pypco Documentation

Release 1.2.0

Bill Deitrick

Mar 04, 2023

Contents

Pypco provides a simple Python interface to communicate with the Planning Center Online REST API. With automatic rate limit handling, pagination handling, object templating, and support for the full breadth of the PCO API, pypco can support any Python application or script needing to utilize the PCO API.

CHAPTER 1

Contents

1.1 Getting Started

1.1.1 Installation

`pip` is the easiest way to install pypco. Run the single command below and you're done.

```
pip install pypco
```

The excellent `pipenv` from Kenneth Reitz is also highly recommended. Assuming you already have pipenv installed, run the single command below and you're ready to go.

```
pipenv install pypco
```

Alternatively, if you want the bleeding edge or you want the source more readily available, you can install from GitHub.

Either clone the repository:

```
git clone git://github.com/billdeitrick/pypco.git
```

Or download a tarball:

```
curl -OL https://github.com/billdeitrick/pypco/tarball/master
```

You can also substitute "zipball" for "tarball" in the URL above to get a zip file instead.

Once you've extracted the code and have a shell in the extracted directory, you can install pypco to your Python's site-packages with:

```
python setup.py install
```

Alternatively, you could simply embed the pypco source directory in your own Python package.

Once you've gotten pypco installed, make sure you can import it in your Python script or at the interactive Python prompt with:

```
import pypco
```

Once you can import the module, you're ready to go!

1.1.2 Authentication

The PCO API offers two options for authentication: OAuth or Personal Access Tokens (hereafter referred to as PAT). Typically, you would use PAT for applications that will access your own data (i.e. applications that you create and use yourself) and OAuth for applications that will access a third-party's data (i.e. applications you will package and make available for others to use.)

PCO provides much more detail about authentication in their API Documentation.

Personal Access Token (PAT) Authentication

PAT authentication is the simplest method (and probably what you'll use if you're wanting to kick the tires), so we'll start there.

First, you'll need to generate your application id and secret. Sign into your account at api.planningcenteronline.com. Click **Applications** in the toolbar and then click **New Personal Access Token** under the Personal Access Tokens heading. Enter a description of your choosing, and select the desired API version for your app. Pypco can support any API version or combination of versions. Click **Submit**, and you'll see your generated token.

Now, you're ready to connect to the PCO API. The example below demonstrates authentication with a PAT and executes a simple query to get and display single person from your account.

```
import pypco

# Get an instance of the PCO object using your personal access token.
pco = pypco.PCO("<APPLICATION_ID_HERE>", "<APPLICATION_SECRET_HERE>")

# Get a single person from your account and display their information
# The iterate() function provides an easy way to retrieve lists of objects
# from an API endpoint, and automatically handles pagination
people = pco.iterate('/people/v2/people')
person = next(people)
print(person)
```

If you can run the above example and see output for one of the people in your PCO account, you have successfully connected to the API. Continue to the *API Tour* to learn more, or learn about OAuth Authentication below.

OAuth Authentication

OAuth is the more complex method for authenticating against PCO, but is what you'll want to use if you're building an app that accesses third-party data.

Before diving in, it's helpful to have an understanding of OAuth basics, both in general and as they apply to the PCO API specifically. You'll want to familiarize yourself with PCO's Authentication docs, and if you're looking to learn more about OAuth in particular you can learn everything you need to know over at oauth.net.

To get started, you'll need to register your OAuth app with PCO. To do this, sign into your account at api.planningcenteronline.com. Click **Applications** in the toolbar and then click **New Application** under the My Developer Tokens (OAuth) heading. Fill out the required information and click **Submit**, and you'll see your generated client id and secret.

