python-icat Documentation

Release 0.18.1

Rolf Krahl

Apr 14, 2021

Contents

The ICAT server is a metadata catalogue to support Large Facility experimental data, linking all aspects of the research chain from proposal through to publication. It provides SOAP and RESTful web service interfaces to an underlying database.

python-icat is a Python package that provides a collection of modules for writing programs that access an ICAT service using the SOAP interface. It is based on Suds and extends it with ICAT specific features.

The most important features include:

- Provide clients for ICAT and IDS.
- Keep the general structure and flexibility of Suds.
- Define Python classes to represent the entity object types from the ICAT schema.
- Read configuration from various sources, such as command line arguments, environment variables, and configuration files.
- Build JPQL expressions to search the ICAT server.
- Dump and restore ICAT content to and from a flat file. This is suitable as a general ingestion tool to ICAT.

CHAPTER 1

Parts of the documentation

1.1 Tutorial

This tutorial provides a step by step introduction to the usage of python-icat. It intents to give an overview of python-icat's most noteworthy features. You need to have python-icat installed to run the examples. You also need a running ICAT server and IDS server to connect to. Some examples in the tutorial assume to have root access and modify data, therefore it is not advisable to use a production ICAT. Rather setup a dedicated test ICAT to run the tutorial.

During the tutorial you will create some simple Python scripts and other files in your local file system. It is advisable to create a new empty folder for that purpose and change into that:

```
$ mkdir python-icat-tutorial
$ cd python-icat-tutorial
```

Some of the more advanced tutorial sections will require some example content in the ICAT server. You'll need the file *icatdump-4.10.yaml* to set it up. This file can be found in the *doc/examples* directory in the python-icat source distribution.

The tutorial assumes some basic knowledge in programming with Python as well as some basic understanding of ICAT. The tutorial contains the following sections:

1.1.1 Hello World!

The minimal task to start any program environment is to print a simple message. The minimal interaction with an ICAT server is to connect to it and get its version. We'll combine both in a simple program:

```
#! /usr/bin/python

from __future__ import print_function
import icat.client

url = "https://icat.example.com:8181"
client = icat.client.Client(url)
print("Connect to %s\nICAT version %s" % (url, client.apiversion))
```

If you run this script, you should get something like the following as output:

```
$ python hello.py
Connect to https://icat.example.com:8181
ICAT version 4.10
```

The constructor of `icat.client.Client` takes the URL of the ICAT service as argument. It contacts the ICAT server, queries the version and stores the result to the attribute `apiversion` of the client object. Obviously, you'll need to change the variable *url* in this example to point to your ICAT server.

If your ICAT server does not have a trusted SSL certificate you may get (depending on your Python version) an error instead:

```
$ python hello.py
Traceback (most recent call last):
  ...
urllib.error.URLError: <urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate␣
↪verify failed (_ssl.c:548)>
```

In this case, you may either install a trusted certificate in your server now or modify your hello program and add a flag `checkCert=False` to the constructor call like this:

```
#! /usr/bin/python

from __future__ import print_function
import icat.client

url = "https://icat.example.com:8181"
client = icat.client.Client(url, checkCert=False)
print("Connect to %s\nICAT version %s" % (url, client.apiversion))
```

The future statement at the beginning of the example is only needed to compile *print* as a built-in function rather than a statement. We'll use it throughout the tutorial to ensure that the examples will work with Python 2 as well as with Python 3.

The class `icat.client.Client` plays the central role in python-icat programs. Most of your code will deal with objects of this class. For this reason, the class is imported by default in the `icat` package. The above example could also be written as:

```
#! /usr/bin/python

from __future__ import print_function
import icat

url = "https://icat.example.com:8181"
client = icat.Client(url)
print("Connect to %s\nICAT version %s" % (url, client.apiversion))
```

1.1.2 Configuration

The example from the last section had the URL of the ICAT service hard coded in the program. This is certainly not the best way to do it. You could make it a command line argument, but then you would need to indicate it each time you run the program, which is also not very convenient. The module `icat.config` has been created to solve this. It manages several configuration variables that most ICAT client programs need.

Let's modify the example program as follows:

```
#! /usr/bin/python

from __future__ import print_function
import icat
import icat.config
```

(continues on next page)

(continued from previous page)

```
config = icat.config.Config(needlogin=False, ids=False)
client, conf = config.getconfig()
print("Connect to %s\nICAT version %s" % (conf.url, client.apiversion))
```

If we run this without any command line arguments, we get an error:

```
$ python config.py
Traceback (most recent call last):
  ...
icat.exception.ConfigError: Config option 'url' not given.
```

Apparently, there is a configuration option named *url* and we didn't specify it. Let's have a look on the command line options, that this program now accepts:

```
$ python config.py -h
usage: config.py [-h] [-c CONFIGFILE] [-s SECTION] [-w URL]
                 [--no-check-certificate] [--http-proxy HTTP_PROXY]
                 [--https-proxy HTTPS_PROXY] [--no-proxy NO_PROXY]

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIGFILE, --configfile CONFIGFILE
                        config file
  -s SECTION, --configsection SECTION
                        section in the config file
  -w URL, --url URL     URL to the web service description
  --no-check-certificate
                        don't verify the server certificate
  --http-proxy HTTP_PROXY
                        proxy to use for http requests
  --https-proxy HTTPS_PROXY
                        proxy to use for https requests
  --no-proxy NO_PROXY   list of exclusions for proxy use
```

So there is a command line option `-w URL`. Let's try:

```
$ python config.py -w 'https://icat.example.com:8181'
Connect to https://icat.example.com:8181
ICAT version 4.10
```

(Again, you may need to add the `--no-check-certificate` flag to the command line if your ICAT server does not have a trusted SSL certificate.) This does the job. But as mentioned above, it's not very convenient having to indicate the URL each time you run the program. But in the command line arguments, there is also a mention of a configuration file. Create a text file named `icat.cfg` in the current working directory with the following content:

```
[myicat]
url = https://icat.example.com:8181
# uncomment, if your server does not have a trusted certificate
#checkCert = No
```

Now you can do the following:

```
$ python config.py -s myicat
Connect to https://icat.example.com:8181
ICAT version 4.10
```

The command line option `-s SECTION` selects a section in the configuration file to read options from.

python-icat is not only a client for ICAT, but also for IDS. Since both may be on a different server, we need to tell python-icat also about the URL to IDS. Modify the example program to read as:

```
#! /usr/bin/python

from __future__ import print_function
import icat
import icat.config

config = icat.config.Config(needlogin=False, ids="optional")
client, conf = config.getconfig()
print("Connect to %s\nICAT version %s" % (conf.url, client.apiversion))
if conf.idsurl:
    print("Connect to %s\nIDS version %s"
          % (conf.idsurl, client.ids.apiversion))
else:
    print("No IDS configured")
```

If you run this in the same way as above, you'll get:

```
$ python config-with-ids.py -s myicat
Connect to https://icat.example.com:8181
ICAT version 4.10
No IDS configured
```

But if you indicate the URL to IDS with the command line option `--idsurl`, or even better in the configuration file as follows:

```
[myicat]
url = https://icat.example.com:8181
idsurl = https://icat.example.com:8181
# uncomment, if your server does not have a trusted certificate
#checkCert = No
```

You'll get something like:

```
$ python config-with-ids.py -s myicat
Connect to https://icat.example.com:8181
ICAT version 4.10
Connect to https://icat.example.com:8181
IDS version 1.10.1
```

1.1.3 Authentication and login

Until now, we only connected the ICAT server to query its version. This doesn't require a login to the server and hence the flag `needlogin=False` in the constructor call of *icat.config.Config* in our example program. If we leave this flag out, we get a bunch of new configuration variables. Consider the following example program:

```
#! /usr/bin/python

from __future__ import print_function
import icat
import icat.config

config = icat.config.Config(ids="optional")
client, conf = config.getconfig()
client.login(conf.auth, conf.credentials)

print("Login to %s was successful." % (conf.url))
print("User: %s" % (client.getUserName()))
```

Let's check the available command line options now:

```
$ python login.py -h
usage: login.py [-h] [-c CONFIGFILE] [-s SECTION] [-w URL] [--idsurl IDSURL]
                [--no-check-certificate] [--http-proxy HTTP_PROXY]
                [--https-proxy HTTPS_PROXY] [--no-proxy NO_PROXY] [-a AUTH]
                [-u USERNAME] [-P] [-p PASSWORD]

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIGFILE, --configfile CONFIGFILE
                        config file
  -s SECTION, --configsection SECTION
                        section in the config file
  -w URL, --url URL     URL to the web service description
  --idsurl IDSURL       URL to the ICAT Data Service
  --no-check-certificate
                        don't verify the server certificate
  --http-proxy HTTP_PROXY
                        proxy to use for http requests
  --https-proxy HTTPS_PROXY
                        proxy to use for https requests
  --no-proxy NO_PROXY   list of exclusions for proxy use
  -a AUTH, --auth AUTH  authentication plugin
  -u USERNAME, --user USERNAME
                        username
  -P, --prompt-pass     prompt for the password
  -p PASSWORD, --pass PASSWORD
                        password
```

Now call this program indicating the name of the authentication plugin and a user name:

```
$ python login.py -s myicat -a db -u jdoe
Password:
Login to https://icat.example.com:8181 was successful.
User: db/jdoe
```

Note that the program prompted us for a password, since we didn't provide one. Of course you need to specify an authentication plugin, user name, and password that is actually configured in your ICAT. Furthermore, the user name printed by the program may be different from the one indicated in the command line. This depends on the configuration of the authentication plugin in your ICAT. It is common praxis to prefix the user name with the name of the authentication plugin as shown in this example.

Note: For this tutorial we assume that the root user in the ICAT server has the user name *root* and is configured in the *simple* authenticator and that there are two users with name *jdoe* and *nbour* configured in the *db* authenticator. If this is not the case in your ICAT, you'll need to adapt the examples accordingly.

All configuration variables aside from *configFile* and *configSection* can be set in the configuration file. Edit your `icat.cfg` file to read:

```
[myicat_jdoe]
url = https://icat.example.com:8181
auth = db
username = jdoe
password = secret
idsurl = https://icat.example.com:8181
# uncomment, if your server does not have a trusted certificate
#checkCert = No
```

You should protect this file from unauthorized read access if you store passwords in it. Now you can do:

```
$ python login.py -s myicat_jdoe
Login to https://icat.example.com:8181 was successful.
User: db/jdoe
```

Command line options override the settings in the configuration file. This way, you can still log in as another user not configured in the file:

```
$ python login.py -s myicat_jdoe -u nbour
Password:
Login to https://icat.example.com:8181 was successful.
User: db/nbour
```

You might have noticed that the program again prompted us for a password even though there is one set in the config file. The *icat.config* module is smart enough to assume that if we overrode the user name on the command line, the password in the config file will likely not be valid for that user.

Configuration files can have many sections. It may come handy to be able to quickly switch between different users to log into the ICAT. Edit `icat.cfg` again to read as follows:

```
[myicat_root]
url = https://icat.example.com:8181
auth = simple
username = root
password = secret
idsurl = https://icat.example.com:8181
# uncomment, if your server does not have a trusted certificate
#checkCert = No

[myicat_jdoe]
url = https://icat.example.com:8181
auth = db
username = jdoe
password = secret
idsurl = https://icat.example.com:8181
#checkCert = No

[myicat_nbour]
url = https://icat.example.com:8181
auth = db
username = nbour
password = secret
idsurl = https://icat.example.com:8181
#checkCert = No
```

We shall use some of this configuration in the following sections of the tutorial. Do not forget to adapt the URLs, the authenticator names, and the passwords to what is configured in your ICAT.

1.1.4 Creating stuff in the ICAT server

The ICAT server is pretty useless if it is void of content. So let's start creating some objects.

We could do it by writing and running a small Python script each time as in the last sections. But python-icat may also be used interactively at the Python prompt, so let's try this out:

```
$ python -i login.py -s myicat_root
Login to https://icat.example.com:8181 was successful.
User: simple/root
>>> client.search("SELECT f FROM Facility f")
[]
```

The `-i` command line option tells Python to enter interactive mode after executing the `login.py` script from last section.

Creating simple objects

The *search()* result shows that there is no `Facility` object in ICAT. Let's create one. In the same session, type:

```
>>> f1 = client.new("facility")
>>> f1.name = "Fac1"
>>> f1.fullName = "Facility 1"
>>> f1.id = client.create(f1)
```

The *new()* method instantiates a new `Facility` object locally in the client. We set some of the attributes of this new object. Finally, we call *create()* to create it in the ICAT server. The return value is the ID of the new `Facility` object in ICAT. The result can be verified by repeating the search from above:

```
>>> client.search("SELECT f FROM Facility f")
[(facility){
   createId = "simple/root"
   createTime = 2019-11-26 12:39:18+01:00
   id = 1
   modId = "simple/root"
   modTime = 2019-11-26 12:39:18+01:00
   fullName = "Facility 1"
   name = "Fac1"
 }]
```

The same result could also have been obtained slightly differently: the *new()* method optionally accepts keyword arguments to set the attributes of the new object. Furthermore, the *Entity* object itself also has a *create()* method to create this object in the ICAT server. We thus could achieve the same as above like this:

```
>>> f2 = client.new("facility", name="Fac2", fullName="Facility 2")
>>> f2.create()
```

To verify the result, we check again:

```
>>> client.search("SELECT f FROM Facility f")
[(facility){
   createId = "simple/root"
   createTime = 2019-11-26 12:39:18+01:00
   id = 1
   modId = "simple/root"
   modTime = 2019-11-26 12:39:18+01:00
   fullName = "Facility 1"
   name = "Fac1"
 }, (facility){
   createId = "simple/root"
   createTime = 2019-11-26 12:40:02+01:00
   id = 2
   modId = "simple/root"
   modTime = 2019-11-26 12:40:02+01:00
   fullName = "Facility 2"
   name = "Fac2"
 }]
```

Relationships to other objects

Most objects in the ICAT are related to other objects. Let's first retrieve again the first facility created above using the *get()* method:

```
>>> f1 = client.get("Facility", 1)
```

The arguments are the name of the entity object class and the ID of the object to fetch. You might need to adapt that second argument, if the ICAT server attributed a different ID to your first facility, see the output from the *search()* call above.

Now consider the following example:

```
>>> pt1 = client.new("parameterType")
>>> pt1.name = "Test parameter type 1"
>>> pt1.units = "pct"
>>> pt1.applicableToDataset = True
>>> pt1.valueType = "NUMERIC"
>>> pt1.facility = f1
>>> pt1.create()
```

The `ParameterType` has a many to one relationship to a `Facility`. This relationship is established by setting the corresponding attribute in the `ParameterType` object before creating it in the ICAT. The `Facility` must already exist at this point.

On the other hand, there is also a one to many relationship between `ParameterType` and `PermissibleStringValue` in the ICAT schema. Let's create a `ParameterType` with string values:

```
>>> pt2 = client.new("parameterType")
>>> pt2.name = "Test parameter type 2"
>>> pt2.units = "N/A"
>>> pt2.applicableToDataset = True
>>> pt2.valueType = "STRING"
>>> pt2.facility = f1
>>> for v in ["buono", "brutto", "cattivo"]:
...     psv = client.new("permissibleStringValue", value=v)
...     pt2.permissibleStringValues.append(psv)
...
>>> pt2.create()
```

The `permissibleStringValues` attribute of `ParameterType` is a list. We may add new `PermissibleStringValue` instances to this list before creating the object. The `PermissibleStringValue` instances should not yet exist in ICAT at this point, they will be created together with the `ParameterType` object.

We can verify this by searching for the newly created objects:

```
>>> query = "SELECT pt FROM ParameterType pt INCLUDE pt.facility, pt.
↪permissibleStringValues"
>>> client.search(query)
[(parameterType){
   createId = "simple/root"
   createTime = 2019-11-26 12:40:54+01:00
   id = 1
   modId = "simple/root"
   modTime = 2019-11-26 12:40:54+01:00
   applicableToDataCollection = False
   applicableToDatafile = False
   applicableToDataset = True
   applicableToInvestigation = False
   applicableToSample = False
   enforced = False
   facility =
      (facility){
         createId = "simple/root"
         createTime = 2019-11-26 12:39:18+01:00
         id = 1
```

(continues on next page)

(continued from previous page)

```
        modId = "simple/root"
        modTime = 2019-11-26 12:39:18+01:00
        fullName = "Facility 1"
        name = "Fac1"
     }
  name = "Test parameter type 1"
  units = "pct"
  valueType = "NUMERIC"
  verified = False
}, (parameterType){
  createId = "simple/root"
  createTime = 2019-11-26 12:41:30+01:00
  id = 2
  modId = "simple/root"
  modTime = 2019-11-26 12:41:30+01:00
  applicableToDataCollection = False
  applicableToDatafile = False
  applicableToDataset = True
  applicableToInvestigation = False
  applicableToSample = False
  enforced = False
  facility =
     (facility){
        createId = "simple/root"
        createTime = 2019-11-26 12:39:18+01:00
        id = 1
        modId = "simple/root"
        modTime = 2019-11-26 12:39:18+01:00
        fullName = "Facility 1"
        name = "Fac1"
     }
  name = "Test parameter type 2"
  permissibleStringValues[] =
     (permissibleStringValue){
        createId = "simple/root"
        createTime = 2019-11-26 12:41:30+01:00
        id = 1
        modId = "simple/root"
        modTime = 2019-11-26 12:41:30+01:00
        value = "buono"
     },
     (permissibleStringValue){
        createId = "simple/root"
        createTime = 2019-11-26 12:41:30+01:00
        id = 2
        modId = "simple/root"
        modTime = 2019-11-26 12:41:30+01:00
        value = "brutto"
     },
     (permissibleStringValue){
        createId = "simple/root"
        createTime = 2019-11-26 12:41:30+01:00
        id = 3
        modId = "simple/root"
        modTime = 2019-11-26 12:41:30+01:00
        value = "cattivo"
     },
  units = "N/A"
  valueType = "STRING"
  verified = False
}]
```

As expected, we get a list of two `ParameterType` objects as result, one of them related to a couple of `PermissibleStringValue` objects that have been created at the same time as the related `ParameterType` object.

Access rules

Until now, we connected the ICAT server as the `root` user. Let's try what happens if we choose another user:

```
$ python -i login.py -s myicat_jdoe
Login to https://icat.example.com:8181 was successful.
User: db/jdoe
>>> client.search("SELECT pt FROM ParameterType pt INCLUDE pt.facility")
[]
```

We can't get any of the objects created above from ICAT. The reason is that we don't have the permission to access these objects. ICAT has a default deny access policy: only the `root` user has read and write access to everything, all other users get only access, if there is a rule that explicitely allows it.

Let's add some rules to allow public read access to some object types. Connect again as `root` and enter:

```
$ python -i login.py -s myicat_root
Login to https://icat.example.com:8181 was successful.
User: simple/root
>>> publicTables = [ "Application", "DatafileFormat", "DatasetType",
...                  "Facility", "FacilityCycle", "Instrument",
...                  "InvestigationType", "ParameterType",
...                  "PermissibleStringValue", "SampleType", ]
>>> queries = [ "SELECT o FROM %s o" % t for t in publicTables ]
>>> client.createRules("R", queries)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The *`createRules()`* method takes an access mode and a list of search queries (and optionally a group) as arguments. It will add rules that allow access to all objects yielded by a search for any of the queries. The access mode is `"R"` for read access in this example. *`createRules()`* is a convenience method in python-icat roughly equivalent to:

```
>>> rules = []
>>> for w in queries:
...     r = client.new("rule", crudFlags="R", what=w)
...     rules.append(r)
...
>>> client.createMany(rules)
```

If we now try again to search for the objects as normal user, we get:

```
$ python -i login.py -s myicat_jdoe
Login to https://icat.example.com:8181 was successful.
User: db/jdoe
>>> client.search("SELECT pt FROM ParameterType pt INCLUDE pt.facility")
[(parameterType){
   createId = "simple/root"
   createTime = 2019-11-26 12:40:54+01:00
   id = 1
   modId = "simple/root"
   modTime = 2019-11-26 12:40:54+01:00
   applicableToDataCollection = False
   applicableToDatafile = False
   applicableToDataset = True
   applicableToInvestigation = False
   applicableToSample = False
   enforced = False
```

(continues on next page)

(continued from previous page)

```
   facility =
      (facility){
         createId = "simple/root"
         createTime = 2019-11-26 12:39:18+01:00
         id = 1
         modId = "simple/root"
         modTime = 2019-11-26 12:39:18+01:00
         fullName = "Facility 1"
         name = "Fac1"
      }
   name = "Test parameter type 1"
   units = "pct"
   valueType = "NUMERIC"
   verified = False
 }, (parameterType){
   createId = "simple/root"
   createTime = 2019-11-26 12:41:30+01:00
   id = 2
   modId = "simple/root"
   modTime = 2019-11-26 12:41:30+01:00
   applicableToDataCollection = False
   applicableToDatafile = False
   applicableToDataset = True
   applicableToInvestigation = False
   applicableToSample = False
   enforced = False
   facility =
      (facility){
         createId = "simple/root"
         createTime = 2019-11-26 12:39:18+01:00
         id = 1
         modId = "simple/root"
         modTime = 2019-11-26 12:39:18+01:00
         fullName = "Facility 1"
         name = "Fac1"
      }
   name = "Test parameter type 2"
   units = "N/A"
   valueType = "STRING"
   verified = False
 }]
```

1.1.5 Working with objects in the ICAT server

In the previous section of this tutorial, we created two `Facility` objects:

```
$ python -i login.py -s myicat_root
Login to https://icat.example.com:8181 was successful.
User: simple/root
>>> client.search("SELECT f FROM Facility f")
[(facility){
   createId = "simple/root"
   createTime = 2019-11-26 12:39:18+01:00
   id = 1
   modId = "simple/root"
   modTime = 2019-11-26 12:39:18+01:00
   fullName = "Facility 1"
   name = "Fac1"
 }, (facility){
```

(continues on next page)

(continued from previous page)

```
   createId = "simple/root"
   createTime = 2019-11-26 12:40:02+01:00
   id = 2
   modId = "simple/root"
   modTime = 2019-11-26 12:40:02+01:00
   fullName = "Facility 2"
   name = "Fac2"
 }]
```

Let's see what we can do with these objects.

Editing the attributes of objects

We can edit the attributes of existing objects by assigning values to the corresponding *Entity* object. To write these changes back into ICAT, we can either call the *icat.client.Client.update()* method, or simply invoke the object's own *update()* method instead.

Let's loop over our `Facility` objects to add some new attributes and to edit existing ones:

```
>>> for facility in client.search("SELECT f FROM Facility f"):
...     facility.description = "An example facility"
...     facility.daysUntilRelease = 1826
...     facility.fullName = "%s Facility" % facility.name
...     client.update(facility)
...
```

We can verify the changes by performing another search:

```
>>> client.search("SELECT f FROM Facility f")
[(facility){
   createId = "simple/root"
   createTime = 2019-11-26 12:39:18+01:00
   id = 1
   modId = "simple/root"
   modTime = 2019-11-26 12:48:46+01:00
   daysUntilRelease = 1826
   description = "An example facility"
   fullName = "Fac1 Facility"
   name = "Fac1"
 }, (facility){
   createId = "simple/root"
   createTime = 2019-11-26 12:40:02+01:00
   id = 2
   modId = "simple/root"
   modTime = 2019-11-26 12:48:46+01:00
   daysUntilRelease = 1826
   description = "An example facility"
   fullName = "Fac2 Facility"
   name = "Fac2"
 }]
```

To remove a particular attribute value, we usually just have to assign `None` to it:

```
>>> for facility in client.search("SELECT f FROM Facility f"):
...     facility.description = None
...     facility.update()
...
```

If we search again now, the descriptions are gone:

```
>>> client.search("SELECT f FROM Facility f")
[(facility){
   createId = "simple/root"
   createTime = 2019-11-26 12:39:18+01:00
   id = 1
   modId = "simple/root"
   modTime = 2019-11-26 12:49:28+01:00
   daysUntilRelease = 1826
   fullName = "Fac1 Facility"
   name = "Fac1"
 }, (facility){
   createId = "simple/root"
   createTime = 2019-11-26 12:40:02+01:00
   id = 2
   modId = "simple/root"
   modTime = 2019-11-26 12:49:28+01:00
   daysUntilRelease = 1826
   fullName = "Fac2 Facility"
   name = "Fac2"
 }]
```

Copying objects

By calling the `copy()` method on an existing object, we can create a new object that has all attributes set to a copy of the corresponding values of the original object. The relations are copied by reference, i.e. the original and the copy refer to the same related object.

To demonstrate this, we use one of the `Facility` objects we created earlier, including its referenced `ParameterType` objects:

```
>>> fac = client.get("Facility f INCLUDE f.parameterTypes", 1)
>>> print(fac)
(facility){
   createId = "simple/root"
   createTime = 2019-11-26 12:39:18+01:00
   id = 1
   modId = "simple/root"
   modTime = 2019-11-26 12:49:28+01:00
   daysUntilRelease = 1826
   fullName = "Fac1 Facility"
   name = "Fac1"
   parameterTypes[] =
      (parameterType){
         createId = "simple/root"
         createTime = 2019-11-26 12:40:54+01:00
         id = 1
         modId = "simple/root"
         modTime = 2019-11-26 12:40:54+01:00
         applicableToDataCollection = False
         applicableToDatafile = False
         applicableToDataset = False
         applicableToInvestigation = False
         applicableToSample = False
         enforced = False
         name = "Test parameter type 1"
         units = "pct"
         valueType = "NUMERIC"
         verified = False
      },
      (parameterType){
```

(continues on next page)

(continued from previous page)

```
        createId = "simple/root"
        createTime = 2019-11-26 12:41:30+01:00
        id = 2
        modId = "simple/root"
        modTime = 2019-11-26 12:41:30+01:00
        applicableToDataCollection = False
        applicableToDatafile = False
        applicableToDataset = False
        applicableToInvestigation = False
        applicableToSample = False
        enforced = False
        name = "Test parameter type 2"
        units = "N/A"
        valueType = "STRING"
        verified = False
    },
}
```

Now we create a copy of this object and modify its attributes. The attributes of the original object remain unchanged. However, any changes to the referenced `ParameterType` objects are reflected in both the copy and the original:

```
>>> facc = fac.copy()
>>> print(facc.name)
Fac1
>>> print(facc.parameterTypes[0].name)
Test parameter type 1
>>> facc.name = "Fac0"
>>> facc.parameterTypes[0].name = "Test parameter type 0"
>>> print(fac.name)
Fac1
>>> print(fac.parameterTypes[0].name)
Test parameter type 0
```

When working with objects from ICAT, it can be a bit cumbersome to keep the (possibly large) tree of related objects in local memory. If you only need to keep the object's attributes, you can use the *`truncateRelations()`* method to delete all references to other objects from this object. Note that this is a local operation on the object in the client only. It does neither affect the corresponding object at the ICAT server, nor any copies of the object:

```
>>> fac.truncateRelations()
>>> print(fac)
(facility){
   createId = "simple/root"
   createTime = 2019-11-26 12:39:18+01:00
   id = 1
   modId = "simple/root"
   modTime = 2019-11-26 12:49:28+01:00
   daysUntilRelease = 1826
   fullName = "Fac1 Facility"
   name = "Fac1"
 }
>>> print(facc)
(facility){
   createId = None
   createTime = None
   id = 1
   modId = None
   modTime = None
   daysUntilRelease = 1826
   description = None
   fullName = "Fac1 Facility"
```

(continues on next page)

(continued from previous page)

```
   name = "Fac0"
   parameterTypes[] =
      (parameterType){
         createId = "simple/root"
         createTime = 2019-11-26 12:40:54+01:00
         id = 1
         modId = "simple/root"
         modTime = 2019-11-26 12:40:54+01:00
         applicableToDataCollection = False
         applicableToDatafile = False
         applicableToDataset = False
         applicableToInvestigation = False
         applicableToSample = False
         enforced = False
         name = "Test parameter type 0"
         units = "pct"
         valueType = "NUMERIC"
         verified = False
      },
      (parameterType){
         createId = "simple/root"
         createTime = 2019-11-26 12:41:30+01:00
         id = 2
         modId = "simple/root"
         modTime = 2019-11-26 12:41:30+01:00
         applicableToDataCollection = False
         applicableToDatafile = False
         applicableToDataset = False
         applicableToInvestigation = False
         applicableToSample = False
         enforced = False
         name = "Test parameter type 2"
         units = "N/A"
         valueType = "STRING"
         verified = False
      },
   url = None
 }
```

1.1.6 Searching for objects in the ICAT server

There are many ways to search for objects in ICAT using python-icat. Until now, we have seen how we can manually write JPQL query strings and pass them to the `search()` method:

```
$ python -i login.py -s myicat_root
Login to https://icat.example.com:8181 was successful.
User: simple/root
>>> client.search("SELECT f FROM Facility f INCLUDE f.parameterTypes LIMIT 1,1")
[(facility){
   createId = "simple/root"
   createTime = 2019-11-26 12:40:02+01:00
   id = 2
   modId = "simple/root"
   modTime = 2019-11-26 12:49:28+01:00
   daysUntilRelease = 1826
   fullName = "Fac2 Facility"
   name = "Fac2"
 }]
```

However, as our queries get more complicated, this can be a bit inconvenient. The `icat.query` module provides

an easier and less error-prone way to build queries. In addition, the `icat.client.Client` class has some useful methods as well.

But before we get into that, we will make sure that we actually have some well defined and rich content to search for. Run the following commands at the command line:

```
$ wipeicat -s myicat_root
$ icatingest -s myicat_root -i icatdump-4.10.yaml
```

wipeicat and *icatingest* are two scripts that get installed with python-icat. Depending on the situation, these scripts may be installed either with or without a trailing `.py` extension. The file *icatdump-4.10.yaml* can be found in the python-icat source distribution. The first command deletes all content from the ICAT server that we may have created in the previous sections. The second command reads the *icatdump-4.10.yaml* file and creates all objects listed therein in the ICAT server.

Note: As the name suggests, the content in *icatdump-4.10.yaml* requires an ICAT server version 4.10 or newer. If you are using an older ICAT, you may just as well use the *icatdump-4.7.yaml* or *icatdump-4.4.yaml* file instead, matching the respective older versions. For the sake of this tutorial, the difference does not matter.

Note: The search results in the following examples may depend on the user you log into ICAT as, because not all users have read access to all data. The examples assume that your user name (as displayed by the *login.py* script) is *db/nbour*. If that does not work for you, you may as well log in as root.

Building advanced queries

The `icat.query` module provides the `Query` class. We need to import it first:

```
$ python -i login.py -s myicat_nbour
Login to https://icat.example.com:8181 was successful.
User: db/nbour
>>> from icat.query import Query
```

Now let's have a look at some examples. We start with a simple query that lists all investigations:

```
>>> query = Query(client, "Investigation")
>>> print(query)
SELECT o FROM Investigation o
>>> client.search(query)
[(investigation){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:27+01:00
   id = 1
   modId = "simple/root"
   modTime = 2020-02-05 16:49:27+01:00
   name = "08100122-EF"
   startDate = 2008-03-13 11:39:42+01:00
   title = "Durol single crystal"
   visitId = "1.1-P"
 }, (investigation){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:28+01:00
   id = 2
   modId = "simple/root"
   modTime = 2020-02-05 16:49:28+01:00
   endDate = 2010-10-12 17:00:00+02:00
   name = "10100601-ST"
   startDate = 2010-09-30 12:27:24+02:00
```

(continues on next page)

(continued from previous page)

```
  title = "Ni-Mn-Ga flat cone"
  visitId = "1.1-N"
 }, (investigation){
  createId = "simple/root"
  createTime = 2020-02-05 16:49:33+01:00
  id = 3
  modId = "simple/root"
  modTime = 2020-02-05 16:49:33+01:00
  endDate = 2012-08-06 03:10:08+02:00
  name = "12100409-ST"
  startDate = 2012-07-26 17:44:24+02:00
  title = "NiO SC OF1 JUH HHL"
  visitId = "1.1-P"
 }]
```

In order to search for a particular investigation, we may add an appropriate condition. The *conditions* argument to `Query` should be a mapping of attribute names to conditions on that attribute:

```
>>> query = Query(client, "Investigation", conditions={"name": "= '10100601-ST'"})
>>> print(query)
SELECT o FROM Investigation o WHERE o.name = '10100601-ST'
>>> client.search(query)
[(investigation){
  createId = "simple/root"
  createTime = 2020-02-05 16:49:28+01:00
  id = 2
  modId = "simple/root"
  modTime = 2020-02-05 16:49:28+01:00
  endDate = 2010-10-12 17:00:00+02:00
  name = "10100601-ST"
  startDate = 2010-09-30 12:27:24+02:00
  title = "Ni-Mn-Ga flat cone"
  visitId = "1.1-N"
 }]
```

We may also include related objects in the search results:

```
>>> query = Query(client, "Investigation", conditions={"name": "= '10100601-ST'"},␣
↪includes=["datasets"])
>>> print(query)
SELECT o FROM Investigation o WHERE o.name = '10100601-ST' INCLUDE o.datasets
>>> client.search(query)
[(investigation){
  createId = "simple/root"
  createTime = 2020-02-05 16:49:28+01:00
  id = 2
  modId = "simple/root"
  modTime = 2020-02-05 16:49:28+01:00
  datasets[] =
    (dataset){
      createId = "simple/root"
      createTime = 2020-02-05 16:49:29+01:00
      id = 3
      modId = "simple/root"
      modTime = 2020-02-05 16:49:29+01:00
      complete = False
      endDate = 2010-10-01 08:17:48+02:00
      name = "e208339"
      startDate = 2010-09-30 12:27:24+02:00
    },
    (dataset){
```

(continues on next page)

(continued from previous page)

```
         createId = "simple/root"
         createTime = 2020-02-05 16:49:32+01:00
         id = 4
         modId = "simple/root"
         modTime = 2020-02-05 16:49:32+01:00
         complete = False
         endDate = 2010-10-05 10:32:21+02:00
         name = "e208341"
         startDate = 2010-10-02 04:00:21+02:00
      },
      (dataset){
         createId = "simple/root"
         createTime = 2020-02-05 16:49:32+01:00
         id = 5
         modId = "simple/root"
         modTime = 2020-02-05 16:49:32+01:00
         complete = False
         endDate = 2010-10-12 17:00:00+02:00
         name = "e208342"
         startDate = 2010-10-09 07:00:00+02:00
      },
   endDate = 2010-10-12 17:00:00+02:00
   name = "10100601-ST"
   startDate = 2010-09-30 12:27:24+02:00
   title = "Ni-Mn-Ga flat cone"
   visitId = "1.1-N"
 }]
```

The conditions in a query may also be put on the attributes of related objects. This allows rather complex queries. Let us search for the datasets in this investigation that have been measured in a magnetic field larger then 5 Tesla and include its parameters in the result:

```
>>> conditions = {
...     "investigation.name": "= '10100601-ST'",
...     "parameters.type.name": "= 'Magnetic field'",
...     "parameters.type.units": "= 'T'",
...     "parameters.numericValue": "> 5.0",
... }
>>> query = Query(client, "Dataset", conditions=conditions, includes=["parameters.
↪type"])
>>> print(query)
SELECT o FROM Dataset o JOIN o.investigation AS i JOIN o.parameters AS p JOIN p.
↪type AS pt WHERE i.name = '10100601-ST' AND p.numericValue > 5.0 AND pt.name =
↪'Magnetic field' AND pt.units = 'T' INCLUDE o.parameters AS p, p.type
>>> client.search(query)
[(dataset){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:29+01:00
   id = 3
   modId = "simple/root"
   modTime = 2020-02-05 16:49:29+01:00
   complete = False
   endDate = 2010-10-01 08:17:48+02:00
   name = "e208339"
   parameters[] =
      (datasetParameter){
         createId = "simple/root"
         createTime = 2020-02-05 16:49:29+01:00
         id = 1
         modId = "simple/root"
         modTime = 2020-02-05 16:49:29+01:00
```

(continues on next page)

(continued from previous page)

```
        numericValue = 7.3
        type =
           (parameterType){
              createId = "simple/root"
              createTime = 2020-02-05 16:49:24+01:00
              id = 5
              modId = "simple/root"
              modTime = 2020-02-05 16:49:24+01:00
              applicableToDataCollection = False
              applicableToDatafile = False
              applicableToDataset = True
              applicableToInvestigation = False
              applicableToSample = False
              enforced = False
              name = "Magnetic field"
              units = "T"
              unitsFullName = "Tesla"
              valueType = "NUMERIC"
              verified = False
           }
     },
     (datasetParameter){
        createId = "simple/root"
        createTime = 2020-02-05 16:49:29+01:00
        id = 2
        modId = "simple/root"
        modTime = 2020-02-05 16:49:29+01:00
        numericValue = 5.0
        type =
           (parameterType){
              createId = "simple/root"
              createTime = 2020-02-05 16:49:24+01:00
              id = 7
              modId = "simple/root"
              modTime = 2020-02-05 16:49:24+01:00
              applicableToDataCollection = False
              applicableToDatafile = False
              applicableToDataset = True
              applicableToInvestigation = False
              applicableToSample = False
              enforced = False
              name = "Reactor power"
              units = "MW"
              unitsFullName = "Megawatt"
              valueType = "NUMERIC"
              verified = False
           }
     },
  startDate = 2010-09-30 12:27:24+02:00
}]
```

We may incrementally add conditions to a query. This is particularly useful if the presence of some of the conditions depend on the logic of your Python program. Consider:

```
>>> def get_investigation(client, name, visitId=None):
...     query = Query(client, "Investigation")
...     query.addConditions({"name": "= '%s'" % name})
...     if visitId is not None:
...         query.addConditions({"visitId": "= '%s'" % visitId})
...     print(query)
...     return client.assertedSearch(query)[0]
```

(continues on next page)

(continued from previous page)

```
...
>>> get_investigation(client, "08100122-EF")
SELECT o FROM Investigation o WHERE o.name = '08100122-EF'
(investigation){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:27+01:00
   id = 1
   modId = "simple/root"
   modTime = 2020-02-05 16:49:27+01:00
   name = "08100122-EF"
   startDate = 2008-03-13 11:39:42+01:00
   title = "Durol single crystal"
   visitId = "1.1-P"
 }
>>> get_investigation(client, "12100409-ST", "1.1-P")
SELECT o FROM Investigation o WHERE o.name = '12100409-ST' AND o.visitId = '1.1-P'
(investigation){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:33+01:00
   id = 3
   modId = "simple/root"
   modTime = 2020-02-05 16:49:33+01:00
   endDate = 2012-08-06 03:10:08+02:00
   name = "12100409-ST"
   startDate = 2012-07-26 17:44:24+02:00
   title = "NiO SC OF1 JUH HHL"
   visitId = "1.1-P"
 }
```

This *get_investigation()* function will search for investigations, either by *name* alone or by *name* and *visitId*, depending on the arguments.

It is also possible to put more then one conditions on a single attribute: setting the corresponding value in the *conditions* argument to a list of strings will result in combining the conditions on that attribute. Search for all datafiles created in 2012:

```
>>> conditions = {
...     "datafileCreateTime": [">= '2012-01-01'", "< '2013-01-01'"]
... }
>>> query = Query(client, "Datafile", conditions=conditions)
>>> print(query)
SELECT o FROM Datafile o WHERE o.datafileCreateTime >= '2012-01-01' AND o.
↪datafileCreateTime < '2013-01-01'
>>> client.search(query)
[(datafile){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:34+01:00
   id = 7
   modId = "simple/root"
   modTime = 2020-02-05 16:49:34+01:00
   datafileCreateTime = 2012-07-16 16:30:17+02:00
   datafileModTime = 2012-07-16 16:30:17+02:00
   fileSize = 28937
   name = "e208945-2.nxs"
 }, (datafile){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:34+01:00
   id = 8
   modId = "simple/root"
   modTime = 2020-02-05 16:49:34+01:00
   checksum = "bd55affa"
```

(continues on next page)

(continued from previous page)

```
  datafileCreateTime = 2012-07-30 03:10:08+02:00
  datafileModTime = 2012-07-30 03:10:08+02:00
  fileSize = 459
  name = "e208945.dat"
}, (datafile){
  createId = "simple/root"
  createTime = 2020-02-05 16:49:34+01:00
  id = 10
  modId = "simple/root"
  modTime = 2020-02-05 16:49:34+01:00
  datafileCreateTime = 2012-07-16 16:30:17+02:00
  datafileModTime = 2012-07-16 16:30:17+02:00
  fileSize = 14965
  name = "e208947.nxs"
}]
```

Of course, that last example also works when adding the conditions incrementally:

```
>>> query = Query(client, "Datafile")
>>> query.addConditions({"datafileCreateTime": ">= '2012-01-01'"})
>>> query.addConditions({"datafileCreateTime": "< '2013-01-01'"})
>>> print(query)
SELECT o FROM Datafile o WHERE o.datafileCreateTime >= '2012-01-01' AND o.
↪datafileCreateTime < '2013-01-01'
```

Instead of returning a list of the matching objects, we may also request single attributes. The result will be a list of the attribute values of the matching objects. Listing the names of all datasets:

```
>>> query = Query(client, "Dataset", attributes="name")
>>> print(query)
SELECT o.name FROM Dataset o
>>> client.search(query)
[e201215, e201216, e208339, e208341, e208342, e208945, e208946, e208947]
```

As the name of that keyword argument suggests, we may also search for multiple attributes at once. The result will be a tuple of attribute values rather then a single value for each object found in the query. This requires an ICAT server version 4.11 or newer though:

```
>>> query = Query(client, "Dataset", attributes=["investigation.name", "name",
↪"complete", "type.name"])
>>> print(query)
SELECT i.name, o.name, o.complete, t.name FROM Dataset o JOIN o.investigation AS i␣
↪JOIN o.type AS t
>>> client.search(query)
[(08100122-EF, e201215, False, raw), (08100122-EF, e201216, False, raw), (10100601-
↪ST, e208339, False, raw), (10100601-ST, e208341, False, raw), (10100601-ST,␣
↪e208342, False, raw), (12100409-ST, e208945, False, raw), (12100409-ST, e208946,␣
↪False, raw), (12100409-ST, e208947, True, analyzed)]
```

There are also some aggregate functions that may be applied to search results. Let's count all datasets:

```
>>> query = Query(client, "Dataset", aggregate="COUNT")
>>> print(query)
SELECT COUNT(o) FROM Dataset o
>>> client.search(query)
[8]
```

Using such aggregate functions in a query may result in a huge performance gain, because the counting is done directly in the database backend of ICAT, instead of compiling a list of all datasets, transferring them to the client, and counting them at client side.

Let's check for a given investigation, the minimum, maximum, and average magnetic field applied in the measurements:

```
>>> conditions = {
...     "dataset.investigation.name": "= '10100601-ST'",
...     "type.name": "= 'Magnetic field'",
...     "type.units": "= 'T'",
... }
>>> query = Query(client, "DatasetParameter", conditions=conditions, attributes=
↪"numericValue")
>>> print(query)
SELECT o.numericValue FROM DatasetParameter o JOIN o.dataset AS ds JOIN ds.
↪investigation AS i JOIN o.type AS t WHERE i.name = '10100601-ST' AND t.name =
↪'Magnetic field' AND t.units = 'T'
>>> client.search(query)
[7.3, 2.7]
>>> query.setAggregate("MIN")
>>> print(query)
SELECT MIN(o.numericValue) FROM DatasetParameter o JOIN o.dataset AS ds JOIN ds.
↪investigation AS i JOIN o.type AS t WHERE i.name = '10100601-ST' AND t.name =
↪'Magnetic field' AND t.units = 'T'
>>> client.search(query)
[2.7]
>>> query.setAggregate("MAX")
>>> print(query)
SELECT MAX(o.numericValue) FROM DatasetParameter o JOIN o.dataset AS ds JOIN ds.
↪investigation AS i JOIN o.type AS t WHERE i.name = '10100601-ST' AND t.name =
↪'Magnetic field' AND t.units = 'T'
>>> client.search(query)
[7.3]
>>> query.setAggregate("AVG")
>>> print(query)
SELECT AVG(o.numericValue) FROM DatasetParameter o JOIN o.dataset AS ds JOIN ds.
↪investigation AS i JOIN o.type AS t WHERE i.name = '10100601-ST' AND t.name =
↪'Magnetic field' AND t.units = 'T'
>>> client.search(query)
[5.0]
```

For another example, let's search for all investigations, having any dataset with a magnetic field parameter set:

```
>>> conditions = {
...     "datasets.parameters.type.name": "= 'Magnetic field'",
...     "datasets.parameters.type.units": "= 'T'",
... }
>>> query = Query(client, "Investigation", conditions=conditions)
>>> print(query)
SELECT o FROM Investigation o JOIN o.datasets AS s1 JOIN s1.parameters AS s2 JOIN␣
↪s2.type AS s3 WHERE s3.name = 'Magnetic field' AND s3.units = 'T'
>>> client.search(query)
[(investigation){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:28+01:00
   id = 2
   modId = "simple/root"
   modTime = 2020-02-05 16:49:28+01:00
   endDate = 2010-10-12 17:00:00+02:00
   name = "10100601-ST"
   startDate = 2010-09-30 12:27:24+02:00
   title = "Ni-Mn-Ga flat cone"
   visitId = "1.1-N"
 }, (investigation){
   createId = "simple/root"
```

(continues on next page)

(continued from previous page)

```
   createTime = 2020-02-05 16:49:28+01:00
   id = 2
   modId = "simple/root"
   modTime = 2020-02-05 16:49:28+01:00
   endDate = 2010-10-12 17:00:00+02:00
   name = "10100601-ST"
   startDate = 2010-09-30 12:27:24+02:00
   title = "Ni-Mn-Ga flat cone"
   visitId = "1.1-N"
 }]
```

We get the same investigation twice! The reason is that this investigation has two datasets, both having a magnetic field parameter respectively. We may fix that by applying *DISTINCT*:

```
>>> query.setAggregate("DISTINCT")
>>> print(query)
SELECT DISTINCT(o) FROM Investigation o JOIN o.datasets AS s1 JOIN s1.parameters␣
↪AS s2 JOIN s2.type AS s3 WHERE s3.name = 'Magnetic field' AND s3.units = 'T'
>>> client.search(query)
[(investigation){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:28+01:00
   id = 2
   modId = "simple/root"
   modTime = 2020-02-05 16:49:28+01:00
   endDate = 2010-10-12 17:00:00+02:00
   name = "10100601-ST"
   startDate = 2010-09-30 12:27:24+02:00
   title = "Ni-Mn-Ga flat cone"
   visitId = "1.1-N"
 }]
```

DISTINCT may be combined with *COUNT*, *AVG*, and *SUM* in order to make sure not to count the same object more then once:

```
>>> conditions = {
...     "datasets.parameters.type.name": "= 'Magnetic field'",
...     "datasets.parameters.type.units": "= 'T'",
... }
>>> query = Query(client, "Investigation", conditions=conditions, aggregate="COUNT
↪")
>>> print(query)
SELECT COUNT(o) FROM Investigation o JOIN o.datasets AS s1 JOIN s1.parameters AS␣
↪s2 JOIN s2.type AS s3 WHERE s3.name = 'Magnetic field' AND s3.units = 'T'
>>> client.search(query)
[2]
>>> query.setAggregate("COUNT:DISTINCT")
>>> print(query)
SELECT COUNT(DISTINCT(o)) FROM Investigation o JOIN o.datasets AS s1 JOIN s1.
↪parameters AS s2 JOIN s2.type AS s3 WHERE s3.name = 'Magnetic field' AND s3.
↪units = 'T'
>>> client.search(query)
[1]
```

The JPQL queries support sorting of the results. Search for all dataset parameter, ordered by parameter type name (ascending), units (ascending), and value (descending):

```
>>> order = ["type.name", "type.units", ("numericValue", "DESC")]
>>> query = Query(client, "DatasetParameter", includes=["type"], order=order)
>>> print(query)
SELECT o FROM DatasetParameter o JOIN o.type AS t ORDER BY t.name, t.units, o.
↪numericValue DESC INCLUDE o.type
```

(continues on next page)

(continued from previous page)

