Python API guide Documentation

Release 0.1.0

Open Microscopy Environment

Apr 25, 2022

CONTENTS

The section demonstrates how to install and use the Python API. Jupyter notebooks are used to analyze data and save results back the server.

CHAPTER

ONE

CONTENTS

1.1 Install omero-py

In this section, we show how to install omero-py together with additional packages for image analysis in a Conda environment.

We will use the Python API to access data stored in an OMERO server. The image analysis is typically done with packages like scikit-image or dask.

If you want to use the Python API only for interacting with the OMERO server, like scripting management / import workflows you can just install the omero-py package by itself, see omero-py.

1.1.1 Setup

For using the examples and notebooks of this guide we recommend using Conda (Option 1). Conda manages programming environments in a manner similar to virtualenv.

Alternatively you can use `repo2docker` to build and run a Docker image locally (Option 2). This Docker image will provide the Conda environment and Jupyter notebooks with some image analysis workflows.

When the installation is done, you should be ready to use the OMERO Python API, see *Getting started with the OMERO Python API*.

Option 1

Install omero-py and additional packages for image analysis (see `environment.yml`) via Conda.

- Install Miniconda if necessary.
- If you do not have a local copy of the omero-guide-python repository, first clone the repository:

```
$ git clone https://github.com/ome/omero-guide-python.git
```

- Go into the directory:

```
$ cd omero-guide-python
```

- Create a programming environment using Conda:

```
$ conda create -n omeropy python=3.6
```

- Install `omero-py` and other packages useful for demonstration purposes in order to connect to an OMERO server using an installation file:

```
$ conda env update -n omeropy --file binder/environment.yml
```

- Activate the environment:

```
$ conda activate omeropy
```

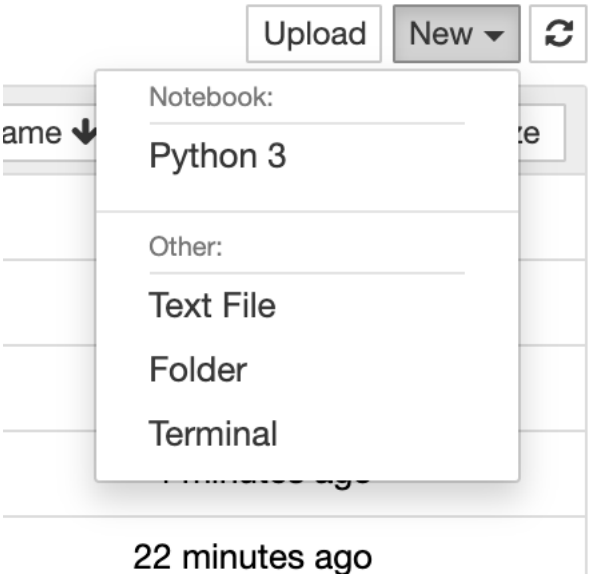
Option 2

Create a local Docker Image using `repo2docker`, see `README.md`:

```
$ pip install jupyter-repo2docker
$ git clone https://github.com/ome/omero-guide-python.git
$ cd omero-guide-python
$ repo2docker .
```

When the Image is ready:

- Copy the URL displayed in the terminal in your favorite browser
- Click the `New` button on the right-hand side of the window
- Select `Terminal`



- A Terminal will open in a new Tab
- A Conda environment has already been created when the Docker Image was built
- To list all the Conda environments, run:

```
$ conda env list
```

- The environment with the OMERO Python bindings and a few other libraries is named `notebook`, activate it:

```
$ conda activate notebook
```

1.2 Getting started with the OMERO Python API

1.2.1 Description

We will show:

- Connect to a server.
- Load images.
- Run a simple FRAP analysis measuring the intensity within a pre-existing ellipse ROI in a named Channel.
- Save the generated mean intensities and link them to the image(s).
- Save the generated plot on the server.

1.2.2 Setup

We recommend to use a Conda environment to install the OMERO Python bindings. Please read first *Install omero-py*.

For the FRAP analysis you need a fluorescence time-lapse image, available at https://downloads.openmicroscopy.org/images/DV/will/FRAP/.

The bleached spot has to be marked with an ellipse. Make sure that the ellipse ROI spans the whole timelapse.

1.2.3 Step-by-Step

In this section, we go through the steps required to analyze the data. The script used in this document is `simple_frap.py`.

It is also available as the 'SimpleFRAP' Jupyter notebook in the notebooks section.

