# qixnat

***Release***

**Jun 15, 2017**

# Contents

CHAPTER 1

# Synopsis

qixnat provides a XNAT facade.

**API** http://qixnat.readthedocs.org/en/latest/api/index.html

**Git** https://github.com/ohsu-qin/qixnat

CHAPTER 2

# Feature List

1. XNAT facade API.
2. XNAT list, transfer and delete utilities.

CHAPTER 3

# Installation

qixnat depends on the pyxnat module, which can in turn can be tricky to install. Furthermore, the pyxnat installation guide is itself insufficient to install pyxnat dependencies consistently in all environments. Consequently, the following installation steps must be followed to ensure a correct build:

1. On Linux only, install the libxslt dev package. For Ubuntu or other Debian-based systems, execute:

   ```
   sudo aptitude install libxslt-dev
   ```

   For Red Hat, execute:

   ```
   sudo yum install libxslt-dev
   ```

2. Anaconda is recommended for ensuring package and library consistency. Install Anaconda in $HOME/anaconda on your workstation according to the Anaconda Installation Instructions. Preferably, install the Anaconda Accelerate add-on as well. Note that a Anaconda Accelerate Academic User License is available.

3. Add $HOME/anaconda/bin to your PATH, if necessary:

   ```
   export PATH=$HOME/anaconda/bin:$PATH
   ```

4. Make an Anaconda virtual environment initialized with pip, e.g.:

   ```
   conda create --name qin pip
   ```

5. Activate the Anaconda environment, e.g.:

   ```
   source activate qin
   ```

   As a convenience, you can initialize this environment at login by prepending Anaconda and your virtual environment to PATH in your shell login script. E.g., for Linux or Mac OS X, open an editor on $HOME/.bashrc or $HOME/.bash_profile and add the following lines:

   ```
   # Prepend the Anaconda base applications.
   export PATH=$HOME/anaconda/bin:$PATH
   ```

```
# Prepend the virtual environment.
source activate qin
```

and refresh your environment:

```
. $HOME/.bash_profile
```

6. Install the `qixnat` dependencies hosted by Anaconda:

```
wget -q --no-check-certificate -O \
  - https://www.github.com/ohsu-qin/qixnat/raw/master/requirements_conda.txt \
  | xargs conda install --yes
```

7. On Mac only, reinstall the lxml package using the `-f` force option to work around a lxml install issue:

```
conda install -f lxml
```

8. Install the `qixnat` package:

```
pip install qixnat
```

CHAPTER 4

# Usage

Run the following command for the utility options:

```
cpxnat --help
lsxnat --help
rmxnat --help
```

The primary read API interface of interest is the XNAT facade class.

CHAPTER 5

# Development

See the qipipe Development Guide for project download, testing and documentation.

## API Documentation

### command

Command XNAT options.

qixnat.command.**add_options**(*parser*)

Adds the logging, project and config options to the given command line arugment parser.

**Parameters** **parser** – the Python `argparse` parser

### configuration

qixnat.configuration.**load**(*config=None*)

Loads the configuration as follows:

- If the *config* parameter is set, then that file is loaded.
- Otherwise, if there is a default configuration file as specified below, then the default file is loaded.
- Otherwise, this method returns an empty dictionary

The default configuration file is the first file found in the following precedence order:

1. The `XNAT_CFG` environment variable, if it is set.
2. `xnat.cfg` in the current working directory
3. `xnat.cfg` in the home `.xnat` subdirectory
4. `xnat.cfg` in the home directory

5. `xnat.cfg` in the `/etc` directory

**Parameters** **config** – the configuration file location

**Returns** the configuration dictionary

## connection

`qixnat.connection.`**connect**(**args*, ***kwds*)

Yields a *qixnat.facade.XNAT* instance.

If this is the first connect call, then this method yields a new XNAT instance which connects to the XNAT server. Otherwise, this method yields the existing XNAT instance. The XNAT connection is closed when the outermost connection block finishes.

