PyCap Documentation

Release 1.0

Scott Burns

October 12, 2018

Contents

PyCap is a python module to communicate with REDCap databases through the Redcap API.

Contents:

CHAPTER 1

Introduction

PyCap is designed to be a minimal interface between you and the REDCap Application Programming Interface (API). All required and optional parameters for the API should be exposed through PyCap and it should extremely easy to discern their mapping.

Philosophy

The REDCap API is pretty minimal. There is no built-in search or pagination, for example. However, it does expose all the functionality required to build advanced data management services on top of the API.

In the same way, PyCap is minimal by design. It doesn't do anything fancy behind the scenes and will not prevent you from shooting yourself in the foot. However, it should be very easy to understand and mentally-map PyCap functionality to the REDCap API.

License

CHAPTER 2

Installation

Install the latest version with pip:

```
$ pip install PyCap
```

To install the bleeding edge from the github repo, use the following:

```
$ pip install -e git+https://github.com/sburns/PyCap.git#egg=PyCap
```

The only requirement is requests which will be installed automatically for you by `pip`.

CHAPTER 3

Quickstart

PyCap makes it very simple to interact with the data stored in your REDCap projects:

```
from redcap import Project
api_url = 'https://redcap.example.edu/api/'
api_key = 'SomeSuperSecretAPIKeyThatNobodyElseShouldHave'
project = Project(api_url, api_key)
```

Export all the data:

```
data = project.export_records()
```

Import all the data:

```
to_import = [{'record': 'foo', 'test_score': 'bar'}]
response = project.import_records(to_import)
```

Import a file:

```
fname = 'something_to_upload.txt'
with open(fname, 'r') as fobj:
    project.import_file('1', 'file', fname, fobj)
```

Export a file:

```
content, headers = project.export_file('1', 'file')
with open(headers['name'], 'w') as fobj:
    fobj.write(content)
```

Delete a file:

```
try:
    project.delete_file('1', 'file')
except redcap.RedcapError:
    # Throws this if file wasn't successfully deleted
    pass
except ValueError:
    # You screwed up and gave it a bad field name, etc
    pass
```

The deep-dive into all the methods can be found in the full *Documentation*.

CHAPTER 4

Documentation

Connecting to Projects

The main class of PyCap is `redcap.Project`. It must be instantiated with the API URL of your REDCap site and a API token:

```
from redcap import Project, RedcapError
URL = 'https://redcap.example.com/api/'
API_KEY = 'ExampleKey'
project = Project(URL, API_KEY)
```

> **note** You will have one API key per redcap project to which you have access. To communicate between projects, you would create multiple `redcap.Project` instances.

API Keys are effectively your username and password for a particular project. If you have read/write access to a project (which is most likely), anyone with your redcap URL (public knowledge) and your API key has read/write access.

Since REDCap projects are often used to store Personal Health Information (PHI), it is of the **utmost importance** to:

- **Never share your API key**.
- **Delete the key (through the web interface) after you're done using it**.

Ignoring SSL Certificates

If you're connecting to a REDCap server whose SSL certificate can't be verified for whatever reason, you can add a `verify_ssl=False` argument in the `Project` constructor and no subsequent API calls to the REDCap server will attempt to verify the certficate.

By default though, the certificate will always be verified. Obviously, use this "feature" at your own risk. You are exposing yourself to man-in-the-middle attacks by using this.

Using a local CA_BUNDLE

Because PyCap uses requests under the hood, you can pass a path to your own CA_BUNDLE in the `verify_ssl` argument during `Project` instantiation and it will be used.

Project Attributes

When creating a `Project` object, PyCap goes ahead and makes some useful API calls and creates these attributes:

- `def_field`: What REDCap refers to as the unique key
- `forms`: A tuple of form names within the project
- `field_names`: A tuple of the raw fields
- `field_labels`: A tuple of the field's labels
- `events`: Unique event names (longitudinal projects)
- `arm_nums`: Unique arm numbers (longitudinal projects)
- `arm_names`: Unique arm names (longitudinal projects)

For non-longitudinal projects, `events`, `arm_nums`, and `arm_names` are empty tuples.

Metadata

Every Project object has an attribute named metadata. This is a list of dicts with the following keys (in no particular order):

- `field_name`: The raw field name
- `field_type`: The field type (text, radio, mult choice, etc.)
- `field_note`: Any notes for this field
- `required_field`: Whether the field is required
- `custom_alignment`: (Web-only) Determines how the field looks visually
- `matrix_group_name`: The matrix name this field belongs to
- `field_label`: The field label
- `section_header`: Under which section in the form page this field belongs
- `text_validation_min`: Minimum value for validation
- `branching_logic`: Any branching logic that REDCap uses to show or hide fields
- `select_choices_or_calculations`: For radio fields, the choices
- `question_number`: For survey fields, the survey number
- `form_name`: Form under which this field exists
- `text_validation_type_or_show_slider_number`: Validation type
- `identifier`: Whether this field has been marked as containing identifying information
- `text_validation_max`: Maximum value for validation

You can export the metadata on your own using the `export_metadata` method on `Project` objects.