Modules and methods which need to be imported:

```
from omero.gateway import BlitzGateway, MapAnnotationWrapper
from omero.model import EllipseI
from PIL import Image
import numpy as np
import matplotlib
from matplotlib import pyplot as plt
from getpass import getpass
```

Connect to the server. It is also important to close the connection again to clear up potential resources held on the server. This is done in the `disconnect` method.

```
def connect(hostname, username, password):
    """
    Connect to an OMERO server
    :param hostname: Host name
    :param username: User
    :param password: Password
    :return: Connected BlitzGateway
    """
```

(continues on next page)

(continued from previous page)

```
    conn = BlitzGateway(username, password,
                        host=hostname, secure=True)
    conn.connect()
    conn.c.enableKeepAlive(60)
    return conn


def disconnect(conn):
    """
    Disconnect from an OMERO server
    :param conn: The BlitzGateway
    """
    conn.close()
```

Load the image:

```
    img = conn.getObject("Image", image_id)
```

Get relevant channel index:

```
def get_channel_index(image, label):
    """
    Get the channel index of a specific channel
    :param image: The image
    :param label:  The channel name
    :return: The channel index (None if not found)
    """
    labels = image.getChannelLabels()
    if label in labels:
        idx = labels.index(label)
        return idx
    return None
```

Get Ellipse ROI:

```
def get_ellipse_roi(conn, image):
    """
    Get the first ellipse ROI found in the image
    :param conn: The BlitzGateway
    :param image: The Image
    :return: The shape ID of the first ellipse ROI found
    """
    roi_service = conn.getRoiService()
    result = roi_service.findByImage(image.getId(), None)
    shape_id = None
    for roi in result.rois:
        for s in roi.copyShapes():
            if type(s) == EllipseI:
                shape_id = s.id.val
```

(continues on next page)

(continued from previous page)

```
    return shape_id
```

Get intensity values:

```
def get_mean_intensities(conn, image, the_c, shape_id):
    """
    Get the mean pixel intensities of an roi in a time series image
    :param conn: The BlitzGateway
    :param image: The image
    :param the_c: The channel index
    :param shape_id: The ROI shape id
    :return: List of mean intensity values (one for each timepoint)
    """
    roi_service = conn.getRoiService()
    the_z = 0
    size_t = image.getSizeT()
    meanvalues = []
    for t in range(size_t):
        stats = roi_service.getShapeStatsRestricted([shape_id],
                                                    the_z, t, [the_c])
        meanvalues.append(stats[0].mean[0])
    return meanvalues
```

Plot the data using `matplotlib`:

```
def plot(values, plot_filename):
    """
    Create a simple plot of the given values
    and saves it.
    :param values: The values
    :param plot_filename: The file name
    :return: Nothing
    """
    matplotlib.use('Agg')
    fig = plt.figure()
    plt.subplot(111)
    plt.plot(values)
    fig.canvas.draw()
    fig.savefig(plot_filename)
    pil_img = Image.open(plot_filename)
    pil_img.show()
```

Save the results as Map annotation:

```
def save_values(conn, image, values):
    """
    Attach the values as map annotation to the image
```

(continues on next page)

(continued from previous page)

```
    :param conn: The BlitzGateway
    :param image: The image
    :param values: The values
    :return: Nothing
    """
    namespace = "demo.simple_frap_data"
    key_value_data = [[str(t), str(value)] for t, value in enumerate(values)]
    map_ann = MapAnnotationWrapper(conn)
    map_ann.setNs(namespace)
    map_ann.setValue(key_value_data)
    map_ann.save()
    image.linkAnnotation(map_ann)
```

Save the plot:

```
def save_plot(conn, image, plot_filename):
    """
    Save the plot to OMERO
    :param conn: The BlitzGateway
    :param image: The image
    :param plot_filename: The path to the plot image
    :return: Nothing
    """
    pil_img = Image.open(plot_filename)
    np_array = np.asarray(pil_img)
    red = np_array[::, ::, 0]
    green = np_array[::, ::, 1]
    blue = np_array[::, ::, 2]
    plane_gen = iter([red, green, blue])
    conn.createImageFromNumpySeq(plane_gen, plot_filename, sizeC=3,
                                 dataset=image.getParent())
```

In order the use the methods implemented above in a proper standalone script: Wrap it all up in an `analyse` method and call it from `main`:

```
def analyse(conn, image_id, channel_name):
    # Step 2 - Load iamge
    img = conn.getObject("Image", image_id)
    # -
    ci = get_channel_index(img, channel_name)
    shape_id = get_ellipse_roi(conn, img)
    values = get_mean_intensities(conn, img, ci, shape_id)
    plot(values, 'plot.png')
    save_values(conn, img, values)
    save_plot(conn, img, 'plot.png')


def main():
```