Now, you're ready to connect to the PCO API. The example below demonstrates authentication with OAuth. Note that you'll have significantly more work to do than with PAT; you'll need to use a browser to display PCO's authentication page with the appropriate parameters and have a redirect page which will be able to hand you the returned code parameter (which you'll use to get your access and refresh tokens). While most of the heavy lifting is up to you, pypco does provide a few convenience functions to help with the process as demonstrated in the example below.

```
import pypco

# Generate the login URI
redirect_url = pypco.get_browser_redirect_url(
    "<CLIENT_ID_HERE>",
    "<REDIRECT_URI_HERE>",
    ["scope_1", "scope_2"]
)

# Now, you'll have the URI to which you need to send the user for authentication
# Here is where you would handle that and get back the code parameter PCO returns.

# For this example, we'll assume you've handled this and now have the code
# parameter returned from the API

# Now, we'll get the OAuth access token json response using the code we received from␣
↪PCO
token_response = pypco.get_oauth_access_token(
    "<CLIENT_ID_HERE>",
    "<CLIENT_SECRET_HERE>",
    "<CODE_HERE>",
    "<REDIRECT_URI_HERE>"
)

# The response you'll receive from the get_oauth_access_token function will include␣
↪your
# access token, your refresh token, and other metadata you may need later.
# You may wish/need to store this entire response on disk as securely as possible.
# Once you've gotten your access token, you can initialize a pypco object like this:
pco = pypco.PCO(token=token_response['access_token'])

# Now, you're ready to go.
# The iterate() function provides an easy way to retrieve lists of objects
# from an API endpoint, and automatically handles pagination
people = pco.iterate('/people/v2/people')
person = next(people)
print(person)
```

OAuth tokens will work for up to two hours after they have been issued, and can be renewed with a refresh token. Again, pypco helps you out here by providing a simple convenience function you can use to refresh OAuth tokens.

```
import pypco

# Refresh the access token
token_response = pypco.get_oauth_refresh_token("<CLIENT_ID_HERE>", "<CLIENT_SECRET_
↪HERE>", "<REFRESH_TOKEN_HERE>")
```

(continues on next page)

(continued from previous page)

```
# You'll get back a response similar to what you got calling get_oauth_access_token
# the first time you authenticated your user against PCO. Now, you can initialize a␣
↪PCO object and make some API calls.
pco = pypco.PCO(token=token_response['access_token'])
people = pco.iterate('/people/v2/people')
person = next(people)
print(person)
```

Church Center API Organization Token (OrganizationToken) Authentication

If you want to access api.churchcenter.com endpoints you need to use an OrganizationToken. We have added the ability for pypco to get these OrganizationTokens for you using `cc_name` as an auth option.

You need to pass the vanity portion of the church center url as cc_name when initializing the PCO object. To auth for https://carlsbad.churchcenter.com use `carlsbad` as the `cc_name`.

Now, you're ready to connect to the Church Center API. The example below demonstrates authentication with an Org Token, the steps needed to change the base url, and executes a simple query to get and display events from the church center api.

```
import pypco

# Get an instance of the PCO object using your personal access token.
# Set the api_base to https://api.churchcenter.com
pco = pypco.PCO(cc_name='carlsbad',
                api_base="https://api.churchcenter.com")

# Get the events from api.churchcenter.com
# The iterate() function provides an easy way to retrieve lists of objects
# from an API endpoint, and automatically handles pagination

events = pco.iterate('/calendar/v2/events')
event = next(events)
print(event)
```

If you can run the above example and see output for one of the events in Test Church Center account, you have successfully connected to the API. Continue to the *API Tour* to learn more.

OrgTokens are generated with every request to the Church Center API so they should always be fresh.

1.1.3 Conclusion

Once you've authenticated and been able to make a simple API call, you're good to go. Head over to the *API Tour* document for a brief tour of the pypco API; this document will show you how pypco calls relate to their PCO API counterparts. Once you've read through the API Tour, you should be ready to fully leverage the capabilities of pypco (and hopefully be done reading pypco documentation. . . you'll be able to know exactly what pypco calls to make by reading the PCO API docs).

1.2 API Tour

1.2.1 Introduction

This API Tour will quickly introduce you to pypco conventions and so you'll be on your way to building cool stuff. Once you've spent a few minutes learning the ropes with pypco, you'll spend the rest of your time directly in the PCO API docs to figure out how to craft your requests.

For the purposes of this document, we'll assume you've already been able to authenticate successfully using the methods described in the *Getting Started Guide*. Thus, each of the examples below will assume that you've already done something like the following to import the pypco module and initialize an instance of the PCO object:

```
>>> import pypco

>>> pco = pypco.PCO("<APP_ID_HERE>", "<APP_SECRET_HERE>")
```

Also for purposes of this guide, we'll assume you're already somewhat familiar with the PCO API (docs here). If you're not, you might want to read the Introduction and then come back here.

