Census Geocoder

Release 0.1.0

Insight Industry Inc.

Sep 06, 2021

CONTENTS:

(Unofficial) Python Binding for the US Census Geocoder API

Version Compatibility

The **US Census Geocoder** is designed to be compatible with:

- Python 3.6 or higher

Branch	Unit Tests
latest	
v.0.5	
develop	

CHAPTER

ONE

QUICKSTART: PATTERNS AND BEST PRACTICES

1.1 Installation

To install the **US Census Geocoder**, just execute:

```
$ pip install census-geocoder
```

1.2 Importing the Library

Importing the **Census Geocoder** is very straightforward. You can either import its components precisely (see *API Reference*) or simply import the entire module:

```
# Import the entire module.
import census_geocoder as geocoder

result = geocoder.location.from_address('4600 Silver Hill Rd, Washington, DC 20233')
result = geocoder.geography.from_address('4600 Silver Hill Rd, Washington, DC 20233')

# Import precise components.
from census_geocoder import Location, Geography

result = Location.from_address('4600 Silver Hill Rd, Washington, DC 20233')
result = Geography.from_address('4600 Silver Hill Rd, Washington, DC 20233')
```

1.3 Getting Location Data

Retrieving data about canonical locations is very straightforward. You have four different ways to get this information, depending on what information you have about the location you want to geocode:

Single-line Address

Parametrized Address

Coordinates

Batch File

```
import census_geocoder as geocoder

result = geocoder.location.from_address('4600 Silver Hill Rd, Washington, DC 20233')
```

See also:

- *Location.from_address()*

```
import census_geocoder as geocoder

result = geocoder.location.from_address(street = '4600 Silver Hill Rd',
                                        city = 'Washington',
                                        state = 'DC',
                                        zip_code = '20233')
```

See also:

- *Location.from_address()*

```
import census_geocoder as geocoder

result = geocoder.location.from_coordinates(longitude = -76.92744,
                                            latitude = 38.845985)
```

See also:

- *Location.from_coordinates()*

```
import census_geocoder as geocoder

result = geocoder.location.from_batch(file_ = '/my-csv-file.csv')
```

Caution: The batch file indicated can have a maximum of 10,000 records.

Warning: While the Census Geocoder API supports CSV, TXT, XLSX, and DAT formats the **Census Geocoder** library only supports CSV and TXT formats so as to avoid dependency-bloat (read: Why rely on other libraries to read XLSX format data?).

See also:

- *Location.from_batch()*

1.4 Getting Geographical Area Data

Retrieving data about the geographic areas that contain a given location/place is just as straightforward as *getting location data*. In fact, the syntax is almost identical. Just swap out the word `'location'` for `'geography'` and you're done!

Here's how to do it:

Single-line Address

Parametrized Address

Coordinates

Batch File

```
import census_geocoder as geocoder

result = geocoder.geography.from_address('4600 Silver Hill Rd, Washington, DC 20233')
```

See also:

- *GeographicArea.from_address()*

```
import census_geocoder as geocoder

result = geocoder.geography.from_address(street = '4600 Silver Hill Rd',
                                         city = 'Washington',
                                         state = 'DC',
                                         zip_code = '20233')
```

See also:

- *GeographicArea.from_address()*

```
import census_geocoder as geocoder

result = geocoder.geography.from_coordinates(longitude = -76.92744,
                                             latitude = 38.845985)
```

See also:

- *GeographicArea.from_coordinates()*

```
import census_geocoder as geocoder

result = geocoder.geography.from_batch(file_ = '/my-csv-file.csv')
```

Caution: The batch file indicated can have a maximum of 10,000 records.

Warning: While the Census Geocoder API supports CSV, TXT, XLSX, and DAT formats the **Census Geocoder** library only supports CSV and TXT formats so as to avoid dependency-bloat (read: Why rely on other libraries to read XLSX format data?).

See also:

- *GeographicArea.from_batch()*

CHAPTER

TWO

USING THE US CENSUS GEOCODER

2.1 Introduction

2.1.1 What is Geocoding?

Hint: The act of determining a specific, canonical location based on some input data.

See also:

- *Forward Geocoding*
- *Reverse Geocoding*

What we typically know about a specific location or geographical area is fuzzy. We might know part of the address, or refer to the address with abbreviations, or describe a general area, etc. It's ambiguous, fuzzy, and unclear. That makes getting specific, canonical, and precise data about that geographic location challenging. Which is where the process of *geocoding* comes into play.

Geocoding is the process of getting a specific, precise, and canonical determination of a geographical location (a place or geographic feature) or of a geographical area (encompassing multiple places or geographic features).

A canonical determination of a geographical location or geographical area is defined by the meta-data that is returned for that location/area. Things like the canonical address, or various characteristics of the geographical area, etc. represent the "canonical" information about that location / area.

The process of geocoding returns exactly that kind of canonical / official / unambiguous meta-data about one or more geographical locations and areas based on a set of inputs. Some inputs may be expected to be imprecise or partial (e.g. addresses, typically used for *forward geocoding*) while others are expected to be precise but with incomplete information (e.g. longitude and latitude coordinates used in *reverse geocoding*).

2.1.2 Why the Census Geocoder?

Geocoding is used for many thing, but the Census Geocoder API in particular is meant to provide the US Census Bureau's canonical meta-data about identified locations and areas. This meta-data is then typically used when executing more in-depth analysis on data published by the US Census Bureau and other departments of the US federal and state governments.

Because the US government uses a very complicated and overlapping hierarchy of geographic areas, it is essential when working with US government data to start from the precise identification of the geographic areas and locations of interest.

But using the Census Geocoder API to get this information is non-trivial in its complexity. That's both because the API has limited documentation on the one hand, and because its syntax is non-pythonic and requires extensive familiarity with the internals of the (complicated) datasets that the US Census Bureau manages/publishes.

