# CenDatHelper Documentation

Alpha 0.1

Your Name

August 11, 2025

## Contents

# 1 Introduction

The `CenDatHelper` library provides a high-level interface for exploring and retrieving data from the U.S. Census Bureau's API. It simplifies the process of discovering available datasets, geographies, and variables, and it provides a robust method for fetching data, handling complex geographic hierarchies automatically.

This document details the two main classes: `CenDatHelper`, which is used to build and execute queries, and `CenDatResponse`, which is a container for the data returned by a query and provides methods for easy conversion to popular data analysis libraries.

# 2 Installation and Dependencies

The library requires the following Python packages:

- `requests`

The `CenDatResponse` class has optional dependencies on `pandas` and `polars`. These are only required if you use the corresponding `to_pandas()` or `to_polars()` methods.

# 3 The `CenDatHelper` Class

This is the main class for interacting with the Census API.

## 3.1 `__init__`

Initializes the helper object.

```
def __init__(self, years=None, key=None):
```

**Parameters**:

- `years` (int or list[int], optional): The year or years of interest. Can be set later with `set_years()`.
- `key` (str, optional): Your Census API key. Can be loaded later with `load_key()`.

## 3.2 `set_years`

Sets the primary year or years for the query.

```
def set_years(self, years):
```

## 3.3 `load_key`

Loads your Census API key for authenticated requests.

```
def load_key(self, key=None):
```

## 3.4 `list_products`

Lists available data products, with options to filter by year and title patterns. The results are cached for use with `set_products()`.

```
def list_products(self, years=None, patterns=None, to_dicts=True, logic=all):
```

**Parameters**:

- `patterns` (str or list[str], optional): Regex patterns to filter products by title.
- `logic` (callable, optional): Use `all` (default) for AND logic or `any` for OR logic when applying multiple patterns.

### 3.5 set_products

Sets the active data product(s). Can be called without arguments to use the cached results from the last `list_products()` call.

```
def set_products(self, titles=None):
```

### 3.6 list_geos

Lists available geographies for the currently set products.

```
def list_geos(self, to_dicts=False, patterns=None, logic=all):
```

### 3.7 set_geos

Sets the active geography or geographies. When successful, it prints the required parent geographies for the 'within' parameter of `get_data()`.

```
def set_geos(self, sumlevs=None):
```

### 3.8 list_variables

Lists available variables for the set products.

```
def list_variables(self, to_dicts=True, patterns=None, logic=all):
```

### 3.9 set_variables

Sets the active variables. The success message confirms which variables have been set for each product and vintage.

```
def set_variables(self, names=None):
```

### 3.10 get_data

The primary method for retrieving data. It builds and executes all necessary API calls based on the set products, geos, and variables.

```
def get_data(self, within='us', max_workers=None):
```

**Parameters**:

- `within` (str, dict, or list[dict]): Defines the geographic scope.
  - `'us'` (default): Fetches data for all required geographies nationwide (for aggregate data).
  - `{'state': '06'}`: A dictionary specifying parent geographies.
  - A list of dictionaries for batch requests.
- `max_workers` (int, optional): The number of concurrent threads to use for API calls.

**Returns**: An instance of the `CenDatResponse` class.

## 4 The CenDatResponse Class

This class is a container for the results of a `get_data()` call and provides methods to transform the data.

## 4.1 to_polars

Converts the raw data into a list of Polars DataFrames. Each DataFrame corresponds to a parameter set from the query and includes context columns.

```
def to_polars(self) -> List["pl.DataFrame"]:
```

## 4.2 to_pandas

Converts the raw data into a list of Pandas DataFrames.

```
def to_pandas(self) -> List["pd.DataFrame"]:
```

# 5 Usage Examples

## 5.1 Example 1: Microdata (PUMS) Request

This example demonstrates a complete workflow for retrieving Public Use Microdata Sample (PUMS) data for specific geographic areas in Alabama and Arizona.

```
import sys
import polars as pl

sys.modules.pop("CensusData", None)
from CensusData import CenDatHelper
import os
from dotenv import load_dotenv

load_dotenv()

# Initialize the helper for a specific year and provide API key
cd = CenDatHelper(years=[2017], key=os.getenv("CENSUS_API_KEY"))

# --- Step 1: Find and select the desired data product ---
# Use patterns to find the 5-year ACS PUMS product for 2017
potential_products = cd.list_products(
    patterns=[
        "american community|acs",
        "public use micro|pums",
        "5-year",
        "^(?!.*puerto rico).*$",
    ]
)
# Call set_products() with no arguments to use the filtered results
cd.set_products()

# --- Step 2: Find and select the desired geography ---
# PUMS data uses 'public use microdata area'
cd.list_geos(to_dicts=True)
cd.set_geos("795")

# --- Step 3: Find and select variables ---
# Find variables related to income, weights, and location
cd.list_variables(
    to_dicts=True,
    patterns=["income", "person weight", "state", "public.*area"],
    logic=any,
)
cd.set_variables(["PUMA", "PWGTP", "HINCP", "ADJINC"])

# --- Step 4: Get the data ---
# Provide a list of dictionaries to `within` to make multiple
# specific geographic requests in one call.
response = cd.get_data(
    within=[
        {"state": "1", "public use microdata area": ["400", "2500"]},
        {"state": "4", "public use microdata area": "105"},
    ]
)
```

```
# --- Step 5: Convert to a DataFrame ---
# The response object can be converted to a list of Polars DataFrames
pums_data = response.to_polars()[0]
print(pums_data.head())
```

## 5.2 Example 2: Aggregate Data Request

This example shows how to retrieve aggregate data for a more complex geography that requires parent-level information (in this case, ‘place‘ requires ‘state‘).

```
import sys
import polars as pl

sys.modules.pop("CensusData", None)
from CensusData import CenDatHelper
import os
from dotenv import load_dotenv

load_dotenv()

# Initialize for multiple years
cdh = CenDatHelper(years=[2022, 2023], key=os.getenv("CENSUS_API_KEY"))

# --- Step 1: Find and select products ---
# Find the standard 5-year detailed tables, excluding special tables
potential_products = cdh.list_products(
    to_dicts=True,
    patterns=[
        "american community|acs",
        "5-year",
        "detailed",
        "^(?!.*(alaska|aian|selected)).*$",
    ],
)
cdh.set_products()

# --- Step 2: Find and select geography ---
# Set the geography to 'place'. The success message will inform us
# that this geography requires 'state' to be specified.
cdh.set_geos(["160"])

# --- Step 3: Find and select variables ---
potential_variables = cdh.list_variables(
    to_dicts=True, patterns=["total", "less.*high"]
)
cdh.set_variables(["B07009_002E", "B16010_009E"])

# --- Step 4: Get the data ---
# Provide a list of dictionaries to `within`. This will fetch data for
# two specific places in New York (state 36) and ALL places in
# California (state 06). The helper will automatically find all places
# in California for you.
response = cdh.get_data(
    max_workers=200,
```

```
    within=[
        {"state": "36", "place": ["61797", "61621"]},
        {"state": "06"},
    ],
)

# --- Step 5: Convert and combine DataFrames ---
# The result will be a list of DataFrames (one for each product/vintage).
# We can concatenate them into a single DataFrame for analysis.
final_df = pl.concat(response.to_polars())
print(final_df.head())
```