URL Passing and `api_base`

As you'll see shortly, for most requests you'll specify portions of the URL corresponding to the API endpoint against which you would like to make a REST call. You don't need to specify the protocol and hostname portions of the URL; these are automatically prepended for you whenever you pass a URL into pypco. Pypco refers to the automatically prepended protocol and hostname as `api_base`. By default, `api_base` is `https://api.planningcenteronline.com` (though an alternative can be passed as an argument). So, you'll want to include a forward slash at the beginning of any URL argument you pass. Don't worry if this is confusing right now; it will all make sense once you've read the examples below.

Often times, you may find it would be easier to pass a URL to pypco that includes `api_base`. You might want to do this in situations where you've pulled a URL to a specific object in the PCO API directly from an attribute on an object you've already retrieved (such as a "links" attribute). Pypco has no problem if you include the `api_base` in a URL you pass in; it's smart enough to detect that it doesn't need to prepend `api_base` again in this situation so there's no need for you to worry about stripping it out.

1.2.2 The Basics: GET, POST, PATCH, and DELETE

GET: Retrieving Objects

Let's start with a simple GET request where we retrieve a specific person from the People API and print their name:

```
# Retrieve the person with ID 71059458
# (GET https://api.planningcenteronline.com/people/v2/people/71059458)
>>> person = pco.get('/people/v2/people/71059458') # Returns a Dict
>>> print(person['data']['attributes']['name'])
John Smith
```

If you need to pass any URL parameters, you can pass them to `get()` as keyword arguments, like this:

```
>>> person = pco.get('/people/v2/people/71059458', test='hello', test2='world')
```

This would result in the following request to the API: `GET https://api.planningcenteronline.com/people/v2/people/71059458?test=hello&test2=world`

Using keyword arguments you can pass any parameters supported by the PCO API for a given endpoint.

> **NOTE:** URL parameter keyword arguments must be passed as strings. If you are passing an object like a `dict` or a `list` for instance, cast it to a `str` and verify it is in the format required by the PCO API!

There may be times where you need to pass URL parameters that you can't pass as function arguments (for example, when the URL parameters contain characters that can't be used in a Python variable name). In these situations, create a `dict` and pass the keyword arguments using the double splat operator:

```
>>> params = {
  'where[first_name]': 'pico',
  'where[last_name]': 'robot'
}
>>> result = pco.get('/people/v2/people', **params)
```

You can learn more about the `get()` function in the PCO module docs.

PATCH: Updating Objects

When altering existing objects in the PCO API, you only need to pass the attributes in your request payload that you wish to change. The easiest way to to generate the necessary payload for your request is using the `template()` function. The `template()` function takes the object type and attributes as arguments and returns a `dict` object that you can pass to `patch()` (which will serialize the object to JSON for you). There is, of course, no reason you have to use the `template()` function, but this is provided for you to help speed the process of generating request payloads.

In this example we'll change an existing person object's last name, using the `template()` function to generate the appropriate payload.

```
# First we'll retrieve the existing person and print their last name
>>> person = pco.get('/people/v2/people/71059458') # Returns a Dict
>>> print(person['data']['attributes']['name'])
John Smith

# Next, we'll use the template() function to build our JSON payload
# for the PATCH request
>>> update_payload = pco.template('Person', {'last_name': 'Rolfe'})

# Perform the PATCH request; patch() will return the updated object
>>> updated_person = pco.patch(person['data']['links']['self'], payload=update_
↪payload)
>>> print(updated_person['data']['attributes']['name'])
John Rolfe
```

Be sure to consult the PCO API Docs for whatever object you are attempting to update to ensure you are passing assignable attributes your payload. If you receive an exception when attempting to update an object, be sure to read *exception handling* below to learn how to find the most information possible about what went wrong.

Aside from the `payload` keyword argument, any additional arguments you pass to the `patch()` function will be sent as query parameters with your request.

You can learn more about the `patch()` function in the PCO module docs.