The **Census Geocoder** library is meant to simplify all of that, by providing an easy-to-use, batteries-included, pythonic wrapper around the Census Geocoder API.

2.1.3 Census Geocoder vs. Alternatives

While we're partial to the **US Census Geocoder** as our primary means of interacting with the Census Geocoder API, there are obviously alternatives for you to consider. Some might be better for your use specific use cases, so here's how we think about them:

Roll Your Own

Census Geocode

CensusBatchGeocoder

geocoder/geopy

The Census Geocoder API is a straightforward RESTful API. Which means that you can just execute your own HTTP requests against it, retrieve the JSON results, and work with the resulting data entirely yourself. This is what I did for years, until I got tired of repeating the same patterns over and over again, and decided to build the **Census Geocoder** instead.

For a super-simple use case, probably the most expedient way to do it. But of course, more robust use cases would require your own scaffolding with built-in retry-logic, object representation, error handling, etc. which becomes non-trivial.

Why not use a library with batteries included?

Tip: When to use it?

In practice, I find that rolling my own solution is great when it's an extremely simple use case, or a one-time operation (e.g. in a Jupyter Notebook) with no business logic to speak of. It's a "quick-and-dirty" solution, where I'm trading rapid implementation (yay!) for less flexibility/functionality (boo!).

Considering how easy the **Census Geocoder** is to use, however, I find that I never really roll my own scaffolding when working with the Census Geocoder API.

The Census Geocode library is fantastic, and it was what I had used before building the **Census Geocoder** library. However, it has a number of significant limitations when compared to the **US Census Geocoder**:

- Results are returned as-is from the Census Geocoder API. This means that:
  - Results are essentially JSON objects represented as `dict`, which makes interacting with them in Python a little more cumbersome (one has to navigate nested `dict` objects).
  - Property/field names are as in the original Census data. This means that if you do not have the documentation handy, it is hard to intuitively understand what the data represents.
- The library is licensed under GPL3, which may complicate or limit its utilization in commercial or closed-source software operating under different (non-GPL) licenses.
- The library requires you to remember / apply a lot of the internals of the Census Geocoder API as-is (e.g. benchmark vintages) which is complicated given the API's limited documentation.
- The library does not support custom *layers*, and only returns the default set of layers for any request.

The **Census Geocoder** explicitly addresses all of these concerns:

- The library uses native Python classes to represent results, providing a more pythonic syntax for interacting with those classes.
- Properties / fields have been renamed to more human-understandable names.
- The **Census Geocoder** is made available under the more flexible *MIT License*.
- The library streamlines the configuration of *benchmarks* and *vintages*, and provides extensive *documentation*.

- The library supports any and all layers supported by the Census Geocoder API.

Tip: When to use it?

Census Geocode has one advantage over the **US Census Geocoder**: It has a CLI.

I haven't found much use for a CLI in the work I've done with the Census Geocoder API, so have not implemented it in the **US Census Geocoder**. Might add it in the future, if there are enough feature requests for it.

Given the above, it may be worth using Census Geocode instead of the **Census Geocoder** if you expect to be using a CLI.

The CensusBatchGeocoder is a fantastic library produced by the team at the Los Angeles Times Data Desk. It is specifically designed to provide a fairly pythonic interface for doing bulk geocoding operations, with great pandas serialization / de-serialization support.

However, it does have a couple of limitations:

- **Stale / Unmaintained**? The library does not seem to have been updated since 2017, leading me to believe that it is stale and unmaintained. There are numerous open issues dating back to 2020, 2018, and 2017 that have seen no activity.
- **No benchmark/vintage/layer support**. The library does not support the configuration of *benchmarks*, *vintages*, or *layers*.
- **Limited error handling**. The library has somewhat limited error handling, judging by the issues that have been reported in the repository.
- **Optimized for bulk operations**. The design of the library has been optimized for geocoding in bulk, which makes transactional one-off requests cumbersome to execute.

The **Census Geocoder** is obviously fresh / maintained, and has explicitly implemented robust error handling, and support for *benchmarks*, *vintages*, and *layers*. It is also designed to support bulk operations *and* transactional one-off requests.

Tip: When to use it?

CensusBatchGeocoder has one advantage over the **US Census Geocoder**: It can serialize results to a pandas `DataFrame` seamlessly and simply.

This is a useful feature, and one that I have added/pinned for the **US Census Geocoder**. If there are enough requests / up-votes on the issue, I may extend the library with this support in the future.

Given all this, it may be worth using CensusBatchGeocoder instead of the **US Census Geocoder** if you expect to be doing a lot of bulk operations using the default benchmark/vintage/layers.

---

geocoder and geopy are two of my favorite geocoding libraries in the Python ecosystem. They are both inherently pythonic, elegant, easy to use, and support most of the major geocoding providers out there with a standardized / unified API.

So at first blush, one might think: Why not just use one of these great libraries to handle requests against the Census Geocoder API?

Well, the problem is that neither geocoder nor geopy supports the Census Geocoder API as a geocoding provider. So… you can't just use either of them if you specifically want US Census geocoding data.

Secondly, both the geocoder and geopy libraries are optimized around providing coordinates and feature information (e.g. matched address), which the Census Geocoder API results go beyond (and are not natively compatible with).

So really, if you want to interact with the Census Geocoder API, the **Census Geocoder** library is designed to do exactly that.

Tip: When to use them?

If you only need relatively simple, coordinate/feature-focused *forward* or *reverse* geocoding from a different provider than the US Census Bureau, and you specifically do not need data from the US Census Bureau.