```
>>> client.search(query)
[(datasetParameter){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:29+01:00
   id = 1
   modId = "simple/root"
   modTime = 2020-02-05 16:49:29+01:00
   numericValue = 7.3
   type =
      (parameterType){
         createId = "simple/root"
         createTime = 2020-02-05 16:49:24+01:00
         id = 5
         modId = "simple/root"
         modTime = 2020-02-05 16:49:24+01:00
         applicableToDataCollection = False
         applicableToDatafile = False
         applicableToDataset = True
         applicableToInvestigation = False
         applicableToSample = False
         enforced = False
         name = "Magnetic field"
         units = "T"
         unitsFullName = "Tesla"
         valueType = "NUMERIC"
         verified = False
      }
 }, (datasetParameter){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:32+01:00
   id = 4
   modId = "simple/root"
   modTime = 2020-02-05 16:49:32+01:00
   numericValue = 2.7
   type =
      (parameterType){
         createId = "simple/root"
         createTime = 2020-02-05 16:49:24+01:00
         id = 5
         modId = "simple/root"
         modTime = 2020-02-05 16:49:24+01:00
         applicableToDataCollection = False
         applicableToDatafile = False
         applicableToDataset = True
         applicableToInvestigation = False
         applicableToSample = False
         enforced = False
         name = "Magnetic field"
         units = "T"
         unitsFullName = "Tesla"
         valueType = "NUMERIC"
         verified = False
      }
 }, (datasetParameter){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:32+01:00
   id = 3
   modId = "simple/root"
   modTime = 2020-02-05 16:49:32+01:00
   numericValue = 5.0
   type =
```

(continues on next page)

(continued from previous page)

```
      (parameterType){
         createId = "simple/root"
         createTime = 2020-02-05 16:49:24+01:00
         id = 7
         modId = "simple/root"
         modTime = 2020-02-05 16:49:24+01:00
         applicableToDataCollection = False
         applicableToDatafile = False
         applicableToDataset = True
         applicableToInvestigation = False
         applicableToSample = False
         enforced = False
         name = "Reactor power"
         units = "MW"
         unitsFullName = "Megawatt"
         valueType = "NUMERIC"
         verified = False
      }
}, (datasetParameter){
  createId = "simple/root"
  createTime = 2020-02-05 16:49:29+01:00
  id = 2
  modId = "simple/root"
  modTime = 2020-02-05 16:49:29+01:00
  numericValue = 5.0
  type =
      (parameterType){
         createId = "simple/root"
         createTime = 2020-02-05 16:49:24+01:00
         id = 7
         modId = "simple/root"
         modTime = 2020-02-05 16:49:24+01:00
         applicableToDataCollection = False
         applicableToDatafile = False
         applicableToDataset = True
         applicableToInvestigation = False
         applicableToSample = False
         enforced = False
         name = "Reactor power"
         units = "MW"
         unitsFullName = "Megawatt"
         valueType = "NUMERIC"
         verified = False
      }
}, (datasetParameter){
  createId = "simple/root"
  createTime = 2020-02-05 16:49:34+01:00
  id = 5
  modId = "simple/root"
  modTime = 2020-02-05 16:49:34+01:00
  numericValue = 3.92
  type =
      (parameterType){
         createId = "simple/root"
         createTime = 2020-02-05 16:49:25+01:00
         id = 9
         modId = "simple/root"
         modTime = 2020-02-05 16:49:25+01:00
         applicableToDataCollection = False
         applicableToDatafile = False
         applicableToDataset = True
```

(continues on next page)

(continued from previous page)

```
        applicableToInvestigation = False
        applicableToSample = False
        enforced = False
        name = "Sample temperature"
        units = "C"
        unitsFullName = "Celsius"
        valueType = "NUMERIC"
        verified = False
     }
 }, (datasetParameter){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:34+01:00
   id = 6
   modId = "simple/root"
   modTime = 2020-02-05 16:49:34+01:00
   numericValue = 277.07
   type =
      (parameterType){
        createId = "simple/root"
        createTime = 2020-02-05 16:49:25+01:00
        id = 10
        modId = "simple/root"
        modTime = 2020-02-05 16:49:25+01:00
        applicableToDataCollection = False
        applicableToDatafile = False
        applicableToDataset = True
        applicableToInvestigation = False
        applicableToSample = False
        enforced = False
        name = "Sample temperature"
        units = "K"
        unitsFullName = "Kelvin"
        valueType = "NUMERIC"
        verified = False
     }
 }]
```

We may limit the number of returned items. Search for the second to last dataset to have been finished:

```
>>> query = Query(client, "Dataset", order=[("endDate", "DESC")], limit=(1, 1))
>>> print(query)
SELECT o FROM Dataset o ORDER BY o.endDate DESC LIMIT 1, 1
>>> client.search(query)
[(dataset){
   createId = "simple/root"
   createTime = 2020-02-05 16:49:34+01:00
   id = 6
   modId = "simple/root"
   modTime = 2020-02-05 16:49:34+01:00
   complete = False
   endDate = 2012-07-30 03:10:08+02:00
   name = "e208945"
   startDate = 2012-07-26 17:44:24+02:00
 }]
```

Useful search methods

Additionally to the generic `search()` method defined in the ICAT API, python-icat provides a few custom search methods that are useful in particular situations.

assertedSearch

The generic search returns a list of matching objects. Often, the number of objects to expect in the result is known from the context. In the most common case, you would expect exactly one object in the result and would raise an error if this is not the case. This is what `assertedSearch()` does. Example: in many production ICAT installations there is one and only one facility object and you often need to fetch that in your scripts in order to create a new investigation or a new parameter type. Using the generic search method you would write the following boiler plate code over and over:

```
res = client.search(Query(client, "Facility"))
if not res:
    raise RuntimeError("Facility not found")
elif len(res) > 1:
    raise RuntimeError("Facility not unique")
facility = res[0]
```

(Note that you cannot safely subscript the result unless you know it's not empty.) Using `assertedSearch()`, you can write the same as:

```
facility = client.assertedSearch(Query(client, "Facility"))[0]
```

searchChunked

A production ICAT has many datasets and datafiles. You cannot search for all of them at once, because the result might not fit in your client's memory. Furthermore, ICAT has a configured limit for the maximum of objects to return in one search call, so you might hit that wall if you are not careful. The `searchChunked()` method comes handy if you need to iterate over a potentially large set of results. It can be used as a drop in replacement for the generic search method most of the times, see the reference documentation for some subtle differences. You can safely do things like:

```
for ds in client.searchChunked(Query(client, "Dataset")):
    # do something useful with the dataset ds ...
    print(ds.name)
```

searchMatching

Given an object having all the attributes and related objects set that form the uniqueness constraint for the object type, the `searchMatching()` method searches this very object from the ICAT server. While this may not sound very useful at first glance, it has a particular use case:

```
def get_dataset(client, inv_name, ds_name, ds_type="raw"):
    """Get a dataset in an investigation.
    If it already exists, search and return it, create it, if not.
    """
    try:
        dataset = client.new("dataset")
        query = Query(client, "Investigation", conditions={
            "name": "= '%s'" % inv_name
        })
        dataset.investigation = client.assertedSearch(query)[0]
        query = Query(client, "DatasetType", conditions={
            "name": "= '%s'" % ds_type
        })
        dataset.type = client.assertedSearch(query)[0]
        dataset.complete = False
        dataset.name = ds_name
        dataset.create()
```

(continues on next page)

(continued from previous page)

```
    except icat.ICATObjectExistsError:
        dataset = client.searchMatching(dataset)
    return dataset
```

1.1.7 Upload and download files to and from IDS

The ICAT Data Service (IDS) is the component that manages the storage in ICAT. It implements file upload and download. You can use python-icat not only as a client for ICAT, but also for IDS. In this tutorial section, we look at some basic examples of this. The examples below assume to have a running IDS that is ready to accept our requests.

If the *idsurl* configuration variable is set (see *Configuration* for details), python-icat will provide an IDS client in the `ids` attribute of the `Client` class. This `IDSClient` provides methods for the IDS API calls:

```
$ python -i login.py -s myicat_nbour
Login to https://icat.example.com:8181 was successful.
User: db/nbour
>>> client.ids.isReadOnly()
False
```

Additionally, the `Client` class directly provides methods for some of the mosten often needed IDS calls. These custom IDS methods are based on the low level IDS client methods but are somewhat more convenient to use and integrate better in the python-icat data structures.

This tutorial section uses the same example content in ICAT as the previous section. This content can be set up with the following commands at the command line:

```
$ wipeicat -s myicat_root
$ icatingest -s myicat_root -i icatdump-4.10.yaml
```

If you already did that for the previous section, you don't need to repeat it. Take notice of the hint on the content of the *icatdump-4.10.yaml* file and ICAT server versions from the previous section.

Upload files

Obviously, we would need some local files first, if we want to upload them. Let's create a few:

```
>>> users = [("jdoe", "John"), ("nbour", "Nicolas"), ("rbeck", "Rudolph")]
>>> for user, name in users:
...     with open("greet-%s.txt" % user, "wt") as f:
...         print("Hello %s!" % name, file=f)
...
```

We need a dataset in ICAT that the uploaded files should be put into, so let's create one:

```
>>> from icat.query import Query
>>> investigation = client.assertedSearch(Query(client, "Investigation",␣
↪conditions={"name": "= '12100409-ST'"}))[0]
>>> dataset = client.new("dataset")
>>> dataset.investigation = investigation
>>> dataset.type = client.assertedSearch(Query(client, "DatasetType", conditions={
↪"name": "= 'other'"}))[0]
>>> dataset.name = "greetings"
>>> dataset.complete = False
>>> dataset.create()
```

For each of the files, we create a new datafile object and call the `putData()` method to upload it:

```
>>> df_format = client.assertedSearch(Query(client, "DatafileFormat", conditions={
↪"name": "= 'Text'"}))[0]
>>> for fname in ("greet-jdoe.txt", "greet-nbour.txt", "greet-rbeck.txt"):
...     datafile = client.new("datafile", name=fname, dataset=dataset,␣
↪datafileFormat=df_format)
...     client.putData(fname, datafile)
...
(datafile){
   createId = "db/nbour"
   createTime = 2020-02-21 14:57:16+01:00
   id = 11
   modId = "db/nbour"
   modTime = 2020-02-21 14:57:16+01:00
   checksum = "bef32c73"
   datafileCreateTime = 2020-02-21 13:45:16+01:00
   datafileModTime = 2020-02-21 13:45:16+01:00
   fileSize = 12
   location = "3/9/f3b5c400-0a24-4915-b7a7-d4f976ec3e73"
   name = "greet-jdoe.txt"
 }
(datafile){
   createId = "db/nbour"
   createTime = 2020-02-21 14:57:16+01:00
   id = 12
   modId = "db/nbour"
   modTime = 2020-02-21 14:57:16+01:00
   checksum = "9012de77"
   datafileCreateTime = 2020-02-21 13:45:16+01:00
   datafileModTime = 2020-02-21 13:45:16+01:00
   fileSize = 15
   location = "3/9/392d4c49-d9c4-40fa-b4cb-5bdcbb4414e6"
   name = "greet-nbour.txt"
 }
(datafile){
   createId = "db/nbour"
   createTime = 2020-02-21 14:57:16+01:00
   id = 13
   modId = "db/nbour"
   modTime = 2020-02-21 14:57:16+01:00
   checksum = "cc830993"
   datafileCreateTime = 2020-02-21 13:45:16+01:00
   datafileModTime = 2020-02-21 13:45:16+01:00
   fileSize = 15
   location = "3/9/dd4c6f7f-05f6-418d-8c1f-8a87ca727e5a"
   name = "greet-rbeck.txt"
 }
```

Note that we did not create these datafiles in ICAT. IDS did this for us in response to the `putData()` call. IDS also calculated the checksum and set the file size. The location attribute is also set by IDS and is mostly only relevant internally in IDS. The value depends on the IDS storage plugin and may be different. The datafileCreateTime and the datafileModTime has been determined by fstat'ing the local files in `putData()`.

Download files

We can request a download of a set of data using the `getData()` method:

```
>>> query = Query(client, "Datafile", conditions={"name": "= 'greet-jdoe.txt'",
↪"dataset.name": "= 'greetings'"})
>>> df = client.assertedSearch(query)[0]
>>> data = client.getData([df])
```

(continues on next page)

(continued from previous page)

```
>>> type(data)
<class 'http.client.HTTPResponse'>
>>> data.read().decode('utf8')
'Hello John!\n'
```

This method takes a list of investigation, dataset, or datafile objects as argument. It returns a `HTTPResponse` object, which is a file like object that we can read the body of the HTTP response from. If we requested only one single file, this response will contain the file content. If more then a single file is requested, either by passing multiple files in the argument or by requesting a dataset having multiple files, IDS will send a zip file with the requested files:

```
>>> from io import BytesIO
>>> from zipfile import ZipFile
>>> query = Query(client, "Dataset", conditions={"name": "= 'greetings'"})
>>> ds = client.assertedSearch(query)[0]
>>> data = client.getData([ds])
>>> buffer = BytesIO(data.read())
>>> with ZipFile(buffer) as zipfile:
...     for f in zipfile.namelist():
...         print("file name: %s" % f)
...         print("content: %r" % zipfile.open(f).read().decode('utf8'))
...
file name: ids/ESNF/12100409-ST/1.1-P/greetings/greet-jdoe.txt
content: 'Hello John!\n'
file name: ids/ESNF/12100409-ST/1.1-P/greetings/greet-nbour.txt
content: 'Hello Nicolas!\n'
file name: ids/ESNF/12100409-ST/1.1-P/greetings/greet-rbeck.txt
content: 'Hello Rudolph!\n'
```

The internal file names in the zip file depend on the IDS storage plugin and may be different.

Note that it may happen that the files we request are not readily available because they are archived to tape. We create this condition by explicitely requesting IDS to archive our dataset:

```
>>> from icat.ids import DataSelection
>>> selection = DataSelection([ds])
>>> client.ids.archive(selection)
```

Note that we needed to resort to a low level call from the IDS client for that. This method requires the selected data to be wrapped in a `DataSelection` object. We may also check that status:

```
>>> client.ids.getStatus(selection)
'ARCHIVED'
```

If we request the data now, we will get an error from IDS:

```
>>> data = client.getData([ds])
Traceback (most recent call last):
  ...
icat.exception.IDSDataNotOnlineError: Before putting, getting or deleting a␣
↪datafile, its dataset has to be restored, restoration requested automatically
```

As the error message hints, a restoration of the data has been requested automatically. So we can just repeat the request again after a short while:

```
>>> client.ids.getStatus(selection)
'ONLINE'
>>> data = client.getData([ds])
>>> len(data.read())
665
```

We can ask IDS with the `prepareData()` call to store a selection of data objects internally for later referral:

```
>>> preparedId = client.prepareData(selection)
>>> preparedId
'eb0dd942-7ce9-4ea9-b342-ea326edd4dfe'
```

The return value is a random id. We can use that preparedId to query the status or to download the data:

```
>>> client.isDataPrepared(preparedId)
True
>>> data = client.getData(preparedId)
>>> buffer = BytesIO(data.read())
>>> with ZipFile(buffer) as zipfile:
...     zipfile.namelist()
...
['ids/ESNF/12100409-ST/1.1-P/greetings/greet-jdoe.txt', 'ids/ESNF/12100409-ST/1.1-
↪P/greetings/greet-nbour.txt', 'ids/ESNF/12100409-ST/1.1-P/greetings/greet-rbeck.
↪txt']
```

1.1.8 Advanced configuration

So far, we have relied on the `icat.config` module to provide configuration variables for us (such as *url* or *idsurl*). However, programs may also define their own custom configuration variables.

Custom configuration variables

Let's add the option to redirect the output of our example program to a file. The output file path shall be passed via the command line as a configuration variable. To set this up, we can use the `add_variable()` method:

```
#! /usr/bin/python

from __future__ import print_function
import sys
import icat
import icat.config

config = icat.config.Config(ids="optional")
config.add_variable("outfile", ("-o", "--outputfile"),
                    dict(help="output file name or '-' for stdout"),
                    default="-")
client, conf = config.getconfig()
client.login(conf.auth, conf.credentials)

if conf.outfile == "-":
    out = sys.stdout
else:
    out = open(conf.outfile, "wt")

print("Login to %s was successful." % (conf.url), file=out)
print("User: %s" % (client.getUserName()), file=out)

out.close()
```

This adds a new configuration variable *outfile*. It can be specified on the command line as `-o OUTFILE` or `--outputfile OUTFILE` and it defaults to the string `-` if not specified. We can check this on the list of available command line options:

```
$ python config-custom.py -h
usage: config-custom.py [-h] [-c CONFIGFILE] [-s SECTION] [-w URL]
                        [--idsurl IDSURL] [--no-check-certificate]
                        [--http-proxy HTTP_PROXY] [--https-proxy HTTPS_PROXY]
```

(continues on next page)

(continued from previous page)

```
                        [--no-proxy NO_PROXY] [-a AUTH] [-u USERNAME] [-P]
                        [-p PASSWORD] [-o OUTFILE]

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIGFILE, --configfile CONFIGFILE
                        config file
  -s SECTION, --configsection SECTION
                        section in the config file
  -w URL, --url URL     URL to the web service description
  --idsurl IDSURL       URL to the ICAT Data Service
  --no-check-certificate
                        don't verify the server certificate
  --http-proxy HTTP_PROXY
                        proxy to use for http requests
  --https-proxy HTTPS_PROXY
                        proxy to use for https requests
  --no-proxy NO_PROXY   list of exclusions for proxy use
  -a AUTH, --auth AUTH  authentication plugin
  -u USERNAME, --user USERNAME
                        username
  -P, --prompt-pass     prompt for the password
  -p PASSWORD, --pass PASSWORD
                        password
  -o OUTFILE, --outputfile OUTFILE
                        output file name or '-' for stdout
```

This new option is optional, so the program can be used as before:

```
$ python config-custom.py -s myicat_jdoe
Login to https://icat.example.com:8181 was successful.
User: db/jdoe
```

If we add the option on the command line, it has the expected effect:

```
$ python config-custom.py -s myicat_jdoe -o out.txt
$ cat out.txt
Login to https://icat.example.com:8181 was successful.
User: db/jdoe
```

Alternatively, we may also specify the option in the configuration file as follows:

```
[myicat_jdoe]
url = https://icat.example.com:8181
auth = db
username = jdoe
password = secret
idsurl = https://icat.example.com:8181
#checkCert = No
outfile = out.txt
```

Flag configuration variables

Instead of passing a string value to our program, we can also define different variable types using the *type* parameter. Among other things, this allows us to pass boolean/flag parameters. Let's add another configuration variable to our example program that lets us control the output via a flag:

```
#! /usr/bin/python
```

(continues on next page)

(continued from previous page)

```
from __future__ import print_function
import sys
import icat
import icat.config

config = icat.config.Config(ids="optional")
config.add_variable("outfile", ("-o", "--outputfile"),
                    dict(help="output file name or '-' for stdout"),
                    default="-")
config.add_variable("hide", ["--hide-user-name"],
                    dict(help="do not display the user after login"),
                    default=False, type=icat.config.flag)
client, conf = config.getconfig()
client.login(conf.auth, conf.credentials)

if conf.outfile == "-":
    out = sys.stdout
else:
    out = open(conf.outfile, "wt")

print("Login to %s was successful." % (conf.url), file=out)
if not conf.hide:
    print("User: %s" % (client.getUserName()), file=out)

out.close()
```

If we call our program normally, we get the same output as before:

```
$ python config-flag.py -s myicat_jdoe
Login to https://icat.example.com:8181 was successful.
User: db/jdoe
```

But if we pass the flag parameter, it produces a different output:

```
$ python config-flag.py -s myicat_jdoe --hide-user-name
Login to https://icat.example.com:8181 was successful.
```

A flag type configuration variable also adds a negated form of the command line flag:

```
$ python config-flag.py -s myicat_jdoe --no-hide-user-name
Login to https://icat.example.com:8181 was successful.
User: db/jdoe
```

This may look somewhat pointless at first glance as it only affirms the default. It becomes useful if we set this flag in the configuration file as in:

```
[myicat_jdoe]
url = https://icat.example.com:8181
auth = db
username = jdoe
password = secret
idsurl = https://icat.example.com:8181
#checkCert = No
hide = true
```

In that case we can override this setting on the command line with `--no-hide-user-name`.

Defining sub-commands

Many programs split up their functionality into sub-commands. For instance, the `git` program can be called as `git clone`, `git checkout`, `git commit`, and so on. In general, each sub-command will take their own

set of configuration variables.

You can create programs like this and manage the configuration of each sub-command with `icat.config` using the `add_subcommands()` method. It adds a special `ConfigSubCmds` configuration variable representing the sub-command. This object provides the `add_subconfig()` method to register a new sub-command value. On the sub-config object in turn you can then define specific configuration variables using the familiar `add_variable()` method.

To put it all together, consider the following example program:

```
#! /usr/bin/python

from __future__ import print_function
import icat
import icat.config

config = icat.config.Config(ids="optional")

# add a global configuration variable 'entity' common for all sub-commands
config.add_variable("entity", ("-e", "--entity"),
                    dict(help="an entity from the ICAT schema",
                         choices=["User", "Study"]))

# add the configuration variable representing the sub-commands
subcmds = config.add_subcommands("mode")

# register three possible values for the sub-commands {list,create,delete}
subconfig_list = subcmds.add_subconfig("list",
                                       dict(help="list existing ICAT objects"))
subconfig_create = subcmds.add_subconfig("create",
                                         dict(help="create a new ICAT object"))
subconfig_delete = subcmds.add_subconfig("delete",
                                         dict(help="delete an ICAT object"))

# add two additional configuration variables 'name' and 'id' that are
# specific to the 'create' and 'delete' sub-commands respectively.
subconfig_create.add_variable("name", ("-n", "--name"),
                              dict(help="name for the new ICAT object"))
subconfig_delete.add_variable("id", ("-i", "--id"),
                              dict(help="ID of the ICAT object"))

client, conf = config.getconfig()
client.login(conf.auth, conf.credentials)

# check which sub-command (mode) was called
if conf.mode.name == "list":
    print("listing existing %s objects..." % conf.entity)
    print(client.search(conf.entity))
elif conf.mode.name == "create":
    print("creating a new %s object named %s..." % (conf.entity, conf.name))
    obj = client.new(conf.entity.lower(), name=conf.name)
    obj.create()
elif conf.mode.name == "delete":
    print("deleting the %s object with ID %s..." % (conf.entity, conf.id))
    obj = client.get(conf.entity, conf.id)
    client.delete(obj)

print("done")
```

If we check the available commands for the above program, our three sub-commands should be listed:

```
$ python config-sub-commands.py -h
usage: config-sub-commands.py [-h] [-c CONFIGFILE] [-s SECTION] [-w URL]
```

(continues on next page)

(continued from previous page)

```
                              [--idsurl IDSURL] [--no-check-certificate]
                              [--http-proxy HTTP_PROXY]
                              [--https-proxy HTTPS_PROXY]
                              [--no-proxy NO_PROXY] [-a AUTH] [-u USERNAME]
                              [-P] [-p PASSWORD] [-e {User,Study}]
                              {list,create,delete} ...

