Exploring ConnectomeDB with Python

All of the publicly available data from the Human Connectome Project are accessible through ConnectomeDB, a web site built on the XNAT imaging informatics platform. In addition to the imaging data, a great deal of imaging metadata and associated non-imaging data are stored on ConnectomeDB.

pyxnat is a library that provides a Python language API to XNAT's RESTful web services. In this tutorial, we'll use pyxnat to access behavioral measures stored in ConnectomeDB and to view and download the imaging data. Even if you're not a Pythonista, read on, as the underlying XNAT REST API can be accessed from just about any language. I have small examples of code using the REST API in bash, Java , Clojure, and Haskell, and I'd probably find it amusing to cook up an example in your favorite language; send me mail if you'd like details.

Getting started

You'll need Python (version 2.7.x recommended) and pyxnat to follow along. I've been adding features (some HCP-specific) and fixing bugs in pyxnat, so I recommend creating a Python virtual environment and installing our customized version of pyxnat. I'm writing this using Python 2.7.1 on Mac OS X 10.7.5, but I regularly use pyxnat on Gentoo Linux; other people use pyxnat on other Linuxes and even Windows. It's also possible to use pyxnat on an Amazon EC2 instance. In principle, this all should work just about anywhere you can run Python, but send me mail if you run into trouble.

You'll also need to create an account on ConnectomeDB and agree to the HCP Open Access Data Use Terms.

We'll look at some behavioral measures in ConnectomeDB: the Non-Toolbox Data Measures, a variety of tests that aren't part of the NIH Toolbox. The non-Toolbox measures are documented in detail here. nontoolbox.xsd is an XML Schema document that specifies the non-Toolbox data type in ConnectomeDB; it's not particularly readable, but it does provide the exact naming conventions used in ConnectomeDB.

Let's start by firing up a Python session, loading pyxnat, and setting up a connection to ConnectomeDB.

```
bash-3.2$ python
Python 2.7.1 (r271:86832, Jul 31 2011, 19:30:53)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on
darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyxnat
>>> cdb=pyxnat.Interface('https://db.humanconnectome.
org','mylogin','mypasswd')
>>>
```

This Interface object creates a session on ConnectomeDB. Be warned: if the session is idle for a while – say, for example, you're too busy reading documentation to keep typing -- ConnectomeDB may close the session. You can tell that the session has gone stale if, when you try to do a query:

```
>>> cdb.select.project('HCP_Q2').subject('100307').id()
```

you get a plateful of nonsense that looks like:

```
['status', 'content-location', 'content-language', ...
200
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
...
```

The same error occurs if you provide the wrong username or password.

If this happens, whether due to timeout or mistyping, just create a new Interface:

Unknown macro: 'html'

Table of Contents

Unknown macro: 'html'

```
>>> cdb=pyxnat.Interface('https://db.humanconnectome.
org','mylogin','mypasswd')
```

Any query result objects that you created from the stale Interface will also need to be refreshed. There's an example later in this tutorial.

Exploring the ConnectomeDB data hierarchy

ConnectomeDB's data is organized into *projects*, which are the main access control structure in XNAT. If you have access to a project, you can see that project's data. Let's see what projects we have access to:

```
>>> cdb.select.projects().get()
['HCP_Q1','HCP_Q2']            # maybe others, depending on your access
settings
>>>
```

`cdb.select.projects()` asks ConnectomeDB for project details and turns the result into a collection of project objects. The `get()` method returns the identifiers for each object in the collection. We could get the same result using a list comprehension; let's try that now, because that will be a more convenient form in general:

```
>>> [project.id() for project in cdb.select.projects()]
['HCP_Q1', 'HCP_Q2', ...]
>>>
```

Let's use the project for the current (Q2) data release, HCP_Q2. Here we get a handle on just that project:

```
>>> q2 = cdb.select.project('HCP_Q2')
>>>
```

Note that if the session goes stale, so will this object `q2`. So in addition to refreshing cdb, you'll probably need to refresh `q2`, too, by reissuing this command:

```
>>> q2 = cdb.select.project('HCP_Q2')
>>>
```