POST: Creating Objects

Similarly to altering existing objects via a PATCH request, the first step towards creating new objects in the PCO API is generally using the `template()` function to generate the necessary payload.

In the following example, we'll create a new person in PCO using the `template()` function to generate the payload for the request.

```
# Create a payload for the request
>>> create_payload = pco.template(
  'Person',
  {
    'first_name': 'Benjamin',
    'last_name': 'Franklin',
    'nickname': 'Ben'
  }
)

# Create the person object and print the name attribute
>>> person = pco.post('/people/v2/people', payload=create_payload)
>>> print(person['data']['attributes']['name'])
Benjamin Franklin
```

Just like `patch()`, always be sure to consult the PCO API docs for the object type you are attempting to create to be sure you are passing assignable attributes in the correct format. If you do get stuck, the *exception handling* section below will help you learn how to get the most possible information about what went wrong.

Also just like `patch()`, any keyword arguments you pass to `post()` aside from the `payload` argument will be added as parameters to your API request.

Aside from object creation, HTTP POST requests are also used by various PCO API endpoints for "Actions". These are endpoint-specific operations supported by various endpoints, such as the Song Endpoint. You can use the `post()` function for Action operations as needed; be sure to pass in the appropriate argument to the `payload` parameter (as a `dict`, which will automatically be serialized to JSON for you).

You can learn more about the `post()` function in the PCO module docs.

DELETE: Removing Objects

Removing objects is probably the simplest operation to perform. Simply pass the desired object's URL to the `delete()` function:

```
>>> response = pco.delete('/people/v2/people/71661010')
>>> print(response)
<Response [204]>
```

Note that the `delete()` function returns a Requests `Response` object instead of a `dict` since the PCO API always returns an empty payload for a DELETE request. The `Response` object returned by a successful DELETE request will have a `status_code` value of 204.

As usual, any keyword arguments you pass to `delete()` will be passed to the PCO API as query parameters (though you typically won't need query parameters for DELETE requests).

You can learn more about the `delete()` function in the PCO module docs.

1.2.3 Advanced: Object Iteration and File Uploads

Object Iteration with `iterate()`

Querying an API endpoint that returns a (possibly quite large) list of results is probably something you'll need to do at one time or another. To simplify this common use case, pypco provides the `iterate()` function. `iterate()`

is a generator function that performs GET requests against API endpoints that return lists of objects, transparently handling pagination.

Let's look at a simple example, where we iterate through all of the `person` objects in PCO People and print out their names:

```
>>> for person in pco.iterate('/people/v2/people'):
>>>   print(person['data']['attributes']['name'])
John Rolfe
Benjamin Franklin
...
```

Just like `get()`, any keyword arguments you pass to `iterate()` will be added to your HTTP request as query parameters. For many API endpoints, this will allow you to build specific queries to pull data from PCO. In the example below, we demonstrate searching for all `person` objects with the last name "Rolfe". Note the use of the double splat operator to pass parameters as explained *above*.

```
>>> params = {
  'where[last_name]': 'Rolfe'
}
>>> for person in pco.iterate('/people/v2/people', **params):
>>>                print(person['data']['attributes']['name'])
John Rolfe
...
```

Often you will want to use includes to return associated objects with your call to `iterate()`. To accomplish this, you can simply pass `includes` as a keyword argument to the `iterate()` function. To save you from having to find which includes are associated with a particular object yourself, `iterate()` will return objects to you with only their associated includes.

You can learn more about the `iterate()` function in the PCO module docs.

File Uploads with `upload()`

Pypco provides a simple function to support file uploads to PCO (such as song attachments in Services, avatars in People, etc). To facilitate file uploads as described in the PCO API docs for file uploads, you'll first use the `upload()` function to upload files from your disk to PCO. This action will return to you a unique ID (UUID) for your newly uploaded file. Once you have the file UUID, you'll pass this to an endpoint that accepts a file.