The new XNAT connection is established as follows:

If the *config* parameter is set, then the configuration options in that file take precedence over the *opts* options.

Unlike pyxnat, a configuration loaded from the *config* file only needs to specify the connection arguments which differ from the `pyxnat.Interface` default values.

Furthermore, if the options do not include a *cachedir*, then this method sets the *cachedir* option to a new temp directory. When the connection is closed, this directory is deleted.

**Note** If a shared *cachedir* is used in a cluster environment, then concurrency cache conflicts can arise because the pyxnat cache is non-reentrant and unsynchronized (cf. the `qixnat.facade.find()` Note).

The caller is required to either not set the *cachedir* option or set the *cachedir* to a location that is unique for each execution process. This practice ensures cache consistency in a cluster environment.

This practice differs from the standard `pyxnat` configuration file. `pyxnat` load of a configuration file without a *cachedir* option results in an error. By contrast, `qixnat` load of a configuration file without a *cachedir* option results in a new temp cache directory.

Example:

```
>>> import qixnat
>>> with qixnat.connect() as xnat:
...     subject = xnat.find_one('QIN', 'Breast003')
```

**Parameters**

- **config** – the XNAT configuration file, or None for the *qixnat.configuration.load()* default
- **opts** – the *qixnat.facade.XNAT* initialization options

**Yield** the XNAT instance

## constants

XNAT object model constants.

## facade

**class** qixnat.facade.**XNAT**(**opts*)
Bases: *object*

XNAT is a pyxnat facade convenience class. An XNAT instance is created in a `connection()` context, e.g.:

```
from qiutil import qixnat
with qixnat.connect() as xnat:
    sbj = xnat.find('QIN_Test', 'Breast003')
```

This XNAT wrapper class implements methods to access XNAT objects in a hierarchical name space using a labeling convention. The access method parameters are XNAT hierarchy object name values. Here, the *name* refers to a designator for an XNAT object that is unique within the scope of the object parent. For non-scan XNAT objects, the name is the ending portion of the XNAT label that strips out the parent label. Since the XNAT scan label is conventionally the scan number as a string, the scan name converts the label to an integer, i.e. the scan name is the scan number designator.

The name parameters are used to build the XNAT label, as shown in the following example:

| Class | Name | Label |
|---|---|---|
| Project | QIN | QIN |
| Subject | Breast003 | Breast003 |
| Experiment | Session01 | Breast003_Session01 |
| Scan | 1 | 1 |
| Reconstruction | reco_fTkr4Y | Breast003_Session01_reco_fTkr4Y |
| Assessor | pk_4kbEv3r | Breast003_Session01_pk_4kbEv3r |
| Resource | reg_zaK1Bd | reg_zaK1Bd |
| File | volume3.nii | volume3.nii |

Table 1. Example XNAT label generation.

**Note** The XNAT Reconstruction data type is deprecated. An experiment or scan Resource should be used instead.

**Note** An XNAT id and label is a string. Although the scan label is customarily the scan number, pyxnat does not support an integer label. By contrast, this *XNAT* class allows an integer name and converts it to a string to perform a XNAT search.

The XNAT label is set by the user and conforms to the following uniqueness constraints:

- the Project label is unique within the database.
- the Subject, Experiment, Assessor and Reconstruction label is unique within the Project.
- the Scan label is unique within the scope of the Experiment.
- the Resource label is unique within the scope of its parent container.
- the File label is unique within the scope of its parent Resource.