Querying for Subjects in the Q2 Project

What's inside of this project object? Each project contains *subjects* and *experiments*. Let's look at the list of subjects:

```
>>> [subject.label() for subject in q2.subjects()]
['100307', '103515', '103818', '111312', '114924', '117122', '118932',
'119833', '120212', '125525', '128632', '130013', '137128', '138231',
'142828', '143325', '144226', '149337', '150423', '153429', '156637',
'159239', '161731', '162329', '167743', '172332', '182739', '191437',
'192439', '192540', '194140', '197550', '199150', '199251', '200614',
'201111', '210617', '217429', '249947', '250427', '255639', '304020',
'307127', '329440', '499566', '530635', '559053', '585862', '638049',
'665254', '672756', '685058', '729557', '732243', '792564', '826353',
'856766', '859671', '861456', '865363', '877168', '889579', '894673',
'896778', '896879', '901139', '917255', '937160', '131621', '355542',
'611231', '144428', '230926', '235128', '707244', '733548', '103414',
'209733', '212318', '105115', '214019', '110411', '214221', '113619',
'214423', '115320', '221319', '116120', '298051', '118730', '293748',
'123117', '124422', '397760', '129028', '414229', '133827', '133928',
'134324', '448347', '135932', '485757', '136833', '528446', '139637',
'552544', '579665', '140420', '581349', '149539', '598568', '151223',
'151627', '627549', '157336', '645551', '158035', '654754', '677968',
'158540', '163432', '702133', '169343', '734045', '175439', '748258',
'177746', '753251', '182840', '788876', '185139', '193239', '857263',
'195647', '872158', '196144', '885975', '205119', '887373', '205725',
'932554', '207628', '984472', '992774']
>>>
```

We used `subject.label()` instead of `subject.id()`, which inside the list comprehension would have given the same result as `q2.get()`. Why `label()` instead of `id()`? The label is the human-readable name for the subject within a specified project (HCP_Q2 in our case); the first label in the list is 100307, which is the HCP-assigned name for that subject. The subject id is the XNAT site-wide unique identifer for that subject, a not-intended-for-human-consumption identifier; the id for subject 100307 is 'ConnectomeDB_S00230'. In principle, different projects might assign different labels to the same subject, or different subjects might share the same label in different projects. We aren't engaging in those sorts of shenanigans on ConnectomeDB, but we do inherit a little complexity from XNAT's flexibility.

Querying for Experiments for each Subject

What data are available for subject 100307? Let's ask:

```
>>> [expt.label() for expt in q2.subject('100307').experiments()]
['100307_3T', '100307_SubjMeta', '100307_Toolbox', '100307_NonToolbox',
'100307_Alertness']
>>>
```

There are five "experiments" here: 100307_3T contains the imaging data and associated metadata acquired on the HCP 3T Skyra; 100307_SubjMeta holds some bookkeeping about what data have been collected for this subject; 100307_Toolbox has NIH Toolbox scores, 100307_NonToolbox the non-Toolbox scores, and 100307_Alertness has scores for the Mini Mental Status Exam (MMSE) and Pittsburgh Sleep Questionnaire (PSQI). Again we use `label()` instead of `id()` (or `get()` on the experiments collection), because each project has a human-readable label for the experiment, whereas the id is the site-wide, XNAT-generated identifier.