2.2 Census Geocoder Features

- **Easy to adopt**. Just install and import the library, and you can be *forward geocoding* and *reverse geocoding* with just two lines of code.
- **Extensive documentation**. One of the main limitations of the Geocoder API is that its documentation is scattered across the different datasets released by the Census Bureau, making it hard to navigate and understand. We've tried to fix that.
- Location Search
  - Using Geographic Coordinates (reverse geocoding)
  - Using a One-line Address
  - Using a Parametrized Address
  - Using Batched Addresses
- Geography Search
  - Using Geographic Coordinates (reverse geocoding)
  - Using a One-line Address
  - Using a Parametrized Address
  - Using Batched Addresses
- Supports all available *benchmarks*, *vintages*, and *layers*.
- Simplified syntax for indicating benchmarks, vintages, and layers.
- No more hard to interpret field names. The library uses simplified (read: human understandable) names for location and geography properties.

2.3 Overview

2.3.1 How the Census Geocoder Works

The **Census Geocoder** works with the Census Geocoder API by providing a thin Python wrapper around the APIs functionality. Rather than having to construct your own HTTP requests against the API itself, you can instead work with Python objects and functions the way you normally would.

In other words, the process is very straightforward:

1. Install the **Census Geocoder** library. (see *here*)
2. Import the geocoder. (see *here*)
3. Geocode something - either *locations* or *geographies*. (see *here*)
4. Work with your geocoded *locations* or *geographical areas*. (see *here*)

And that's it! Once you've done the steps above, you can easily geocode one-off requests or batch many requests into a single transaction.

2.4 1. Installing the Census Geocoder

To install the **US Census Geocoder**, just execute:

```
$ pip install census-geocoder
```

2.4.1 Dependencies

- Validator-Collection v1.5.0 or higher
- Backoff-Utils v1.0.1 or higher
- Requests v2.26 or higher

2.5 2. Import the Census Geocoder

Importing the **Census Geocoder** is very straightforward. You can either import its components precisely (see *API Reference*) or simply import the entire module:

```
# Import the entire module.
import census_geocoder as geocoder

result = geocoder.location.from_address('4600 Silver Hill Rd, Washington, DC 20233')
result = geocoder.geography.from_address('4600 Silver Hill Rd, Washington, DC 20233')

# Import precise components.
from census_geocoder import Location, Geography

result = Location.from_address('4600 Silver Hill Rd, Washington, DC 20233')
result = Geography.from_address('4600 Silver Hill Rd, Washington, DC 20233')
```

2.6 3. Geocoding

Geocoding a location means to retrieve canonical meta-data about that location. Think of it as getting the "official" details for a given place. Using the **Census Geocoder**, you can geocode locations given:

- A single-line address (whole or partial)
- A *parametrized address* where you know its components parts
- A set of longitude and latitude coordinates
- A batch file in CSV or TXT format

However, the Census Geocoder API provides two different sets of meta-data for any canonical location:

- **Location Data**. Think of it as the canonical address for a given location/place.
- **Geographic Area Data**. Think of it as canonical information about the (different) areas that contain the given location/place.

Using the **Census Geocoder** library you can retrieve both types of information.

Hint: When retrieving geographic area data, you *also* get location data.

2.6.1 Getting Location Data

Retrieving data about canonical locations is very straightforward. You have four different ways to get this information, depending on what information you have about the location you want to geocode:

Single-line Address

Parametrized Address

Coordinates

Batch File

```
import census_geocoder as geocoder

result = geocoder.location.from_address('4600 Silver Hill Rd, Washington, DC 20233')
```

See also:

- *Location.from_address()*

```
import census_geocoder as geocoder

result = geocoder.location.from_address(street = '4600 Silver Hill Rd',
                                        city = 'Washington',
                                        state = 'DC',
                                        zip_code = '20233')
```

See also:

- *Location.from_address()*

```
import census_geocoder as geocoder

result = geocoder.location.from_coordinates(longitude = -76.92744,
                                            latitude = 38.845985)
```

See also:

- *Location.from_coordinates()*

```
import census_geocoder as geocoder

result = geocoder.location.from_batch(file_ = '/my-csv-file.csv')
```

Caution: The batch file indicated can have a maximum of 10,000 records.

Warning: While the Census Geocoder API supports CSV, TXT, XLSX, and DAT formats the **Census Geocoder** library only supports CSV and TXT formats so as to avoid dependency-bloat (read: Why rely on other libraries to read XLSX format data?).

See also:

- *Location.from_batch()*

2.6.2 Getting Geographic Area Data

Retrieving data about the geographic areas that contain a given location/place is just as straightforward as *getting location data*. In fact, the syntax is almost identical. Just swap out the word `'location'` for `'geography'` and you're done!

Here's how to do it:

Single-line Address

Parametrized Address

Coordinates

Batch File

```
import census_geocoder as geocoder

result = geocoder.geography.from_address('4600 Silver Hill Rd, Washington, DC 20233')
```

See also:

- *GeographicArea.from_address()*

```
import census_geocoder as geocoder

result = geocoder.geography.from_address(street = '4600 Silver Hill Rd',
                                         city = 'Washington',
```

(continues on next page)

(continued from previous page)

```
                                          state = 'DC',
                                          zip_code = '20233')
```

See also:

- *GeographicArea.from_address()*

```
import census_geocoder as geocoder

result = geocoder.geography.from_coordinates(longitude = -76.92744,
                                             latitude = 38.845985)
```

See also:

- *GeographicArea.from_coordinates()*

```
import census_geocoder as geocoder

result = geocoder.geography.from_batch(file_ = '/my-csv-file.csv')
```

Caution: The batch file indicated can have a maximum of 10,000 records.

Warning: While the Census Geocoder API supports CSV, TXT, XLSX, and DAT formats the **Census Geocoder** library only supports CSV and TXT formats so as to avoid dependency-bloat (read: Why rely on other libraries to read XLSX format data?).