All but one constraint violation results in a `DatabaseError` with REST HTTP status 409 or 500. The one constraint violation which does not raise an exception results in the following data anomaly:

- If a *subject1* Experiment is created with the same label as a *subject2* Experiment in the same Project, then the *subject1* Experiment is moved to *subject2*, e.g.:

```
>>> from qixnat.facade import XNAT
>>> xnat = XNAT().interface
>>> sbj1 = xnat.select('/project/QIN_Test/subject/Breast003')
```

```
>>> exp1 = sbj1.experiment('Session01').create()
>>> sbj2 = xnat.select('/project/QIN_Test/subject/Breast004')
>>> exp2 = sbj2.experiment('Session01').create()
>>> exp1.exists()
False
```

The pyxnat access methods specify an *id* parameter, but in fact either the id or label can be used for the *id* parameter, consistent with the XNAT REST interface, e.g.:

```
>>> from qixnat.facade import XNAT
>>> xnat = XNAT().interface
>>> sbj1 = xnat.select('/project/QIN_Test/subject/Breast003')
>>> exp1 = sbj1.experiment('Breast003_Session01')
>>> if exp1.exists():
...     exp1.delete()
>>> exp1.id() == None
True
>>> exp1.label() == None
True
>>> exp1.create().id() == None
False
>>> exp1.label()
'Breast003_Session01'
>>> exp2 = sbj1.experiment(exp1.id())
>>> exp1.label() == exp2.label()
True
```

XNAT generates the id as follows:

- The Project, Scan and File id and label are identical
- A Reconstruction does not have an XNAT id
- The Subject id is an opaque XNAT-generated value starting with *project*_S.
- The Experiment and Assessor id is an opaque XNAT-generated value starting with *project*_E.
- The Resource is an opaque XNAT-generated string value consisting of digits.

The Project and opaque XNAT-generated identifiers are unique within the database. The Scan and File id is unique within its parent.

The following table shows example XNAT ids and labels:

| Class | Id | Label |
|---|---|---|
| Project | QIN | QIN |
| Subject | QIN_S00580 | Breast003 |
| Experiment | QIN_E00604 | Breast003_Session01 |
| Scan | 1 | 1 |
| Reconstruction | – | Breast003_Session01_reco_fTkr4Y |
| Assessor | QIN_E00868 | Breast003_Session01_pk_4kbEv3r |
| Resource | 3187 | reg_zaK1Bd |
| File | image3.nii | image3.nii |

Table 2. Example XNAT ids and labels.

In the example above, the XNAT assessor object is obtained as follows:

```
>>> import qixnat
>>> with qixnat.connect() as xnat:
```

```
...     recon = xnat.find_one('QIN', 'Breast003', 'Session01',
...                           assessor='pk_4kbEv3r')
```

XNAT files are always placed in an existing XNAT resource, e.g.:

```
import qixnat
with qixnat.connect() as xnat:
    rsc = xnat.find_or_create('QIN', 'Breast003', 'Session01',
                              scan=1, resource='DICOM')
    xnat.upload(rsc, *dicom_files)
```

**Parameters** **opts** – the XNAT configuration options

**__init__**(***opts*)

**Parameters** **opts** – the XNAT configuration options

**close**()

Drops the XNAT connection.

**delete**(**args*, ***opts*)

Deletes the XNAT objects which match the given search criteria. The object search is described in *find()*.

**Note** XNAT project deletion is unsupported.

**Note** pyxnat file object deletion is unsupported.

**Note** XNAT delete is recursive. In particular, all files contained in the deletion target are deleted. Use this method with caution.

Examples:

```
from qiutil import qixnat
with qixnat.connect() as xnat:
    # Delete all resources which begin with 'pk_'.
    xnat.delete('QIN', 'Sarcoma003', 'Session01', scan=1,
                        resource='pk_*')
```

**Parameters**

- **args** – the *find()* positional search key
- **opts** – the *find()* keyword hierarchy options search key

**Raises** **XNATError** – if a project or file object is specified

**download**(**args*, ***opts*)

Downloads the files contained in XNAT resource or resources. The parameters and options specify the target XNAT file search condition, as described in *find()*, augmented as follows: - if there is no *resource* option, then all resources

are included

•if there is no *file* option, then all files in the selected resources are downloaded

Example:

```
download('QIN', 'Breast001', '*', scan=1, resource='reg_*')
```

downloads the files for all `QIN` `Breast001` scan `1` resources whose label begins with `reg_`.