Exporting Data

Exporting data is very easy with PyCap:

```
data = project.export_records()
```

`data` is a list of `dicts` with the raw field names as keys.

We can request slices to reduce the size of the transmitted data, which can be useful for large projects:

```
# Known record identifiers
ids_of_interest = ['1', '2', '3']
subset = project.export_records(records=ids_of_interest)
# Contains all fields, but only three records

# Known fields of interest
fields_of_interest = ['age', 'test_score1', 'test_score2']
subset = project.export_records(fields=fields_of_interest)
# All records, but only three columns

# Only want the first two forms
forms = project.forms[:2]
subset = project.export_records(forms=forms)
# All records, all fields within the first two forms
```

Note, no matter which fields or forms are requested, the `project.def_field` key will always be in the returned dicts.

Finally, you can tweak the how the data is labeled or formatted:

```
# Same data, but keys will the field labels
data = project.export_records(raw_or_label='label')

# You can also get the data in different formats
csv_data = project.export_records(format='csv') # or format='xml'

# quickly make a pandas.DataFrame
data_frame = project.export_records(format='df')
other_df = project.export_records(format='df', df_kwargs={'index_col': project.field_names[1]})
```

When you request a `DataFrame`, PyCap exports the data as csv and passes it to the `pandas.read_csv` function. The `df_kwargs` dict can be used to guide the conversion from csv to `DataFrame`.

Previously, PyCap enforced a strict intersection between the passed fields and `project.field_names` but that requirement was dropped in PyCap v0.5:

```
non_fields = ['foo', 'bar', 'bat']
response = project.export_records(fields=non_fields)
# response will contain dicts with only the def_field
```

Dealing with large exports

> **note** If your databases are smaller than about 1 million cells (X records x Y columns), you can safely ignore this section.

Exporting large projects will fail on REDCap's backend and PyCap will throw a `redcap.RedcapError`. The threshold for failure seems to be around 1 million cells but I haven't studied this empirically. So for large projects, the export call with default values will fail:

```
>>> project = Project(url, 'TokenToALargeProject')
>>> try:
>>>     data = project.export_records()
>>> except RedcapError:
```

```
>>>     print "Failure"
Failure
```

Here's an exporting function that trades speed for robustness:

```
def chunked_export(project, chunk_size=100):
    def chunks(l, n):
        """Yield successive n-sized chunks from list l"""
        for i in xrange(0, len(l), n):
            yield l[i:i+n]
    record_list = project.export_records(fields=[project.def_field])
    records = [r[project.def_field] for r in record_list]
    try:
        response = []
        for record_chunk in chunks(records, chunk_size):
            chunked_response = project.export_records(records=record_chunk)
            response.extend(chunked_response)
    except RedcapError:
        msg = "Chunked export failed for chunk_size={:d}".format(chunk_size)
        raise ValueError(msg)
    else:
        return response
```

The gist of the function:

- Define a sub-function that will yield successive n-sized chunks from a list.
- Export only the record identifiers. If this times out because you have a million records in your project, you effectively can't interact with the project through the API. Sorry.
- **Build a list of just the record identifiers and iterate on the chunks:**
  - Export the data for just this chunk of identifiers.
  - Extend an ongoing list of responses with this list of data.
- If any `export_records` call fails during the loop, a `ValueError` is raised. You should try again with a smaller chunk size. Otherwise, the list of responses is returned.

Caveats:

- You can do this with json responses because each chunked response is a list of dictionaries with no structure between records. This becomes much more difficult if you want csv or xml as there is much more structure in these responses.
- You could also do this with `pandas.DataFrame` but you'll want to `.append` the chunked dataframe, not extend.

I'm hesitant to include this as a method on `Project` because of these issues. I'm also not sure how often this is encountered in the real world. But feel free to use this function if you need it.

Regardless, you should remember that the REDCap instance you're working with is most likely a shared resource and you should always try to limit your API export requests to just the information you need at that point in time.