In the example below, we upload a avatar image for a person in PCO People and associate it with the appropriate person object:

```
# Upload the file, receive response containing UUID
>>> upload_response = pco.upload('john.jpg')
# Update the avatar attribute on the appropriate person object
# and print the resulting URL
>>> avatar_update = pco.template(
                  'Person',
                  {'avatar': upload_response['data'][0]['id']}
                )
>>> person = pco.patch('/people/v2/people/71059458', payload=avatar_update)
>>> print(person['data']['attributes']['avatar'])
https://avatars.planningcenteronline.com/uploads/person/71059458-1578368234/avatar.2.
↪jpg
```

As usual, any keyword arguments you pass to `upload()` will be passed to the PCO API as query parameters (though you typically won't need query parameters for file uploads).

You can learn more about the `upload()` function in the PCO module docs.

1.2.4 Exception Handling

Pypco provides custom exception types for error handling purposes. All exceptions are defined in the exceptions module, and inherit from the base PCOExceptions class.

Most of the pypco exception classes are fairly mundane, though the PCORequestException class is worth a closer look. This exception is raised in circumstances where a connection was made to the API, but the API responds with a status code indicative of an error (other than a rate limit error, as these are handled transparently as discussed below). To provide as much helpful diagnostic information as possible, `PCORequestException` provides three attributes with more data about the failed request: `status_code`, `message`, and `response_body`. You can find more details about each of these attributes in the PCORequestException Docs. A brief example is provided below showing what sort of information each of these variables might contain when a request raises this exception:

```
# Create an invalid payload to use as an example
>>> bad_payload = pco.template(
                        'Person',
                    {'bogus': 'bogus'}
        )

# Our bad payload will raise an exception...print out attributes
# from PCORequestException
>>> try:
>>>    result = pco.patch('/people/v2/people/71059458', payload=bad_payload)
>>>         except Exception as e:
>>>    print(f'{e.status_code}\n-\n{e.message}\n-\n{e.response_body}')
422
-
422 Client Error: Unprocessable Entity for url:
-
https://api.planningcenteronline.com/people/v2/people/71059458
{"errors":[{"status":"422","title":"Forbidden Attribute","detail":"bogus cannot be␣
↪assigned"}]}
```

You can find more information about all types of exceptions raised by pypco in the PCOExceptions module docs.

1.2.5 Rate Limit Handling

Pypco automatically handles rate limiting for you. When you've hit your rate limit, pypco will look at the value of the `Retry-After` header from the PCO API and automatically pause your requests until your rate limit for the current period has expired. Pypco uses the `sleep()` function from Python's `time` package to do this. While the `sleep()` function isn't reliable as a measure of time per se because of the underlying kernel-level mechanisms on which it relies, it has proven accurate enough for this use case.

1.3 pypco

1.3.1 pypco package

Submodules

pypco.auth_config module

Internal authentication helper objects for pypco.

class pypco.auth_config.**PCOAuthConfig**(*application_id: Optional[str] = None*, *secret: Optional[str] = None*, *token: Optional[str] = None*, *cc_name: Optional[str] = None*)

Bases: object

Auth configuration for PCO.

Parameters

- **application_id** (*str*) – The application ID for your application (PAT).
- **secret** (*str*) – The secret for your application (PAT).
- **token** (*str*) – The token for your application (OAUTH).
- **cc_name** (*str*) – The vanity name portion of the <vanity_name>.churchcenter.com url
- **auth_type** (PCOAuthType) – The authentication type specified by this config object.

auth_header

Get the authorization header for this authentication configuration scheme.

Returns The authorization header text to pass as a request header.

Return type str

auth_type

The authentication type specified by this configuration.

Raises PCOCredentialsException – You have specified invalid authentication information.

Returns The authentication type for this config.

Return type *PCOAuthType*

class pypco.auth_config.**PCOAuthType**

Bases: enum.Enum

Defines PCO authentication types.

OAUTH = 2

ORGTOKEN = 3

PAT = 1

pypco.exceptions module

All pypco exceptions.

exception pypco.exceptions.**PCOCredentialsException**

Bases: *pypco.exceptions.PCOException*

Unusable credentials are supplied to pypco.

exception pypco.exceptions.**PCOException**

Bases: Exception

A base class for all pypco exceptions.

exception pypco.exceptions.**PCORequestException**(*status_code*, *message*, *response_body=None*)

Bases: *pypco.exceptions.PCOException*

The response from the PCO API indicated an error with your request.

