DNAnexus Python Bindings Documentation

Release

DNAnexus

January 17, 2015

Contents

The DNAnexus `dxpy` Python library provides Python bindings to interact with the DNAnexus Platform via its API. The bindings are available to Python apps running within the DNAnexus Execution Environment, and can also be used in scripts you run that access the API from the outside. Before you start digging in, you may want to take a look at the Introduction to the DNAnexus Platform. The following references may also be useful to you as you develop your own scripts and apps.

API Specification Complete details on the API, the lifecycle of different data objects, project permissions, and more. Many methods in the bindings translate directly into API calls, so it will be helpful to have at least a general understanding of the API's structure.

Execution Environment Reference How applets and apps are run within the DNAnexus Platform.

Developer Portal Links to all other docs, e.g. useful command-line tools.

CHAPTER 1

Table of Contents

1.1 `dxpy` Package

This Python 2.7 package includes three key modules:

- `dxpy.bindings`: Contains useful Pythonic bindings for interacting with remote objects via the DNAnexus API server. For convenience, this is automatically imported directly into the namespace under `dxpy` when `dxpy` is imported.
- `dxpy.api`: Contains low-level wrappers that can be called directly to make the respective API calls to the API server.
- `dxpy.exceptions`: Contains exceptions used in the other `dxpy` modules.

It has the following external dependencies:

- `requests`: To install on Linux, use `sudo pip install requests`. Other installation options can be found at http://docs.python-requests.org/en/latest/user/install
- `futures`: To install on Linux, use `sudo pip install futures`. Other installation options can be found at http://code.google.com/p/pythonfutures/

1.1.1 Package Configuration

1.2 `dxpy.bindings` Module

Documentation on classes and methods:

1.2.1 `dxpy` Object Handlers

1.2.2 Utility Functions

1.2.3 Projects and Containers

1.2.4 Records

1.2.5 Files

Files can be used to store an immutable opaque sequence of bytes.

You can obtain a handle to a new or existing file object with `new_dxfile()` or `open_dxfile()`, respectively. Both return a remote file handler, which is a Python file-like object. There are also helper functions (`download_dxfile()`, `upload_local_file()`, and `upload_string()`) for directly downloading and uploading existing files or strings in a single operation.

Files are tristate objects:

- When initially created, a file is in the "open" state and data can be written to it. After you have written your data to the file, call the `close()` method.
- The file enters the "closing" state while it is finalized in the platform.
- Some time later, the file enters the "closed" state and can be read.

Many methods that return a `DXFile` object take a *mode* parameter. In general the available modes are as follows (some methods create a new file and consequently do not support immediate reading from it with the "r" mode):

- "r" (read): for read-only access. No data is expected to be written to the file.
- "w" (write): for writing. When the object exits scope, all buffers are flushed and closing commences.
- "a" (append): for writing. When the object exits scope, all buffers are flushed but the file is left open.

Note: The automatic flush and close operations implied by the "w" or "a" modes **only** happen if the `DXFile` object is used in a Python context-managed scope (see the following examples).

Here is an example of writing to a file object via a context-managed file handle:

```
# Open a file for writing
with open_dxfile('file-xxxx', mode='w') as fd:
    for line in input_file:
        fd.write(line)
```

The use of the context-managed file is optional for read-only objects; that is, you may use the object without a "with" block (and omit the *mode* parameter), for example:

```
# Open a file for reading
fd = open_dxfile('file-xxxx')
for line in fd:
    print line
```

Warning: If you write any data to a file and you choose to use a non context-managed file handle, you **must** call `flush()` or `close()` when you are done, for example:

```
# Open a file for writing; we will flush it explicitly ourselves
fd = open_dxfile('file-xxxx')
for line in input_file:
    fd.write(line)
fd.flush()
```

If you do not do so, and there is still unflushed data when the `DXFile` object is garbage collected, the `DXFile` will attempt to flush it then, in the destructor. However, any errors in the resulting API calls (or, in general, any exception in a destructor) are **not** propagated back to your program! That is, *your writes can silently fail if you rely on the destructor to flush your data*.
`DXFile` will print a warning if it detects unflushed data as the destructor is running (but again, it will attempt to flush it anyway).

Note: Writing to a file with the "w" mode calls `close()` but does not wait for the file to finish closing. If the file you are writing is one of the outputs of your app or applet, you can use job-based object references, which will make

downstream jobs wait for closing to finish before they can begin. However, if you intend to subsequently read from the file in the same process, you will need to call `wait_on_close()` to ensure the file is ready to be read.

1.2.6 GenomicTables

A GenomicTable (GTable) is an immutable tabular dataset suitable for large-scale genomic applications.