**Parameters**

- **project** – the XNAT project id
- **subject** – the XNAT subject name
- **experiment** – the XNAT experiment name
- **opts** – the *find()* hierarchy and *download_file()* options, as well the following option:
- **dest** – the optional download location (default current directory)

**Raises** **XNATError** – if the options do not specify a resource

**Returns** the downloaded file names

**download_file**(*file_obj*, *dest*, ***opts*)

Downloads the given XNAT file to the target directory.

**Parameters**

- **file_obj** – the XNAT File object
- **dest** – the required target directory
- **opts** – the following option:
- **skip_existing** – ignore the source XNAT file if it a file of the same name already exists at the target location (default False)
- **force** – overwrite existing file (default False)

**Returns** the downloaded file path

**Raises**

- **XNATError** – if both the *skip_existing force* options are set
- **XNATError** – if the XNAT file already exists and the *force* option is not set

**find**(**args*, ***opts*)

Finds the XNAT objects which match the given search criteria.

The positional parameters and keyword options extend the *object()* interface to allow glob wildcards (`*`). A key in the hierarchy which contains a wildcard matches those XNAT objects whose *qixnat.helpers.xnat_name()* match the wildcard expression.

The return value is a list of those XNAT objects which match the specification and exist in the database.

Examples:

```
>>> from qiutil import qixnat
>>> with qixnat.connect() as xnat:
...     subjects = xnat.find('QIN', 'Sarcoma*')
...     scan = xnat.find('QIN', 'Sarcoma003', '*', scan=1)
```

**Parameters**

- **args** – the *object()* positional search keys
- **opts** – the *object()* keyword hierarchy options search key

**Returns** the XNAT objects

**find_one**(**args*, ***opts*)

Finds the XNAT object which match the given search criteria. as described in *find()*. Unlike *find()*, this find_one method returns a single object, or None if there is no match.

Examples:

```
from qiutil import qixnat
with qixnat.connect() as xnat:
    subject = xnat.find_one('QIN', 'Sarcoma003')
    scan = xnat.find_one('QIN', 'Sarcoma003', 'Session01', scan=1)
    file_obj = xnat.find_one('QIN', 'Sarcoma003', 'Session01',
                             scan=1, resource='NIFTI',
                             file='image12.nii.gz')
```

**Parameters**

- **args** – the *find()* positional search key
- **opts** – the *find()* keyword hierarchy options search key

**Returns** the matching XNAT object, or None if not found

**find_or_create**(**args*, ***opts*)

Extends *find_one()* to create the object if it doesn't exist. If the target object doesn't exist, then every non-existing object in the hierarchy leading to and including the target object is created.

The *find()* hierarchy keyword options are extended with the scan *modality* keyword option, e.g. `MR` or `CT`. The modality argument is case-insensitive. Direct or indirect creation of an experiment or scan requires the *modality* option.

The positional parameters and hierarchy keyword options extend the *find()* interface to allow either the XNAT search key or a (search key, non-key {attribute: value}) tuple. The non-key content is set if and only if the object in the hierarchy specified by the option is created.

Examples:

```
from qiutil import qixnat
with qixnat.connect() as xnat:
    date = datetime(2004, 12, 3)
    exp_opt = (('Session01', dict(date=date))
    experiment = find_or_create('QIN', 'Breast003',
                                exp_opt, modality='MR')

    scan_opt = (1, dict(series_description='T1'))
    resource = xnat.find_or_create(
        'QIN', 'Sarcoma003', 'Session02', scan=scan_opt,
        resource='NIFTI', modality='MR'
    )
```

In the previous example, if the ancestor XNAT scan 1 and resource 'NIFTI' don't exist, then they are created. The *series_description* attribute is set if and only if the XNAT scan object is created.