Parameters

- **status_code** (*int*) – The HTTP status code corresponding to the error.
- **message** (*str*) – The error message string.
- **response_body** (*str*) – The body of the response (may include helpful information). Defaults to None.

status_code
The HTTP status code returned.

Type int

message
The error message string.

Type str

response_body
Text included in the response body. Often includes additional informative errors describing the problem encountered.

Type str

exception pypco.exceptions.**PCORequestTimeoutException**
Bases: *pypco.exceptions.PCOException*

Request to PCO timed out after the maximum number of retries.

exception pypco.exceptions.**PCOUnexpectedRequestException**
Bases: *pypco.exceptions.PCOException*

An unexpected exception has occurred attempting to make the request.

We don't have any additional information associated with this exception.

pypco.pco module

The primary module for pypco containing main wrapper logic.

class pypco.pco.**PCO**(*application_id: Optional[str] = None*, *secret: Optional[str] = None*, *token: Optional[str] = None*, *cc_name: Optional[str] = None*, *api_base: str = 'https://api.planningcenteronline.com'*, *timeout: int = 60*, *upload_url: str = 'https://upload.planningcenteronline.com/v2/files'*, *upload_timeout: int = 300*, *timeout_retries: int = 3*)

Bases: object

The entry point to the PCO API.

Note: You must specify either an application ID and a secret or an oauth token. If you specify an invalid combination of these arguments, an exception will be raised when you attempt to make API calls.

Parameters

- **application_id** (*str*) – The application_id; secret must also be specified.

- **secret** (*str*) – The secret for your app; application_id must also be specified.
- **token** (*str*) – OAUTH token for your app; application_id and secret must not be specified.
- **api_base** (*str*) – The base URL against which REST calls will be made. Default: https://api.planningcenteronline.com
- **timeout** (*int*) – How long to wait (seconds) for requests to timeout. Default 60.
- **upload_url** (*str*) – The URL to which files will be uploaded. Default: https://upload.planningcenteronline.com/v2/files
- **upload_timeout** (*int*) – How long to wait (seconds) for uploads to timeout. Default 300.
- **timeout_retries** (*int*) – How many times to retry requests that have timed out. Default 3.

delete(*url: str*, ***params*) → requests.models.Response

Perform a DELETE request against the PCO API.

Performs a fully managed DELETE request (handles ratelimiting, timeouts, etc.).

Parameters

- **url** (*str*) – The URL against which to perform the request. Can include what's been set as api_base, which will be ignored if this value is also present in your URL.
- **params** – Any named arguments will be passed as query parameters. Values must be of type str!

Raises

- `PCORequestTimeoutException` – The request to PCO timed out the maximum number of times.
- `PCOUnexpectedRequestException` – An unexpected error occurred when making your request.
- `PCORequestException` – The response from the PCO API indicated an error with your request.

Returns The response object returned by the API for this request. A successful delete request will return a response with an empty payload, so we return the response object here instead.

Return type requests.Response

get(*url: str*, ***params*) → Optional[dict]

Perform a GET request against the PCO API.

Performs a fully managed GET request (handles ratelimiting, timeouts, etc.).

Parameters

- **url** (*str*) – The URL against which to perform the request. Can include what's been set as api_base, which will be ignored if this value is also present in your URL.
- **params** – Any named arguments will be passed as query parameters.

Raises

- `PCORequestTimeoutException` – The request to PCO timed out the maximum number of times.
- `PCOUnexpectedRequestException` – An unexpected error occurred when making your request.

- `PCORequestException` – The response from the PCO API indicated an error with your request.

Returns The payload returned by the API for this request.

Return type dict

iterate(*url: str*, *offset: int = 0*, *per_page: int = 25*, ***params*) → Iterator[dict]
Iterate a list of objects in a response, handling pagination.

Basically, this function wraps get in a generator function designed for processing requests that will return multiple objects. Pagination is transparently handled.

Objects specified as includes will be injected into their associated object and returned.

Parameters

- **url** (*str*) – The URL against which to perform the request. Can include what's been set as api_base, which will be ignored if this value is also present in your URL.
- **offset** (*int*) – The offset at which to start. Usually going to be 0 (the default).
- **per_page** (*int*) – The number of results that should be requested in a single page. Valid values are 1 - 100, defaults to the PCO default of 25.
- **params** – Any additional named arguments will be passed as query parameters. Values must be of type str!