See also:

- *GeographicArea.from_batch()*

2.6.3 Benchmarks and Vintages

The data returned by the Census Geocoder API is different from typical geocoding services, in that it is time-sensitive. A geocoding service like the Google Maps API or Here.com only cares about the *current* location. But the US Census Bureau's information is inherently linked to the statistical data collected by the US Census Bureau at particular moments in time.

Thus, when making requests against the Census Geocoder API you are always asking for geographic location data or geographic area data as of a particular date. You might think "geographies don't change", but in actuality they are constantly evolving. Congressional districts, school districts, town lines, county lines, street names, house numbers, etc. are all constantly evolving. And to ensure that the statistical data is tied to the locations properly, that alignment needs to be maintained through two key concepts:

- *Benchmarks*
- *Vintages*

The *benchmark* is the time period when geographic information was snapshotted for use / publication in the Census Geocoder API. This is typically done twice per year, and represents the "geographic definitions as of the time period indicated by the benchmark".

The *vintage* is the census or survey data that the geographies are linked to. Thus, the geographic identifiers or statistical data associated with locations or geographic areas within a given benchmark are *also* linked to a particular vintage

of census/survey data. Trying to use those identifiers or statistical data with a different vintage of data may produce inaccurate results.

The Census Geocoder API supports a variety of benchmarks and vintages, and they are unfortunately poorly documented and difficult to interpret. Therefore, the **Census Geocoder** has been designed to streamline and simplify their usage.

Vintages are only available for a given benchmark. The table below provides guidance on the vintages and benchmarks supported by the **Census Geocoder**:

	BENCHMARKS	
	Current	Census2020
VINTAGES	Current	Census2020
	Census2020	Census2010
	ACS2019	
	ACS2018	
	ACS2017	
	Census2010	

When using the **Census Geocoder**, you can supply the *benchmark* and *vintage* directly when executing your geocoding request:

Single-line Address

Parametrized Address

Coordinates

Batch File

```
import census_geocoder as geocoder

result = geocoder.location.from_address('4600 Silver Hill Rd, Washington, DC 20233',
                                        benchmark = 'Current',
                                        vintage = 'ACS2019')

result = geocoder.geography.from_address('4600 Silver Hill Rd, Washington, DC 20233',
                                         benchmark = 'Current',
                                         vintage = 'ACS2019')
```

See also:

- *Location.from_address()*
- *GeographicArea.from_address()*

```
import census_geocoder as geocoder

result = geocoder.location.from_address(street = '4600 Silver Hill Rd',
                                        city = 'Washington',
                                        state = 'DC',
                                        zip_code = '20233',
                                        benchmark = 'Current',
                                        vintage = 'ACS2019')

result = geocoder.geography.from_address(street = '4600 Silver Hill Rd',
                                         city = 'Washington',
```

(continues on next page)

(continued from previous page)

```
                                          state = 'DC',
                                          zip_code = '20233',
                                          benchmark = 'Current',
                                          vintage = 'ACS2019')
```

See also:

- *Location.from_address()*
- *GeographicArea.from_address()*

```
import census_geocoder as geocoder

result = geocoder.location.from_coordinates(longitude = -76.92744,
                                            latitude = 38.845985,
                                            benchmark = 'Current',
                                            vintage = 'ACS2019')

result = geocoder.geography.from_coordinates(longitude = -76.92744,
                                             latitude = 38.845985,
                                             benchmark = 'Current',
                                             vintage = 'ACS2019')
```

See also:

- *Location.from_coordinates()*
- *GeographicArea.from_coordinates()*

```
import census_geocoder as geocoder

result = geocoder.location.from_batch(file_ = '/my-csv-file.csv',
                                      benchmark = 'Current',
                                      vintage = 'ACS2019')

result = geocoder.geography.from_batch(file_ = '/my-csv-file.csv',
                                       benchmark = 'Current',
                                       vintage = 'ACS2019')
```

See also:

- *Location.from_batch()*
- *GeographicArea.from_batch()*

Hint: Several important things to be aware of when it comes to benchmarks and vintages in the **Census Geocoder** library:

Unless over-ridden by the `CENSUS_GEOCODER_BENCHMARK` or `CENSUS_GEOCODER_VINTAGE` environment variables, the benchmark and vintage default to `'Current'` and `'Current'` respectively.

The benchmark and vintage are case-insensitive. This means that you can supply `'Current'`, `'CURRENT'`, or `'current'` and it will all work the same.

If you want to set a different default benchmark or vintage, you can do so by setting `CENSUS_GEOCODER_BENCHMARK` and `CENSUS_GEOCODER_VINTAGE` environment variables to the defaults you want to use.

2.6.4 Layers

When working with the Census Geocoder API (particularly when *getting geographic area data*), you have the ability to control which *types* of geographic area get returned. These types of geographic area are called "*layers*".

An example of two different "layers" might be "State" and "County". These are two different types of geographic area, one of which (County) may be encompassed by the other (State). In general, geographic areas within the same layer cannot and do not overlap. However different layers can and *do* overlap, where one layer (State) may contain multiple other layers (Counties), or one layer (Metropolitan Statistical Areas) may partially overlap multiple entities within a different layer (States).

When using the **Census Geocoder** you can easily specify the layers of data that you want returned. Unless overridden by the `CENSUS_GEOCODER_LAYERS` environment variable, the layers returned will always default to `'all'`.