Exploring Experiment Data

The individual experiments are represented by XML documents; we can view the XML for 100307_NonToolbox to see what's inside:

```
>>> nt_100307 = q2.subject('100307').experiment('100307_NonToolbox')
>>> print(nt_100307.get())
<?xml version="1.0" encoding="UTF-8"?>
<nt:NTScores ID="ConnectomeDB_E00299" project="HCP_Subjects" label="
100307_NonToolbox" xmlns:arc="http://nrg.wustl.edu/arc" xmlns:val="
http://nrg.wustl.edu/val" xmlns:pipe="http://nrg.wustl.edu/pipe" xmlns:
hcp="http://nrg.wustl.edu/hcp" xmlns:fs="http://nrg.wustl.edu/fs" xmlns:
wrk="http://nrg.wustl.edu/workflow" xmlns:scr="http://nrg.wustl.edu/scr"
xmlns:xdat="http://nrg.wustl.edu/security" xmlns:nt="http://nrg.wustl.edu
/nt" xmlns:cat="http://nrg.wustl.edu/catalog" xmlns:prov="http://www.nbirn.
net/prov" xmlns:xnat="http://nrg.wustl.edu/xnat" xmlns:xnat_a="http://nrg.
wustl.edu/xnat_assessments" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://nrg.wustl.edu/fs https://db.
humanconnectome.org/schemas/freesurfer/fs.xsd http://nrg.wustl.edu
/workflow https://db.humanconnectome.org/schemas/pipeline/workflow.xsd
http://nrg.wustl.edu/catalog https://db.humanconnectome.org/schemas/catalog
/catalog.xsd http://nrg.wustl.edu/pipe https://db.humanconnectome.org
/schemas/pipeline/repository.xsd http://nrg.wustl.edu/hcp https://db.
humanconnectome.org/schemas/HCPMetadata/metadata.xsd http://nrg.wustl.edu
/nt https://db.humanconnectome.org/schemas/nontoolbox/nontoolbox.xsd
http://nrg.wustl.edu/hcp https://db.humanconnectome.org/schemas/alertness
/alertness.xsd http://nrg.wustl.edu/scr https://db.humanconnectome.org
/schemas/screening/screeningAssessment.xsd http://nrg.wustl.edu/arc
https://db.humanconnectome.org/schemas/project/project.xsd http://nrg.
wustl.edu/hcp https://db.humanconnectome.org/schemas/toolbox/toolbox.xsd
http://nrg.wustl.edu/hcp https://db.humanconnectome.org/schemas/restricted
/restrictedASR.xsd http://nrg.wustl.edu/val https://db.humanconnectome.org
/schemas/validation/protocolValidation.xsd http://nrg.wustl.edu/xnat
https://db.humanconnectome.org/schemas/xnat/xnat.xsd http://nrg.wustl.edu
/hcp https://db.humanconnectome.org/schemas/restricted/restrictedTier1.xsd
http://nrg.wustl.edu/xnat_assessments https://db.humanconnectome.org
/schemas/assessments/assessments.xsd http://www.nbirn.net/prov https://db.
humanconnectome.org/schemas/birn/birnprov.xsd http://nrg.wustl.edu
/security https://db.humanconnectome.org/schemas/security/security.xsd">
<xnat:sharing>
<xnat:share label="100307_NonToolbox" project="HCP_Q2">
<!--hidden_fields[xnat_experimentData_share_id="233",
sharing_share_xnat_experimentDa_id="ConnectomeDB_E00299"]-->
</xnat:share>
<xnat:share label="100307_NonToolbox" project="HCP_Q1">
<!--hidden_fields[xnat_experimentData_share_id="234",
sharing_share_xnat_experimentDa_id="ConnectomeDB_E00299"]-->
</xnat:share>
</xnat:sharing>
<xnat:subject_ID>ConnectomeDB_S00230</xnat:subject_ID>
<nt:HCPPNP>
<nt:mars_log_score>1.76</nt:mars_log_score>
<nt:mars_errs>0</nt:mars_errs>
<nt:mars_final>1.76</nt:mars_final>
</nt:HCPPNP>
<nt:DDISC>
<nt:SV_1mo_200>103.13</nt:SV_1mo_200>
<nt:SV_6mo_200>46.88</nt:SV_6mo_200>
<nt:SV_1yr_200>103.13</nt:SV_1yr_200>
<nt:SV_3yr_200>21.88</nt:SV_3yr_200>
<nt:SV_5yr_200>21.88</nt:SV_5yr_200>
<nt:SV_10yr_200>9.38</nt:SV_10yr_200>
<nt:SV_1mo_40000>19375.0</nt:SV_1mo_40000>
<nt:SV_6mo_40000>29375.0</nt:SV_6mo_40000>
<nt:SV_1yr_40000>24375.0</nt:SV_1yr_40000>
<nt:SV_3yr_40000>9375.0</nt:SV_3yr_40000>
<nt:SV_5yr_40000>9375.0</nt:SV_5yr_40000>
<nt:SV_10yr_40000>9375.0</nt:SV_10yr_40000>
<nt:AUC_200>0.16217604</nt:AUC_200>
<nt:AUC_40000>0.31145853</nt:AUC_40000>
</nt:DDISC>
<nt:NEO>
<nt:NEO>144</nt:NEO>
<nt:NEOFAC_A>33</nt:NEOFAC_A>
<nt:NEOFAC_O>24</nt:NEOFAC_O>
```

```
<nt:NEOFAC_C>35</nt:NEOFAC_C>
<nt:NEOFAC_N>15</nt:NEOFAC_N>
<nt:NEOFAC_E>37</nt:NEOFAC_E>
</nt:NEO>
<nt:SPCPTNL>
<nt:SCPT_TP>59</nt:SCPT_TP>
<nt:SCPT_TN>115</nt:SCPT_TN>
<nt:SCPT_FP>5</nt:SCPT_FP>
<nt:SCPT_FN>1</nt:SCPT_FN>
<nt:SCPT_TPRT>412.0</nt:SCPT_TPRT>
<nt:SCPT_SEN>0.9833</nt:SCPT_SEN>
<nt:SCPT_SPEC>0.9583</nt:SCPT_SPEC>
<nt:SCPT_LRNR>11</nt:SCPT_LRNR>
</nt:SPCPTNL>
<nt:CPW>
<nt:IWRD_TOT>35</nt:IWRD_TOT>
<nt:IWRD_RTC>1442.0</nt:IWRD_RTC>
</nt:CPW>
<nt:PMAT24A>
<nt:PMAT24_A_CR>17</nt:PMAT24_A_CR>
<nt:PMAT24_A_SI>2</nt:PMAT24_A_SI>
<nt:PMAT24_A_RTCR>11839.0</nt:PMAT24_A_RTCR>
</nt:PMAT24A>
<nt:VSPLOT24>
<nt:VSPLOT_TC>9</nt:VSPLOT_TC>
<nt:VSPLOT_CRTE>834.3</nt:VSPLOT_CRTE>
<nt:VSPLOT_OFF>29</nt:VSPLOT_OFF>
</nt:VSPLOT24>
<nt:ER40>
<nt:ER40_CR>39</nt:ER40_CR>
<nt:ER40_CRT>1471.0</nt:ER40_CRT>
<nt:ER40ANG>8</nt:ER40ANG>
<nt:ER40FEAR>8</nt:ER40FEAR>
<nt:ER40HAP>8</nt:ER40HAP>
<nt:ER40NOE>8</nt:ER40NOE>
<nt:ER40SAD>7</nt:ER40SAD>
</nt:ER40>
</nt:NTScores>
>>>
```

