xnatpy tutorial

June 10, 2016

1 xnatpy: a pythonic feeling interface to XNAT

xnatpy attempts to expose objects in XNAT as native feeling Python objects. The objects reflect the state of XNAT and changes to the objects automatically update the server.

To facilitate this xnatpy scans the server xnat.xsd and creates a Python class structure to mimic this is well as possible.

Current features: * automatic generate of most data structures from the xnat.xsd * easy exploration of data * easy getting/setting of custom variables * easy downloading/uploading of data * using the prearchive * the import service

Missing features (aka my TODO list): * good support for the creation of objects * good support for searches

1.0.1 Some imports and helper code used later on

```
In [1]: import os
        import random
```

1.1 getting started

First we need to set up an xnatpy session. The session scans the xnat.xsd, creates classes, logs in into XNAT, and keeps the connection alive using a hearbeat.

```
In [2]: import xnat
        session = xnat.connect('https://central.xnat.org', user='nosetests', passwo
```

```
[INFO] Retrieving schema from https://central.xnat.org/schemas/xnat/xnat.xsd
```

To save your login you set up a .netrc file with the correct information about the target host. A simple example of a .netrc file can be found at http://www.mavetju.org/unix/netrc.php.

It is possible to set the login information on connect without using a netrc file.

```
In [3]: xnat.connect?
```

The session is the main entry point for the module. It exposes part of the archive as objects.

```
In [4]: sandbox = session.projects['nosetests']
        print(sandbox)
```

```
<ProjectData nosetests>
```

```
In [5]: sandbox.description
```

```
Out[5]: 'Random_project_description_89'
```

```
In [6]: new_description = 'Random_project_description_{}'.format(random.randint(0,
        print('Changing description to: {}'.format(new_description))
        sandbox.description = new_description
```

```
Changing description to: Random_project_description_36
```

```
In [7]: sandbox.description
```

```
Out[7]: 'Random_project_description_36'
```

```
In [8]: # Get a list of the subjects
        sandbox.subjects
```

```
Out[8]: <XNATListing (CENTRAL_S04325, 5f45e50ffe5311e49e3fa8206633b88a): <SubjectDa
```

Note that the entries are in the form `(CENTRAL_S01824, custom_label): <SubjectData CENTRAL_S01824>`. This does not mean that the key is `(CENTRAL_S01824, custom_label)`, but that both the keys `CENTRAL_S01824` and `custom_label` can be used for lookup. The first key is always the XNAT internal id, the second key is defined as: * project: the name * subject: the label * experiment: the label * scan: the scantype * resource: label * file: filename

```
In [9]: subject = sandbox.subjects['bla_subject']
```

```
In [10]: print('Before change:')
         print('Gender: {}'.format(subject.demographics.gender))
         print('Initials: {}'.format(subject.initials))

         # Change gender and initials. Flip them between male and female and JC and
         subject.demographics.gender = 'female' if subject.demographics.gender ==
         subject.initials = 'JC' if subject.initials == 'PI' else 'PI'

         print('After change:')
         print('Gender: {}'.format(subject.demographics.gender))
         print('Initials: {}'.format(subject.initials))
```

```
Before change:
Gender: female
Initials: PI
After change:
Gender: male
Initials: JC
```

There is some basic value checking before assignment are carried out. It uses the xsd directives when available. For example:

```
In [11]: subject.demographics.gender = 'martian'


         ---------------------------------------------------------------------

         ValueError                                Traceback (most recent call last)

         <ipython-input-11-ef509675a021> in <module>()
    ----> 1 subject.demographics.gender = 'martian'


         /tmp/tmp9LFxKr_generated_xnat.py in gender(self, value)
        6971         # Restrictions for value
        6972         if value not in ["male", "female", "other", "unknown", "M", "F"
    -> 6973             raise ValueError('gender has to be one of: "male", "female"
        6974
        6975         # Generate automatically, type: xs:string


         ValueError: gender has to be one of: "male", "female", "other", "unknown",
```

1.2 Custom Variables

In xnatpy custom variables are exposed as a simple mapping type that is very similar to a dictionary.

```
In [12]: print(subject.fields)

<VariableMap {u'test_field': u'1337'}>


In [13]: # Add something
         subject.fields['test_field'] = 42
         print(subject.fields)

<VariableMap {u'test_field': u'42'}>


In [14]: subject.fields['test_field']

Out[14]: u'42'
```

Note that custom variables are always stored as string in the database. So the value is always casted to a string. Also the length of a value is limited because they are passed on the requested url.

The custom variables are by default not visible in the UI, there are special settings in the UI to make them appear. Defined variables in the UI that are not set, are just not appearing in the fields dictionary.

To avoid `KeyError`, it would be best to use `subject.fields.get('field_name')` which returns None when not available.

1.2.1 Downloading stuff

Downloading with xnatpy is fairly simple. Most objects that are downloadable have a `.download` method. There is also a `download_dir` method that downloads the zip and unpacks it to the target directory.

```
In [15]: download_dir = os.path.expanduser('~/xnatpy_temp')
         print('Using {} as download directory'.format(download_dir))
         if not os.path.exists(download_dir):
             os.makedirs(download_dir)
         sandbox.subjects['515502d2da9111df848d001c2304e1df'].download_dir(downloac
```

```
Using /home/hachterberg/xnatpy_temp as download directory
Downloaded subject to /home/hachterberg/xnatpy_temp/515502d2da9111df848d001c2304e1c
```

1.2.2 Close the session!

```
In [16]: # Don't forget to disconnect to close cleanly and clean up temporary thing
         session.disconnect()
```

1.3 Context operator

It is also possible to use xnatpy in a context, which guarantees clean closure of the connections etc.

```
In [ ]: with xnat.connect('https://central.xnat.org', user='nosetests', password='n
            print('Nosetests project description: {}'.format(session.projects['nose

        # Here the session will be closed properly, even if there were exceptions w
```

1.4 Ideas for the future

Currently I am thinking on two different additions and how to implement that best. * Creation of new objects * XNAT searches

This illustrates my current ideas, but these are not implemented yet. If you have an opinion about this, let me know!

Object creation xnat.SubjectData(parent=projectA, initials='X') or alternatively projectA.subjects['subjectX'] = xnat.SubjectData(initials='X')

Searches The idea is to create something similar to SQLAlchemy xnat.MrSessionData.query(session).filter(xnat.MrSessionData.age > 65, xnat.MrSessionData.age <= 80).all()