Which layers are available is ultimately determined by the *vintage* of the data you are retrieving. The following represents the list of layers available in each vintage:

Current

- 2010 Census Public Use Microdata Areas
- 2010 Census PUMAs
- 2010 PUMAs
- Census Public Use Microdata Areas
- Census PUMAs
- PUMAs
- 2020 Census ZIP Code Tabulation Areas
- 2020 Census ZCTAs
- Census ZCTAs
- ZCTAs
- Tribal Census Tracts
- Tribal Block Groups
- Census Tracts
- Census Block Groups
- 2020 Census Blocks
- Census Blocks
- Blocks
- Unified School Districts
- Secondary School Districts
- Elementary School Districts
- Estates
- County Subdivisions
- Subbarrios
- Consolidated Cities
- Incorporated Places

- Census Designated Places
- CDPs
- Alaska Native Regional Corporations
- Tribal Subdivisions
- Federal American Indian Reservations
- Off-Reservation Trust Lands
- State American Indian Reservations
- Hawaiian Home Lands
- Alaska Native Village Statistical Areas
- Oklahoma Tribal Statistical Areas
- State Designated Tribal Stastical Areas
- Tribal Designated Statistical Areas
- American Indian Joint-Use Areas
- 116th Congressional Districts
- Congressional Districts
- 2018 State Legislative Districts - Upper
- State Legislative Districts - Upper
- 2018 State Legislative Districts - Lower
- State Legislative Districts - Lower
- Census Divisions
- Divisions
- Census Regions
- Regions
- Combined New England City and Town Areas
- Combined NECTAs
- New England City and Town Area Divisions
- NECTA Divisions
- Metropolitan New England City and Town Areas
- Metropolitan NECTAs
- Micropolitan New England City and Town Areas
- Micropolitan NECTAs
- Combined Statistical Areas
- CSAs
- Metropolitan Divisions
- Metropolitan Statistical Areas
- Micropolitan Statistical Areas

- States
- Counties

Census2020

- Urban Growth Areas
- Tribal Census Tracts
- Tribal Block Groups
- Census Tracts
- Census Block Groups
- Block Groups
- Census Blocks
- Blocks
- Unified School Districts
- Secondary School Districts
- Elementary School Districts
- Estates
- County Subdivisions
- Subbarrios
- Consolidated Cities
- Incorporated Places
- Census Designated Places
- CDPs
- Alaska Native Regional Corporations
- Tribal Subdivisions
- Federal American Indian Reservations
- Off-Reservation Trust Lands
- State American Indian Reservations
- Hawaiian Home Lands
- Alaska Native Village Statistical Areas
- Oklahoma Tribal Statistical Areas
- State Designated Tribal Stastical Areas
- Tribal Designated Statistical Areas
- American Indian Joint-Use Areas
- 116th Congressional Districts
- Congressional Districts
- 2018 State Legislative Districts - Upper
- State Legislative Districts - Upper

- 2018 State Legislative Districts - Lower
- State Legislative Districts - Lower
- Voting Districts
- Census Divisions
- Divisions
- Census Regions
- Regions
- Combined New England City and Town Areas
- Combined NECTAs
- New England City and Town Area Divisions
- NECTA Divisions
- Metropolitan New England City and Town Areas
- Metropolitan NECTAs
- Micropolitan New England City and Town Areas
- Micropolitan NECTAs
- Combined Statistical Areas
- CSAs
- Metropolitan Divisions
- Metropolitan Statistical Areas
- Micropolitan Statistical Areas
- States
- Counties
- Zip Code Tabulation Areas
- ZCTAs

ACS2019

- 2010 Census Public Use Microdata Areas
- 2010 Census PUMAs
- 2010 PUMAs
- Census Public Use Microdata Areas
- Census PUMAs
- PUMAs
- 2010 Census ZIP Code Tabulation Areas
- 2010 Census ZCTAs
- Census ZCTAs
- ZCTAs
- Tribal Census Tracts

- Tribal Block Groups
- Census Tracts
- Census Block Groups
- Unified School Districts
- Secondary School Districts
- Elementary School Districts
- Estates
- County Subdivisions
- Subbarrios
- Consolidated Cities
- Incorporated Places
- Census Designated Places
- CDPs
- Alaska Native Regional Corporations
- Tribal Subdivisions
- Federal American Indian Reservations
- Off-Reservation Trust Lands
- State American Indian Reservations
- Hawaiian Home Lands
- Alaska Native Village Statistical Areas
- Oklahoma Tribal Statistical Areas
- State Designated Tribal Stastical Areas
- Tribal Designated Statistical Areas
- American Indian Joint-Use Areas
- 116th Congressional Districts
- Congressional Districts
- 2018 State Legislative Districts - Upper
- State Legislative Districts - Upper
- 2018 State Legislative Districts - Lower
- State Legislative Districts - Lower
- Census Divisions
- Divisions
- Census Regions
- Regions
- 2010 Census Urbanized Areas
- Census Urbanized Areas

- Urbanized Areas
- 2010 Census Urban Clusters
- Census Urban Clusters
- Urban Clusters
- Combined New England City and Town Areas
- Combined NECTAs
- New England City and Town Area Divisions
- NECTA Divisions
- Metropolitan New England City and Town Areas
- Metropolitan NECTAs
- Micropolitan New England City and Town Areas
- Micropolitan NECTAs
- Combined Statistical Areas
- CSAs
- Metropolitan Divisions
- Metropolitan Statistical Areas
- Micropolitan Statistical Areas
- States
- Counties

ACS2018

- 2010 Census Public Use Microdata Areas
- 2010 Census PUMAs
- 2010 PUMAs
- Census Public Use Microdata Areas
- Census PUMAs
- PUMAs
- 2010 Census ZIP Code Tabulation Areas
- 2010 Census ZCTAs
- Census ZCTAs
- ZCTAs
- Tribal Census Tracts
- Tribal Block Groups
- Census Tracts
- Census Block Groups
- Unified School Districts
- Secondary School Districts