That's a lot of stuff. Let's take it line-by-line.

The first line, `<?xml version="1.0" ...` , just tells us that this is an XML document.

The second line, `<nt:NTScores ID="ConnectomeDB_E00299" ...`, is the start of the actual content. It tells us that this is a N(on)T(oolbox)Scores document, gives us the experiment ID (the XNAT site-wide identifier), the project ID, the experiment labels (the human-readable, in-project-context name), and ends with a bunch of namespace information in case we want to validate this document against the schema we were looking at earlier. (I don't. You're welcome to if you like.)

The next few lines, `<xnat:sharing>` through `</xnat:sharing>`, tell us what projects know about this experiment. We can skip over this.

Next comes the subject ID; again, this is the XNAT site-wide ID, not the human-readable name (label). We can use pyxnat to ask ConnectomeDB for the label in a specified project:

```
>>> q2_proj.subject('ConnectomeDB_S00230').label()
'100307'
>>>
```

After that come the scores (and lots of them), organized into a few groups. The schema document nontoolbox.xsd may be useful in helping to decipher this. We can ask for individual scores by walking the XML DOM:

```
>>> nt = q2_proj.subject('100307').experiment('100307_NonToolbox')
>>> nt.xpath('nt:ER40/nt:ER40_CR')
[<Element {http://nrg.wustl.edu/nt}ER40_CR at 0x102065370>]
>>> nt.xpath('nt:ER40/nt:ER40_CR')[0].text()
'39'
>>>
```

That's a slow way of retreiving scores, since we need a full HTTP request and response for each field. (Actually, pyxnat does some caching so the requests aren't repeated. Probably. Usually. I'd still recommend doing something else.) If we want multiple scores -- either more than one score from a single experiment, or one or more scores from each of multiple experiments, there are more efficient methods.

Let's start with selecting multiple scores for a single experiment. A reasonable approach is to grab and parse the entire experiment XML document, using the Python standard library module ElementTree:

```
>>> import xml.etree.ElementTree as ET
>>> nt_dom = ET.fromstring(nt.get())
>>> nt_dom.tag
'{http://nrg.wustl.edu/nt}NTScores'
>>> er40 = nt_dom.find('{http://nrg.wustl.edu/nt}ER40')
>>> [[e.tag,e.text] for e in er40]
[['{http://nrg.wustl.edu/nt}ER40_CR', '39'], ['{http://nrg.wustl.edu/nt}
ER40_CRT', '1471.0'], ['{http://nrg.wustl.edu/nt}ER40ANG', '8'],
['{http://nrg.wustl.edu/nt}ER40FEAR', '8'], ['{http://nrg.wustl.edu/nt}
ER40HAP', '8'], ['{http://nrg.wustl.edu/nt}ER40NOE', '8'], ['{http://nrg.
wustl.edu/nt}ER40SAD', '7']]
>>>
```

Getting scores from multiple experiments can be done either by iterating over experiment IDs with the methods described above (single-attribute or XML document requests), or by using the pyxnat search interface.

Searching on ConnectomeDB

Retrieving values for multiple subjects or experiments is usually best done through the pyxnat search interface.

Simple searching

Let's start with an example: getting all of the NEO-FFI scores for all subjects in project 'HCP_Q2', with the corresponding subject labels. Parts of this example won't yet make sense, but we'll march ahead to a result, then backtrack to fill in the missing details.