The supported non-key attributes are defined by the XNAT schema. For example, the standard non-key scan attributes are as follows: * note * quality * condition * series_description * documentation

This method accomodates the following pyxnat oddities:

- The pyxnat convention is to specify a modality-specific subtype as a {pluralized type name: REST type} option, where the option key is pluralized, e.g. `experiments`, and the option value is the *qixnat.helpers.rest_type()*.
- The pyxnat idiom is to set non-key attributes on a create by the options {<REST type>/<attribute>: value}.
- The pyxnat date setter argument is a XML Schema xs:date string with format YYYY-MM-DD. pyxnat does not convert a date value to a Python datetime.

This *find_or_create()* method handles these and other pyxnat create idiosyncracies described in the pyxnat operations guide.

See the note below for important information about XNAT object creation in a cluster or other concurrent processing environment.

It is an error to use this method to create a *file* object. The *upload()* method is used for this purpose instead.

**Note** Concurrent XNAT object find-or-create fails unpredictably, possibly arising from one of the following causes: * the pyxnat config in $HOME/.xnat/xnat.cfg specifies a temp

directory that *is not* shared by all concurrent jobs, resulting in inconsistent cache content

- the pyxnat config in $HOME/.xnat/xnat.cfg specifies a temp directory that *is* shared by some concurrent jobs, resulting in unsynchronized pyxnat write conflicts across jobs
- the non-reentrant pyxnat's custom non-http2lib cache is corrupted
- an XNAT archive directory access race condition

Update 05/12/2015 - there are two potential failure points: * Concurrent pyxnat cache access corrupts the cache resulting in

unpredictable errors, e.g. attempt to create an existing XNAT object

- Concurrent XNAT resource file upload corrupts the archive such that the files are stored in the archive but are not recognized by XNAT

Update 05/14/2015 - per https://github.com/pyxnat/pyxnat/issues/61, a shared pyxnat cache is a likely point of failure. However, sporadic failures also occur without a shared cache. The following practices are recommended: * set an isolated pyxnat cache_dir for each execution context * serialize common XNAT object find-or-create access across all

concurrent execution contexts

**Parameters**

- **args** – the *find()* positional search key or (search key, {attribute: value}) tuple
- **opts** – the *find()* keyword hierarchy options search key or (search key, {attribute: value}), as well as the following create options:
- **modality** – the scan modality (default `MR`)

**Raises** **XNATError** – if the options specify a non-existing XNAT file object

**find_path**(*path*)

Returns the XNAT object children in the given XNAT object path. The *path* string argument must conform to the *qixnat.helpers.path_hierarchy()* requirements.

**Parameters** **path** – the path string

**Returns** the XNAT child label list

**Raise** XNATError if there is no such child

**object** (*project*, *subject=None*, *experiment=None*, ***opts*)

Return the XNAT object with the given search specification. The parameters and options specify a XNAT object in the XNAT REST hierarchy. This hierarchy is summarized as follows:

```
/project/PROJECT/
    [subject/SUBJECT/
        [experiment/EXPERIMENT/
            [<container type>/CONTAINER]]]
                [resource/RESOURCE
                    [file/FILE]]
```

where *container type* is the experiment child type, e.g. `scan`. The brackets in the hierarchy specification above denote optionality, namely: - Only the project is required. - If there is an experiment, then there must be a subject. - If there is a container, then there must be an experiment. - A resource can be associated with any ancestor type. - A file requires a resource.

The positional parameters specify the XNAT project/subject/experiment hierarchy from the root project down to and including the XNAT experiment object. The keyword arguments specify the object hierarchy below the experiment, e.g. `scan=1`. The experiment child can be a `scan`, `reconstruction` or `assessment`. The `assessment` container type value corresponds to the XNAT `assessor` Image Assessment type. `analysis`, `assessment` and `assessor` are synonymous. A resource is specified by the *resource* keyword.

The positional parameter and keyword option values are the XNAT object names, as described in the *qixnat.facade.XNAT* class documentation. The names are prefixed, if necessary, by *qixnat.helpers.hierarchical_label()* to form the standard XNAT label.