- Elementary School Districts
- Estates
- County Subdivisions
- Subbarrios
- Consolidated Cities
- Incorporated Places
- Census Designated Places
- CDPs
- Alaska Native Regional Corporations
- Tribal Subdivisions
- Federal American Indian Reservations
- Off-Reservation Trust Lands
- State American Indian Reservations
- Hawaiian Home Lands
- Alaska Native Village Statistical Areas
- Oklahoma Tribal Statistical Areas
- State Designated Tribal Stastical Areas
- Tribal Designated Statistical Areas
- American Indian Joint-Use Areas
- 116th Congressional Districts
- Congressional Districts
- 2018 State Legislative Districts - Upper
- State Legislative Districts - Upper
- 2018 State Legislative Districts - Lower
- State Legislative Districts - Lower
- Census Divisions
- Divisions
- Census Regions
- Regions
- 2010 Census Urbanized Areas
- Census Urbanized Areas
- Urbanized Areas
- 2010 Census Urban Clusters
- Census Urban Clusters
- Urban Clusters
- Combined New England City and Town Areas

- Combined NECTAs
- New England City and Town Area Divisions
- NECTA Divisions
- Metropolitan New England City and Town Areas
- Metropolitan NECTAs
- Micropolitan New England City and Town Areas
- Micropolitan NECTAs
- Combined Statistical Areas
- CSAs
- Metropolitan Divisions
- Metropolitan Statistical Areas
- Micropolitan Statistical Areas
- States
- Counties

ACS2017

- 2010 Census Public Use Microdata Areas
- 2010 Census PUMAs
- 2010 PUMAs
- Census Public Use Microdata Areas
- Census PUMAs
- PUMAs
- 2010 Census ZIP Code Tabulation Areas
- 2010 Census ZCTAs
- Census ZCTAs
- ZCTAs
- Tribal Census Tracts
- Tribal Block Groups
- Census Tracts
- Census Block Groups
- Unified School Districts
- Secondary School Districts
- Elementary School Districts
- Estates
- County Subdivisions
- Subbarrios
- Consolidated Cities

- Incorporated Places
- Census Designated Places
- CDPs
- Alaska Native Regional Corporations
- Tribal Subdivisions
- Federal American Indian Reservations
- Off-Reservation Trust Lands
- State American Indian Reservations
- Hawaiian Home Lands
- Alaska Native Village Statistical Areas
- Oklahoma Tribal Statistical Areas
- State Designated Tribal Stastical Areas
- Tribal Designated Statistical Areas
- American Indian Joint-Use Areas
- 115th Congressional Districts
- Congressional Districts
- 2016 State Legislative Districts - Upper
- State Legislative Districts - Upper
- 2016 State Legislative Districts - Lower
- State Legislative Districts - Lower
- Census Divisions
- Divisions
- Census Regions
- Regions
- 2010 Census Urbanized Areas
- Census Urbanized Areas
- Urbanized Areas
- 2010 Census Urban Clusters
- Census Urban Clusters
- Urban Clusters
- Combined New England City and Town Areas
- Combined NECTAs
- New England City and Town Area Divisions
- NECTA Divisions
- Metropolitan New England City and Town Areas
- Metropolitan NECTAs

- Micropolitan New England City and Town Areas
- Micropolitan NECTAs
- Combined Statistical Areas
- CSAs
- Metropolitan Divisions
- Metropolitan Statistical Areas
- Micropolitan Statistical Areas
- States
- Counties

Census2010

- Public Use Microdata Areas
- PUMAs
- Traffic Analysis Districts
- TADs
- Traffic Analysis Zones
- TAZs
- Urban Growth Areas
- ZIP Code Tabulation Areas
- Zip Code Tabulation Areas
- ZCTAs
- Tribal Census Tracts
- Tribal Block Groups
- Census Tracts
- Census Block Groups
- Census Blocks
- Blocks
- Unified School Districts
- Secondary School Districts
- Elementary School Districts
- Estates
- County Subdivisions
- Subbarrios
- Consolidated Cities
- Incorporated Places
- Census Designated Places
- CDPs

- Alaska Native Regional Corporations
- Tribal Subdivisions
- Federal American Indian Reservations
- Off-Reservation Trust Lands
- State American Indian Reservations
- Hawaiian Home Lands
- Alaska Native Village Statistical Areas
- Oklahoma Tribal Statistical Areas
- State Designated Tribal Stastical Areas
- Tribal Designated Statistical Areas
- American Indian Joint-Use Areas
- 113th Congressional Districts
- 111th Congressional Districts
- 2012 State Legislative Districts - Upper
- 2012 State Legislative Districts - Lower
- 2010 State Legislative Districts - Upper
- 2010 State Legislative Districts - Lower
- Voting Districts
- Census Divisions
- Divisions
- Census Regions
- Regions
- Urbanized Areas
- Urban Clusters
- Combined New England City and Town Areas
- Combined NECTAs
- New England City and Town Area Divisions
- NECTA Divisions
- Metropolitan New England City and Town Areas
- Metropolitan NECTAs
- Micropolitan New England City and Town Areas
- Micropolitan NECTAs
- Combined Statistical Areas
- CSAs
- Metropolitan Divisions
- Metropolitan Statistical Areas

- Micropolitan Statistical Areas
- States
- Counties

Note: You may notice that there are (logical) duplicate layers in the lists above, for example "2010 Census PUMAs" and "2010 Census Public Use Microdata Areas". This is because there are multiple ways that users of Census data may refer to particular layers in their work. This duplication is purely for the convenience of **Census Geocoder** users, since the Census Geocoder API actually uses numerical identifiers for the layers returned.