(continues on next page)

(continued from previous page)

```
    try:
        hostname = input("Host [wss://workshop.openmicroscopy.org/omero-ws]: \
                         ") or "wss://workshop.openmicroscopy.org/omero-ws"
        username = input("Username [trainer-1]: ") or "trainer-1"
        password = getpass("Password: ")
        image_id = int(input("Image ID [28662]: ") or 28662)
        channel = input("Channel name [528.0]: ") or "528.0"
        conn = connect(hostname, username, password)
        analyse(conn, image_id, channel)
    finally:
        if conn:
            disconnect(conn)


if __name__ == "__main__":
    main()
```

1.2.4 Further Reading

How to turn a script into an OMERO script which runs on the server and can be directly launched via the web interface, see *How to convert a client-side script into a server-side script*.

This `simple_frap.py` example as server-side OMERO.script: `simple_frap_server.py`.

1.3 How to convert a client-side script into a server-side script

In this section, we show how to convert a Python script into a script that can be run server-side. We recommend that you read OMERO.scripts guide.

1.3.1 Description

We will make a simple `Hello World` script as client-side Python script and show how to convert it into a server-side script. The script will:

- Connect to the server
- Load images in a dataset

1.3.2 Setup

We recommend to use a Conda environment to install the OMERO Python bindings. Please read first *Install omero-py*.

1.3.3 Step-by-step

The scripts used in this document are `hello_world.py` and `hello_world_server.py`.

It is also available as the 'OMEROHelloWorld' Jupyter notebook in the notebooks section.

Client-side script

Let's first start by writing a **client-side** script named `hello_world.py`.

Connect to the server:

```
def connect(hostname, username, password):
    """
    Connect to an OMERO server
    :param hostname: Host name
    :param username: User
    :param password: Password
    :return: Connected BlitzGateway
    """
    conn = BlitzGateway(username, password,
                        host=hostname, secure=True)
    conn.connect()
    conn.c.enableKeepAlive(60)
    return conn
```

Load the images in the dataset:

```
def load_images(conn, dataset_id):
    """
    Load the images in the specified dataset
    :param conn: The BlitzGateway
    :param dataset_id: The dataset's id
    :return: The Images or None
    """
    dataset = conn.getObject("Dataset", dataset_id)
    images = []
    for image in dataset.listChildren():
        images.append(image)
    if len(images) == 0:
        return None

    for image in images:
        print("---- Processing image", image.id)
    return images
```

Collect the script parameters:

```
        host = input("Host [wss://workshop.openmicroscopy.org/omero-ws]: ") or 'wss://
↪workshop.openmicroscopy.org/omero-ws'  # noqa
        username = input("Username [trainer-1]: ") or 'trainer-1'
        password = getpass("Password: ")
        dataset_id = input("Dataset ID [2391]: ") or '2391'
```

In order to use the methods implemented above in a proper standalone script: Wrap it all up and call them from `main`:

```
if __name__ == "__main__":
    try:
        # Collect parameters
        host = input("Host [wss://workshop.openmicroscopy.org/omero-ws]: ") or 'wss://
↪workshop.openmicroscopy.org/omero-ws'  # noqa
        username = input("Username [trainer-1]: ") or 'trainer-1'
        password = getpass("Password: ")
        dataset_id = input("Dataset ID [2391]: ") or '2391'

        # Connect to the server
        conn = connect(host, username, password)

        # Load the images container in the specified dataset
        load_images(conn, dataset_id)
    finally:
        conn.close()
    print("done")
```

Server-side script

Now let's see how to convert the script above into a **server-side** script.

The first step is to declare the parameters, the UI components will be built automatically from it. This script only needs to collect the dataset ID:

```
    # Define the script name and description, and a single 'required' parameter
    client = scripts.client(
        'Hello World.py',
        """
    This script does connect to OMERO.
        """,
        scripts.Long("datasetId", optional=False),

        authors=["OME Team", "OME Team"],
        institutions=["University of Dundee"],
        contact="ome-users@lists.openmicroscopy.org.uk",
    )
```

Process the arguments:

```
        script_params = {}
        for key in client.getInputKeys():
            if client.getInput(key):
```

(continues on next page)

(continued from previous page)

```
                script_params[key] = client.getInput(key, unwrap=True)

        dataset_id = script_params["datasetId"]
```

Access the data using the Python gateway:

```
        conn = BlitzGateway(client_obj=client)
```

