New Python client usage

The Python client allows to interact with Cytomine-core through the RESTful API using Python.

Changelog

This new version is a complete rewrite of the Python client and improves and adds a lot of features:

- Compatibility Python 2.7 / 3+
- Logging with verbosity levels (standard Python logger)
- Use requests library to handle HTTP
- Collections are iterable
- Model's attributes are written in the code: code is self-documented and ease auto-completion
- Cytomine class is a singleton and its no more necessary to pass it as parameter in every functions
- Model and Collection are self managed
- CytomineJob class extends Cytomine and is thus a singleton where the current user is a user-job.

Connection

You need 3 parameters to connect :

1. The host (example: demo.cytomine.be)
2. Your public key
3. Your private key

It is recommended to use the with statement to establish the HTTP session between the client and the sever. The "with statement" block is a Python feature to efficiently deal with data flows.

```
host = "demo.cytomine.be"
public_key = "XXX"
private_key = "XXX"

with Cytomine(host=host, public_key=public_key, private_key=private_key,
              verbose=logging.INFO) as cytomine:
        print(cytomine.current_user)
```

The verbose parameter allows to change the verbosity level. By default, it is set to INFO, which prints a summary of a resource when fetched, saved, updated or deleted. See the Python standard logging module for more information.

Another possibility to connect, especially for short scripts is to use the connect() method:

```
cytomine = Cytomine.connect(host, public_key, private_key)
print(cytomine.current_user)
```

It is still possible to connect by simply instantiate the Cytomine class for backwards compatibility reasons but is deprecated.

Model

A Cytomine resource (e.g. a project A) is an instance of a domain (e.g. Project). In the client, a resource is a Python object which is an instance of a model class describing its domain. Each object (an instance of model):

- has a set of attributes corresponding to resource attributes
- is managed itself through fetch(), save(), update() and delete() methods that communicate with the Cytomine-core server
- has some utilities to be serialized to JSON
- a dictionary of query parameters that will appear in the request: resource.json?param1=value1 (seldom used for models)

A simple example with a project resource:

```
# ... Assume we are connected
id_ontology = 0

# Add a new project
# The save() method makes the appropriate HTTP POST request.
# Option 1
my_new_project = Project("my project name", id_ontology).save()
# Option 2
my_new_project = Project()
my_new_project.name = "my project name"
my_new_project.ontology = id_ontology
my_new_project.save()


# Get the project from server.
# The fetch() method makes the appropriate HTTP GET request.
# Option 1
my_new_project2 = Project().fetch(id=my_new_project.id)
# Option 2
my_new_project2 = Project()
my_new_project2.id = my_new_project.id
my_new_project2.fetch()


# Update a project
# The update() method makes the appropriate HTTP PUT request.
# Option 1
my_new_project.name = "Another name"
my_new_project.update()
# Option 2
my_new_project.update(name="Another name")


# Delete a project
# The delete() method makes the appropriate HTTP DELETE request.
# Option 1
my_new_project.delete()
# Option 2
Project().delete(id=my_new_project.id)
```

Supplementary actions

In some cases, the server provides supplementary actions for some resources. These actions are encapsulated in additional methods for the model.

Example 1

Download the original file behind an abstract image.

```
id_image = 0
image = AbstractImage().fetch(id_image)
image.download("/tmp/{originalFilename}")
```

The download() method will get the image and copy it into the temporary directory. The name of the file is the original filename stored into Cytomine-core because the destination path is parsed, and every model attribute can be used. For example, it is possible to save the image with a filename which contains its Cytomine ID, its width and height:

```
image.download("/tmp/{id}_{width}px_{height}px.jpg")
```

Example 2

Get the spectrum for a given pixel in an Image Group HDF5.

```
id_image_group_hdf5 = 0
ighdf5 = ImageGroupHDF5().fetch(id_image_group_hdf5)
spectrum = ighdf5.pixel(100, 100)
print(spectrum) # Print spectrum (Numpy array) at position (100, 100)
```

The pixel request makes the appropriate GET request in background.

Collection

A collection is a list of objects that have the same model. A collection is fetched with fetch() method.

For example, to get the list of all projects you have access to:

```
projects = ProjectCollection().fetch()
print(projects)
for project in projects:
        print(project)
        print(project.name)
```

The Collection class extends the Python MutableSequence class. You can thus work with a collection like with any Python list:

```
projects = ProjectCollection().fetch()
print(len(projects))
print(projects[0])
print([project.name for project in projects])
```

Filtering

The API often allows to filter a list of resources with an other resource, by specifiying it in the URI. The filtering by URI is performed using the fetch_with_filter() method. The list of allowed filters for a collection is given by mycollection.filters().

```
projects = ProjectCollection().fetch()
print(projects.filters()) # None, user, software, ontology

# None indicates the collection can be fetched without any filter
ProjectCollection().fetch()

# Get the list of projects of a particular user
id_user = 0
ProjectCollection().fetch_with_filter("user", id_user)

# Get the list of projects linked to a particular software
id_software = 0
ProjectCollection().fetch_with_filter("software", id_software)

# Get all projects that share the same ontology
id_ontology = 0
ProjectCollection().fetch_with_filter("ontology", id_ontology)
```

It will make the appropriate GET request, e.g. : GET /api/ontology/{id_ontology}/project.json

The behavior of collection requests can sometimes be modified with some query parameters. In the case of collections, these query parameters are the attributes of the Collection class. For example, we can obtain the list of online users with a query parameter:

```
UserCollection(online=True).fetch()
# Get the list of online users
# => GET /api/user.json?online=true

id_project = 0
UserCollection(online=True).fetch_with_filter("project", id_project)
# Get the list of online users on a particular project
# => GET /api/project/{id_project}/user.json?online=true
```

Particular case: a collection of annotations

A list of annotations can be filtered with many filters (project, user, image, term, ...). The only way given by the API to filter a list of annotations is to use query parameters.

```
id_project = 0
id_users = [1, 2]
annotations = AnnotationCollection(project=id_project, users=id_users, showWKT=True).fetch()
# => GET /api/annotation.json?showBasic=true&showMeta=true&showWKT=true&project=0&users=1,2
print(annotations)
```