When geocoding data, you can simply supply the layers you want using the `layers` keyword argument as below:

Single-line Address

Parametrized Address

Coordinates

Batch File

```
import census_geocoder as geocoder

result = geocoder.location.from_address('4600 Silver Hill Rd, Washington, DC 20233',
                                        benchmark = 'Current',
                                        vintage = 'ACS2019',
                                        layers = 'Census Tracts, States, CDPs, Divisions
↪')

result = geocoder.geography.from_address('4600 Silver Hill Rd, Washington, DC 20233',
                                         benchmark = 'Current',
                                         vintage = 'ACS2019',
                                         layers = 'Census Tracts, States, CDPs, Divisions
↪')
```

See also:

- *Location.from_address()*
- *GeographicArea.from_address()*

```
import census_geocoder as geocoder

result = geocoder.location.from_address(street = '4600 Silver Hill Rd',
                                        city = 'Washington',
                                        state = 'DC',
                                        zip_code = '20233',
                                        benchmark = 'Current',
                                        vintage = 'ACS2019',
                                        layers = 'Census Tracts, States, CDPs, Divisions
↪')

result = geocoder.geography.from_address(street = '4600 Silver Hill Rd',
                                         city = 'Washington',
                                         state = 'DC',
                                         zip_code = '20233',
                                         benchmark = 'Current',
```

(continues on next page)

(continued from previous page)

```
                                          vintage = 'ACS2019',
                                          layers = 'Census Tracts, States, CDPs, Divisions
↪')
```

See also:

- *Location.from_address()*
- *GeographicArea.from_address()*

```
import census_geocoder as geocoder

result = geocoder.location.from_coordinates(longitude = -76.92744,
                                            latitude = 38.845985,
                                            benchmark = 'Current',
                                            vintage = 'ACS2019',
                                            layers = 'Census Tracts, States, CDPs,␣
↪Divisions')

result = geocoder.geography.from_coordinates(longitude = -76.92744,
                                             latitude = 38.845985,
                                             benchmark = 'Current',
                                             vintage = 'ACS2019',
                                             layers = 'Census Tracts, States, CDPs,␣
↪Divisions')
```

See also:

- *Location.from_coordinates()*
- *GeographicArea.from_coordinates()*

```
import census_geocoder as geocoder

result = geocoder.location.from_batch(file_ = '/my-csv-file.csv',
                                      benchmark = 'Current',
                                      vintage = 'ACS2019')

result = geocoder.geography.from_batch(file_ = '/my-csv-file.csv',
                                       benchmark = 'Current',
                                       vintage = 'ACS2019',
                                       layers = 'Census Tracts, States, CDPs, Divisions')
```

See also:

- *Location.from_batch()*
- *GeographicArea.from_batch()*

Hint: When using the **Census Geocoder** to return geographic area data, you can request multiple layers worth of data by passing them in a comma-delimited string. This will return separate data for each layer indicated. The comma-delimited string can include white-space for easy readability, which means that the following two values are considered identical:

- `layers = 'Census Tracts, States, CDPs, Divisions'`
- `layers = 'Census Tracts,States,CDPs,Divisions'`

To retrieve all available layers that have data for a given location, you can submit `'all'`. Unless you have set the `CENSUS_GEOCODER_LAYERS` environment variable to a different value, `'all'` is the default set of layers that will be returned.

Note that layer names in the **Census Geocoder** are case-insensitive.

2.7 4. Working with Results

Locations vs Geographical Areas?

If all geographical area data is contained within a *`Location`*, why differentiate between *working with location data* and *working with geographical area data* at all?

The answer is two-fold: use case and performance. The act of geocoding is very simple and occurs at the level of a given *`Location`*. This process is done as soon as the Census Geocoder API has determined a canonical location (a *`MatchedAddress`*). Typically, use cases that need that geocoded canonical address require it to be very fast, and that's how the Census Geocoder API has been optimized.

However, pulling geographical area data *relies* on first determining the canonical location. And then, it has to pull a set of additional geographical area meta-data for that canonical location's geographical surroundings. That takes time, and the more *layers* you request, the longer that process will take.

Therefore, both the Census Geocoder API and the **Census Geocoder** library differentiate between the two so that you can use the more-performant location-only API calls when appropriate, and the less-performant but more robust geographical area API calls as needed.

Now that you've geocoded some data using the **Census Geocoder**, you probably want to work with your data. Well, that's pretty easy since the **Census Geocoder** returns native Python objects containing your location or geographical area data.

2.7.1 Shared Methods

Most of what you will do with your results is read properties from them so as to consume or use the canonical location/geographic meta-data in your application. However, there are a number of methods that are shared between both location data and geographic area data that may prove helpful:

`inspect`(*as_census_fields=False*)

> **Parameters** **`as_census_fields`** (`bool`) – If `True`, returns the properties using the Census field name rather than the **Census Geocoder** (user-friendly) property name. Defaults to `False`.
>
> Returns a list of the properties that are populated with values in the object.
>
> **Return type** `list` of `str`

`to_dict`()

> Serializes the data for the location/geographic area into a `dict` that conforms directly to the output from the Census Geocoder API.
>
> **Return type** `dict`

`to_json()`
Serializes the data for the location/geographic area into a `str` containing a JSON object that conforms directly to the output from the Census Geocoder API.

Return type `str`

2.7.2 Location Data

When working with location data, there are two principle sets of meta-data made available:

- **Input**. This is the input that was submitted to the Census Geocoder API, and it includes:
  - The address that you submitted.
  - The *benchmark* requested.
  - The *vintage* requested.
- **Matched Addresses**. This is a collection of addresses that the Census Geocoder API returned as the canonical addresses for your inputs.

Each matched address exposes its key meta-data, including:

- The address components in a term:*parametrized* <*parametrized address*> form.
- The address in a single-line form.
- The *Tigerline* identifier information for the address.
- The side of the street where the address can be found, per the *Tigerline* data.

See also:

- *Location*
- *MatchedAddress*

2.7.3 Geographical Area Data

Geographical area data is always returned within the context of a *MatchedAddress* instance, which itself is always contained within a *Location* instance. That matched address will have a `.geographies` property, which will contain a *GeographyCollection*. That `.geographies` property is what contains the detailed geographical area meta-data for all geographical areas returned in response to your API request.