We can then use the same method that the one in the client-side script to load the images:

```
def load_images(conn, dataset_id):
    """
    Load the images in the specified dataset
    :param conn: The BlitzGateway
    :param dataset_id: The dataset's id
    :return: The Images or None
    """
    dataset = conn.getObject("Dataset", dataset_id)
    images = []
    if dataset is None:
        return None
    for image in dataset.listChildren():
        images.append(image)
    if len(images) == 0:
        return None

    for image in images:
        print("---- Processing image", image.id)
    return images
```

In order to use the methods implemented above in a proper standalone script and return the output to the users, wrap it all up and call them from main:

```
if __name__ == "__main__":
    # Start declaration
    # Define the script name and description, and a single 'required' parameter
    client = scripts.client(
        'Hello World.py',
        """
    This script does connect to OMERO.
        """,
        scripts.Long("datasetId", optional=False),

        authors=["OME Team", "OME Team"],
        institutions=["University of Dundee"],
        contact="ome-users@lists.openmicroscopy.org.uk",
    )
    # Start script
    try:
```

(continues on next page)

(continued from previous page)

```
        # process the list of arguments
        script_params = {}
        for key in client.getInputKeys():
            if client.getInput(key):
                script_params[key] = client.getInput(key, unwrap=True)

        dataset_id = script_params["datasetId"]

        # wrap client to use the Blitz Gateway
        conn = BlitzGateway(client_obj=client)

        # load the images
        images = load_images(conn, dataset_id)

        # return output to the user
        if images is None:
            message = "No images found"
        else:
            message = "Returned %s images" % len(images)
            # return first image:
            client.setOutput("Image", robject(images[0]._obj))

        client.setOutput("Message", rstring(message))
        # end output
    finally:
        client.closeSession()
```

1.4 Analyze data stored in a public S3 repository in parallel

1.4.1 Description

We will show how to use dask to analyze an IDR image stored in a public S3 repository

We will show:

- How to connect to IDR to retrieve the image metadata.
- How to load the Zarr binary stored in a public repository.
- How to run a segmentation on each plane in parallel.

1.4.2 Setup

We recommend to use a Conda environment to install the OMERO Python bindings. Please read first *Install omero-py*.

1.4.3 Step-by-Step

In this section, we go through the steps required to analyze the data. The script used in this document is `public_s3_segmentation_parallel.py`.

Connect to the server:

Load the image:

Create a dask array from the Zarr storage format:

```
def load_binary_from_s3(id, resolution='4'):
    endpoint_url = 'https://uk1s3.embassy.ebi.ac.uk/'
    root = 'idr/zarr/v0.1/%s.zarr/%s/' % (id, resolution)
    return da.from_zarr(endpoint_url + root)
```

Define the analysis function:

```
def analyze(t, c, z):
    plane = data[t, c, z, :, :]
    smoothed_image = dask_image.ndfilters.gaussian_filter(plane, sigma=[1, 1])
    threshold_value = 0.33 * da.max(smoothed_image).compute()
    threshold_image = smoothed_image > threshold_value
    label_image, num_labels = dask_image.ndmeasure.label(threshold_image)
    name = 't:%s, c: %s, z:%s' % (t, c, z)
    print("Plane coordinates: %s" % name)
    ref = 't_%s_c_%s_z_%s' % (t, c, z)
    return label_image, ref
```

Make our function lazy using `dask.delayed`. It records what we want to compute as a task into a graph that we will run later in parallel:

```
def prepare_call(dimensions):
    middle_z = dimensions[2] // 2
    middle_t = dimensions[0] // 2
    range_t = 2
    range_z = 2
    number_c = dimensions[1]
    lazy_results = []
    for t in range(middle_t - range_t, middle_t + range_t):
        for z in range(middle_z - range_z, middle_z + range_z):
            for c in range(number_c):
                lazy_result = dask.delayed(analyze)(t, c, z)
                lazy_results.append(lazy_result)
    return lazy_results
```

We are now ready to run in parallel using the default number of workers see Configure dask.compute:

When done, close the session:

In order to use the methods implemented above in a proper standalone script: **Wrap it all up** in `main`:

```
def main():
    # Collect image ID
    image_id = "4007801"

    global data
    data = load_binary_from_s3(image_id)
    print("Dask array: %s" % data)

    lazy_results = prepare_call(data.shape)

    start = time.time()
    results = compute(lazy_results)
    elapsed = time.time() - start
    print('Compute time (in seconds): %s' % elapsed)
    save_results(results)
    print('done')


if __name__ == "__main__":
    main()
```

CHAPTER

TWO

CONTRIBUTE

Changes to the documentation or the materials should be made directly in the omero-guide-python repository.