You can obtain a handle to a new or existing GTable object with `new_dxgtable()` or `open_dxgtable()`, respectively.

GTables are tristate objects:

- When initially created, a GTable is in the "open" state and row data can be written to it. After you have written your row data to the GTable, call the `close()` method.
- The GTable enters the "closing" state while it is finalized in the platform.
- Some time later, the GTable enters the "closed" state and can be read.

Many methods that return a `DXGTable` object take a *mode* parameter. In general the available modes are as follows (some methods create a new GTable and consequently do not support immediate reading from it with the "r" mode):

- "r" (read): for read-only access. No data is expected to be written to the GTable.
- "w" (write): for writing. When the object exits scope, all buffers are flushed and GTable closing commences.
- "a" (append): for writing. When the object exits scope, all buffers are flushed but the GTable is left open.

Note: The automatic flush and close operations implied by the "w" or "a" modes **only** happen if the `DXGTable` object is used in a Python context-managed scope (see the following examples).

The "w" mode can be used to create a new GTable, populate it, and close it within a single process. The "a" mode leaves the GTable open after all data is written. You can use it when you wish to open a GTable for writing in multiple sub-jobs in parallel. (You then need to close the table, once, after all those jobs have finished.)

Here is an example of writing to a GTable via a context-managed GTable handle:

```
# Open a GTable for writing
with open_dxgtable('gtable-xxxx', mode='w') as gtable:
    for line in input_file:
        gtable.add_row(line.split(','))
```

The use of the context-managed GTable is optional for read-only objects; that is, you may use the object without a "with" block (and omit the *mode* parameter), for example:

```
# Open a GTable for reading
gtable = open_dxgtable('gtable-xxxx')
for row in gtable.iterate_rows(...):
    process(row)
```

Warning: If you write any data to a GTable (with `add_row()` or `add_rows()`) and you choose to use a non context-managed GTable handle, you **must** call `flush()` or `close()` when you are done, for example:

```
# Open a GTable for writing; we will flush it explicitly ourselves
gtable = open_dxgtable('gtable-xxxx')
for line in input_file:
    gtable.add_row(line.split(','))
gtable.flush()
```

If you do not do so, and there is still unflushed data when the `DXGTable` object is garbage collected, the `DXGTable` will attempt to flush it then, in the destructor. However, any errors in the resulting API calls (or, in general, any exception in a destructor) are **not** propagated back to your program! That is, *your writes can silently fail if you rely on the destructor to flush your data*.
`DXGTable` will print a warning if it detects unflushed data as the destructor is running (but again, it will attempt to flush it anyway).

Note: Writing to a GTable with the "w" mode calls `close()` but does not wait for the GTable to finish closing. If the GTable you are writing is one of the outputs of your app or applet, you can use job-based object references, which will make downstream jobs wait for closing to finish before they can begin. However, if you intend to subsequently read from the GTable in the same process, you will need to call `wait_on_close()` to ensure the GTable is ready to be read.

1.2.7 Applets, Apps, Workflows, and Jobs

An executable (applet or app) defines application logic that is to be run in the DNAnexus Platform's Execution Environment. In order to facilitate parallel processing, an executable may define multiple functions (or *entry points*) that can invoke each other; a running job can use `new_dxjob()` to invoke any function defined in the same executable, creating a new job that runs that function on a different machine (possibly even launching multiple such jobs in parallel).

To create an executable from scratch, we encourage you to use the command-line tools dx-app-wizard and dx build rather than using the API or bindings directly. The following handlers for applets, apps, and jobs are most useful for running preexisting executables and monitoring their resulting jobs.

Workflows created from the website UI can also be run using the `DXWorkflow` workflow handler.

For **DXApp.run()**, see `run()`.

1.2.8 Search

This module contains useful Python bindings for calling API methods on the DNAnexus Platform. Data objects (such as records, files, GenomicTables, and applets) are represented locally by a handler that inherits from the abstract class `DXDataObject`. This abstract base class supports functionality common to all of the data object classes–for example, setting properties and types, as well as cloning the object to a different project, moving it to a different folder in the same project, or removing the object from a project.

Note: While this documentation will largely refer to data containers as simply "projects", both project and container IDs can generally be provided as input and will be returned as output anywhere a "project" is expected, except in methods of the `DXProject` class specifically or where otherwise noted.

Object and Project IDs

A remote handler for a data object has two IDs associated with it: one ID representing the underlying data and a project ID to indicate which project's copy it represents. (If not explicitly specified for a particular handler, the project defaults to the default data container.) The ID of a data object remains the same when it is moved within a project or cloned to another project.