The XNAT object need not exist in the database. By contrast, the *find_one()* method returns an object if and only if it exists in the database.

Examples:

```
>>> from qiutil import qixnat
>>> with qixnat.connect() as xnat:
...     subject = xnat.object('QIN', 'Sarcoma003')
```

**Parameters**

- **project** – the XNAT project id or (id, {attribute: value}) tuple
- **subject** – the XNAT subject name or (name, {attribute: value}) tuple
- **experiment** – the XNAT experiment name or (name, {attribute: value}) tuple
- **opts** – the following container options:
- **scan** – the scan number or (number, {attribute: value}) tuple
- **reconstruction** – the reconstruction name or (name, {attribute: value}) tuple
- **analysis** – the analysis name or (name, {attribute: value}) tuple
- **resource** – the resource name or (name, {attribute: value}) tuple
- **in_resource** – the XNAT *in_resource* name or (name, {attribute: value}) tuple
- **out_resource** – the XNAT *out_resource* name or (name, {attribute: value}) tuple
- **file** – the file name

- **inout** – the resource direction (`in` or `out`)

**Returns** the XNAT object, which may not exist

**update**(*obj*, ***mods*)

Sets the given attributes and saves the object.

This method handles python datetime values correctly by working around the pyxnat date oddity described in **:mneth:'find_or_create'**.

**Parameters**

- **obj** – the object to change
- **mods** – the {attribute, value} modifications

**upload**(*resource*, **in_files*, ***opts*)

Imports the given files into XNAT. The parameters and options specify a target XNAT resource object in the XNAT object hierarchy, as described in *find()*. The resource and its hierarchy ancestor objects are created as necessary by the *find()* method.

Example:

```
from glob import glob
from qiutil import qixnat
in_files = glob('/path/to/images/*.nii.gz')
with qixnat.connect() as xnat:
    rsc = xnat.find_or_create(
        'QIN', 'Sarcoma003', 'Session01', scan=4,
        resource='NIFTI', modality='MR'
    )
    xnat.upload(rsc, *in_files)
```

**Parameters**

- **resource** – the existing XNAT resource object
- **in_files** – the input files to upload
- **opts** – the following keyword options:
- **name** – the XNAT file object name (default is the input file base name)
- **skip_existing** – flag indicating whether to forego overwriting an existing file (default False)
- **force** – flag indicating whether to replace an existing file (default False)

**Returns** the new XNAT file names

**Raises**

- **XNATError** – if there are no input files
- **XNATError** – if the input file does not exist
- **XNATError** – if both the *skip_existing force* options are set
- **XNATError** – if the XNAT file already exists and neither the *skip_existing* nor the *force* option is set

## helpers

qixnat.helpers.**hierarchical_label**(**names*)

Returns the XNAT label for the given hierarchical name, qualified by a prefix if necessary.

Example:

```
>>> from qixnat.helpers import hierarchical_label
>>> hierarchical_label('Breast003', 'Session01')
'Breast003_Session01'
>>> hierarchical_label('Breast003', 'Breast003_Session01')
'Breast003_Session01'
>>> hierarchical_label(3)   # for scan number 3
3
```

**Parameters** **names** – the object names

**Returns** the corresponding XNAT label

qixnat.helpers.**is_pluralized_type_designator**(*designator*)

**Parameters** **designator** – the XNAT type name or synonym

**Returns** whether the designator is a pluralized XNAT type designator

qixnat.helpers.**parse_xnat_date**(*value*)

**The XNAT REST client unfortunately returns date fields as a string.** experiment must have a date. This method converts the string input to a datetime.

**Parameters** **value** – the input string in `qixnat.constants.DATE_FMT` format or None

**Returns** None, if the input is None, otherwise the input parsed as a datetime object

**Return type** datetime.datetime

qixnat.helpers.**path_hierarchy**(*path*)

Transforms the given XNAT path into a list of *(type, value)* tuples.