Importing Data

PyCap aims to make importing as easy as exporting:

```
# toy
def increment_score(record):
    record['score'] += 5

data = project.export_records(fields=['score'])
map(increment_score, data)
response = project.import_records(data)
# response['count'] is the number of records successfully updated

# import other formats too
response = project.import_records(csv_string, format='csv')

# PyCap will convert a DataFrame to csv and import it automatically
response = project.import_records(df)
```

Date String Formatting

If the REDCap server you're working with is older than version 5.9 (look at the footer on the main page of your site to find your version), date strings to be imported can be formatted as either `'YYYY-MM-DD'` **or** `'MM/DD/YYYY'`. Beginning with v5.9, the API will **only** accept `'YYYY-MM-DD'` formatting unless you specify the `date_format` parameter in the `import_records` call. Possible values are `'YMD'` (default), `'DMY'` or `'MDY'`:

```
to_import = [{'record': '1', 'date_of_birth': '02/14/2000'}]
response = project.import_records(to_import, date_format='MDY')
```

Working with Files

You can download files in a REDCap project (exporting) and upload local files (import) to a REDCap project. You can also delete them but there is no undo button for this operation.

> **note** Unlike exporting and importing data, exporting/importing/deleting files can only be done for a single record at a time.

Generally, you will be given bytes from the file export method so binary-formatted data can be written properly and you are expected to pass an open file object for file importing. Of course, you should open a file you wish to import with a well-chosen mode.

The REDCap API doesn't send any return message for file methods. Therefore, it's important to watch out for `redcap.RedcapError` exceptions that may occur when a request fails on the server. If this isn't thrown, you can assume your request worked:

```
try:
    file_content, headers = project.export_file(record='1', field='file')
except RedcapError:
    # file_content will actually contain an error message now that might be useful to look at.
    pass
else:
    # Note, you may want to change the mode in which you're opening files
    # based on the header['name'] value, but that is completely up to you.
    mode = 'wb' if headers['name'].endswith('.pdf') else 'w'
    with open(headers['name'], mode) as f:
        f.write(file_content)


existing_fname = 'to_upload.pdf'
```

```
fobj = open(existing_fname, 'rb')
field = 'data_file'
# In the REDCap UI, the link to download the file will be named the fname you pass as the ``fname`` p
try:
    response = project.import_file(record='1', field=field, fname=existing_fname, fobj=fobj)
except RedcapError:
    # Your import didn't work
    pass
finally:
    fobj.close()

# And deleting...
try:
    project.delete_file('1', field)
except RedcapError:
    # The file wasn't deleted
    pass
else:
    # It's gone
    pass

# Attempting to do any file-related operation on a non-file field will raise a ValueError quickly
try:
    project.import_file(record='1', field='numeric_field', fname, fobj)
except ValueError:
    # Bingo
```

Exporting Users

You can also export data related to the fellow users of your REDCap project:

```
users = project.export_users()
for user in users:
    assert 'firstname' in user
    assert 'lastname' in user
    assert 'email' in user
    assert 'username' in user
    assert 'expiration' in user
    assert 'data_access_group' in user
    assert 'data_export' in user
    assert 'forms' in user
```

So each dict in the exported users list contains the following key, value pairs:

- `firstname`: First name of the user
- `lastname`: Last name of the user
- `email`: Email address for the user
- `username`: The username of the user
- `expiration`: The user's access expiration date (empty if no expiration)
- `data_access_group`: Data access group of the user
- `data_export`: An integer where 0 means they have no access, 2 means they get a De-Identified data, and 1 means they can export the full data set

- `forms`: A list of dicts, each having one key (the form name) and an integer value, where 0 means they have no access, 1 means they can view records/responses and edit records (survey responses are read-only), 2 means they can only read surveys and forms, and 3 means they can edit survey responses as well as forms

You can also specify the `format` argument to `project.export_users` to be `'csv'` or `'xml'` and get strings in those respective formats, though `json` is default and will return the decoded objects.

Exporting Form-Event Mappings

Longitudinal projects have a mapping of what forms are available to collect data within each event. These mappings can be exported from the `Project`:

```
fem = project.export_fem()
# Only ask for particular arms
subset = project.export_fem(arms=['arm1'])

# You can also get a DataFrame of the FEM
fem_df = project.export_fem(format='df')
```

Full API

Full API documentation can be found in the *API* docs.

CHAPTER 5

API

PyCap is structured into two layers: a high level that exposes `redcap.Project` and a low-level, `redcap.request`. Users (like yourself) should generally only worry about working the `redcap.Project` class.

JSON Handling

For any request with a `format='json'` argument, the API will respond with a JSON-formatted string representation of the response. This is automatically decoded by PyCap into a list of python dictionaries. This is the default format for all requests.