Raises

- `PCORequestTimeoutException` – The request to PCO timed out the maximum number of times.
- `PCOUnexpectedRequestException` – An unexpected error occurred when making your request.
- `PCORequestException` – The response from the PCO API indicated an error with your request.

Yields *dict* – Each object returned by the API for this request. Returns "data", "included", and "meta" nodes for each response. Note that data is processed somewhat before being returned from the API. Namely, includes are injected into the object(s) with which they are associated. This makes it easier to process includes associated with specific objects since they are accessible directly from each returned object.

patch(*url: str*, *payload: Optional[dict] = None*, ***params*) → Optional[dict]
Perform a PATCH request against the PCO API.

Performs a fully managed PATCH request (handles ratelimiting, timeouts, etc.).

Parameters

- **url** (*str*) – The URL against which to perform the request. Can include what's been set as api_base, which will be ignored if this value is also present in your URL.
- **payload** (*dict*) – The payload for the PUT request. Must be serializable to JSON!
- **params** – Any named arguments will be passed as query parameters. Values must be of type str!

Raises

- `PCORequestTimeoutException` – The request to PCO timed out the maximum number of times.

- `PCOUnexpectedRequestException` – An unexpected error occurred when making your request.
- `PCORequestException` – The response from the PCO API indicated an error with your request.

Returns The payload returned by the API for this request.

Return type dict

post(*url: str*, *payload: Optional[dict] = None*, ***params*) → Optional[dict]

Perform a POST request against the PCO API.

Performs a fully managed POST request (handles ratelimiting, timeouts, etc.).

Parameters

- **url** (*str*) – The URL against which to perform the request. Can include what's been set as api_base, which will be ignored if this value is also present in your URL.
- **payload** (*dict*) – The payload for the POST request. Must be serializable to JSON!
- **params** – Any named arguments will be passed as query parameters. Values must be of type str!

Raises

- `PCORequestTimeoutException` – The request to PCO timed out the maximum number of times.
- `PCOUnexpectedRequestException` – An unexpected error occurred when making your request.
- `PCORequestException` – The response from the PCO API indicated an error with your request.

Returns The payload returned by the API for this request.

Return type dict

request_json(*method: str*, *url: str*, *payload: Optional[Any] = None*, *upload: Optional[str] = None*, ***params*) → Optional[dict]

A generic entry point for making a managed request against PCO.

This function will return the payload from the PCO response (a dict).

Parameters

- **method** (*str*) – The HTTP method to use for this request.
- **url** (*str*) – The URL against which this request will be executed.
- **payload** (*obj*) – A json-serializable Python object to be sent as the post/put payload.
- **upload** (*str*) – The path to a file to upload.
- **params** (*obj*) – A dictionary or list of tuples or bytes to send in the query string.

Raises

- `PCORequestTimeoutException` – The request to PCO timed out the maximum number of times.
- `PCOUnexpectedRequestException` – An unexpected error occurred when making your request.
- `PCORequestException` – The response from the PCO API indicated an error with your request.

Returns The payload from the response to this request.

Return type dict

request_response(*method: str*, *url: str*, *payload: Optional[Any] = None*, *upload: Optional[str] = None*, ***params*) → requests.models.Response

A generic entry point for making a managed request against PCO.

This function will return a Requests response object, allowing access to all request data and metadata. Executed request could be one of the standard HTTP verbs or a file upload. If you're just looking for your data (json), use the request_json() function or get(), post(), etc.

Parameters

- **method** (*str*) – The HTTP method to use for this request.
- **url** (*str*) – The URL against which this request will be executed.
- **payload** (*obj*) – A json-serializable Python object to be sent as the post/put payload.
- **upload** (*str*) – The path to a file to upload.
- **params** (*obj*) – A dictionary or list of tuples or bytes to send in the query string.

Raises

- PCORequestTimeoutException – The request to PCO timed out the maximum number of times.
- PCOUnexpectedRequestException – An unexpected error occurred when making your request.
- PCORequestException – The response from the PCO API indicated an error with your request.

Returns The response to this request.