The project ID is **only** relevant when using certain metadata fields that are tied to a particular project. These are the name, properties, and tags fields, and are read and updated using the following methods: `describe()` ("name", "properties", and "tags" fields), `rename()`, `get_properties()`, `set_properties()`, `add_tags()`, `remove_tags()`.

Creating new handlers and remote objects

To access a preexisting object, a remote handler for that class can be set up via two methods: the constructor or the `set_ids()` method. For example:

```
# Provide ID in constructor
dxFileHandle = DXFile("file-1234")

# Provide no ID initially, then call set_ids()
dxOtherFH = DXFile()
dxOtherFH.set_ids("file-4321")
```

Neither of these methods perform API calls; they merely set the local state of the remote file handler. The object ID and project ID stored in the handler can be overwritten with subsequent calls to `set_ids()`.

The object handler `__init__` methods do not create new remote objects; they only initialize whatever local state the handler needs. Creation of a new remote object can be performed using the method `dxpy.bindings.DXDataObject.new()`. In each subclass of `DXDataObject` the method can take class-specific arguments, for example:

```
newDXFileHandle = DXFile()
newDXFileHandle.new(media_type="application/json")
```

Some of the classes provide additional functions that are shorthand for some of these common use cases. For instance, `dxpy.bindings.dxfile_functions.open_dxfile()` opens a preexisting file, and `dxpy.bindings.dxfile_functions.new_dxfile()` creates a new file and opens it for writing. Both of those methods return a remote object handler on which additional methods can be called.

In addition, class-specific handlers provide extra functionality for their respective classes. For example, `DXFile` provides functionality for reading, writing, downloading, and uploading files.

Though not explicitly documented in each method as such, all methods that interact with the API server may raise the exception `dxpy.exceptions.DXAPIError`.

1.3 `dxpy.app_builder` Module

This module contains high-level subroutines for creating app and applet objects.

If you are developing apps, we strongly recommend that you use the command-line application builder tool dx build to compile and deploy applets and apps onto the platform. (Those command-line tools are implemented using the methods in this module.)

Warning: This module is mostly intended for DNAnexus internal use. You probably only need to use the methods here directly if you are implementing a specialized development or publishing workflow that is not supported by the stock command-line builders.

1.4 `dxpy.utils` Module

1.5 `dxpy.api` Module

This module is automatically generated from a list of available routes on the API server. Functions in this module take a remote object ID (when appropriate) and an optional argument *input_params* to set the request body. The request must be a list or a dict, as it will be converted to JSON. If *input_params* is not provided, the JSON of an empty dict will be sent. Each function returns the Pythonized JSON (a list or dict) that is returned from the API server.

An example API call

dxpy.api.**classname_methodname**(*object_id*, *input_params={}*, *always_retry=False*, ***kwargs*)

Parameters

- **object_id** (*string*) – Object ID of remote object to be manipulated
- **input_params** (*list or dict*) – Request body (will be converted to JSON)
- **always_retry** (*bool*) – True if the request is idempotent and is safe to retry. Note that this parameter is misleadingly named; setting it to True (False, respectively) does not guarantee that the request will always (never) be retried. Rather, it is advisory and whether the request is retried depends on the specific error condition: see the retry logic specification for more information.
- **kwargs** – Additional arguments to be passed to the `dxpy.DXHTTPRequest` object (such as headers)

Returns Contents of response from API server (Pythonized JSON object)

Return type list or dict

Raises `DXAPIError` if an HTTP response code other than 200 is received from the API server.

For apps, the signature is slightly different, because apps can also be named by their name and version (or tag), in addition to their app ID.

An example API call on an app instance

dxpy.api.**app_methodname**(*app_name_or_id*, *alias=None*, *input_params={}*, *always_retry=False*, ***kwargs*)

Parameters

- **app_name_or_id** (*string*) – Either "app-NAME" or the hash ID "app-xxxx"
- **alias** (*string*) – If *app_name_or_id* is given using its name, then a version or tag string (if none is given, then the tag "default" will be used). If *app_name_or_id* is a hash ID, this value should be `None`.
- **input_params** (*list or dict*) – Request body (will be converted to JSON)

- **always_retry** (*bool*) – True if the request is idempotent and is safe to retry. Note that this parameter is misleadingly named; setting it to True (False, respectively) does not guarantee that the request will always (never) be retried. Rather, it is advisory and whether the request is retried depends on the specific error condition: see the retry logic specification for more information.
- **kwargs** – Additional arguments to be passed to the `dxpy.DXHTTPRequest` object (such as headers)

Returns Contents of response from API server (Pythonized JSON object)

Return type list or dict

Raises `DXAPIError` if an HTTP response code other than 200 is received from the API server.

The specific functions provided in this module are enumerated below.

1.6 `dxpy.exceptions` Module

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

A

C