First, we identify the fields we want to retrieve:

```
>>> xsitype = 'nt:scores'
>>> fields = ['SUBJECT_ID']+['NEO_NEOFAC_{}'.format(e) for e in 'AOCNE']
>>> fields = ['{}/{}'.format(xsitype, f) for f in fields]
>>> fields
['nt:scores/SUBJECT_ID',
 'nt:scores/NEO_NEOFAC_A', 'nt:scores/NEO_NEOFAC_O',
'nt:scores/NEO_NEOFAC_C', 'nt:scores/NEO_NEOFAC_N',
'nt:scores/NEO_NEOFAC_E']
>>>
```

Next, we build a constraint to use only subjects in project HCP_Q2:

```
>>> constraints = [(xsitype+'/PROJECTS', 'LIKE', '%HCP_Q2%')]
>>>
```

Now, we run the search:

```
>>> table = cdb.select(xsitype,fields).where(constraints)
>>> table
<JsonTable 143:6> subject_id,neo_neofac_a ... neo_neofac_n,neo_neofac_o
>>>
```

The result is an instance of a pyxnat-defined class (JsonTable). The most useful methods on this class are headers(), which shows the ordering of each row; and items(), which returns the content as an array of rows, each row a list:

```
>>> table.headers()
['subject_id', 'neo_neofac_a', 'neo_neofac_c', 'neo_neofac_e',
'neo_neofac_n', 'neo_neofac_o']
>>> table.items()
[('ConnectomeDB_S00230', '33', '24', '35', '15', '37'),
('ConnectomeDB_S00231', '19', '27', '32', '25', '25'), ...
```

ConnectomeDB gave us subject IDs instead of labels. Let's reorder this result into a dictionary with subject labels as the keys:

```
>>> neoffi_fields = table.headers()[1:]
>>> neoffi = {q2.subject(row[0]).label():row[1:] for row in table.items()}
>>> neoffi.keys()
['788876', '197550', '110411', '729557', '169343', ... ]
>>>
```

Now it's easy to view the scores for a single subject:

```
>>> dict(zip(neoffi_fields, neoffi['143325']))
{'neo_neofac_a': '28', 'neo_neofac_c': '25', 'neo_neofac_e': '32',
'neo_neofac_n': '25', 'neo_neofac_o': '15'}
>>>
```

We can also do simple local searches with data in this form. For example, here are all the subjects with agreeableness (neo_neofac_a) score higher than 40:

```
>>> ia = neoffi_fields.index('neo_neofac_a')
>>> [k for [k,v] in neoffi.iteritems() if v[ia] and int(v[ia]) > 40]
['702133', '732243', '149337', '792564', '877168']
>>>
```

Simple searching (once more, with details)

Now that we've seen that it's possible to do something with the search interface, let's dig into the details.

Naming fields in searches

Our first step was to build a list naming the fields of interest:

```
>>> xsitype = 'nt:scores'
>>> fields = ['SUBJECT_ID']+['NEO_NEOFAC_{}'.format(e) for e in 'AOCNE']
>>> fields = ['{}/{}'.format(xsitype, f) for f in fields]
>>> fields
['nt:scores/SUBJECT_ID',
 'nt:scores/NEO_NEOFAC_A', 'nt:scores/NEO_NEOFAC_O',
'nt:scores/NEO_NEOFAC_C', 'nt:scores/NEO_NEOFAC_N',
'nt:scores/NEO_NEOFAC_E']
>>>
```

All of the fields we're retrieving are defined in the nt:scores datatype; we need to include the datatype name as a prefix to the field names. If you look at nontolbox.xsd, the XML Schema document where the scores datatype is defined, you'll see the definitions of all these fields, but with a twist: none of the names in the schema match the names we're using above. For example, the NEO fields are all defined inside a wrapper element. We might expect the fields to have names like NEO/NEOFAC_A; instead, we have names like NEO_NEOFAC_A.