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIGFILE, --configfile CONFIGFILE
                        config file
  -s SECTION, --configsection SECTION
                        section in the config file
  -w URL, --url URL     URL to the web service description
  --idsurl IDSURL       URL to the ICAT Data Service
  --no-check-certificate
                        don't verify the server certificate
  --http-proxy HTTP_PROXY
                        proxy to use for http requests
  --https-proxy HTTPS_PROXY
                        proxy to use for https requests
  --no-proxy NO_PROXY   list of exclusions for proxy use
  -a AUTH, --auth AUTH  authentication plugin
  -u USERNAME, --user USERNAME
                        username
  -P, --prompt-pass     prompt for the password
  -p PASSWORD, --pass PASSWORD
                        password
  -e {User,Study}, --entity {User,Study}
                        an entity from the ICAT schema

subcommands:
  {list,create,delete}
    list                list existing ICAT objects
    create              create a new ICAT object
    delete              delete an ICAT object
```

This looks good. Let's try calling our program with the *list* sub-command. Of course we must also provide a *section* from our config file (`-s SECTION`) as well as the *entity* variable (`-e {User,Study}`) we defined earlier:

```
$ python config-sub-commands.py -s myicat_root -e User list
listing existing User objects...
[(user){
   createId = "simple/root"
   createTime = 2020-02-20 15:55:39+01:00
   id = 1
   modId = "simple/root"
   modTime = 2020-02-20 15:55:39+01:00
   fullName = "Aelius Cordus"
   name = "db/acord"
 }, (user){
   createId = "simple/root"
   createTime = 2020-02-20 15:55:39+01:00
   id = 2
   modId = "simple/root"
   modTime = 2020-02-20 15:55:39+01:00
   fullName = "Arnold Hau"
   name = "db/ahau"
 }, (user){
   createId = "simple/root"
```

(continues on next page)

(continued from previous page)

```
  createTime = 2020-02-20 15:55:39+01:00
  id = 3
  modId = "simple/root"
  modTime = 2020-02-20 15:55:39+01:00
  fullName = "Jean-Baptiste Botul"
  name = "db/jbotu"
 }, (user){
  createId = "simple/root"
  createTime = 2020-02-20 15:55:39+01:00
  id = 4
  modId = "simple/root"
  modTime = 2020-02-20 15:55:39+01:00
  fullName = "John Doe"
  name = "db/jdoe"
 }, (user){
  createId = "simple/root"
  createTime = 2020-02-20 15:55:39+01:00
  id = 5
  modId = "simple/root"
  modTime = 2020-02-20 15:55:39+01:00
  fullName = "Nicolas Bourbaki"
  name = "db/nbour"
 }, (user){
  createId = "simple/root"
  createTime = 2020-02-20 15:55:39+01:00
  id = 6
  modId = "simple/root"
  modTime = 2020-02-20 15:55:39+01:00
  fullName = "Rudolph Beck-Dülmen"
  name = "db/rbeck"
 }, (user){
  createId = "simple/root"
  createTime = 2020-02-20 15:55:39+01:00
  id = 7
  modId = "simple/root"
  modTime = 2020-02-20 15:55:39+01:00
  fullName = "IDS reader"
  name = "simple/idsreader"
 }, (user){
  createId = "simple/root"
  createTime = 2020-02-20 15:55:39+01:00
  id = 8
  modId = "simple/root"
  modTime = 2020-02-20 15:55:39+01:00
  fullName = "Root"
  name = "simple/root"
 }, (user){
  createId = "simple/root"
  createTime = 2020-02-20 15:55:39+01:00
  id = 9
  modId = "simple/root"
  modTime = 2020-02-20 15:55:39+01:00
  fullName = "User Office"
  name = "simple/useroffice"
 }]
done
```

We see the users defined in the example content created in the previous tutorial sections. Let's add a new user. We will use the *create* sub-command to do this. Earlier, we defined a configuration variable *name* (`-n NAME`) that is specific to the *create* sub-command. We can check this by calling:

```
$ python config-sub-commands.py create -h
usage: config-sub-commands.py create [-h] [-n NAME]

optional arguments:
  -h, --help            show this help message and exit
  -n NAME, --name NAME  name for the new ICAT object
```

Let's create a new `User` object named "db/alice". Note that we must provide the 'global' configuration variables (*section* and *entity*) before the sub-command, and the sub-command-specific option (*name*) after it:

```
$ python config-sub-commands.py -s myicat_root -e User create -n db/alice
creating a new User object named db/alice...
done
```

If we now list the `User` objects again, we can see a new object with name "db/alice":

```
$ python config-sub-commands.py -s myicat_root -e User list
listing existing User objects...
[(user){
   createId = "simple/root"
   createTime = 2020-02-20 15:55:39+01:00
   id = 1
   modId = "simple/root"
   modTime = 2020-02-20 15:55:39+01:00
   fullName = "Aelius Cordus"
   name = "db/acord"
 }, (user){
   createId = "simple/root"
   createTime = 2020-02-20 15:55:39+01:00
   id = 2
   modId = "simple/root"
   modTime = 2020-02-20 15:55:39+01:00
   fullName = "Arnold Hau"
   name = "db/ahau"
 }, (user){
   createId = "simple/root"
   createTime = 2020-02-20 15:55:39+01:00
   id = 3
   modId = "simple/root"
   modTime = 2020-02-20 15:55:39+01:00
   fullName = "Jean-Baptiste Botul"
   name = "db/jbotu"
 }, (user){
   createId = "simple/root"
   createTime = 2020-02-20 15:55:39+01:00
   id = 4
   modId = "simple/root"
   modTime = 2020-02-20 15:55:39+01:00
   fullName = "John Doe"
   name = "db/jdoe"
 }, (user){
   createId = "simple/root"
   createTime = 2020-02-20 15:55:39+01:00
   id = 5
   modId = "simple/root"
   modTime = 2020-02-20 15:55:39+01:00
   fullName = "Nicolas Bourbaki"
   name = "db/nbour"
 }, (user){
   createId = "simple/root"
   createTime = 2020-02-20 15:55:39+01:00
   id = 6
```

(continues on next page)

(continued from previous page)

```
    modId = "simple/root"
    modTime = 2020-02-20 15:55:39+01:00
    fullName = "Rudolph Beck-Dülmen"
    name = "db/rbeck"
  }, (user){
    createId = "simple/root"
    createTime = 2020-02-20 15:55:39+01:00
    id = 7
    modId = "simple/root"
    modTime = 2020-02-20 15:55:39+01:00
    fullName = "IDS reader"
    name = "simple/idsreader"
  }, (user){
    createId = "simple/root"
    createTime = 2020-02-20 15:55:39+01:00
    id = 8
    modId = "simple/root"
    modTime = 2020-02-20 15:55:39+01:00
    fullName = "Root"
    name = "simple/root"
  }, (user){
    createId = "simple/root"
    createTime = 2020-02-20 15:55:39+01:00
    id = 9
    modId = "simple/root"
    modTime = 2020-02-20 15:55:39+01:00
    fullName = "User Office"
    name = "simple/useroffice"
  }, (user){
    createId = "simple/root"
    createTime = 2020-02-21 18:34:29+01:00
    id = 10
    modId = "simple/root"
    modTime = 2020-02-21 18:34:29+01:00
    name = "db/alice"
  }]
done
```

Finally, let's delete this new object using the *delete* sub-command. To do this, we must specify the sub-command-specific configuration variable *id* (`-i ID`). In the above output, we can see that the object's ID is 10, so we write:

```
$ python config-sub-commands.py -s myicat_root -e User delete -i 10
deleting the User object with ID 10...
done
```

1.2 Module reference

This section provides a reference for the python-icat modules.

1.2.1 Modules defining the python-icat API

These modules define classes or functions that common python-icat programs interact with.

`icat.client` — Provide the Client class

The `icat.client` defines the `Client` class that manages the interaction with an ICAT service as a client.

class icat.client.**Client**(*url*, *idsurl=None*, *checkCert=True*, *caFile=None*, *caPath=None*, *sslContext=None*, *proxy=None*, ***kwargs*)

Bases: suds.client.Client

A client accessing an ICAT service.

This is a subclass of suds.client.Client and inherits most of its behavior. It adds methods for the instantiation of ICAT entities and implementations of the ICAT API methods.

Parameters

- **url** (str) – The URL pointing to the WSDL of the ICAT service. If the URL does not contain a path, e.g. contains only a URL scheme and network location part, a default path is assumend.
- **idsurl** (str) – The URL pointing to the IDS service. If set, an *icat.ids.IDSClient* instance will be created.
- **checkCert** (bool) – Flag whether the server's SSL certificate should be verified if connecting ICAT with HTTPS.
- **caFile** (str) – Path to a file of concatenated trusted CA certificates. If neither *caFile* nor *caPath* is set, the system's default certificates will be used.
- **caPath** (str) – Path to a directory containing trusted CA certificates. If neither *caFile* nor *caPath* is set, the system's default certificates will be used.
- **sslContext** (ssl.SSLContext) – A SSL context describing various SSL options to be used in HTTPS connections. If set, this will override *checkCert*, *caFile*, and *caPath*.
- **proxy** (dict) – HTTP proxy settings. A map with the keys *http_proxy* and *https_proxy* and the URL of the respective proxy to use as values.
- **kwargs** – additional keyword arguments that will be passed to suds.client.Client, see suds.options.Options for details.

Class attributes

Register
The register of all active clients.

AutoRefreshRemain
Number of minutes to leave in the session before automatic refresh should be called.

Instance attributes

url
The URL to the web service description of the ICAT server.

kwargs
A copy of the kwargs used in the constructor.

apiversion
Version of the ICAT server this client connects to.

autoLogout
Flag whether the client should logout automatically on exit.

ids
The *icat.ids.IDSClient* instance used for IDS calls.

sessionId
The session id as returned from *login()*.

sslContext
The ssl.SSLContext instance that has been used to establish the HTTPS conection to the ICAT and IDS server. This is None for old Python versions that do not have the ssl.SSLContext class.

typemap
A `dict` that maps type names from the ICAT WSDL schema to the corresponding classes in the `icat.entity.Entity` hierarchy.

Class and instance methods

classmethod cleanupall()
Cleanup all class instances.

Call `cleanup()` on all registered class instances, e.g. on all clients that have not yet been cleaned up.

cleanup()
Release resources allocated by the client.

Logout from the active ICAT session (if `autoLogout` is `True`). The client should not be used any more after calling this method.

add_ids(*url*, *proxy=None*)
Add the URL to an ICAT Data Service.

clone()
Create a clone.

Return a clone of the `Client` object. That is, a client that connects to the same ICAT server and has been created with the same kwargs. The clone will be in the state as returned from the constructor. In particular, it does not share the same session if this client object is logged in.

Returns a clone of the client object.

Return type `Client`

new(*obj*, ***kwargs*)
Instantiate a new `icat.entity.Entity` object.

If obj is a string, take it as the name of an instance type. Create a new instance object of this type and lookup the class for the object in the `typemap` using this type name. If obj is an instance object, look up its class name in the typemap to determine the class. If obj is `None`, do nothing and return `None`.

Parameters

- **obj** (`suds.sudsobject.Object` or `str`) – either a Suds instance object, a name of an instance type, or `None`.
- **kwargs** – attributes passed to the constructor of `icat.entity.Entity`.

Returns the new entity object or `None`.

Return type `icat.entity.Entity`

Raises ***EntityTypeError*** – if obj is neither a valid instance object, nor a valid name of an entity type, nor None.

getEntityClass(*name*)
Return the Entity class corresponding to a BeanName.

getEntity(*obj*)
Get the corresponding `icat.entity.Entity` for an object.

if obj is a *fieldSet*, return a tuple of the fields. If obj is any other Suds instance object, create a new entity object with `new()`. Otherwise do nothing and return obj unchanged.

Parameters **obj** (`suds.sudsobject.Object` or any type) – either a Suds instance object or anything.

Returns the new entity object or obj.

Return type `tuple` or `icat.entity.Entity` or any type

Changed in version 0.18.0: add support of *fieldSet*.

Changed in version 0.18.1: changed the return type from `list` to `tuple` in the case of *fieldSet*.

ICAT API methods

These methods implement the low level API calls of the ICAT server. See the documentation in the ICAT SOAP Manual. (Note: the Python examples in that manual are based on plain Suds, not on python-icat.)

login(*auth*, *credentials*)

logout()

create(*bean*)

createMany(*beans*)

delete(*bean*)

deleteMany(*beans*)

get(*query*, *primaryKey*)

getApiVersion()

getAuthenticatorInfo()

getEntityInfo(*beanName*)

getEntityNames()

getProperties()

getRemainingMinutes()

getUserName()

getVersion()

isAccessAllowed(*bean*, *accessType*)

refresh()

search(*query*)

update(*bean*)

Custom API methods

These higher level methods build on top of the ICAT API methods.

autoRefresh()

Call `refresh()` only if needed.

Call `refresh()` if less then `AutoRefreshRemain` minutes remain in the current session. Do not make any client calls if not. This method is supposed to be very cheap if enough time remains in the session so that it may be called often in a loop without causing too much needless load.

assertedSearch(*query*, *assertmin=1*, *assertmax=1*)

Search with an assertion on the result.

Perform a search and verify that the number of items found lies within the bounds of *assertmin* and *assertmax*. Raise an error if this assertion fails.

Parameters

- **query** (`icat.query.Query` or `str`) – the search query.
- **assertmin** (`int`) – minimum number of expected results.
- **assertmax** (`int`) – maximum number of expected results. A value of `None` is treated as infinity.

Returns search result.

Return type `list`

Raises

- **`ValueError`** – in case of inconsistent arguments.
- ***`SearchAssertionError`*** – if the assertion on the number of results fails.
- ***`ICATError`*** – in case of exceptions raised by the ICAT server.

`searchChunked`(*query*, *skip=0*, *count=None*, *chunksize=100*)

Search the ICAT server.

Call the ICAT `search()` API method, limiting the number of results in each call and repeat the call as often as needed to retrieve all the results.

This can be used as a drop in replacement for the search API method most of the times. It avoids the error if the number of items in the result exceeds the limit imposed by the ICAT server. There are a few subtle differences though: the query must not contain a LIMIT clause (use the skip and count arguments instead) and should contain an ORDER BY clause. The return value is a generator yielding successively the items in the search result rather then a list. The individual search calls are done lazily, e.g. they are not done until needed to yield the next item from the generator.

Note: The result may be defective (omissions, duplicates) if the content in the ICAT server changes between individual search calls in a way that would affect the result. It is a common mistake when looping over items returned from this method to have code with side effects on the search result in the body of the loop. Example:

```
# Mark all datasets as complete
# This will *not* work as expected!
query = Query(client, "Dataset", conditions={
    "complete": "= False"
}, includes="1", order=["id"])
for ds in client.searchChunked(query):
    ds.complete = True
    ds.update()
```

This should rather be formulated as:

```
# Mark all datasets as complete
# This version works!
query = Query(client, "Dataset", includes="1", order=["id"])
for ds in client.searchChunked(query):
    if not ds.complete:
        continue
    ds.complete = True
    ds.update()
```

Parameters

- **`query`** (*`icat.query.Query`* or `str`) – the search query.
- **`skip`** (`int`) – offset from within the full list of available results.
- **`count`** (`int`) – maximum number of items to return. A value of `None` means no limit.
- **`chunksize`** (`int`) – number of items to query in each search call. This is an internal tuning parameter and does not affect the result.

Returns a generator that successively yields the items in the search result.

Return type generator

searchUniqueKey (*key*, *objindex=None*)

Search the object that belongs to a unique key.

This is in a sense the inverse method to `icat.entity.Entity.getUniqueKey()`, the key must previously have been generated by it. This method searches the entity object that the key has been generated for from the server.

if objindex is not `None`, it is used as a cache of previously retrieved objects. It must be a dict that maps keys to entity objects. The object retrieved by this method call will be added to this index.

Parameters

- **key** (`str`) – the unique key of the object to search for.
- **objindex** (`dict`) – cache of entity objects.

Returns the object corresponding to the key.

Return type `icat.entity.Entity`

Raises

- ***SearchResultError*** – if the object has not been found.
- **ValueError** – if the key is not well formed.

searchMatching (*obj*, *includes=None*)

Search the matching object.

Search the object from the ICAT server that matches the given object in the uniqueness constraint.