Return type requests.Response

static template(*object_type: str*, *attributes: Optional[dict] = None*) → dict

Get template JSON for creating a new object.

Parameters

- **object_type** (*str*) – The type of object to be created.
- **attributes** (*dict*) – The new objects attributes. Defaults to empty.

Returns A template from which to set the new object's attributes.

Return type dict

upload(*file_path: str*, ***params*) → Optional[dict]

Upload the file at the specified path to PCO.

Parameters

- **file_path** (*str*) – The path to the file to be uploaded to PCO.
- **params** – Any named arguments will be passed as query parameters. Values must be of type str!

Raises

- PCORequestTimeoutException – The request to PCO timed out the maximum number of times.

- `PCOUnexpectedRequestException` – An unexpected error occurred when making your request.
- `PCORequestException` – The response from the PCO API indicated an error with your request.

Returns The PCO response from the file upload.

Return type dict

pypco.user_auth_helpers module

User-facing authentication helper functions for pypco.

`pypco.user_auth_helpers.`**`get_browser_redirect_url`**(*client_id: str, redirect_uri: str, scopes: List[str]*) → str

Get the URL to which the user's browser should be redirected.

This helps you perform step 1 of PCO OAUTH as described at: https://developer.planning.center/docs/#/introduction/authentication

Parameters

- **`client_id`** (*str*) – The client id for your app.
- **`redirect_uri`** (*str*) – The redirect URI.
- **`scopes`** (*list*) – A list of the scopes to which you will authenticate (see above).

Returns The url to which a user's browser should be directed for OAUTH.

Return type str

`pypco.user_auth_helpers.`**`get_cc_org_token`**(*cc_name: Optional[str] = None*) → Optional[str]

Get a non-authenticated Church Center OrganizationToken.

Parameters **`cc_name`** (*str*) – The organization_name part of the organization_name.churchcenter.com url.

Raises:

Returns String of organization token

Return type str

`pypco.user_auth_helpers.`**`get_oauth_access_token`**(*client_id: str, client_secret: str, code: int, redirect_uri: str*) → dict

Get the access token for the client.

This assumes you have already completed steps 1 and 2 as described at: https://developer.planning.center/docs/#/introduction/authentication

Parameters

- **`client_id`** (*str*) – The client id for your app.
- **`client_secret`** (*str*) – The client secret for your app.
- **`code`** (*int*) – The code returned by step one of your OAUTH sequence.
- **`redirect_uri`** (*str*) – The redirect URI, identical to what was used in step 1.

Raises

- `PCORequestTimeoutException` – The request timed out.

- `PCOUnexpectedRequestException` – Something unexpected went wrong with the request.
- `PCORequestException` – The HTTP response from PCO indicated an error.

Returns The PCO response to your OAUTH request.

Return type dict

`pypco.user_auth_helpers.`**`get_oauth_refresh_token`**(*client_id: str*, *client_secret: str*, *refresh_token: str*) → dict

Refresh the access token.

This assumes you have already completed steps 1, 2, and 3 as described at: https://developer.planning.center/docs/#/introduction/authentication

Parameters

- **`client_id`** (*str*) – The client id for your app.
- **`client_secret`** (*str*) – The client secret for your app.
- **`refresh_token`** (*str*) – The refresh token for the user.

Raises

- `PCORequestTimeoutException` – The request timed out.
- `PCOUnexpectedRequestException` – Something unexpected went wrong with the request.
- `PCORequestException` – The HTTP response from PCO indicated an error.

Returns The PCO response to your token refresh request.

Return type dict

Module contents

A Pythonic Object-Oriented wrapper to the PCO API

pypco is a Python wrapper for the Planning Center Online (PCO) REST API intended to help you accomplish useful things with Python and the PCO API more quickly. pypco provides simple helpers wrapping the REST calls you'll place against the PCO API, meaning that you'll be spending your time directly in the PCO API docs rather than those specific to your API wrapper tool of choice.

usage:

```
>>> import pypco
>>> pco = pypco.PCO()
```

pypco supports both OAUTH and Personal Access Token (PAT) authentication.

CHAPTER 2

Indices and tables

- Index
- *Modules*
- Search

Python Module Index

p

Index