Why does the pyxnat search interface use a different field naming convention than the datatype definitions? It's a historical accident, and XNAT's fault. The search interface uses an field naming system that is independent of field naming elsewhere in the application. pyxnat provides a way to see what search terms are available:

```
>>> cdb.inspect.datatypes('nt:scores')
['nt:scores/SUBJECT_ID', 'nt:scores/EXPT_ID', 'nt:scores/PROJECT', 'nt:
scores/INSERT_DATE', 'nt:scores/INSERT_USER', 'nt:scores
/HCPPNP_MARS_LOG_SCORE', 'nt:scores/HCPPNP_MARS_ERRS', 'nt:scores
/HCPPNP_MARS_FINAL', 'nt:scores/DDISC_SV_1MO_200', 'nt:scores
/DDISC_SV_6MO_200', 'nt:scores/DDISC_SV_1YR_200', 'nt:scores
/DDISC_SV_3YR_200', 'nt:scores/DDISC_SV_5YR_200', 'nt:scores
/DDISC_SV_10YR_200', 'nt:scores/DDISC_SV_1MO_40000', 'nt:scores
/DDISC_SV_6MO_40000', 'nt:scores/DDISC_SV_1YR_40000', 'nt:scores
/DDISC_SV_3YR_40000', 'nt:scores/DDISC_SV_5YR_40000', 'nt:scores
/DDISC_SV_10YR_40000', 'nt:scores/DDISC_AUC_200', 'nt:scores
/DDISC_AUC_40000', 'nt:scores/NEO_NEO', 'nt:scores/NEO_NEOFAC_A', 'nt:
scores/NEO_NEOFAC_O', 'nt:scores/NEO_NEOFAC_C', 'nt:scores/NEO_NEOFAC_N',
'nt:scores/NEO_NEOFAC_E', 'nt:scores/SPCPTNL_SCPT_TP', 'nt:scores
/SPCPTNL_SCPT_TN', 'nt:scores/SPCPTNL_SCPT_FP', 'nt:scores
/SPCPTNL_SCPT_FN', 'nt:scores/SPCPTNL_SCPT_TPRT', 'nt:scores
/SPCPTNL_SCPT_SEN', 'nt:scores/SPCPTNL_SCPT_SPEC', 'nt:scores
/SPCPTNL_SCPT_LRNR', 'nt:scores/CPW_IWRD_TOT', 'nt:scores/CPW_IWRD_RTC',
'nt:scores/PMAT24A_PMAT24_A_CR', 'nt:scores/PMAT24A_PMAT24_A_SI', 'nt:
scores/PMAT24A_PMAT24_A_RTCR', 'nt:scores/VSPLOT24_VSPLOT_TC', 'nt:scores
/VSPLOT24_VSPLOT_CRTE', 'nt:scores/VSPLOT24_VSPLOT_OFF', 'nt:scores
/ER40_ER40_CR', 'nt:scores/ER40_ER40_CRT', 'nt:scores/ER40_ER40ANG', 'nt:
scores/ER40_ER40FEAR', 'nt:scores/ER40_ER40HAP', 'nt:scores/ER40_ER40NOE',
'nt:scores/ER40_ER40SAD']
```

That list is useful, but doesn't really tell you what each field is. The mapping between search-service field names and everywhere-else field names is defined in the display document, which primarily specifies how data types appear in the web application. The display document for nt:scores lists all the fields that can be displayed in the web application, with some information about how they are to be displayed. An excerpt follows:

nt:scores display document excerpt

```
<?xml version="1.0" encoding="UTF-8"?>
<Displays xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
noNamespaceSchemaLocation="../../xdat/display.xsd" schema-element="nt:
scores" full-description="NTScores" brief-description="NTScores">
        <Arc name="PARTICIPANT_EXPERIMENT">
                <CommonField id="PART_ID" local-field="SUBJECT_ID"/>
                <CommonField id="DATE" local-field="DATE"/>
                <CommonField id="EXPT_ID" local-field="EXPT_ID"/>
        </Arc>
        <DisplayField id="SUBJECT_ID" header="Subject" visible="true"
searchable="true">
                <DisplayFieldElement name="Field1" schema-element="nt:
scores.subject_ID"/>
                <HTML-Link>
                        <Property name="HREF" value="none"/>
                        <Property name="ONCLICK" value="return rpt
('@Field1','xnat:subjectData','xnat:subjectData.ID');">
                                <InsertValue id="Field1" field="SUBJECT_ID"
/>
                        </Property>
                </HTML-Link>
                <description>Subject Label</description>
        </DisplayField>
        <DisplayField id="EXPT_ID" header="ID" visible="true" searchable="
true">
                <DisplayFieldElement name="Field1" schema-element="nt:
scores.ID"/>
                <HTML-Link>
                        <Property name="HREF" value="none"/>
                        <Property name="ONCLICK" value="return rpt
('@Field1','nt:scores','nt:scores.ID');">
                                <InsertValue id="Field1" field="EXPT_ID"/>
                        </Property>
                </HTML-Link>
                <description>Experiment ID</description>
        </DisplayField>
...
        <DisplayField id="NEO_NEOFAC_A" header="NEOFAC_A" visible="true"
searchable="true">
                <DisplayFieldElement name="Field1" schema-element="nt:
scores/NEO/NEOFAC_A"/>
                <description>NEO Factor A (Agreeableness)</description>
        </DisplayField>
 ...
```

Each field that can be accessed through the search interface has a corresponding DisplayField element; the id attribute is the name by which the field is addressed in searches. Look carefully at the DisplayField element with id="NEO_NEOFAC_A" ; note that that element has a DisplayFieldElement subelement, which includes an attribute "schema-element" containing the field's name as we'd find it in the schema.

The tedious but reliable strategy for finding the search interface field identifiers is:

1. Determine from the data type documentation what fields you'll be using.
2. Find the field in the schema document (nontoolbox.xsd for the non-Toolbox Data Measures)
3. Find the corresponding DisplayElement in the display document (nt_scores_display.xml for the non-Toolbox Data Measures scores data type nt:scores), then use the id attribute value, with the data type prefix.