```
>>> dataset = client.new("dataset", investigation=inv, name=dsname)
>>> dataset = client.searchMatching(dataset)
>>> dataset.id
172383
```

Parameters

- **obj** (`icat.entity.Entity`) – an entity object having the attrinutes for the uniqueness constraint set accordingly.
- **includes** (iterable of `str`) – list of related objects to add to the INCLUDE clause of the search query. See `icat.query.Query.addIncludes()` for details.

Returns the corresponding object.

Return type `icat.entity.Entity`

Raises

- ***SearchResultError*** – if the object has not been found.
- **ValueError** – if the object's class does not have a uniqueness constraint or if any attribute needed for the constraint is not set.

createUser (*name*, *search=False*, ***kwargs*)

Search a user by name or create a new user.

If search is `True` search a user by the given name. If search is `False` or no user is found, create a new user.

Parameters

- **name** (`str`) – username.
- **search** (`bool`) – flag whether a user should be searched first.
- **kwargs** – attributes of the user passed to *new*.

Returns the user.

Return type `icat.entity.Entity`

`createGroup`(*name*, *users=()*)

Create a group and add users to it.

Parameters

- **`name`** (`str`) – the name of the group.
- **`users`** (`list` of `icat.entity.Entity`) – a list of users.

Returns the group.

Return type `icat.entity.Entity`

`createRules`(*crudFlags*, *what*, *group=None*)

Create access rules.

Parameters

- **`crudFlags`** (`str`) – access mode.
- **`what`** (`list`) – list of items subject to the rule. The items must be either ICAT search expression strings or `icat.query.Query` objects.
- **`group`** (`icat.entity.Entity`) – the group that should be granted access or `None` for everybody.

Returns list of the ids of the created rules.

Return type `list` of `int`

Custom IDS methods

These methods provide the most commonly needed IDS functionality and build on top of the low level IDS API methods provided by `icat.ids.IDSClient`.

`putData`(*infile*, *datafile*)

Upload a datafile to IDS.

The content of the file to upload is read from *infile*, either directly if it is an open file, or a file by that name will be opened for reading.

The *datafile* object must be initialized but not yet created at the ICAT server. It will be created by the IDS. The ids of the Dataset and the DatafileFormat as well as the attributes description, doi, datafileCreateTime, and datafileModTime will be taken from *datafile*. If datafileModTime is not set, the method will try to `os.fstat()` *infile* and use the last modification time from the file system, if available. If datafileCreateTime is not set, it will be set to datafileModTime.

Note that only the attributes datafileFormat, dataset, description, doi, datafileCreateTime, and datafileModTime of *datafile* will be taken into account as described above. All other attributes are ignored and the Datafile object created in the ICAT server might end up with different values for those other attribues.

Parameters

- **`infile`** (`file` or `str`) – either a file opened for reading or a file name.
- **`datafile`** (`icat.entity.Entity`) – A Datafile object.

Returns The Datafile object created by IDS.

Return type `icat.entity.Entity`

`getData`(*objs*, *compressFlag=False*, *zipFlag=False*, *outname=None*, *offset=0*)

Retrieve the requested data from IDS.

The data objects to retrieve are given in objs. This can be any combination of single Datafiles, Datasets, or complete Investigations.

Parameters

- **objs** (dict, list of *icat.entity.Entity*, *icat.ids.DataSelection*, or str) – either a dict having some of the keys *investigationIds*, *datasetIds*, and *datafileIds* with a list of object ids as value respectively, or a list of entity objects, or a data selection, or an id returned by *prepareData()*.
- **compressFlag** (bool) – flag whether to use a zip format with an implementation defined compression level, otherwise use no (or minimal) compression.
- **zipFlag** (bool) – flag whether return a single datafile in zip format. For multiple files zip format is always used.
- **outname** (str) – the preferred name for the downloaded file to specify in the Content-Disposition header.
- **offset** (int) – if larger then zero, add Range header to the HTTP request with the indicated bytes offset.

Returns a file-like object as returned by urllib.request.OpenerDirector.open().

getDataUrl(*objs*, *compressFlag=False*, *zipFlag=False*, *outname=None*)

Get the URL to retrieve the requested data from IDS.

The data objects to retrieve are given in objs. This can be any combination of single Datafiles, Datasets, or complete Investigations.

Note that the URL contains the session id of the current ICAT session. It will become invalid if the client logs out.

Parameters

- **objs** (dict, list of *icat.entity.Entity*, *icat.ids.DataSelection*, or str) – either a dict having some of the keys *investigationIds*, *datasetIds*, and *datafileIds* with a list of object ids as value respectively, or a list of entity objects, or a data selection, or an id returned by *prepareData()*.
- **compressFlag** (bool) – flag whether to use a zip format with an implementation defined compression level, otherwise use no (or minimal) compression.
- **zipFlag** (bool) – flag whether return a single datafile in zip format. For multiple files zip format is always used.
- **outname** (str) – the preferred name for the downloaded file to specify in the Content-Disposition header.

Returns the URL for the data at the IDS.

Return type str

prepareData(*objs*, *compressFlag=False*, *zipFlag=False*)

Prepare data at IDS to be retrieved in subsequent calls.

The data objects to retrieve are given in objs. This can be any combination of single Datafiles, Datasets, or complete Investigations.

Parameters

- **objs** (dict, list of *icat.entity.Entity*, or *icat.ids.DataSelection*) – either a dict having some of the keys *investigationIds*, *datasetIds*, and *datafileIds* with a list of object ids as value respectively, or a list of entity objects, or a data selection.
- **compressFlag** (bool) – flag whether to use a zip format with an implementation defined compression level, otherwise use no (or minimal) compression.
- **zipFlag** (bool) – flag whether return a single datafile in zip format. For multiple files zip format is always used.

Returns *preparedId*, an opaque string which may be used as an argument to `isDataPrepared()` and `getData()` calls.

Return type `str`

isDataPrepared(*preparedId*)
Check if prepared data is ready at IDS.

Parameters **preparedId** (`str`) – the id returned by `prepareData()`.

Returns `True` if the data is ready, otherwise `False`.

Return type `bool`

getPreparedData(*preparedId*, *outname=None*, *offset=0*)
Retrieve prepared data from IDS.

Parameters

- **preparedId** (`str`) – the id returned by `prepareData()`.
- **outname** (`str`) – the preferred name for the downloaded file to specify in the Content-Disposition header.
- **offset** (`int`) – if larger then zero, add Range header to the HTTP request with the indicated bytes offset.

Returns a file-like object as returned by `urllib.request.OpenerDirector.open()`.

Deprecated since version 0.17.0: Call `getData()` instead.

getPreparedDataUrl(*preparedId*, *outname=None*)
Get the URL to retrieve prepared data from IDS.

Parameters

- **preparedId** (`str`) – the id returned by `prepareData()`.
- **outname** (`str`) – the preferred name for the downloaded file to specify in the Content-Disposition header.

Returns the URL for tha data at the IDS.

Return type `str`

Deprecated since version 0.17.0: Call `getDataUrl()` instead.

deleteData(*objs*)
Delete data from IDS.

The data objects to delete are given in objs. This can be any combination of single Datafiles, Datasets, or complete Investigations.

Parameters **objs** (`dict`, `list` of `icat.entity.Entity`, or `icat.ids.DataSelection`) – either a dict having some of the keys *investigationIds*, *datasetIds*, and *datafileIds* with a list of object ids as value respectively, or a list of entity objects, or a data selection.

`icat.config` — Manage configuration

This module reads configuration variables from different sources, such as command line arguments, environment variables, and configuration files. A set of configuration variables that any ICAT client program typically needs is predefined. Custom configuration variables may be added. The main class that client programs interact with is `icat.config.Config`.

icat.config.**cfgdirs**
Search path for the configuration file. The value depends on the operating system.

icat.config.**cfgfile = 'icat.cfg'**
Configuration file name

icat.config.**defaultsection = None**
Default value for *configSection*

icat.config.**boolean**(*value*)
Test truth value.

Convert the string representation of a truth value, such as '0', '1', 'yes', 'no', 'true', or 'false' to `bool`. This function is suitable to be passed as type to `icat.config.BaseConfig.add_variable()`.

icat.config.**flag**
Variant of `icat.config.boolean()` that defines two command line arguments to switch the value on and off respectively. May be passed as type to `icat.config.BaseConfig.add_variable()`.

icat.config.**cfgpath**(*p*)
Search for a file in some default directories.

The argument *p* should be a file path name. If *p* is absolute, it will be returned unchanged. Otherwise, *p* will be resolved against the directories in `icat.config.cfgdirs` in reversed order. If a file with the resulting path is found to exist, this path will be returned, first match wins. If no file exists in any of the directories, *p* will be returned unchanged.

This function is suitable to be passed as *type* argument to `icat.config.BaseConfig.add_variable()`.

class icat.config.**ConfigVariable**(*name*, *envvar*, *optional*, *default*, *convert*, *subst*)
Bases: `object`

Describe a configuration variable. Configuration variables are created in `icat.config.BaseConfig.add_variable()` and control the behavior of `icat.config.Config.getconfig()`.

class icat.config.**ConfigSubCmds**(*name*, *optional*, *config*, *subparsers*)
Bases: `icat.config.ConfigVariable`

A special configuration variable that selects a subcommand. These subcommand configuration variables are created in `icat.config.BaseConfig.add_subcommand()`. Possible values for the subcommand are then registered calling the `add_subconfig()` method.

add_subconfig(*name*, *arg_kws=None*, *func=None*)
Add a comand to a set of subcommands defined with `icat.config.BaseConfig.add_subcommands()`.

Parameters

- **name** (`str`) – the name of the command.
- **arg_kws** (`dict`) – constructor arguments to be passed to `argparse.ArgumentParser()` to create the subparser. Mostly useful to set *help*.
- **func** – any custom value. The configuration value representing the subcommands in the `icat.config.Configuration` object returned by `icat.config.Config.getconfig()` will have an attribute *func* with this value if this command has been selected. Most useful to set this to a callable that implements the command.

Returns a subconfig object that allows to set specific configuration variables for the command.

Return type `icat.config.SubConfig`

Raises `ValueError` – if the name is already defined.

class icat.config.**Configuration**(*config*)
Bases: `object`

Provide a name space to store the configuration.

icat.config.Config.getconfig() returns a Configuration object having the configuration values stored in the respective attributes.

as_dict()
Return the configuration as a dict.

class icat.config.**BaseConfig**(*argparser*)
Bases: object

Abstract base class for *icat.config.Config* and *icat.config.SubConfig*. This class defines the common API. It is not intended to be instantiated directly.

Class attributes (read only):

ReservedVariables = ['configDir', 'credentials']
Reserved names of configuration variables.

Instance methods:

add_variable(*name*, *arg_opts=()*, *arg_kws=None*, *envvar=None*, *optional=False*, *default=None*, *type=None*, *subst=False*)
Defines a new configuration variable.

Note that the value of some configuration variable may influence the evaluation of other variables. For instance, if *configFile* and *configSection* are set, the values for other configuration variables are searched in this configuration file. Thus, the evaluation order of the configuration variables is important. The variables are evaluated in the order that this method is called to define the respective variable.

Call argparse.ArgumentParser.add_argument() to add a new command line argument if *arg_opts* is set.

Parameters

- **name** (str) – the name of the variable. This will be used as the name of the attribute of *icat.config.Configuration* returned by *icat.config.Config.getconfig()* and as the name of the option to be looked for in the configuration file. The name must be unique and not in icat.config.Config.ReservedVariables. If *arg_opts* corresponds to a positional argument, the name must be equal to this argument name.
- **arg_opts** (tuple of str) – command line flags associated with this variable. This will be passed as *name or flags* to argparse.ArgumentParser.add_argument().
- **arg_kws** (dict) – keyword arguments to be passed to argparse.ArgumentParser.add_argument().
- **envvar** (str) – name of the environment variable or None. If set, the value for the variable may be set from the respective environment variable.
- **optional** (bool) – flag wether the configuration variable is optional. If set to False and *default* is None the variable is mandatory.
- **default** – default value.
- **type** (*callable*) – type to which the value should be converted. This must be a callable that accepts one string argument and returns the desired value. Python builtins int and float or some standard library classes such as Path are fine. If set to None, the string value is taken as is. If applicable, the default value will also be passed through this conversion. The special value *icat.config.flag* may also be used to indicate a variant of *icat.config.boolean()*.
- **subst** (bool) – flag wether substitution of other configuration variables using the % interpolation operator shall be performed. If set to True, the value may contain conversion specifications such as %(othervar)s. This will then be substituted by the value of *othervar*. The referenced variable must have been defined earlier.

Returns the new configuration variable object.

Return type `icat.config.ConfigVariable`

Raises

- **RuntimeError** – if this objects already has subcommands defined with `icat.config.BaseConfig.add_subcommands()`.
- **ValueError** – if the name is not valid.

See the documentation of the `argparse` standard library module for details on *arg_opts* and *arg_kws*.

add_subcommands(*name='subcmd'*, *arg_kws=None*, *optional=False*)

Defines a new configuration variable to select subcommands.

Note: adding a subcommand variable must be the last action of this kind on a `icat.config.BaseConfig` object. Adding any more configuration variables or subcommand variables subsequently is not allowed. As a consequence, a `icat.config.BaseConfig` object may not have more then one subcommand variable.

Parameters

- **name** (`str`) – the name of the variable. This will be used as the name of the attribute of `icat.config.Configuration` returned by `icat.config.Config.getconfig()` and as the name of the option to be looked for in the configuration file. The name must be unique and not in `icat.config.Config.ReservedVariables`.
- **arg_kws** (`dict`) – keyword arguments to be passed to `argparse.ArgumentParser.add_subparsers()`. Mostly useful to set *title* or *help*. Note that *dest* will be overridden and set to the value of *name*.
- **optional** (`bool`) – flag wether providing a subcommand is optional.

Returns the new subcommand object.

Return type `icat.config.ConfigSubCmd`

Raises

- **RuntimeError** – if this objects already has subcommands.
- **ValueError** – if the name is not valid.

See the documentation of the `argparse` standard library module for details on *arg_kws*.

class `icat.config.`**Config**(*defaultvars=True*, *needlogin=True*, *ids='optional'*, *args=None*)

Bases: `icat.config.BaseConfig`

Set configuration variables.

Allow configuration variables to be set via command line arguments, environment variables, configuration files, and default values, in this order. In the case of a hidden credential such as a password, the user may also be prompted for a value. The first value found will be taken. Command line arguments and configuration files are read using the standard Python library modules `argparse` and `configparser` respectively, see the documentation of these modules for details on how to setup custom arguments or for the format of the configuration files.

The constructor sets up some predefined configuration variables.

Parameters

- **defaultvars** (`bool`) – if set to `False`, no default configuration variables other then *configFile* and *configSection* will be defined. The arguments *needlogin* and *ids* will be ignored in this case.

- **needlogin** (bool) – if set to False, the configuration variables *auth*, *username*, *password*, *promptPass*, and *credentials* will be left out.
- **ids** (bool or str) – the configuration variable *idsurl* will not be set up at all, or be set up as a mandatory, or as an optional variable, if this is set to False, to 'mandatory', or to 'optional' respectively.
- **args** (list of str) – list of command line arguments or None. If not set, the command line arguments will be taken from sys.argv.

Instance attributes (read only):

client
The *icat.client.Client* object initialized according to the configuration. This is also the first element in the return value if *getconfig()*.

client_kwargs
The keyword arguments that have been passed to the constructor of *client*.

Instance methods:

getconfig()
Get the configuration.

Parse the command line arguments, evaluate environment variables, read the configuration file, and apply default values (in this order) to get the value for each defined configuration variable. The first defined value found will be taken.

Returns a tuple with two items, a client initialized to connect to an ICAT server according to the configuration and an object having the configuration values set as attributes. The client will be None if the *defaultvars* constructor argument was False.

Return type tuple of *icat.client.Client* and *icat.config.Configuration*

Raises ***ConfigError*** – if *configFile* is defined but the file by this name can not be read, if *configSection* is defined but no section by this name could be found in the configuration file, if an invalid value is given to a variable, or if a mandatory variable is not defined.

class icat.config.**SubConfig**(*argparser*, *parent*, *name=None*, *func=None*)
Bases: *icat.config.BaseConfig*

Set configuration variables for a subcommand.

These subconfig objects are created in *icat.config.ConfigSubCmds.add_subconfig()*. Specific configuration variables for the respective subcommand may be added calling the *add_variable()* method inherited from *icat.config.BaseConfig*.

Predefined configuration variables

The constructor of *icat.config.Config* sets up the following set of configuration variables that an ICAT client typically needs:

configFile Name of the configuration file to read.

configSection Name of the section in the configuration file to apply. If not set, no values will be read from the configuration file.

url URL to the web service description of the ICAT server.

idsurl URL to the ICAT Data Service.

checkCert Verify the server certificate for HTTPS connections. Note that this requires either Python 2.7.9 or 3.2 or newer. With older Python version, this option has no effect.

http_proxy Proxy to use for HTTP requests.

https_proxy Proxy to use for HTTPS requests.

no_proxy Comma separated list of domain extensions proxy should not be used for.

auth Name of the authentication plugin to use for login.

username The ICAT user name.

password The user's password. Will prompt for the password if not set.

promptPass Prompt for the password.

A few derived variables are also set in `icat.config.Config.getconfig()`:

configDir the directory where (the last) configFile has been found.

credentials contains the credentials needed for the indicated authenticator (username and password if authenticator information is not available) suitable to be passed to `icat.client.Client.login()`.

Deprecated since version 0.13: The derived variable *configDir* is deprecated and will be removed in version 1.0.

The command line arguments, environment variables, and default values for the configuration variables are as follows:

Name	Command line	Environment	Default	Mandatory	Notes
configFile	`-c`, `--configfile`	`ICAT_CFG`	depends	no	(1)
configSection	`-s`, `--configsection`	`ICAT_CFG_SECTION`	`None`	no	(2)
url	`-w`, `--url`	`ICAT_SERVICE`		yes	
idsurl	`--idsurl`	`ICAT_DATA_SERVICE`	`None`	depends	(3)
checkCert	`--check-certificate`, `--no-check-certificate`		`True`	no	
http_proxy	`--http-proxy`	`http_proxy`	`None`	no	
https_proxy	`--https-proxy`	`https_proxy`	`None`	no	
no_proxy	`--no-proxy`	`no_proxy`	`None`	no	
auth	`-a`, `--auth`	`ICAT_AUTH`		yes	(4)
username	`-u`, `--user`	`ICAT_USER`		yes	(4),(5)
password	`-p`, `--pass`		interactive	yes	(4),(5),(6)
promptPass	`-P`, `--prompt-pass`		`False`	no	(4),(5),(6)

Mandatory means that an error will be raised in `icat.config.Config.getconfig()` if no value is found for the configuration variable in question.

Notes:

1. The default value for *configFile* depends on the operating system.
2. The default value for *configSection* may be changed in `icat.config.defaultsection`.
3. The configuration variable *idsurl* will not be set up at all, or be set up as a mandatory, or as an optional variable, if the *ids* argument to the constructor of `icat.config.Config` is set to `False`, to "mandatory", or to "optional" respectively.
4. If the argument *needlogin* to the constructor of `icat.config.Config` is set to `False`, the configuration variables *auth*, *username*, *password*, *promptPass*, and *credentials* will be left out.
5. If the ICAT server supports the `icat.client.Client.getAuthenticatorInfo()` API call (icat.server 4.9.0 and newer), the server will be queried about the credentials required for the authenticator indicated by the value of *auth*. The corresponding variables will be setup in the place of *username* and *password*. The variable *promptPass* will be setup only if any of the credentials is marked as hidden in the authenticator information.

6. The user will be prompted for the password if *promptPass* is `True`, if no *password* is provided in the command line or the configuration file, or if the *username*, but not the *password* has been provided by command line arguments. This applies accordingly to credentials marked as hidden if authenticator information is available from the server.

If the argument *defaultvars* to the constructor of `icat.config.Config` is set to `False`, no default configuration variables other then *configFile* and *configSection* will be defined. The configuration mechanism is still intact. In particular, custom configuration variables may be defined and reading the configuration file still works.

`icat.entities` — Provide classes corresponding to the ICAT schema

Provide the classes corresponding to the entities in the ICAT schema.

Entity classes defined in this module are derived from the abstract base class `icat.entity.Entity`. They override the class attributes `icat.entity.Entity.BeanName`, `icat.entity.Entity.Constraint`, `icat.entity.Entity.InstAttr`, `icat.entity.Entity.InstRel`, `icat.entity.Entity.InstMRel`, `icat.entity.Entity.AttrAlias`, and `icat.entity.Entity.SortAttrs` as appropriate.

Furthermore, custom methods are added to a few selected entity classes.

class `icat.entities.`**GroupingMixin**

Bases: `object`

Mixin class to define custom methods for Grouping objects.

addUsers(*users*)

Add users to the group.

getUsers(*attribute=None*)

Get the users in the group. If *attribute* is given, return the corresponding attribute for all users in the group, otherwise return the users.

class `icat.entities.`**InstrumentMixin**

Bases: `object`

Mixin class to define custom methods for Instrument objects.

addInstrumentScientists(*users*)

Add instrument scientists to the instrument.

getInstrumentScientists(*attribute=None*)

Get instrument scientists of the instrument. If *attribute* is given, return the corresponding attribute for all users related to the instrument, otherwise return the users.

class `icat.entities.`**InvestigationMixin**

Bases: `object`

Mixin class to define custom methods for Investigation objects.

addInstrument(*instrument*)

Add an instrument to the investigation.

addKeywords(*keywords*)

Add keywords to the investigation.

addInvestigationUsers(*users*, *role='Investigator'*)

Add investigation users.

class `icat.entities.`**Investigation44Mixin**

Bases: `icat.entities.InvestigationMixin`

Mixin class to define custom methods for Investigation objects for ICAT version 4.4.0 and later.

addInvestigationGroup(*group*, *role=None*)

Add an investigation group.

`icat.entities.getTypeMap(client)`

Generate a type map for the client.

Query the ICAT server about the entity classes defined in the schema and their attributes and relations. Generate corresponding Python classes representing these entities. The Python classes are based on `icat.entity.Entity`.

Parameters **client** (`icat.client.Client`) – a client object configured to connect to an ICAT server.

Returns a mapping of type names from the ICAT web service description to the corresponding Python classes. This mapping may be used as `icat.client.Client.typemap` for the client object.

Return type `dict`

`icat.entity` — Provide the Entity class

class `icat.entity.Entity(client, instance, **kwargs)`

Bases: `object`

The base of the classes representing the entities in the ICAT schema.

Entity is the abstract base for a hierarchy of classes representing the entities in the ICAT schema. It implements the basic behavior of these classes.

Each Entity object is connected to an instance of `suds.sudsobject.Object`, named *instance* in the following. Instances are created by Suds based on the ICAT WSDL schema. Entity objects mimic the behavior of the corresponding instance. Attribute accesses are proxied to the instance. A transparent conversion between Entity objects and Suds instances is performed where appropriate.

BeanName = None

Name of the entity in the ICAT schema, `None` for abstract classes.

Constraint = ('id',)

Attribute or relation names that form a uniqueness constraint.

SelfAttr = frozenset({'instance', 'client', 'validate'})

Attributes stored in the Entity object itself.

InstAttr = frozenset({'id'})

Attributes of the entity in the ICAT schema, stored in the instance.

MetaAttr = frozenset({'createId', 'modId', 'createTime', 'modTime'})

Readonly meta attributes, retrieved from the instance.

InstRel = frozenset()

Many to one relationships in the ICAT schema.

InstMRel = frozenset()

One to many relationships in the ICAT schema.

AttrAlias = {}

Map of alias names for attributes and relationships.

SortAttrs = None

List of attributes used for sorting. Uses Constraint if `None`.

validate = None

Hook to add a pre create validation method.

This may be set to a function that expects one argument, the entity object. It will then be called before creating the object at the ICAT server. The function is expected to raise an exception (preferably ValueError) in case of validation errors.

classmethod getInstance(*obj*)

Get the corresponding instance from an object.

classmethod getInstances(*objs*)
Translate a list of objects into the list of corresponding instances.

classmethod getAttrInfo(*client*, *attr*)
Get information on an attribute.

Query the EntityInfo of the entity from the ICAT server and retrieve information on one of the attributes from it.

Parameters

- **client** (*icat.client.Client*) – the ICAT client.
- **attr** (*str*) – name of the attribute.

Returns information on the attribute.

Raises **ValueError** – if this is an abstract entity class or if no attribute by that name is found.

classmethod getNaturalOrder(*client*)
Return a natural order for this class.

The order is a list of attributes suitable to be used in a ORDER BY clause in an ICAT search expression. The natural order is the one that is as close as possible to sorting the objects by the *__sortkey__()*. It is based on *Constraint* or the *SortAttrs*, if the latter are defined. In any case, one to many relationships and nullable many to one relationships are removed from the list.

copy()
Return a shallow copy of this entity object.

Create a new object that has all attributes set to a copy of the corresponding values of this object. The relations are copied by reference, i.e. the original and the copy refer to the same related object.

```
>>> inv = client.new("investigation", name="Investigation A")
>>> ds = client.new("dataset", investigation=inv, name="Dataset X")
>>> cds = ds.copy()
>>> cds.name
'Dataset X'
>>> cds.investigation.name
'Investigation A'
>>> cds.name = "Dataset Y"
>>> cds.investigation.name = "Investigation B"
>>> ds.name
'Dataset X'
>>> ds.investigation.name
'Investigation B'
```

__sortkey__()
Return a key for sorting.

This is suitable to be passed as *key* to the `list.sort()` method. E.g. if *l* is a list of *Entity* objects, you can sort it using:

```
>>> l.sort(key=icat.entity.Entity.__sortkey__)
```

as_dict()
Return a dict with the object's attributes.

getAttrType(*attr*)
Get the type of an attribute.

Query this object's EntityInfo from the ICAT server and retrieve the type of one of the attributes from it. In the case of a relation attribute, this yields the BeanName of the related object.

Parameters **attr** (*str*) – name of the attribute.

Returns name of the attribute type.

Return type `str`

Raises `ValueError` – if no attribute by that name is found.

truncateRelations()

Delete all relationships.

Delete all attributes having relationships to other objects from this object. Note that this is a local operation on the object in the client only. It does not affect the corresponding object at the ICAT server. This is useful if you only need to keep the object's attributes but not the (possibly large) tree of related objects in local memory.

getUniqueKey(*keyindex=None*)

Return a unique key.

The key is a string that is guaranteed to be unique for all entities in the ICAT. All attributes that form the uniqueness constraint must be set. A `icat.client.Client.search()` or `icat.client.Client.get()` with the appropriate include clause may be required before calling this method.

if *keyindex* is not `None`, it is used as a cache of previously generated keys. It must be a dict that maps entity ids to the keys returned by previous calls of `getUniqueKey()` on other entity objects. The newly generated key will be added to this index.

Parameters **keyindex** (`dict`) – cache of generated keys.

Returns a unique key.

Return type `str`

Raises `DataConsistencyError` – if a relation required in a constraint is not set.

create()

Call `icat.client.Client.create()` to create the object in the ICAT.

update()

Call `icat.client.Client.update()` to update the object in the ICAT.

get(*query=None*)

Call `icat.client.Client.get()` to get the object from the ICAT.

`icat.exception` — Exception handling

This module defines Python counterparts of the exceptions raised by ICAT or IDS server, as well as exceptions raised in python-icat.

Helper

icat.exception.**stripCause**(*e*)

Try to suppress misleading context from an exception.

Deprecated since version 0.14.0: Not needed any more, embedded in `icat.exception._BaseException` now.

exception icat.exception.**_BaseException**(**args*)

Bases: `Exception`

An exception that tries to suppress misleading context.

Exception Chaining and Embedded Tracebacks has been introduced with Python 3. Unfortunately the result is completely misleading most of the times. This class tries to strip the context from the exception traceback.

This is the common base class for for all exceptions defined in `icat.exception`, it is not intented to be raised directly.

Exceptions raised by the ICAT or IDS server

exception icat.exception.**ServerError**(*error*, *status=None*)
Bases: *icat.exception._BaseException*

Errors raised by either the ICAT or the IDS server.

This is the common base class for *icat.exception.ICATError* and *icat.exception.IDSError*, it is not intented to be raised directly.

exception icat.exception.**ICATError**(*error*, *status=None*)
Bases: *icat.exception.ServerError*

Base class for the errors raised by the ICAT server.

exception icat.exception.**ICATParameterError**(*error*, *status=None*)
Bases: *icat.exception.ICATError*

Generally indicates a problem with the arguments made to a call.

exception icat.exception.**ICATInternalError**(*error*, *status=None*)
Bases: *icat.exception.ICATError*

May be caused by network problems, database problems, GlassFish problems or bugs in ICAT.

exception icat.exception.**ICATPrivilegesError**(*error*, *status=None*)
Bases: *icat.exception.ICATError*

Indicates that the authorization rules have not matched your request.

exception icat.exception.**ICATNoObjectError**(*error*, *status=None*)
Bases: *icat.exception.ICATError*

Is thrown when something is not found.

exception icat.exception.**ICATObjectExistsError**(*error*, *status=None*)
Bases: *icat.exception.ICATError*

Is thrown when trying to create something but there is already one with the same values of the constraint fields.

exception icat.exception.**ICATSessionError**(*error*, *status=None*)
Bases: *icat.exception.ICATError*

Is used when the sessionId you have passed into a call is not valid or if you are unable to authenticate.

exception icat.exception.**ICATValidationError**(*error*, *status=None*)
Bases: *icat.exception.ICATError*

Marks an exception which was thrown instead of placing the database in an invalid state.

exception icat.exception.**ICATNotImplementedError**(*error*, *status=None*)
Bases: *icat.exception.ICATError*

exception icat.exception.**IDSError**(*error*, *status=None*)
Bases: *icat.exception.ServerError*

Base class for the errors raised by the IDS server.

exception icat.exception.**IDSBadRequestError**(*error*, *status=None*)
Bases: *icat.exception.IDSError*

Any kind of bad input parameter.

exception icat.exception.**IDSDataNotOnlineError**(*error*, *status=None*)
Bases: *icat.exception.IDSError*

The requested data are not on line.

exception icat.exception.**IDSInsufficientPrivilegesError**(*error*, *status=None*)
Bases: *icat.exception.IDSError*

You are denied access to the data.

exception icat.exception.**IDSInsufficientStorageError**(*error*, *status=None*)
Bases: *icat.exception.IDSError*

There is not sufficient physical storage or you have exceeded some quota.

exception icat.exception.**IDSInternalError**(*error*, *status=None*)
Bases: *icat.exception.IDSError*

Some kind of failure in the server or in communicating with the server.

exception icat.exception.**IDSNotFoundError**(*error*, *status=None*)
Bases: *icat.exception.IDSError*

The requested data do not exist.

exception icat.exception.**IDSNotImplementedError**(*error*, *status=None*)
Bases: *icat.exception.IDSError*

Use of some functionality that is not supported by the implementation.

icat.exception.**translateError**(*error*, *status=None*, *server='ICAT'*)
Translate an error from ICAT or IDS to the corresponding exception.

Exceptions raised by python-icat

exception icat.exception.**InternalError**(**args*)
Bases: *icat.exception._BaseException*

An error that reveals a bug in python-icat.

exception icat.exception.**ConfigError**(**args*)
Bases: *icat.exception._BaseException*

Error getting configuration options.

exception icat.exception.**QueryNullableOrderWarning**(*attr*)
Bases: Warning

Warn about using a nullable relation for ordering.

exception icat.exception.**ClientVersionWarning**(*version=None*, *comment=None*)
Bases: Warning

Warn that the version of the ICAT server is not supported by the client.

exception icat.exception.**ICATDeprecationWarning**(*feature*, *version=None*)
Bases: DeprecationWarning

Warn about using an API feature that may get removed in future ICAT server versions.

exception icat.exception.**EntityTypeError**(**args*)
Bases: *icat.exception._BaseException*, TypeError

An invalid entity type has been used.

Changed in version 0.18.0: Inherit from TypeError.

exception icat.exception.**VersionMethodError**(*method*, *version=None*, *service='ICAT'*)
Bases: *icat.exception._BaseException*

Call of an API method that is not supported in the version of the server.

exception icat.exception.**SearchResultError**(**args*)

Bases: *icat.exception._BaseException*

A search result does not conform to what should have been expected.

exception icat.exception.**SearchAssertionError**(*query*, *assertmin*, *assertmax*, *num*)

Bases: *icat.exception.SearchResultError*

A search result does not conform to an assertion.

This exception is thrown when the number of objects found on a search does not lie within the bounds of an assertion, see *icat.client.Client.assertedSearch()*.

exception icat.exception.**DataConsistencyError**(**args*)

Bases: *icat.exception._BaseException*

Some data is not consistent with rules or constraints.

exception icat.exception.**IDSResponseError**(**args*)

Bases: *icat.exception._BaseException*

The response from the IDS was not what should have been expected.

exception icat.exception.**GenealogyError**(**args*)

Bases: *icat.exception._BaseException*

Error in the genealogy of entity types.

Deprecated since version 0.17: Only used in *icat.icatcheck* which in turn is deprecated.

Exception hierarchy

The class hierarchy for the exceptions is:

```
Exception
 +-- ServerError
 |    +-- ICATError
 |    |    +-- ICATParameterError
 |    |    +-- ICATInternalError
 |    |    +-- ICATPrivilegesError
 |    |    +-- ICATNoObjectError
 |    |    +-- ICATObjectExistsError
 |    |    +-- ICATSessionError
 |    |    +-- ICATValidationError
 |    |    +-- ICATNotImplementedError
 |    +-- IDSError
 |         +-- IDSBadRequestError
 |         +-- IDSDataNotOnlineError
 |         +-- IDSInsufficientPrivilegesError
 |         +-- IDSInsufficientStorageError
 |         +-- IDSInternalError
 |         +-- IDSNotFoundError
 |         +-- IDSNotImplementedError
 +-- InternalError
 +-- ConfigError
 +-- TypeError
 |    +-- EntityTypeError
 +-- VersionMethodError
 +-- SearchResultError
 |    +-- SearchAssertionError
 +-- DataConsistencyError
 +-- IDSResponseError
 +-- GenealogyError
 +-- Warning
      +-- QueryNullableOrderWarning
```

(continues on next page)

(continued from previous page)

```
      +-- ClientVersionWarning
      +-- DeprecationWarning
           +-- ICATDeprecationWarning
```

Here, `Exception`, `TypeError`, `Warning`, and `DeprecationWarning` are build-in exceptions from the Python standard library.

`icat.ids` — Provide the IDSClient class

This module defines the *`IDSClient`* class that connects to an ICAT Data Service (IDS) as a client.

class `icat.ids.`**`DataSelection`**(*objs=None*)

Bases: `object`

A set of data to be processed by the ICAT Data Service.

This can be passed as the *selection* argument to *`icat.ids.IDSClient`* method calls. The *objs* argument is passed to the *`extend()`* method.

extend(*objs*)

Add *objs* to the DataSelection.

Parameters **objs** (`dict`, `list` of *`icat.entity.Entity`*, or *`DataSelection`*) – either a dict having some of the keys *investigationIds*, *datasetIds*, and *datafileIds* with a list of object ids as value respectively, or a list of entity objects, or another data selection.

class `icat.ids.`**`IDSClient`**(*url*, *sessionId=None*, *sslContext=None*, *proxy=None*)

Bases: `object`

A client accessing an ICAT Data Service.

The attribute sessionId must be set to a valid ICAT session id from the ICAT client.

ping()

Check that the server is alive and is an IDS server.

getApiVersion()

Get the version of the IDS server.

Note: the *getApiVersion* call has been added in IDS server version 1.3.0. For older servers, try to guess the server version from features visible in the API. Obviously this cannot always be accurate as we cannot distinguish server version with no visible API changes. In particular, versions older then 1.2.0 will always reported as 1.0.0. Nevertheless, the result of the guess should be fair enough for most use cases.

version()

Get the version of the IDS server.

Note: the *version* call has been added in IDS server version 1.8.0, deprecating *getApiVersion* at the same time. For older servers, we fall back to *getApiVersion* to emulate this call. Note furthermore that *version* returns a dict, while *getApiVersion* returns the plain version number as a string.

getIcatUrl()

Get the URL of the ICAT server connected to this IDS.

isReadOnly()

See if the server is configured to be readonly.

isTwoLevel()

See if the server is configured to use both main and archive storage.

getServiceStatus()

Return information about what the IDS is doing.

If all lists are empty it is quiet. To use this call, the user represented by the sessionId must be in the set of rootUserNames defined in the IDS configuration.

getSize (*selection*)
Return the total size of the datafiles.

getStatus (*selection*)
Return the status of data.

archive (*selection*)
Archive data.

restore (*selection*)
Restore data.

write (*selection*)
Write data.

reset (*selection*)
Reset data so that they can be queried again.

resetPrepared (*preparedId*)
Reset prepared data so that they can be queried again.

Deprecated since version 0.17.0: Call `reset()` instead.

prepareData (*selection*, *compressFlag=False*, *zipFlag=False*)
Prepare data for a subsequent `getData()` call.

isPrepared (*preparedId*)
Check if data is ready.

Returns true if the data identified by the *preparedId* returned by a call to `prepareData()` is ready.

getDatafileIds (*selection*)
Get the list of data file id corresponding to the selection.

getPreparedDatafileIds (*preparedId*)
Get the list of data file id corresponding to the prepared Id.

Deprecated since version 0.17.0: Call `getDatafileIds()` instead.

getData (*selection*, *compressFlag=False*, *zipFlag=False*, *outname=None*, *offset=0*)
Stream the requested data.

getDataUrl (*selection*, *compressFlag=False*, *zipFlag=False*, *outname=None*)
Get the URL to retrieve the requested data.

getPreparedData (*preparedId*, *outname=None*, *offset=0*)
Get prepared data.

Get the data using the *preparedId* returned by a call to `prepareData()`.

Deprecated since version 0.17.0: Call `getData()` instead.

getPreparedDataUrl (*preparedId*, *outname=None*)
Get the URL to retrieve prepared data.

Get the URL to retrieve data using the *preparedId* returned by a call to `prepareData()`.

Deprecated since version 0.17.0: Call `getDataUrl()` instead.

getLink (*datafileId*, *username=None*)
Return a hard link to a data file.

This is only useful in those cases where the user has direct access to the file system where the IDS is storing data. The caller is only granted read access to the file.

put (*inputStream*, *name*, *datasetId*, *datafileFormatId*, *description=None*, *doi=None*, *datafileCreateTime=None*, *datafileModTime=None*)
Put data into IDS.

Put the data in the *inputStream* into a data file and catalogue it. The client generates a checksum which is compared to that produced by the server to detect any transmission errors.

delete(*selection*)
Delete data.

icat.query — Provide the Query class

class icat.query.**Query**(*client*, *entity*, *attributes=None*, *aggregate=None*, *order=None*, *conditions=None*, *includes=None*, *limit=None*, *attribute=None*)

Bases: object

Build a query to search an ICAT server.

The query uses the JPQL inspired syntax introduced with ICAT 4.3.0. It won't work with older ICAT servers.

Parameters

- **client** (*icat.client.Client*) – the ICAT client.
- **entity** – the type of objects to search for. This may either be an *icat.entity.Entity* subclass or the name of an entity type.
- **attributes** – the attributes that the query shall return. See the *setAttributes()* method for details.
- **aggregate** – the aggregate function to be applied in the SELECT clause, if any. See the *setAggregate()* method for details.
- **order** – the sorting attributes to build the ORDER BY clause from. See the *setOrder()* method for details.
- **conditions** – the conditions to build the WHERE clause from. See the *addConditions()* method for details.
- **includes** – list of related objects to add to the INCLUDE clause. See the *addIncludes()* method for details.
- **limit** – a tuple (skip, count) to be used in the LIMIT clause. See the *setLimit()* method for details.
- **attribute** – alias for *attributes*, retained for compatibility. Deprecated, use *attributes* instead.

Raises

- **TypeError** – if *entity* is not a valid entity type or if both *attributes* and *attribute* are provided.
- **ValueError** – if any of the keyword arguments is not valid, see the corresponding method for details.

Changed in version 0.18.0: add support for queries requesting a list of attributes rather then a single one. Consequently, the keyword argument *attribute* has been renamed to *attributes* (in the plural).

setAttributes(*attributes*)
Set the attributes that the query shall return.

Parameters **attributes** (str or iterable of str) – the names of the attributes. This can either be a single name or a list of names. The result of the search will be a list with either a single attribute value or a list of attribute values respectively for each matching entity object. If attributes is None, the result will be the list of matching objects instead.

Raises **ValueError** – if any name in *attributes* is not valid or if multiple attributes are provided, but the ICAT server does not support this.

Changed in version 0.18.0: also accept a list of attribute names. Renamed from `setAttribute()` to `setAttributes()` (in the plural).

setAggregate(*function*)

Set the aggregate function to be applied to the result.

Note that the Query class does not verify whether the aggregate function makes any sense for the selected result. E.g. the SUM of entity objects or the AVG of strings will certainly not work in an ICAT search expression, but it is not within the scope of the Query class to reject such nonsense beforehand. Furthermore, "DISTINCT" requires icat.server 4.7.0 or newer to work. Again, this is not checked by the Query class.

Parameters **function** (`str`) – the aggregate function to be applied in the SELECT clause, if any. Valid values are "DISTINCT", "COUNT", "MIN", "MAX", "AVG", "SUM", or `None`. ":DISTINCT", may be appended to "COUNT", "AVG", and "SUM" to combine the respective function with "DISTINCT".

Raises **ValueError** – if *function* is not valid.

setOrder(*order*)

Set the order to build the ORDER BY clause from.

Parameters **order** (iterable or `bool`) – the list of the attributes used for sorting. A special value of `True` may be used to indicate the natural order of the entity type. Any false value means no ORDER BY clause. Rather then only an attribute name, any item in the list may also be a tuple of an attribute name and an order direction, the latter being either "ASC" or "DESC" for ascending or descending order respectively.

Raises **ValueError** – if *order* contains invalid attributes that either do not exist or contain one to many relationships.

addConditions(*conditions*)

Add conditions to the constraints to build the WHERE clause from.

Parameters **conditions** (`dict`) – the conditions to restrict the search result. This must be a mapping of attribute names to conditions on that attribute. The latter may either be a string with a single condition or a list of strings to add more then one condition on a single attribute. If the query already has a condition on a given attribute, it will be turned into a list with the new condition(s) appended.

Raises **ValueError** – if any key in *conditions* is not valid.

addIncludes(*includes*)

Add related objects to build the INCLUDE clause from.

Parameters **includes** (iterable of `str`) – list of related objects to add to the INCLUDE clause. A special value of "1" may be used to set (the equivalent of) an "INCLUDE 1" clause.

Raises **ValueError** – if any item in *includes* is not a related object.

setLimit(*limit*)

Set the limits to build the LIMIT clause from.

Parameters **limit** (`tuple`) – a tuple (skip, count).

Raises **TypeError** – if *limit* is not a tuple of two elements.

copy()

Return an independent clone of this query.

setAttribute(*attribute*)

Alias for `setAttributes()`.

Deprecated since version 0.18.0: use `setAttributes()` instead.

1.2.2 Special purpose modules

These modules will generally be used in particular cases only.

icat.eval — Evaluate expression in the context of an ICAT session

Evaluate a Python expression in the context of an ICAT session.

This module is intended to be run using the "-m" command line switch to Python. It adds an "-e" command line switch and evaluates the Python expression given as argument to it after having started an ICAT session. This allows one to run simple programs as one liners directly from the command line, as in:

```
# get all Dataset ids
$ python -m icat.eval -e 'client.search("Dataset.id")' -s root
[102284, 102288, 102289, 102293]
```

icat.dumpfile — Backend for icatdump and icatingest

This module provides the base classes *icat.dumpfile.DumpFileReader* and *icat.dumpfile.DumpFileWriter* that define the API and the logic for reading and writing ICAT data files. The actual work is done in file format specific modules that should provide subclasses that must implement the abstract methods.

class icat.dumpfile.**DumpFileReader**(*client*, *infile*)

Bases: object

Base class for backends that read a data file.

mode = 'r'

File mode suitable for the backend.

Subclasses should override this with either "rt" or "rb", according to the mode required for the backend.

getdata()

Iterate over the chunks in the data file.

Yield some data object in each iteration. This data object is specific to the implementing backend and should be passed as the *data* argument to *getobjs_from_data()*.

getobjs_from_data(*data*, *objindex*)

Iterate over the objects in a data chunk.

Yield a new entity object in each iteration. The object is initialized from the data, but not yet created at the client.

getobjs(*objindex=None*)

Iterate over the objects in the data file.

Yield a new entity object in each iteration. The object is initialized from the data, but not yet created at the client.

Parameters **objindex** (dict) – a mapping from keys to entity objects, see *icat.client.Client.searchUniqueKey()* for details. This serves as a cache of previously retrieved objects, used to resolve object relations. If this is None, an internal cache will be used that is purged at the start of every new data chunk.

class icat.dumpfile.**DumpFileWriter**(*client*, *outfile*)

Bases: object

Base class for backends that write a data file.

mode = 'w'

File mode suitable for the backend.

Subclasses should override this with either "wt" or "wb", according to the mode required for the backend.

head()
Write a header with some meta information to the data file.

startdata()
Start a new data chunk.

If the current chunk contains any data, write it to the data file.

writeobj(*key*, *obj*, *keyindex*)
Add an entity object to the current data chunk.

finalize()
Finalize the data file.

writeobjs(*objs*, *keyindex*, *chunksize=100*)
Write some entity objects to the current data chunk.

The objects are searched from the ICAT server. The key index is used to serialize object relations in the data file. For object types that do not have an appropriate uniqueness constraint in the ICAT schema, a generic key is generated. These objects may only be referenced from the same chunk in the data file.

Parameters

- **objs** (*icat.query.Query* or `str` or `list`) – query to search the objects, either a Query object or a string. It must contain an appropriate include clause to include all related objects from many-to-one relations. These related objects must also include all informations needed to generate their unique key, unless they are registered in the key index already.

  Furthermore, related objects from one-to-many relations may be included. These objects will then be embedded with the relating object in the data file. The same requirements for including their respective related objects apply.

  As an alternative to a query, objs may also be a list of entity objects. The same conditions on the inclusion of related objects apply.

- **keyindex** (`dict`) – cache of generated keys. It maps object ids to unique keys. See the *icat.entity.Entity.getUniqueKey()* for details.

- **chunksize** (`int`) – tuning parameter, see *icat.client.Client.searchChunked()* for details.

writedata(*objs*, *keyindex=None*, *chunksize=100*)
Write a data chunk.

Parameters

- **objs** – an iterable that yields either queries to search for the objects or object lists. See *icat.dumpfile.DumpFileWriter.writeobjs()* for details.

- **keyindex** (`dict`) – cache of generated keys, see *icat.dumpfile.DumpFileWriter.writeobjs()* for details. If this is `None`, an internal index will be used.

- **chunksize** (`int`) – tuning parameter, see *icat.client.Client.searchChunked()* for details.

icat.dumpfile.**Backends = {}**
A register of all known backends.

icat.dumpfile.**register_backend**(*formatname*, *reader*, *writer*)
Register a backend.

This function should be called by file format specific backends at initialization.

Parameters

- **formatname** (str) – name of the file format that the backend implements.
- **reader** – class for reading data files. Should be a subclass of *icat.dumpfile.DumpFileReader*.
- **writer** – class for writing data files. Should be a subclass of *icat.dumpfile.DumpFileWriter*.

icat.dumpfile.**open_dumpfile**(*client*, *f*, *formatname*, *mode*)

Open a data file, either for reading or for writing.

Note that depending on the backend, the file must either be opened in binary or in text mode. If f is a file object, it must have been opened in the appropriate mode according to the backend selected by formatname. The backend classes define a corresponding class attribute *mode*. If f is a file name, the file will be opened in the appropriate mode.

The subclasses of *icat.dumpfile.DumpFileReader* and *icat.dumpfile.DumpFileWriter* may be used as context managers. This function is suitable to be used in the `with` statement.

```
>>> with open_dumpfile(client, f, "XML", 'r') as dumpfile:
...     for obj in dumpfile.getobjs():
...         obj.create()
```

Parameters

- **client** (*icat.client.Client*) – the ICAT client.
- **f** – the object to read the data from or write the data to, according to mode. What object types are supported depends on the backend. All backends support at least a file object or the name of file. The special value of "-" may be used as an alias for `sys.stdin` or `sys.stdout`.
- **formatname** (str) – name of the file format that has been registered by the backend.
- **mode** (str) – either "r" or "w" to indicate that the file should be opened for reading or writing respectively.

Returns an instance of the appropriate class. This is either the reader or the writer class, according to the mode, that has been registered by the backend.

Raises **ValueError** – if the format is not known or if the mode is not "r" or "w".

ICAT data files

Data files are partitioned in chunks. This is done to avoid having the whole file, e.g. the complete inventory of the ICAT, at once in memory. The problem is that objects contain references to other objects (e.g. Datafiles refer to Datasets, the latter refer to Investigations, and so forth). We keep an index of the objects in order to resolve these references. But there is a memory versus time tradeoff: we cannot keep all the objects in the index, that would again mean the complete inventory of the ICAT. And we can't know beforehand which object is going to be referenced later on, so we don't know which one to keep and which one to discard from the index. Fortunately we can query objects we discarded once back from the ICAT server with *icat.client.Client.searchUniqueKey()*. But this is expensive. So the strategy is as follows: keep all objects from the current chunk in the index and discard the complete index each time a chunk has been processed. This will work fine if objects are mostly referencing other objects from the same chunk and only a few references go across chunk boundaries.

Therefore, we want these chunks to be small enough to fit into memory, but at the same time large enough to keep as many relations between objects as possible local in a chunk. It is in the responsibility of the writer of the data file to create the chunks in this manner.

The objects that get written to the data file and how this file is organized is controlled by lists of ICAT search expressions, see *icat.dumpfile.DumpFileWriter.writeobjs()*. There is some degree of flexibility:

an object may include related objects in an one-to-many relation, just by including them in the search expression. In this case, these related objects should not have a search expression on their own again. For instance, the search expression for Grouping may include UserGroup. The UserGroups will then be embedded in their respective grouping in the data file. There should not be a search expression for UserGroup then.

Objects related in a many-to-one relation must always be included in the search expression. This is also true if the object is indirectly related to one of the included objects. In this case, only a reference to the related object will be included in the data file. The related object must have its own list entry.

1.2.3 Internal modules

These modules are used internally by python-icat, but most users will not need to care about them.

`icat.authinfo` — Provide the AuthenticatorInfo class

Note: This module is used internally in *`icat.config`*. Most users will not need to use it directly or even care about it.

class `icat.authinfo.`**`AuthenticatorInfo`**(*authInfo*)

Bases: `collections.abc.Sequence`

A wrapper around the authenticator info as returned by the ICAT server.

Parameters **`authInfo`** (`list`) – authenticator information from the ICAT server as returned by *`icat.client.Client.getAuthenticatorInfo()`*.

`getAuthNames`()

Return a list of authenticator names available at the ICAT server.

`getCredentialKeys`(*auth=None*, *hide=None*)

Return credential keys.

Parameters

- **`auth`** (`str`) – authenticator name. If given, return only the credential keys for this authenticator. If `None`, return credential keys for all authenticators.
- **`hide`** (`bool`) – if given, return either only the hidden or the non-hidden credential keys, according to the provided value. If `None`, return credential keys for all authenticators.

Returns names of credential keys.

Return type `set` of `str`

Raises **`KeyError`** – if *auth* is provided, but no authenticator by that name is defined in the authenticator information.

Changed in version 0.17.0: add default value for parameter *auth*.

class `icat.authinfo.`**`LegacyAuthenticatorInfo`**

Bases: `object`

AuthenticatorInfo for old ICAT server.

This is a dummy implementation to emulate AuthenticatorInfo for the case that the server does not support the *`icat.client.Client.getAuthenticatorInfo()`* call.

`getAuthNames`()

Return `None`.

getCredentialKeys(*auth=None*, *hide=None*)

Return credential keys.

Dummy implementation, pretent that all authenticators expect *username* and *password* as credential keys, where *password* is marked as hidden.

Parameters

- **auth** (str) – authenticator name. This parameter is ignored.
- **hide** (bool) – if given, return either only the hidden or the non-hidden credential keys, according to the provided value. If None, return credential keys for all authenticators.

Returns names of credential keys.

Return type set of str

Changed in version 0.17.0: add default value for parameter *auth*.

icat.dumpfile_xml — XML data file backend

This module provides the XML backend for *icat.dumpfile*. See the documentation of that module on how to read and write ICAT data files.

class icat.dumpfile_xml.**XMLDumpFileReader**(*client*, *infile*)

Bases: *icat.dumpfile.DumpFileReader*

Backend for reading ICAT data from a XML file.

This backend accepts a file object, a filename, or a XML tree object (lxml.etree._ElementTree) as input. Note that the latter case requires by definition the complete input to be at once in memory. This is only useful if the input is small enough.

mode = 'rb'

File mode suitable for this backend.

getdata_file()

Iterate over the chunks in the data file.

getdata_etree()

Iterate over the chunks from a XML tree object.

getobjs_from_data(*data*, *objindex*)

Iterate over the objects in a data chunk.

Yield a new entity object in each iteration. The object is initialized from the data, but not yet created at the client.

class icat.dumpfile_xml.**XMLDumpFileWriter**(*client*, *outfile*)

Bases: *icat.dumpfile.DumpFileWriter*

Backend for writing ICAT data to a XML file.

mode = 'wb'

File mode suitable for this backend.

head()

Write a header with some meta information to the data file.

startdata()

Start a new data chunk.

If the current chunk contains any data, write it to the data file.

writeobj(*key*, *obj*, *keyindex*)

Add an entity object to the current data chunk.

finalize()
Finalize the data file.

icat.dumpfile_yaml — YAML data file backend

This module provides the YAML backend for *icat.dumpfile*. See the documentation of that module on how to read and write ICAT data files.

class icat.dumpfile_yaml.**YAMLDumpFileReader**(*client*, *infile*)
Bases: *icat.dumpfile.DumpFileReader*

Backend for reading ICAT data from a YAML file.

mode = 'rt'
File mode suitable for this backend.

getdata()
Iterate over the chunks in the data file.

getobjs_from_data(*data*, *objindex*)
Iterate over the objects in a data chunk.

Yield a new entity object in each iteration. The object is initialized from the data, but not yet created at the client.

class icat.dumpfile_yaml.**YAMLDumpFileWriter**(*client*, *outfile*)
Bases: *icat.dumpfile.DumpFileWriter*

Backend for writing ICAT data to a YAML file.

mode = 'wt'
File mode suitable for this backend.

head()
Write a header with some meta information to the data file.

startdata()
Start a new data chunk.

If the current chunk contains any data, write it to the data file.

writeobj(*key*, *obj*, *keyindex*)
Add an entity object to the current data chunk.

finalize()
Finalize the data file.

icat.dump_queries — Queries needed to dump the ICAT content

Note: This module is mostly intended as a helper for the *icatdump* script. Most users will not need to use it directly or even care about it.

The icatdump data is written in chunks, see the documentation of *icat.dumpfile* for details why this is needed. The partition used here is the following:

1. One chunk with all objects that define authorization (User, Group, Rule, PublicStep).
2. All static content in one chunk, e.g. all objects not related to individual investigations and that need to be present, before we can add investigations.
3. The investigation data. All content related to individual investigations. Each investigation with all its data in one single chunk on its own.
4. One last chunk with all remaining stuff (RelatedDatafile, DataCollection, Job).

The functions defined in this module each return a list of queries needed to fetch all objects to be included in one of these chunkes.

`icat.dump_queries.`**`getAuthQueries`**(*client*)
Return the queries to fetch all objects related to authorization.

`icat.dump_queries.`**`getStaticQueries`**(*client*)
Return the queries to fetch all static objects.

`icat.dump_queries.`**`getInvestigationQueries`**(*client*, *invid*)
Return the queries to fetch all objects related to an investigation.

`icat.dump_queries.`**`getOtherQueries`**(*client*)
Return the queries to fetch all other objects, e.g. not static and not directly related to an investigation.

`icat.helper` — A collection of internal helper routines

Note: This module is intended for the internal use in python-icat. Most users will not need to use it directly or even care about it.

`icat.helper.`**`simpleqp_quote`**(*obj*)
Simple quote in quoted-printable style.

`icat.helper.`**`simpleqp_unquote`**(*qs*)
Simple unquote from quoted-printable style.

`icat.helper.`**`parse_attr_val`**(*avs*)
Parse an attribute value list string.

Parse a string representing a list of attribute and value pairs in the form:

```
attrvaluestring ::= attrvalue
                |   attrvalue '_' attrvaluestring
attrvalue       ::= attr '-' value
value           ::= simplevalue
                |   '(' attrvaluestring ')'
attr            ::= [A-Za-z]+
simplevalue     ::= [0-9A-Za-z=]+
```

Return a dict with the attributes as keys. In the case of an attrvaluestring in parenthesis, this string is set as value in the dict without any further processing.

`icat.helper.`**`parse_attr_string`**(*s*, *attrtype*)
Parse the string representation of an entity attribute.

Note: for Date we use the parser from `suds.sax.date`. If this is the original Suds version, this parser is buggy and might yield wrong results. But the same buggy parser is also applied by Suds internally for the Date values coming from the ICAT server. Since we are mainly interested to compare with values from the ICAT server, we have a fair chance that this comparision nevertheless yields valid results.

`icat.helper.`**`ms_timestamp`**(*dt*)
Convert `datetime.datetime` or string to timestamp in milliseconds since epoch.

`icat.listproxy` — Provide the ListProxy class

Note: This module is mostly intended for the internal use in python-icat. Most users will not need to use it directly or even care about it.

class icat.listproxy.**ListProxy**(*target*)

Bases: collections.abc.MutableSequence

A list that acts as a proxy to another list.

ListProxy mirrors a target list: all items are stored in the target and fetched back from the target on request. In all other aspects, ListProxy tries to mimic as close as possible the behavior of an ordinary list.

This class tries to be a minimal working implementation. Methods like `append()` and `extent()` have deliberately not been implemented here. These operations fall back on the versions inherited from `collections.MutableSequence` that are based on the elementary methods. This may be less efficient then proxying the operations directly to the target, but this way its easier to override the elementary methods.