The *path* string argument must consist of a sequence of slash-delimited XNAT object specifications, where each specification is either a singular XNAT type and value, e.g. `subject/Breast003`, or a pluralized XNAT type, e.g. `resources`.

The path can include wildcards, e.g. `/project/QIN/subject/Breast*`.

If the path starts with a forward slash, then the first three components can elide the XNAT type. Thus, the following are equivalent:

```
path_hierarchy('/project/QIN/subject/Breast003/experiment/Session02')
path_hierarchy('/QIN/Breast003/Session02')
```

The following XNAT object type synonyms are allowed: * `session` => `experiment` * `analysis` or `assessment` => `assessor` Pluralized type synonyms are standardized according to the singular form, e.g. `analyses` => `assessors`.

The path hierarchy result is a list of *(type, value)* tuples. A pluralization value is a wild card.

Examples:

```
>>> from qixnat.helpers import path_hierarchy
>>> path_hierarchy('/QIN/Breast003/Session03/resource/reg_Qzu7R/files')
[('project', 'QIN'), ('subject', 'Breast*'), ('project', 'QIN'),
 ('subject', 'Breast*'), ('experiment', 'Session03'),
 ('resource', 'reg_Qzu7R'), ('file', '*')]
>>> path_hierarchy('/QIN/Breast*/*/resources')
[('project', 'QIN'), ('subjects, 'Breast*'), ('experiments, '*'),
 ('resource', '*')]
```

**Parameters** **path** – the XNAT object path string or list

**Returns** the path hierarchy list

**Return type** list

qixnat.helpers.**pluralize_type_designator**(*designator*)

**Parameters** **designator** – the XNAT type name or synonym

**Returns** the pluralized type designator

qixnat.helpers.**rest_date**(*value*)

**Parameters** **value** – the input `datetime` object or None

**Returns** None, if the input is None, otherwise the input formatted as a string using the `qixnat.constants.DATE_FMT`

**Return type** str

qixnat.helpers.**rest_type**(*type_name*, *modality=None*)

Qualifies the given type name with the modality, e.g.:

```
>>> from qixnat.helpers import rest_type
>>> rest_type('experiment', 'MR')
'xnat:mrSessionData'
```

**Parameters**

- **type_name** – the XNAT type name
- **modality** – the case-insensitive modality, e.g. `MR` or `CT`

**Returns** the full XNAT subtype designation

**Raises** **XNATError** – if the type name is in `qixnat.constants.MODALITY_TYPES` but modality is None

qixnat.helpers.**xnat_children**(*obj*)

Returns the XNAT objects contained in the given parent object.

**Parameters** **obj** – the XNAT parent object

**Returns** the child objects

**Return type** list

qixnat.helpers.**xnat_key**(*obj*)

Returns the XNAT object key unique within the parent scope, determined as follows: * If the object is a Reconstruction, then the XNAT id * Otherwise, the XNAT label

**Parameters** **obj** – the XNAT object

**Returns** the XNAT label or id

qixnat.helpers.**xnat_name**(*obj*)

Returns the canonical XNAT object name determined as the `xnat_key()` with the parent key prefix removed, if necessary, e.g.:

```
>> xnat_key(session)
```

Breast003_Session01 >> xnat_name(session) Session01

The scan name is an integer, the other names are strings.

**Parameters** **obj** – the XNAT object

**Returns** the canonical XNAT name

qixnat.helpers.**xnat_path**(*obj*)

Returns the XNAT object path in the canonical form:

/<project>[/<subject>[/<session>[/type/<name>]*]]

e.g.:

```
/QIN/Breast003/Session02/scan/1/resource/NIFTI/file/volume001.nii.gz
```

**Parameters** **obj** – the XNAT object

**Returns** the XNAT path

# Python Module Index

## h



## q

# Index

## Symbols



## A



## C



## D



## F



## H



## I



## L



## O



## P



## Q



## R



## U



## X