In practice, the display label returned by cdb.inspect.datatypes() is usually enough detail to save you from digging through the various XML documents.

Search constraints

Next, we build a constraint to use only subjects in project HCP_Q2:

```
>>> constraints = [(xsitype+'/PROJECTS', 'LIKE', '%HCP_Q2%')]
```

What's happening here? The nt:scores display document defines a PROJECTS DisplayField that is a comma-separated list of all projects that can see the nt:scores experiment: the project that contains the experiment, plus any projects into which the experiment has been shared. Since XNAT checks this constraint by doing a SQL query, the value is compared with a LIKE operation – a pattern match. % is the wildcard character in SQL, so we're looking for HCP_Q2 anywhere in that comma-separated list.

Aside for the obsessively detail-oriented: We'd have to be more clever if there were a project named, say, HCP_Q2b. It's possible to conjoin LIKE constraints with NOT LIKE constraints to make this work, and I'll update this text if that becomes necessary. For now, the project names on ConnectomeDB are probably sufficiently distinct.

Running the search and transforming the results

This part is mostly straightforward: we need to tell pyxnat what datatype we're searching for, what fields we want from it, and what constraints we want to apply. We use a dictionary comprehension to transform the results into a dictionary where the subject labels are the keys.

```
>>> table = cdb.select(xsitype,fields).where(constraints)
<JsonTable 76:6> subject_id,neo_neofac_a ... neo_neofac_n,neo_neofac_o
>>> neoffi_fields = table.headers()[1:]
>>> neoffi = {q2.subject(row[0]).label():row[1:] for row in table.items()}
```

The first field (index 0) is the subject ID, which we strip out of both the headers and the row contents using Python's slice operator. We then build a dictionary by looking up the subject labels (`q2.subject(row[0]).label()`), using those as keys, and using the rows with the subject ID removed as values.

Once we have the subject-to-values dictionary, we can build single-subject field-name-to-value dictionaries by zipping the field names together with the values for that subject, then building a dictionary from the resulting stream of key-value pairs:

```
>>> dict(zip(neoffi_fields, neoffi['143325']))
{'neo_neofac_a': '28', 'neo_neofac_c': '25', 'neo_neofac_e': '32',
'neo_neofac_n': '25', 'neo_neofac_o': '15'}
```

Finally, since we've downloaded all the NEO scores, we can search in memory without asking ConnectomeDB anything else:

```
>>> ia = neoffi_fields.index('neo_neofac_a')
>>> [k for [k,v] in neoffi if v[ia] and int(v[ia]) > 40]
['792564', '877168', '732243', '149337']
>>>
```

Here we look up the index of the agreeableness score, store it as ia=0, and extract the subject labels where there is an agreeableness score (`v[ia]` evaluates truthy for nonempty scores, false for the empty string) and the value is greater than 40 (`int(v[ia]) > 40`).

Building complex searches

The pyxnat search interface documentation explains how to compose simple searches into complex ones, so I'll just present some examples.

This is the same search that we performed on the NEO-FFI scores above, except this time on the server side:

```
>>> fields = ['SUBJECT_ID']+['NEO_NEOFAC_{}'.format(e) for e in 'AOCNE']
>>> fields = ['nt:scores/{}'.format('nt:scores', f) for f in fields]
>>> constraints = [('nt:scores/PROJECTS','LIKE','%HCP_Q2%'), ('nt:scores
/NEO_NEOFAC_A', '>', '40'), 'AND']
>>> table = cdb.select('nt:scores',fields).where(constraints);
>>> [q2.subject(row[0]).label() for row in table.items()]
['149337', '732243', '792564', '877168']
```

Finding (and counting) subjects scoring from 30 to 34 total correct on the Computerized Penn Word Memory:

```
>>> fields = ["nt:scores/"+n for n in
['SUBJECT_ID','CPW_IWRD_TOT','CPW_IWRD_RTC']]
>>> constraints = [('nt:scores/PROJECTS','LIKE','%HCP_Q2%'), ('nt:scores
/CPW_IWRD_TOT', '>=', '30'), ('nt:scores/CPW_IWRD_TOT', '<', '35'), 'AND']
>>> table = cdb.select('nt:scores',fields).where(constraints);
>>> len(table)
26
>>> [q2.subject(r[0]).label() for r in table.items()]
['111312', '114924', '119833', '125525', '128632', '137128', '144226',
'162329', '167743', '192439', '192540', '199150', '201111', '210617',
'250427', '255639', '672756', '685058', '856766', '894673', '896778',
'901139', '611231', '230926', '235128', '707244']
```

