workflow in progress for ~191.1 seconds.
For 5 trees, test accuracy is 0.8530647
```

(continues on next page)

(continued from previous page)

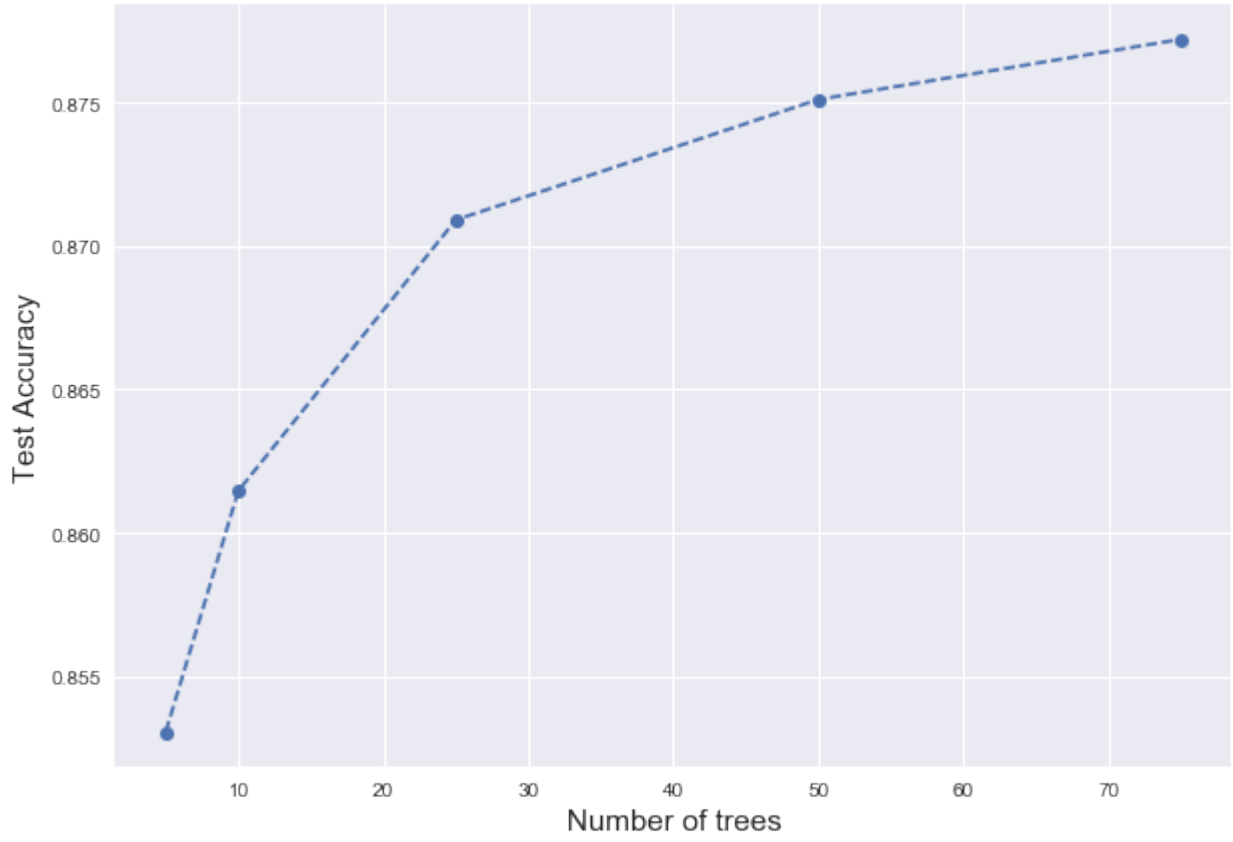
```
Running with workflow variable: [{'name': '@n_trees', 'value': '10'}]
Workflow in progress for ~207.0 seconds.
For 10 trees, test accuracy is 0.861461

Running with workflow variable: [{'name': '@n_trees', 'value': '25'}]
Workflow in progress for ~250.3 seconds.
For 25 trees, test accuracy is 0.8709068

Running with workflow variable: [{'name': '@n_trees', 'value': '50'}]
Workflow in progress for ~311.6 seconds.
For 50 trees, test accuracy is 0.875105

Running with workflow variable: [{'name': '@n_trees', 'value': '75'}]
Workflow in progress for ~363.1 seconds.
For 75 trees, test accuracy is 0.877204
```

[16]:
```
plt.figure(figsize = (10,7))
plt.plot(ntrees, test_acc, "o--")
plt.xlabel("Number of trees", size=15)
plt.ylabel("Test Accuracy", size=15);
```



[ ]:

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

R

S

U

W