High-Level

The `redcap.Project` class is the high-level object of the module. Generally you'll only need this class.

class `redcap.project.`**`Project`**(*url*, *token*, *name=''*, *verify_ssl=True*)

Main class for interacting with REDCap projects

`__init__`(*url*, *token*, *name=''*, *verify_ssl=True*)

Parameters **url** : str

API URL to your REDCap server

token : str

API token to your project

name : str, optional

name for project

verify_ssl : boolean, str

Verify SSL, default True. Can pass path to CA_BUNDLE.

`delete_file`(*record*, *field*, *return_format='json'*, *event=None*)

Delete a file from REDCap

Parameters **record** : str

record ID

field : str

field name

return_format : (`'json'`), `'csv'`, `'xml'`

return format for error message

event : str

If longitudinal project, event to delete file from

Returns **response** : dict, str

response from REDCap after deleting file

Notes

There is no undo button to this.

export_fem(*arms=None*, *format='json'*, *df_kwargs=None*)
Export the project's form to event mapping

Parameters **arms** : list

Limit exported form event mappings to these arm numbers

format : (`'json'`), `'csv'`, `'xml'`

Return the form event mappings in native objects, csv or xml, `'df'` will return a `pandas.DataFrame`

df_kwargs : dict

Passed to pandas.read_csv to control construction of returned DataFrame

Returns **fem** : list, str, `pandas.DataFrame`

form-event mapping for the project

export_file(*record*, *field*, *event=None*, *return_format='json'*)
Export the contents of a file stored for a particular record

Parameters **record** : str

record ID

field : str

field name containing the file to be exported.

event: str :

for longitudinal projects, specify the unique event here

return_format: ('json'), 'csv', 'xml' :

format of error message

Returns **content** : bytes

content of the file

content_map : dict

content-type dictionary

Notes

Unlike other export methods, this works on a single record.

export_metadata(*fields=None*, *forms=None*, *format='json'*, *df_kwargs=None*)
Export the project's metadata

Parameters **fields** : list

Limit exported metadata to these fields

forms : list

Limit exported metadata to these forms

format : ('json'), 'csv', 'xml', 'df'

Return the metadata in native objects, csv or xml. 'df' will return a `pandas.DataFrame`.

df_kwargs : dict

Passed to `pandas.read_csv` to control construction of returned DataFrame. by default `{'index_col': 'field_name'}`

Returns **metadata** : list, str, `pandas.DataFrame`

metadata sttructure for the project.

export_records(*records=None*, *fields=None*, *forms=None*, *events=None*, *raw_or_label='raw'*, *event_name='label'*, *format='json'*, *export_survey_fields=False*, *export_data_access_groups=False*, *df_kwargs=None*)
Export data from the REDCap project.

Parameters **records** : list

array of record names specifying specific records to export. by default, all records are exported

fields : list

array of field names specifying specific fields to pull by default, all fields are exported

forms : list

array of form names to export. If in the web UI, the form name has a space in it, replace the space with an underscore by default, all forms are exported

events : list

an array of unique event names from which to export records

note this only applies to longitudinal projects

raw_or_label : ('raw'), 'label', 'both'

export the raw coded values or labels for the options of multiple choice fields, or both

event_name : ('label'), 'unique'

export the unique event name or the event label

format : ('json'), 'csv', 'xml', 'df'

Format of returned data. 'json' returns json-decoded objects while 'csv' and 'xml' return other formats. 'df' will attempt to return a `pandas.DataFrame`.

export_survey_fields : (`False`), True

specifies whether or not to export the survey identifier field (e.g., "redcap_survey_identifier") or survey timestamp fields (e.g., form_name+"_timestamp") when surveys are utilized in the project.

export_data_access_groups : (`False`), `True`

specifies whether or not to export the `"redcap_data_access_group"` field when data access groups are utilized in the project.

note This flag is only viable if the user whose token is being used to make the API request is *not* in a data access group. If the user is in a group, then this flag will revert to its default value.

df_kwargs : dict

Passed to `pandas.read_csv` to control construction of returned DataFrame. by default, `{'index_col': self.def_field}`

Returns **data** : list, str, `pandas.DataFrame`

exported data

`export_users` (*format='json'*)

Export the users of the Project

Parameters **format** : (`'json'`), `'csv'`, `'xml'`

response return format

Returns **users: list, str** :

list of users dicts when `'format'='json'`, otherwise a string

Notes

Each user will have the following keys:

- `'firstname'` : User's first name
- `'lastname'` : User's last name
- `'email'` : Email address
- `'username'` : User's username
- `'expiration'` : Project access expiration date
- `'data_access_group'` : data access group ID
- `'data_export'` : (0=no access, 2=De-Identified, 1=Full Data Set)
- **`'forms'`** [a list of dicts with a single key as the form name and] value is an integer describing that user's form rights, where: 0=no access, 1=view records/responses and edit records (survey responses are read-only), 2=read only, and 3=edit survey responses,

`filter` (*query*, *output_fields=None*)

Query the database and return subject information for those who match the query logic

Parameters **query: Query or QueryGroup** :

Query(Group) object to process

output_fields: list :

The fields desired for matching subjects

Returns **A list of dictionaries whose keys contains at least the default field** :

and at most each key passed in with output_fields, each dictionary :

representing a surviving row in the database. :

filter_metadata(*key*)

Return a list of values for the metadata key from each field of the project's metadata.

Parameters **key: str** :

A known key in the metadata structure

Returns **filtered :** :

attribute list from each field

import_file(*record*, *field*, *fname*, *fobj*, *event=None*, *return_format='json'*)

Import the contents of a file represented by fobj to a particular records field

Parameters **record** : str

record ID

field : str

field name where the file will go

fname : str

file name visible in REDCap UI

fobj : file object

file object as returned by *open*

event : str

for longitudinal projects, specify the unique event here

return_format : ('json'), 'csv', 'xml'

format of error message

Returns **response :** :

response from server as specified by `return_format`

import_records(*to_import*, *overwrite='normal'*, *format='json'*, *return_format='json'*, *return_content='count'*, *date_format='YMD'*)

Import data into the RedCap Project

Parameters **to_import** : array of dicts, csv/xml string, `pandas.DataFrame`

note If you pass a csv or xml string, you should use the `format` parameter appropriately.

note Keys of the dictionaries should be subset of project's, fields, but this isn't a requirement. If you provide keys that aren't defined fields, the returned response will contain an `'error'` key.

overwrite : ('normal'), 'overwrite'

`'overwrite'` will erase values previously stored in the database if not specified in the to_import dictionaries.

format : ('json'), 'xml', 'csv'

Format of incoming data. By default, to_import will be json-encoded

return_format : ('json'), 'csv', 'xml'

Response format. By default, response will be json-decoded.

return_content : ('count'), 'ids', 'nothing'

By default, the response contains a 'count' key with the number of records just imported. By specifying 'ids', a list of ids imported will be returned. 'nothing' will only return the HTTP status code and no message.

date_format : ('YMD'), 'DMY', 'MDY'

Describes the formatting of dates. By default, date strings are formatted as 'YYYY-MM-DD' corresponding to 'YMD'. If date strings are formatted as 'MM/DD/YYYY' set this parameter as 'MDY' and if formatted as 'DD/MM/YYYY' set as 'DMY'. No other formattings are allowed.

Returns **response** : dict, str

response from REDCap API, json-decoded if `return_format == 'json'`

is_longitudinal()

Returns **boolean :** :

longitudinal status of this project

metadata_type(*field_name*)
If the given field_name is validated by REDCap, return it's type

names_labels(*do_print=False*)
Simple helper function to get all field names and labels

exception redcap.**RedcapError**
This is thrown when an API method fails. Depending on the API call, the REDCap server will return a helpful message. Sometimes it won't :(

Low-Level

The `Project` class makes all HTTP calls to the REDCap API through the `redcap.request.RCRequest` class. You shouldn't need this in day-to-day usage.

exception redcap.request.**RCAPIError**
Errors corresponding to a misuse of the REDCap API

class redcap.request.**RCRequest**(*url*, *payload*, *qtype*)
Private class wrapping the REDCap API. Decodes response from redcap and returns it.

References

https://redcap.vanderbilt.edu/api/help/

Users shouldn't really need to use this, the Project class is the biggest consumer.

__init__(*url*, *payload*, *qtype*)
Constructor

Parameters **url** : str

REDCap API URL

payload : dict

key,values corresponding to the REDCap API

qtype : str

Used to validate payload contents against API

execute(***kwargs*)

Execute the API request and return data

Parameters kwargs : :

passed to requests.post()

Returns response : list, str

data object from JSON decoding process if format==’json’, else return raw string (ie format==’csv’|’xml’)

expect_empty_json()

Some responses are known to send empty responses

get_content(*r*)

Abstraction for grabbing content from a returned response

raise_for_status(*r*)

Given a response, raise for bad status for certain actions

Some redcap api methods don’t return error messages that the user could test for or otherwise use. Therefore, we need to do the testing ourself

Raising for everything wouldn’t let the user see the (hopefully helpful) error message

validate()

Checks that at least required params exist

r

r

Symbols

D

E

F

G

I

M

N

P

R

V