Finding subjects scoring either less than 30 or 35 or more total correct on the Computerized Penn Word Memory:

```
>>> fields = ["nt:scores/"+n for n in
['SUBJECT_ID','CPW_IWRD_TOT','CPW_IWRD_RTC']]
>>> constraints = [('nt:scores/PROJECTS','LIKE','%HCP_Q2%'), [('nt:scores
/CPW_IWRD_TOT', '<', '30'), ('nt:scores/CPW_IWRD_TOT', '>=', '35'), 'OR'],
'AND']
>>> table = cdb.select('nt:scores',fields).where
(constraints);                  >>> len(table)
50
>>> [q2.subject(r[0]).label() for r in table.items()]
['100307', '103515', '103818', '117122', '118932', '120212', '130013',
'138231', '142828', '143325', '149337', '150423', '153429', '156637',
'159239', '161731', '172332', '182739', '191437', '194140', '197550',
'199251', '200614', '217429', '249947', '304020', '307127', '329440',
'499566', '530635', '559053', '585862', '638049', '665254', '729557',
'732243', '792564', '826353', '859671', '861456', '865363', '877168',
'889579', '896879', '917255', '937160', '131621', '355542', '144428',
'733548']
```

Accessing imaging data

Before trying to access the data, it's important to understand what's in the Q2 release. The Q2 data release documentation describes the session structure and file layout in detail.

Once you know what you want, the best method to download the HCP imaging data is to use Aspera's fasp™, a proprietary transport protocol supported by ConnectomeDB. You'll need the free (gratis, not libre) Aspera Connect browser plugin. The easiest way to get this plugin is to log in to ConnectomeDB; if you don't have the plugin installed, a message at the top of the screen will hector you until you do.

The HCP-customized version of pyxnat includes a class for browsing and downloading HCP data using the Aspera plugin. For example, we can list the available package types:

```
>>> cdb.packages.list()
[u'3T_Structural_preproc', u'3T_Structural_unproc',
u'3T_rfMRI_REST1_preproc', u'3T_rfMRI_REST1_unproc',
u'3T_rfMRI_REST2_preproc', u'3T_rfMRI_REST2_unproc',
u'3T_tfMRI_EMOTION_preproc', u'3T_tfMRI_EMOTION_unproc',
u'3T_tfMRI_GAMBLING_preproc', u'3T_tfMRI_GAMBLING_unproc',
u'3T_tfMRI_LANGUAGE_preproc', u'3T_tfMRI_LANGUAGE_unproc',
u'3T_tfMRI_MOTOR_preproc', u'3T_tfMRI_MOTOR_unproc',
u'3T_tfMRI_RELATIONAL_preproc', u'3T_tfMRI_RELATIONAL_unproc',
u'3T_tfMRI_SOCIAL_preproc', u'3T_tfMRI_SOCIAL_unproc',
u'3T_tfMRI_WM_preproc', u'3T_tfMRI_WM_unproc', u'3T_Diffusion_preproc',
u'3T_Diffusion_unproc']
>>>
```

The Aspera-downloadable data are organized a little differently from Connectome-in-a-Box; each package is a zip file that contains the associated files. The zip files unpack into the same layout as C-in-a-B.

We can get archive file metadata about packages available for particular subjects:

```
>>> cdb.packages.for_subjects(['100307','125525'])
{u'packages':[{u'count': 446, u'subjects_total': 2, u'description': u'HCP
Q2 Data Release Structural Preprocessed', u'subjects_ok': 2, u'keywords':
[u'preprocessed', u'structural'], u'label': u'HCP Q2 Structural
Preprocessed', u'id': u'3T_Structural_preproc', u'size': 2551381019}, ...
```

We can use the Aspera plugin to download the archive files:

```
>>> cdb.packages.download(['100307'],
['3T_Structural_preproc','3T_rfMRI_REST1_preproc'])
```

This places the .zip files (and .zip.md5 checksums) in your working directory; you can instead specify a directory to place the downloaded zip files:

```
>>> cdb.packages.download(['100307'],
['3T_Structural_preproc','3T_rfMRI_REST1_preproc'], '/data/hcp/q2')
```

This download will take a while, even with the Aspera transport – and trust me, it's a lot faster than FTP /HTTP, especially if you're far from St. Louis. You are moving gigabytes of data through the rusty old intertubes. If you're going to download more than a few subjects, consider getting Connectome in a Box. The cost is comparable to the extra disk you were going to need to buy to hold the data anyway.

What's next?

I hope this tutorial is enough to get you started scripting against ConnectomeDB. If you're working through this tutorial or diving into the data, and you find rough edges that need to be smoothed (or big gaping holes that need to be filled), please let me know, by either submitting a comment below or sending me mail.