```
>>> l = [ 0, 1, 2, 3, 4 ]
>>> lp = ListProxy(l)
>>> lp
[0, 1, 2, 3, 4]
>>> lp[2]
2
>>> lp[2:4]
[2, 3]
>>> lp == l
True
>>> lp < l
False
>>> l < lp
False
>>> lp < [0, 1, 2, 3, 4, 0]
True
>>> [0, 1, 2, 3, 3] > lp
False
>>> lp[2:4] = ["two", "three"]
>>> lp
[0, 1, 'two', 'three', 4]
>>> l
[0, 1, 'two', 'three', 4]
>>> lp *= 2
>>> l
[0, 1, 'two', 'three', 4, 0, 1, 'two', 'three', 4]
>>> del lp[5:]
>>> l
[0, 1, 'two', 'three', 4]
>>> lp += ['...', 'and', 'so', 'on']
>>> l
[0, 1, 'two', 'three', 4, '...', 'and', 'so', 'on']
>>> l[0:] = [ 1, 'b', 'iii' ]
>>> ll = [ 'x', 'y' ]
>>> lp + ll
[1, 'b', 'iii', 'x', 'y']
>>> ll + lp
['x', 'y', 1, 'b', 'iii']
>>> lp * 3
[1, 'b', 'iii', 1, 'b', 'iii', 1, 'b', 'iii']
```

insert(*index*, *value*)

S.insert(index, value) – insert value before index

icat.sslcontext — Helper functions and classes related to SSL contexts

Note: This module is mostly intended for the internal use in python-icat. Most users will not need to use it directly or even care about it.

`icat.sslcontext.`**`create_ssl_context`**(*verify=True*, *cafile=None*, *capath=None*)
Set up the SSL context.

class `icat.sslcontext.`**`HTTPSTransport`**(*context*, ***kwargs*)
Bases: `suds.transport.http.HttpTransport`

A modified HttpTransport using an explicit SSL context.

`u2handlers`()
Get a collection of urllib handlers.

1.2.4 Obsolete modules

These modules are deprecated and will be removed from future version of python-icat.

`icat.cgi` — Common Gateway Interface support

This module provides tools for writing CGI scripts acting as ICAT clients.

Deprecated since version 0.13: This module is deprecated and will be removed in version 1.0.

class `icat.cgi.`**`SessionCookie`**
Bases: `http.cookies.SimpleCookie`

A cookie to store an ICAT session id.

Extend the parent class by the attribute *sessionId*. Setting this attribute will set the session id in the cookie, getting it will retrieve its value from the cookie.

class `icat.cgi.`**`Session`**(*url*, *cookieName='ICATSESSIONID'*, *cookiePath='/'*, *secure=True*)
Bases: `object`

A persisting ICAT session.

Manage an ICAT session that persist over the life time of the script. The session id is stored in a `icat.cgi.SessionCookie`.

`isActive`()
Check whether there is an active session.

`login`(*auth*, *username*, *password*)
Log in with username and password and start a session.

`logout`()
Log out and terminate the session.

`icat.icatcheck` — Check compatibility with the ICAT server

Note: This module provides tests to check the compatibility of the client with the WSDL description got from the ICAT server. It is mainly useful for the package maintainer.

Deprecated since version 0.17: This module is deprecated and will be removed in version 1.0.

class `icat.icatcheck.`**`ICATChecker`**(*client*)
Bases: `object`

Provide checks for the ICAT schema from a given server.

Check that the entities defined in the ICAT client are in sync with the WSDL schema got from the ICAT server.

gettypes()
Return a list of the types defined in the WSDL.

getentities()
Search for entities defined at the server.

Return a dict with type names as keys and EntityInfo objects as values.

check()
Check consistency of the ICAT client with the server schema.

Report any abnormalities as warnings to the logger. Returns the number of warnings emitted.

checkExceptions()
Check consistency of exceptions.

Check that all icatExceptionTypes defined in the WSDL have a corresponding exception class defined in icat.exception. Report missing exceptions as a warning to the logger. Return the number of warnings emitted.

pythonsrc(*genealogyrules=None*, *baseclassname='Entity'*)
Generate Python source code matching the ICAT schema.

Generate source code for a set of classes that match the entity info found at the server. The source code is returned as a string.

The Python classes are created as a hierarchy. It is assumed that there is one abstract base type which is the root of the genealogy tree. In the case of the ICAT 4.2.* schema, this assumptions holds, the base is `suds.sudsobject.entityBaseBean`.

Entity classes having children in the hierarchy are assumed to be abstract. In this case the attribut *`icat.entity.Entity.BeanName`* is set to `None`.

Parameters

- **genealogyrules** (`list` of `tuple`) – define the rules for the genealogy tree. It must be a list of tuples, each having two elements, a regular expression and the name of a parent type. Each type matching the regular expression is assumed to be derived from the parent. The first match in the list wins. The last element in the list should be a default rule of the form `(r'','base')`, where base is the name of the root.
- **baseclassname** (`str`) – the name for the base class at the root of the genealogy tree that shall be used in the Python output.

Returns Python source.

Return type `str`

Raises ***`GenealogyError`*** – if the genealogy tree defined by *genealogyrules* is not consistent.

1.3 Command line scripts

This section provides a reference for the command line scripts that are alongside with python-icat.

1.3.1 icatdump

Synopsis

icatdump [*standard options*] [-o FILE] [-f FORMAT]

Description

This script queries the content from an ICAT server and serializes into a flat file. The format of that file depends on the backend that can be selected with the `--format` option.

Options

The configuration options may be set in the command line or in a configuration file. Some options may also be set in the environment.

Specific Options

The following options are specific to icatdump:

-o FILE, **--outputfile** FILE
Set the output file name. If the value - is used, the output will be written to standard output. This is also the default.

-f FORMAT, **--format** FORMAT
Select the backend to use and thus the output file format. XML and YAML backends are available.

Standard Options

The following options needed to connect the ICAT service are common for most python-icat scripts:

-h, --help
Display a help message and exit.

-c CONFIGFILE, **--configfile** CONFIGFILE
Name of a configuration file.

-s SECTION, **--configsection** SECTION
Name of a section in the configuration file. If set, the values in this configuration section will be applied to define other options.

-w URL, **--url** URL
URL of the ICAT server. This should point to the web service descriptions. If the URL has no path component, a default path will be added.

--no-check-certificate
Do not verify the ICAT server's TLS certificate. This is only relevant if the URL set with `--url` uses HTTPS. It is mostly only useful for connecting a test server that does not have a trusted certificate.

--http-proxy HTTP_PROXY
Proxy to use for http requests.

--https-proxy HTTPS_PROXY
Proxy to use for https requests.

--no-proxy NO_PROXY
Comma separated list of exclusions for proxy use.

-a AUTH, **--auth** AUTH
Name of the authentication plugin to use for login to the ICAT server.

-u USERNAME, **--user** USERNAME
The ICAT user name.

-p PASSWORD, **--pass** PASSWORD
The user's password. Will prompt for the password if not set.

-P, --prompt-pass
Prompt for the password. This is mostly useful to override a password set in the configuration file.

Known Issues and Limitations

- IDS is not supported: the script only dumps the meta data stored in the ICAT, not the content of the files stored in the IDS.
- The script will only writes objects that the user connecting ICAT has read permissions for. The script may need to connect as the ICAT root user in order to get the full content.
- The following items are deliberately not included in the output:
  - Log objects (ICAT server versions older then 4.7.0),
  - The attributes `id`, `createId`, `createTime`, `modId`, and `modTime` of any object.
- It is assumed that for each Dataset *ds* in the ICAT where *ds.sample* is not NULL, the condition *ds.investigation* = *ds.sample.investigation* holds. If this is not satisfied, this script will fail with a `DataConsistencyError`.
- The partition of the data into chunks is static. It should rather be dynamic, e.g. chunks should be splitted if the number of objects in them grows too large.
- The content in the ICAT server must not be modified while this script is retrieving it. Otherwise the script may fail or the dumpfile be inconsistent.
- The script fails if the data contains any *Study* if the ICAT server version is older then 4.6.0. This is a bug in icat.server.

Environment Variables

ICAT_CFG
Name of a configuration file, see `--configfile`.

ICAT_CFG_SECTION
Name of a section in the configuration file, see `--configsection`.

ICAT_SERVICE
URL of the ICAT server, see `--url`.

http_proxy
Proxy to use for http requests, see `--http-proxy`.

https_proxy
Proxy to use for https requests, see `--https-proxy`.

no_proxy
Exclusions for proxy use, see `--no-proxy`.

ICAT_AUTH
Name of the authentication plugin, see `--auth`.

ICAT_USER
ICAT user name, see `--user`.

See also

- Section *ICAT data files* on the structure of the dump files.
- Section *Predefined configuration variables* on the standard options.
- The *icatingest* script.

1.3.2 icatingest

Synopsis

icatingest [*standard options*] [-i FILE] [-f FORMAT] [–upload-datafiles] [–datafile-dir DATADIR] [–duplicate OPTION]

Description

This script reads an ICAT data file and creates all objects found in an ICAT server. The format of that file depends on the backend that can be selected with the `--format` option.

Options

The configuration options may be set in the command line or in a configuration file. Some options may also be set in the environment.

Specific Options

The following options are specific to icatingest:

-i FILE, **--inputfile** FILE
: Set the input file name. If the value - is used, the input will be read from standard input. This is also the default.

-f FORMAT, **--format** FORMAT
: Select the backend to use and thus the input file format. XML and YAML backends are available.

--upload-datafiles
: If that flag is set, Datafile objects will not be created in the ICAT server directly, but a corresponding file will be uploaded to IDS instead.

--datafile-dir DATADIR
: Directory to search for the files to be uploaded to IDS. This is only relevant if `--upload-datafiles` is set. The default is the current working directory.

--duplicate OPTION
: Set the behavior in the case that any object read from the input already exists in the ICAT server. Valid options are:

  THROW Throw an error. This is the default.

  IGNORE Skip the object read from the input.

  CHECK Compare all attributes from the input object with the already existing object in ICAT. Throw an error of any attribute differs.

  OVERWRITE Overwrite the existing object in ICAT, e.g. update it with all attributes set to the values found in the input object.

  If `--upload-datafiles` is set, this option will be ignored for Datafile objects which will then always raise an error if they already exist.

Standard Options

The following options needed to connect the ICAT service are common for most python-icat scripts:

-h, --help
: Display a help message and exit.

-c CONFIGFILE, **--configfile** CONFIGFILE
Name of a configuration file.

-s SECTION, **--configsection** SECTION
Name of a section in the configuration file. If set, the values in this configuration section will be applied to define other options.

-w URL, **--url** URL
URL of the ICAT server. This should point to the web service descriptions. If the URL has no path component, a default path will be added.

--idsurl URL
URL of the IDS server. This is only relevant if *--upload-datafiles* is set. If the URL has no path component, a default path will be added.

--no-check-certificate
Do not verify the ICAT server's TLS certificate. This is only relevant if the URL set with *--url* or *--idsurl* uses HTTPS. It is mostly only useful for connecting a test server that does not have a trusted certificate.

--http-proxy HTTP_PROXY
Proxy to use for http requests.

--https-proxy HTTPS_PROXY
Proxy to use for https requests.

--no-proxy NO_PROXY
Comma separated list of exclusions for proxy use.

-a AUTH, **--auth** AUTH
Name of the authentication plugin to use for login to the ICAT server.

-u USERNAME, **--user** USERNAME
The ICAT user name.

-p PASSWORD, **--pass** PASSWORD
The user's password. Will prompt for the password if not set.

-P, --prompt-pass
Prompt for the password. This is mostly useful to override a password set in the configuration file.

Known Issues and Limitations

- The user running this script need to have create permission for all objects in the dump file. In the generic case of restoring the entire content on an empty ICAT server, the script must be run by the ICAT root user.
- A dump and restore of an ICAT will not preserve the attributes `id`, `createId`, `createTime`, `modId`, and `modTime` of any object. As a consequence, access rules that are based on the values of these attributes will not work after a restore.
- Dealing with duplicates, see *--duplicate*, is only supported for single objects. If the object contains embedded related objects in one to many relationships that are to be created at once, the only allowed option to deal with duplicates is THROW.

Environment Variables

ICAT_CFG
Name of a configuration file, see *--configfile*.

ICAT_CFG_SECTION
Name of a section in the configuration file, see *--configsection*.

ICAT_SERVICE
URL of the ICAT server, see *--url*.

ICAT_DATA_SERVICE
URL of the IDS server, see `--idsurl`.

http_proxy
Proxy to use for http requests, see `--http-proxy`.

https_proxy
Proxy to use for https requests, see `--https-proxy`.

no_proxy
Exclusions for proxy use, see `--no-proxy`.

ICAT_AUTH
Name of the authentication plugin, see `--auth`.

ICAT_USER
ICAT user name, see `--user`.

See also

- Section *ICAT data files* on the structure of the dump files.
- Section *Predefined configuration variables* on the standard options.
- The *icatdump* script.

1.3.3 wipeicat

Synopsis

wipeicat [*options*]

Description

Delete all content from an ICAT server.

In order to avoid leaving orphan content in the IDS server behind, the script first tries to delete all Datafiles having the `location` attribute set from the IDS server. If deleting the Datafiles succeeded, the remaing content is deleted from ICAT in palatable chunks.

Options

The configuration options may be set in the command line or in a configuration file. Some options may also be set in the environment.

These options are needed to connect the ICAT service and are common for most python-icat scripts.

-h, --help
Display a help message and exit.

-c CONFIGFILE, **--configfile** CONFIGFILE
Name of a configuration file.

-s SECTION, **--configsection** SECTION
Name of a section in the configuration file. If set, the values in this configuration section will be applied to define other options.

-w URL, **--url** URL
URL of the ICAT server. This should point to the web service descriptions. If the URL has no path component, a default path will be added.

--no-check-certificate
Do not verify the ICAT server's TLS certificate. This is only relevant if the URL set with *--url* uses HTTPS. It is mostly only useful for connecting a test server that does not have a trusted certificate.

--http-proxy HTTP_PROXY
Proxy to use for http requests.

--https-proxy HTTPS_PROXY
Proxy to use for https requests.

--no-proxy NO_PROXY
Comma separated list of exclusions for proxy use.

-a AUTH, **--auth** AUTH
Name of the authentication plugin to use for login to the ICAT server.

-u USERNAME, **--user** USERNAME
The ICAT user name.

-p PASSWORD, **--pass** PASSWORD
The user's password. Will prompt for the password if not set.

-P, --prompt-pass
Prompt for the password. This is mostly useful to override a password set in the configuration file.

Known Issues with old IDS Versions

The recommended version of the IDS server is 1.6.0 or newer. The script does not take any particular measure to work around issues in servers older than that. In particular, the script mail fail or leave rubbish behind in the following situations:

- The IDS server is older then 1.6.0 and there is any Dataset with many Datafiles, see IDS Issue #42.
- The IDS server is older then 1.3.0 and restoring of any Dataset takes a significant amount of time, see IDS Issue #14.

The script does however take care not trying to delete any Datafile having a NULL `location` attribute in order to work around IDS Issue #63 in IDS server older then 1.9.0.

Environment Variables

ICAT_CFG
Name of a configuration file, see *--configfile*.

ICAT_CFG_SECTION
Name of a section in the configuration file, see *--configsection*.

ICAT_SERVICE
URL of the ICAT server, see *--url*.

http_proxy
Proxy to use for http requests, see *--http-proxy*.

https_proxy
Proxy to use for https requests, see *--https-proxy*.

no_proxy
Exclusions for proxy use, see *--no-proxy*.

ICAT_AUTH
Name of the authentication plugin, see *--auth*.

ICAT_USER
ICAT user name, see *--user*.

1.4 Changelog

1.4.1 0.18.1 (2021-04-13)

Bug fixes and minor changes

- #82: Change the search result in the case of multiple fields from list to tuple.
- #76, #81: work around an issue in icat.server using *DISTINCT* in search queries for multiple fields.

1.4.2 0.18.0 (2021-03-29)

New features

- #76, #78: add client side support for searching for multiple fields introduced in icat.server 4.11.0. Add support for building the corresponding queries in the in class `icat.query.Query`.

Incompatible changes and deprecations

- Since `icat.query.Query` now also accepts a list of attribute names rather then only a single one, the corresponding keyword argument *attribute* has been renamed to *attributes* (in the plural). Accordingly, the method `icat.query.Query.setAttribute()` has been renamed to `icat.query.Query.setAttributes()`. The old names are retained as aliases, but are deprecated.

Bug fixes and minor changes

- #79: fix an encoding issue in `icat.client.Client.apiversion`, only relevant with Python 2.
- #80: add `TypeError` as additional ancestor of `icat.exception.EntityTypeError`.

1.4.3 0.17.0 (2020-04-30)

New features

- #65: Add support for the extended IDS API calls `icat.ids.IDSClient.getSize()` and `icat.ids.IDSClient.getStatus()` accepting a preparedId as introduced in ids.server 1.11.0. Also extend the methods `icat.ids.IDSClient.reset()`, `icat.ids.IDSClient.getDatafileIds()`, `icat.ids.IDSClient.getData()`, `icat.ids.IDSClient.getDataUrl()`, `icat.client.Client.getData()`, and `icat.client.Client.getDataUrl()` to accept a preparedId in the place of a data selection.
- #63: Set a default path in the URL for ICAT and IDS respectively.

Incompatible changes and deprecations

- Drop support for ICAT 4.2.*, deprecated in 0.13.0.
- #61, #64: Review `icat.entities`. The entity classes from the ICAT schema are now dynamically created based on the information gathered with the `icat.client.Client.getEntityInfo()` ICAT API call. Code that relied on the internals of `icat.entities` such as the class hierarchy or that referenced any of the entity classes directly will need to be revisited. Note that common python-icat programs don't need to do any of that. So it is assumed that most existing programs are not concerned.

- Deprecate `icat.ids.IDSClient.resetPrepared()`, `icat.ids.IDSClient.getPreparedDatafileIds()`, `icat.ids.IDSClient.getPreparedData()`, `icat.ids.IDSClient.getPreparedDataUrl()`, `icat.client.Client.getPreparedData()`, and `icat.client.Client.getPreparedDataUrl()`. Call the corresponding methods without *Prepared* in the name with the same arguments instead.
- Deprecate support for Python 2 and Python 3.3.
- Deprecate module `icat.icatcheck`. This module was not intended to be used in python-icat programs anyway.

Bug fixes and minor changes

- #68: *wipeicat* enters an infinite loop if Datafiles are missing from IDS storage.
- #19, #69: Review documentation and add tutorial.
- #62: Minor fixes in the error handling in *setup.py*.
- Fix icatdata-4.10.xsd: `Study.endDate` was erroneously not marked as optional.
- #70: Fix several errors in the tests.
- #58: Use specific test data for different ICAT versions.
- #67, #71, #72: document the option to use suds-community instead of suds-jurko.

Misc

- Do not include the documentation in the source distribution. Rely on the online documentation (see link in the README.rst) instead.

1.4.4 0.16.0 (2019-09-26)

New features

- #59: Add support for sub-commands in `icat.config`.

Incompatible changes and deprecations

- Drop support for Python 2.6.

Bug fixes and minor changes

- #60: Fix bad coding style dealing with function parameters.
- Use `setuptools_scm` to manage the version number.

1.4.5 0.15.1 (2019-07-12)

Bug fixes and minor changes

- Issue #56: *icatdump* fails to include `Shift.instrument`.
- Issue #57: `icat.client.Client.searchChunked()` still susceptible to LIMIT clause bug in icat.server (Issue icatproject/icat.server#128).
- Call `yaml.safe_load()` rather then `yaml.load()`, fixing a deprecation warning from PyYAML 5.1.

1.4.6 0.15.0 (2019-03-27)

New features

- #53: Add support for ICAT 4.10.0 including schema changes in that version.

Incompatible changes and deprecations

- Require pytest 3.1.0 or newer to run the test suite. Note that this pytest version in turn requires Python 2.6, 2.7, or 3.3 and newer.
- Drop support for Python 3.1 and 3.2. There is no known issue with these Python versions in python-icat (so far). But since we can't test this any more, see above, we drop the claim to support them.

Bug fixes and minor changes

- #49: Module icat.eval is outdated.
- #50, #52: Fix DeprecationWarnings.
- #51: Fix a compatibility issue with pytest 4.1.0 in the tests.
- #54: Fix a UnicodeDecodeError in the tests.

1.4.7 0.14.2 (2018-10-25)

Bug fixes and minor changes

- Add a hook to control internal diverting of `sys.err` in the *`icat.config`* module. This is intentionally not documented as it goes deeply into the internals of this module and most users will probably not need it.

1.4.8 0.14.1 (2018-06-05)

Bug fixes and minor changes

- Fix a misleading error message if the IDS server returns an error for the Write API call.

1.4.9 0.14.0 (2018-06-01)

New features

- #45: Add support for the IDS Write API call introduced in ids.server 1.9.0.
- #46, #47: Add a `icat.client.Client.autoRefresh()` method. The scripts *icatdump* and *icatingest* call this method periodically to prevent the session from expiring.
- #48: Add support for an ordering direction qualifier in class *`icat.query.Query`*.
- #44: Add method *`icat.entity.Entity.as_dict()`*.
- #40: Add method *`icat.client.Client.clone()`*.

Incompatible changes and deprecations

- Deprecate function `icat.exception.stripCause()`.

  This was an internal helper function not really meant to be part of the API. The functionality has been moved in a base class of the exception hierarchy.

Bug fixes and minor changes

- Add the `icat.ids.IDSClient.version()` API call introduced in ids.server 1.8.0.
- #41: Incomprehensible error messages with Python 3.
- #43: `icat.client.Client.logout()` should silently ignore `icat.exception.ICATSessionError`.
- Minor changes in the error handling. Add new exception `icat.exception.EntityTypeError`.
- Documentation fixes.

1.4.10 0.13.1 (2017-07-12)

Bug fixes and minor changes

- #38: There should be a way to access the kwargs used to create the client in config.

1.4.11 0.13.0 (2017-06-09)

New features

- #11: Support discovery of info about available ICAT authenticators.

  If supported by the ICAT server (icat.server 4.9.0 and newer), the `icat.config` module queries the server for information on available authenticators and the credential keys they require for login. The configuration variables for these keys are then adapted accordingly. Note incompatible changes below.
- Review *wipeicat*. This was an example script, but is now promoted to be a regular utility script that gets installed.
- #32: Add support for using aggregate functions in class `icat.query.Query`.
- #30: Add a predefined config variable type `icat.config.cfgpath()`.
- #31: Add a flag to add the default variables to the `icat.config.Config` constructor (default: True).
- `icat.dumpfile_xml.XMLDumpFileReader` also accepts a XML tree object as input.
- Verify support for ICAT 4.9.0. Add new ICAT API method `icat.client.Client.getVersion()`.

Incompatible changes and deprecations

- As a consequence of the discovery of available authenticators, the workflow during configuration need to be changed. Until now, the beginning of a typical python-icat program would look like:

```
config = icat.config.Config()
# Optionally, add custom configuration variables:
# config.add_variable(...)
conf = config.getconfig()
client = icat.Client(conf.url, **conf.client_kwargs)
```

E.g. first the configuration variables are set up, then the configuration is applied and finally the `icat.client.Client` object is created using the configuration values. With the discovery of authenticators, the `icat.config.Config` object itself needs a working `icat.client.Client` object in order to connect to the ICAT server and query the authenticator info. The `icat.client.Client` object will now be created in the `icat.config.Config` constructor and returned along with the configuration values by `icat.config.Config.getconfig()`. You will need to replace the code from above by:

```
config = icat.config.Config()
# Optionally, add custom configuration variables:
# config.add_variable(...)
client, conf = config.getconfig()
```

The derived configuration variable *client_kwargs* that was used to pass additional arguments from the configuration to the Client constructor is no longer needed and has been removed.

The optional argument *args* has been moved from the `icat.config.Config.getconfig()` call to the `icat.config.Config` constructor, retaining the same semantics. E.g. you must change in your code:

```
config = icat.config.Config()
conf = config.getconfig(args)
client = icat.Client(conf.url, **conf.client_kwargs)
```

to:

```
config = icat.config.Config(args)
client, conf = config.getconfig()
```

- Deprecate support for ICAT 4.2.*.

  Note that already now significant parts of python-icat require features from ICAT 4.3 such as the JPQL like query language. The only workaround is to upgrade your icat.server.

- Deprecate module `icat.cgi`.

  It is assumed that this has never actually been used in production. For web applications it is recommended to use the Python Web Server Gateway Interface (WSGI) rather then CGI.

- Deprecate the predefined configuration variable *configDir*.

  The main use case for this variable was to be substituted in the default value for the path of an additional configuration file. The typical usage was the definition of a configuration variable like:

```
config = icat.config.Config()
config.add_variable('extracfg', ("--extracfg",),
                    dict(help="Extra config file"),
                    default="%(configDir)s/extra.xml", subst=True)
```

  This set the default path for the extra config file to the same directory the main configuration file was found in. Using the new config variable type `icat.config.cfgpath()` you can replace this by:

```
config = icat.config.Config()
config.add_variable('extracfg', ("--extracfg",),
                    dict(help="Extra config file"),
                    default="extra.xml", type=icat.config.cfgpath)
```

  This will search the extra config file in all the default config directories, regardless where the main configuration file was found.

- The fixes for #35 and #36 require some changes in the semantics in the *f* and the *mode* argument to `icat.dumpfile.open_dumpfile()`. Most users will probably not notice the difference.

Bug fixes and minor changes

- Changed the default for the `icat.config.Config` constructor argument *ids* from `False` to `"optional"`.
- Improved `icat.client.Client.searchChunked()`. This version is not susceptible to Issue icat-project/icat.server#128 anymore.
- Move the management of dependencies of tests into a separate package pytest-dependency that is distributed independently.
- #34: `TypeError` in the `icat.client.Client` constructor if setting the *sslContext* keyword argument.
- #35: `io.UnsupportedOperation` is raised if `icat.dumpfile.open_dumpfile()` is called with an in-memory stream.
- #36: `icat.dumpfile.DumpFileReader` and `icat.dumpfile.DumpFileWriter` must not close file.
- #37: `TypeError` is raised when writing a YAML dumpfile to `io.StringIO`.

1.4.12 0.12.0 (2016-10-10)

New features

- Verify support for ICAT 4.8.0 and IDS 1.7.0.
- Add methods `icat.ids.IDSClient.reset()` and `icat.ids.IDSClient.resetPrepared()`.
- #28: Add support for searching for attributes in class `icat.query.Query`.

Bug fixes and minor changes

- Sort objects in *icatdump* before writing them to the dump file. This keeps the order independent from the collation used in the ICAT database backend.
- #2: for Python 3.6 (expected to be released in Dec 2016) and newer, use the support for chunked transfer encoding in the standard lib. Keep our own implementation in module `icat.chunkedhttp` only for compatibility with older Python versions.
- Improved the example script *wipeicat*.
- Add an example script *dumprules.py*.
- Add missing schema definition for the ICAT XML data file format for ICAT 4.7.
- Fix an `AttributeError` during error handling.

1.4.13 0.11.0 (2016-06-01)

New features

- #12, #23: add support for ICAT 4.7.0 and IDS 1.6.0. ICAT 4.7.0 had some small schema changes that have been taken into account.

Incompatible changes

- Remove the *autoget* argument from `icat.entity.Entity.getUniqueKey()`. Deprecated since 0.9.0.

Bug fixes and minor changes

- #21: configuration variable *promptPass* is ignored when set in the configuration file.
- #18: Documentation: missing stuff in the module index.
- #20: add test on compatibility with icat.server.
- #24, #25: test failures caused by different timezone settings of the test server.
- Use a separate module distutils_pytest to run the tests from *setup.py*.
- `icat.icatcheck`: move checking of exceptions into a separate method `icat.icatcheck.ICATChecker.checkExceptions()`. Do not report exceptions defined in the client, but not found in the schema.
- Many fixes in the example script *wipeicat*.
- Fix a missing import in the *icatexport.py* example script.
- Somewhat clearer error messages for some special cases of `icat.exception.SearchAssertionError`.

Misc

- Change license to Apache 2.0.

1.4.14 0.10.0 (2015-12-06)

New features

- Add a method `icat.entity.Entity.copy()`.
- Implement setting an INCLUDE 1 clause equivalent in class `icat.query.Query`.
- Add an optional argument *includes* to `icat.client.Client.searchMatching()`.
- Add a hook for a custom method to validate entity objects before creating them at the ICAT server.
- Add support for ids.server 1.5.0:
  - Add `icat.ids.IDSClient.getDatafileIds()` and `icat.ids.IDSClient.getPreparedDatafileIds()` calls.
  - `icat.ids.IDSClient.getStatus()` allows *sessionId* to be None.
- Add new exception class `icat.exception.ICATNotImplementedError` that is supposed to be raised by the upcoming version 4.6.0 of icat.server.

Bug fixes and minor changes

- #13: `icat.client.Client.searchChunked()` raises exception if the query contains a percent character.
- #15: *icatdump* raises `icat.exception.DataConsistencyError` for *DataCollectionParameter*.
- #14: `icat.entity.Entity.__sortkey__()` may raise `RuntimeError` "maximum recursion depth exceeded".
- Allow a `icat.ids.DataSelection` to be created from (almost) any Iterator, not just a `Sequence`. Store the object ids in `icat.ids.DataSelection` internally in a `set` rather then a `list`.
- Add optional arguments *objindex* to `icat.dumpfile.DumpFileReader.getobjs()` and *keyindex* to `icat.dumpfile.DumpFileWriter.writedata()` to allow the caller to control these internal indices.

- Add optional argument *chunksize* to `icat.dumpfile.DumpFileWriter.writedata()`.
- The constructor of class `icat.query.Query` checks the version of the ICAT server and raises an error if too old.
- The `icat.ids.IDSClient.getIcatUrl()` call checks the version of the IDS server.
- Some changes in the test suite, add more tests.

1.4.15 0.9.0 (2015-08-13)

New features

- #4: Extend *icatrestore* to become a generic ingestion tool.

  Rename *icatrestore* to *icatingest*.

  Allow referencing of objects by attribute rather then by unique key in the input file for *icatingest* (only in the XML backend).

  Allow adding references to already existing objects in the input file for *icatingest* (only in the XML backend).

  Change the name of the root element in the input file for *icatingest* (and the output of *icatdump*) from *icatdump* to *icatdata* (only in the XML backend).
- Implement upload of Datafiles to IDS rather then only creating the ICAT object from *icatingest*.
- Implement handling of duplicates in *icatingest*. The same options (*THROW*, *IGNORE*, *CHECK*, and *OVERWRITE*) as in the import call in the ICAT restful interface are supported.
- #1: add a test suite.
- #3: use Sphinx to generate the API documentation.
- Add method `icat.client.Client.searchMatching()`.
- Add the `icat.ids.IDSClient.getIcatUrl()` call introduced with IDS 1.4.0.

Incompatible changes and deprecations

- The Lucene calls that have been removed in ICAT 4.5.0 are also removed from the client.
- Deprecate the use of the *autoget* argument in `icat.entity.Entity.getUniqueKey()`.

Bug fixes and minor changes

- #6: `icat.query.Query`: adding a condition on a meta attribute fails.
- #10: client.putData: IDSInternalError is raised if datafile.datafileCreateTime is set.
- Ignore import errors from the backend modules in *icatingest* and *icatdump*. This means one can use the scripts also if the prerequisites for some backends are not fulfilled, only the concerned backends are not available then.
- #5, compatibility with ICAT 4.5: entity ids are not guaranteed to be unique among all entities, but only for entities of the same type.
- #5, compatibility with ICAT 4.5: `icat.client.Client.getEntityInfo()` also lists *createId*, *createTime*, *modId*, and *modTime* as attributes. This need to be taken into account in `icat.icatcheck`.
- The last fix in 0.8.0 on the string representation operator `icat.query.Query.__str__()` was not complete, the operator still had unwanted side effects.
- Fix a bug in the handling of errors raised from the ICAT or the IDS server. This bug affected only Python 3.

- Add proper type checking and conversion for setting an attribute that corresponds to a one to many relationship in class `icat.entity.Entity`. Accept any iterable of entities as value.
- #9: *icatingest* with *duplicate=CHECK* may fail when attributes are not strings. Note that this bug was only present in an alpha version, but not in any earlier release version.
- Source repository moved to Git. This gives rise to a few tiny changes. To name the most visible ones: python2_6.patch is now auto generated by comparing two source branches and must be applied with *-p1* instead of *-p0*, the format of the icat module variable `icat.__revision__` has changed.
- Review default exports of modules. Mark some helper functions as internal.

1.4.16 0.8.0 (2015-05-08)

New features

- Enable verification of the SSL server certificate in HTTPS connections. Add a new configuration variable *checkCert* to control this. It is set to `True` by default.

  Note that this requires either Python 2.7.9 or 3.2 or newer. With older Python version, this configuration option has no effect.
- Add type conversion of configuration variables.
- Add substituting the values of configuration variables in other variables.
- Add another derived configuration variable *configDir*.
- Default search path for the configuration file: add an appropriate path on Windows, add `/etc/icat` and `~/.config/icat` to the path if not on Windows.
- Add *icatexport.py* and *icatimport.py* example scripts that use the corresponding calls to the ICAT RESTful interface to dump and restore the ICAT content.
- The constructor of `icat.exception.ICATError` and the `icat.exception.translateError()` function are now able to construct exceptions based on a dict such as those returned by the ICAT RESTful interface in case of an error.

  Unified handling of errors raised from the ICAT and the IDS server.

Incompatible changes

- As a consequence of the unified handling of errors, the exception class hierarchy has been reviewed, with a somewhat more clear separation of exceptions raised by other libraries, exceptions raised by the server, and exceptions raised by python-icat respectively.

  If you put assumptions on the exception hierarchy in your code, this might need a review. In particular, `icat.exception.IDSResponseError` is not derived from `icat.exception.IDSError` any more. `icat.exception.IDSServerError` has been removed.

  I.e., replace all references to `icat.exception.IDSServerError` by `icat.exception.IDSError` in your code. Furthermore, if you catch `icat.exception.IDSError` in your code with the intention to catch both, errors from the IDS server and `icat.exception.IDSResponseError` in one branch, replace:

  ```
  try:
      # ...
  except IDSError:
      # ...
  ```

  by

```
try:
    # ...
except (IDSError, IDSResponseError):
    # ...
```

Bug fixes and minor changes

- The `icat.query.Query` class now checks the attributes referenced in conditions and includes for validity.
- Fix a regression introduced with version 0.7.0 that caused non-ASCII characters in queries not to work.
- Fix `icat.exception.ICATError` and `icat.exception.IDSError` to gracefully deal with non-ASCII characters in error messages. Add a common abstract base class `icat.exception.ICATException` that cares about this.
- Fix: the string representation operator `icat.query.Query.__str__()` should not modify the query object.
- Cosmetic improvement in the formal representation operator `icat.query.Query.__repr__()`.

1.4.17 0.7.0 (2015-02-11)

New features

- Add a module `icat.query` with a class `icat.query.Query` that can be used to build ICAT search expressions. Instances of the class may be used in place of search expression strings where appropriate.

  Numerous examples on how to use this new class can be found in *querytest.py* in the examples.
- Add a class method `icat.entity.Entity.getNaturalOrder()` that returns a list of attributes suitable to be used in an ORDER BY clause in an ICAT search expression.
- Add a class method `icat.entity.Entity.getAttrInfo()` that queries the EntityInfo from the ICAT server and extracts the information on an attribute.
- Add a method `icat.client.Client.getEntityClass()` that returns the `icat.entity.Entity` subclass corresponding to a name.
- Add a warning class `icat.exception.QueryNullableOrderWarning`.
- Add an optional argument *username* to the `icat.ids.IDSClient.getLink()` method.

1.4.18 0.6.0 (2014-12-15)

New features

- Add support for ICAT 4.4.0: add new `icat.entity.Entity` type *InvestigationGroup*, *role* has been added to the constraint in *InvestigationUser*.
- Add new API method `icat.ids.IDSClient.getApiVersion()` that will be introduced with the upcoming version 1.3.0 of IDS. This method may also be called with older IDS servers: if it is not available because the server does not support it yet, the server version is guessed from visible features in the API.

  `icat.ids.IDSClient` checks the API version on init.
- Add new API methods `icat.ids.IDSClient.isReadOnly()`, `icat.ids.IDSClient.isTwoLevel()`, `icat.ids.IDSClient.getLink()`, and `icat.ids.IDSClient.getSize()` introduced with IDS 1.2.0.
- Add *no_proxy* support. The proxy configuration variables, *http_proxy*, *https_proxy*, and *no_proxy* are set in the environment. [Suggested by Alistair Mills]

- Rework the dump file backend API for *icatdump* and *icatrestore*. As a result, writing custom dump or restore scripts is much cleaner and easier now.

  This may cause compatibility issues for users who either wrote their own dump file backend or for users who wrote custom dump or restore scripts, using the XML or YAML backends. In the first case, compare the old XML and YAML backends with the new versions and you'll easily see what needs to get adapted. In the latter case, have a look into the new versions of *icatdump* and *icatrestore* to see how to use the new backend API.

- Add method `icat.client.Client.searchChunked()`.
- Add method `icat.entity.Entity.getAttrType()`.

Incompatible changes

- Move the *group* argument to method `icat.client.Client.createRules()` to the last position and make it optional, having default `None`.

  In the client code, replace:

  ```
  client.createRules(group, crudFlags, what)
  ```

  by

  ```
  client.createRules(crudFlags, what, group)
  ```

- The `icat.client.Client.putData()` method returns the new Datafile object created by IDS rather then only its id.

  If you depend on the old behavior in the client code, replace:

  ```
  dfid = client.putData(file, datafile)
  ```

  by

  ```
  df = client.putData(file, datafile)
  dfid = df.id
  ```

Minor changes and fixes

- The `icat.client.Client.searchText()` and `icat.client.Client.luceneSearch()` client method have been deprecated. They are destined to be dropped from the ICAT server or at least changed in version 4.5.0 and might get removed from python-icat in a future release as well.

  The methods now emit a deprecation warning when called. Note however that Python by default ignores deprecation warnings, so you won't see this unless you switch them on.

- Fixed overly strict type checking in the constructor arguments of `icat.ids.DataSelection` and as a consequence also in the arguments of the ICAT client methods `icat.client.Client.getData()`, `icat.client.Client.getDataUrl()`, `icat.client.Client.prepareData()`, and `icat.client.Client.deleteData()`: now, any `Sequence` of entity objects will be accepted, in particular an `icat.entity.EntityList`.
- Change `icat.ids.IDSClient.archive()` and `icat.ids.IDSClient.restore()` to not to return anything. While formally, this might be considered an incompatible change, these methods never returned anything meaningful in the past.
- Slightly modified the `==` and `!=` operator for `icat.entity.Entity`. Add a `icat.entity.Entity.__hash__()` method. The latter means that you will more likely get what you expect when you create a set of `icat.entity.Entity` objects or use them as keys in a dict.

- The module *icat.eval* now only does its work (parsing command line arguments and connecting to an ICAT server) when called from the Python command line. When imported as a regular module, it will essentially do nothing. This avoids errors to occur when imported.
- *setup.py* raises an error with Python 2.6 if python2_6.patch has not been applied.
- Add missing *MANIFEST.in* in the source distribution.
- Remove the work around the Suds datetime value bug (setting the environment variable TZ to `UTC`) from `icat`. Instead, document it along with other known issues in the README.
- Minor fixes in the sorting of entity objects.
- Add an optional argument args to *icat.config.Config.getconfig()*. If set to a list of strings, it replaces `sys.argv`. Mainly useful for testing.
- Add comparison operators to class *icat.listproxy.ListProxy*.

1.4.19 0.5.1 (2014-07-07)

- Add a module *icat.eval* that is intended to be run using the `-m` command line switch to Python. It allows to evaluate Python expressions within an ICAT session as one liners directly from the command line, as for example:

```
# get all Dataset ids
$ python -m icat.eval -e 'client.search("Dataset.id")' -s root
[102284L, 102288L, 102289L, 102293L]
```

- Fix an issue in the error handling in the IDS client that caused an `urllib2.HTTPError` to be raised instead of an `icat.exception.IDSServerError` in the case of an error from the IDS server and thus the loss of all details about the error reported in the reply from the server.
- Add specific exception classes for the different error codes raised by the IDS server.
- Fix compatibility issue with Python 3.3 that caused the HTTP method to be set to `None` in some IDS methods, which in turn caused an internal server error to be raised in the IDS server.
- Fix compatibility issues with Python 3.4: some methods have been removed from class `urllib.request.Request` which caused an `AttributeError` in the *icat.ids.IDSClient*.
- Fix: failed to connect to an ICAT server if it advertises a version number having a trailing "-SNAPSHOT" in *icat.client.Client.getApiVersion()*. For compatibility, a trailing "-SNAPSHOT" will be replaced by "a1" in the client.apiversion attribute.
- Suppress misleading context information introduced with Python 3 (PEP 3134) from the traceback in some error messages. Unfortunately, the fix only works for Python 3.3 and newer.
- Make example files compatible across Python versions without modifications, such as running 2to3 on them.

1.4.20 0.5.0 (2014-06-24)

- Integrate an IDS client in the ICAT client.
- Improved *icatdump* and *icatrestore*:
  - Changed the logical structure of the dump file format which significantly simplified the scripts. Note that old dump files are not compatible with the new versions.
  - Add support for XML dump files. A XML Schema Definition for the dump file format is provided in the doc directory.

  The scripts are now considered to be legitimate tools (though still alpha) rather then mere examples. Consequently, they will be installed into the bin directory.

- Implicitly set a one to many relation to an empty list if it is accessed but not present in an `icat.entity.Entity` object rather then raising an `AttributeError`. See ICAT Issue 112.
- Allow setting one to many relationship attributes and deletion of attributes in `icat.entity.Entity`. Add method `icat.entity.Entity.truncateRelations()`. Truncate dummy relations set by the factory in newly created entity objects.
- Cache the result from `icat.client.Client.getEntityInfo()` in the client.
- Add a method `icat.entity.Entity.__sortkey__()` that return a key that when used as a sorting key in `list.sort()` allows any list of entity objects to have a well defined order. Sorting is based on the Constraint attributes. Add a class variable `icat.entity.Entity.SortAttrs` that overrides this and will be set as a fall back for those entity classes that do not have a suitable Constraint.

1.4.21 0.4.0 (2014-02-11)

- Add support for the jurko fork of Suds and for Python 3.
- Add a new method `icat.client.Client.searchUniqueKey()`.
- Add an optional argument *keyindex* to method `icat.entity.Entity.getUniqueKey()` that is used as a cache of previously generated keys. Remove the argument *addbean*. It had been documented as for internal use only, so this is not considered an incompatible change.
- Add a new exception `icat.exception.DataConsistencyError`. Raise this in `icat.entity.Entity.getUniqueKey()` if a relation that is required in a constraint is not set.
- Rename `icat.exception.SearchResultError` to `icat.exception.SearchAssertionError`. SearchResultError was a misnomer here, as this exception class is very specific to `icat.client.Client.assertedSearch()`. Add a new generic exception class `icat.exception.SearchResultError` and derive `icat.exception.SearchAssertionError` from it. This way, the change should not create any compatibility problems in client programs.
- Add a check in `icat.icatcheck` that the `icat.exception.ICATError` subclasses are in sync with *icatExceptionType* as defined in the schema.
- Bugfix: The code dealing with exceptions raised by the ICAT server did require all attributes in IcatException sent by the server to be set, although some of these attributes are marked as optional in the schema.
- Do not delete the Suds cache directory in `icat.client.Client.cleanup()`.
- Installation: python-icat requires Python 2.6 or newer. Raise an error if *setup.py* is run by a too old Python version.
- Move some internal routines in a separate module `icat.helper`.
- Greatly improved example scripts *icatdump* and *icatrestore*.

1.4.22 0.3.0 (2014-01-10)

- Add support for ICAT 4.3.1. (Compatibility with ICAT 4.3.2 has also been tested but did not require any changes.)
- Implement alias names for entity attributes. This facilitates compatibility of client programs to different ICAT versions. E.g. a client program may use *rule.grouping* regardless of the ICAT version, for ICAT 4.2.* this is aliased to *rule.group*.
- Add a method `icat.client.Client.assertedSearch()`.
- Add a method `icat.entity.Entity.getUniqueKey()`.
- Add entity methods `Group.getUsers()` and `Instrument.getInstrumentScientists()`.

- WARNING, incompatible change!

  Changed entity methods `Instrument.addInstrumentScientist()` and `Investigation.addInvestigationUser()` to not to create the respective user any more, but rather expect a list of existing users as argument. Renamed `Group.addUser()`, `Instrument.addInstrumentScientist()`, and `Investigation.addInvestigationUser()` to `addUsers()`, `addInstrumentScientists()`, and `addInvestigationUsers()` (note the plural "s") respectively.

  In the client code, replace:

```
pi = investigation.addInvestigationUser(uid, fullName=userName,
                                        search=True,
                                        role="Principal Investigator")
```

  by

```
pi = client.createUser(uid, fullName=userName, search=True)
investigation.addInvestigationUsers([pi], role="Principal Investigator")
```

- Work around a bug in the way SUDS deals with datetime values: set the local time zone to `UTC`.
- Add example scripts *icatdump* and *icatrestore*.

1.4.23 0.2.0 (2013-11-18)

- Rework internals of *`icat.config`*.
- Bugfix: *`icat.config.Config`* required a password to be set even if prompt for password was requested.
- Add support for configuration via environment variables.
- Add support of HTTP proxy settings. [Suggested by Alistair Mills]
- WARNING, incompatible change! The configuration read by *`icat.config`* is not stored as attributes on the *`icat.config.Config`* object itself, but rather *`icat.config.Config.getconfig()`* returns an object with these attributes set. This keeps the configuration values cleanly separated from the attributes of the *`icat.config.Config`* object.

  In the client code, replace:

```
conf = icat.config.Config()
conf.getconfig()
```

  by

```
config = icat.config.Config()
conf = config.getconfig()
```

- Move `ConfigError` from *`icat.config`* to *`icat.exception`*.
- Move `GenealogyError` from *`icat.icatcheck`* to *`icat.exception`*.
- Review export of symbols. Most client programs should only need to import `icat` and *`icat.config`*.

1.4.24 0.1.0 (2013-11-01)

- Initial version

CHAPTER 2

Indices and tables

- genindex
- search

Python Module Index

i

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y