Each *layer* requested is contained in a property of the *GeographyCollection*. For example, the relevant regions would be contained in the `.regions` property, while the relevant census tracts would be contained in the `.tracts` property.

See also:

For a full list of the properties/layers that are available within a *GeographyCollection*, please see the detailed API reference:

- *GeographyCollection*

If a *layer* is not requested (or is irrelevant for a given *benchmark* / *vintage*), then its corresponding property in the *GeographyCollection* will be `None`.

Within each layer/property, you will find a collection of *Geography* instances (technically, layer-specific sub-class instances). Each of these instances represents a geographical area returned by the Census Geocoder API, and their properties will contain the meta-data returned by that API.

Because different types of geographical area return different meta-data, there is a useful *.inspect()* method that will tell you what meta-data properties are available / have data.

The most universal properties (and the ones that are going to prove most useful when working with other Census Bureau datasets) are:

- *.geoid* which contains the GEOID (unique consolidated identifier for the geographical area)
- *.name* which contains the human-readable name of the geographical area
- *.geography_type* which contains a human-readable label for the instances's geographical area/layer type
- *.functional_status* which contains a human-readable indication of the geographical area's functional status

See also:

- *GeographyCollection*
- *Geography*

CHAPTER

THREE

GEOGRAPHIES IN THE CENSUS GEOCODER

3.1 Introduction

We like to think that geography is simple. There's a place, and that place has some borders, and it's all easy to understand. Intuitive, right?

Wrong.

Geography is actually extremely complicated, because it is by its very nature ambiguous. The only objectively unambiguous definition of a geographic area is a pair of longitude/latitude coordinates. When you start considering ways in which geographic areas overlap or roll into a hierarchy, it gets even more complicated because then you need to consider how each geographic area gets defined and overlaps.

Then, when you consider how such geographic hierarchies map to data (which itself represents a point-in-time), it gets even more complicated. That's because geographic definitions change all the time. Street names change, town names change, borders shift, etc.

And the Census Geocoder API and the US Census Bureau data that it corresponds to has to inherently account for all of these complexities. Which makes the way the Census Geocoder API handles geographic areas complicated.

3.2 Benchmarks, Vintages, and Layers

3.2.1 Benchmarks and Vintages

The data returned by the Census Geocoder API is different from typical geocoding services, in that it is time-sensitive. A geocoding service like the Google Maps API or Here.com only cares about the *current* location. But the US Census Bureau's information is inherently linked to the statistical data collected by the US Census Bureau at particular moments in time.

Thus, when making requests against the Census Geocoder API you are always asking for geographic location data or geographic area data as of a particular date. You might think "geographies don't change", but in actuality they are constantly evolving. Congressional districts, school districts, town lines, county lines, street names, house numbers, etc. are all constantly evolving. And to ensure that the statistical data is tied to the locations properly, that alignment needs to be maintained through two key concepts:

- *Benchmarks*
- *Vintages*

The *benchmark* is the time period when geographic information was snapshotted for use / publication in the Census Geocoder API. This is typically done twice per year, and represents the "geographic definitions as of the time period indicated by the benchmark".

The *vintage* is the census or survey data that the geographies are linked to. Thus, the geographic identifiers or statistical data associated with locations or geographic areas within a given benchmark are *also* linked to a particular vintage of census/survey data. Trying to use those identifiers or statistical data with a different vintage of data may produce inaccurate results.

The Census Geocoder API supports a variety of benchmarks and vintages, and they are unfortunately poorly documented and difficult to interpret. Therefore, the **Census Geocoder** has been designed to streamline and simplify their usage.

Vintages are only available for a given benchmark. The table below provides guidance on the vintages and benchmarks supported by the **Census Geocoder**:

	BENCHMARKS	
	Current	Census2020
VINTAGES	Current	Census2020
	Census2020	Census2010
	ACS2019	
	ACS2018	
	ACS2017	
	Census2010	

When using the **Census Geocoder**, you can supply the *benchmark* and *vintage* directly when executing your geocoding request:

Single-line Address

Parametrized Address

Coordinates

Batch File

```
import census_geocoder as geocoder
```

(continues on next page)

(continued from previous page)

```
result = geocoder.location.from_address('4600 Silver Hill Rd, Washington, DC 20233',
                                        benchmark = 'Current',
                                        vintage = 'ACS2019')

result = geocoder.geography.from_address('4600 Silver Hill Rd, Washington, DC 20233',
                                         benchmark = 'Current',
                                         vintage = 'ACS2019')
```

See also:

- *Location.from_address()*
- *GeographicArea.from_address()*

```
import census_geocoder as geocoder

result = geocoder.location.from_address(street = '4600 Silver Hill Rd',
                                        city = 'Washington',
                                        state = 'DC',
                                        zip_code = '20233',
                                        benchmark = 'Current',
                                        vintage = 'ACS2019')

result = geocoder.geography.from_address(street = '4600 Silver Hill Rd',
                                         city = 'Washington',
                                         state = 'DC',
                                         zip_code = '20233',
                                         benchmark = 'Current',
                                         vintage = 'ACS2019')
```

See also:

- *Location.from_address()*
- *GeographicArea.from_address()*

```
import census_geocoder as geocoder

result = geocoder.location.from_coordinates(longitude = -76.92744,
                                            latitude = 38.845985,
                                            benchmark = 'Current',
                                            vintage = 'ACS2019')

result = geocoder.geography.from_coordinates(longitude = -76.92744,
                                             latitude = 38.845985,
                                             benchmark = 'Current',
                                             vintage = 'ACS2019')
```

See also:

- *Location.from_coordinates()*
- *GeographicArea.from_coordinates()*

```
import census_geocoder as geocoder

```

(continues on next page)

(continued from previous page)

```
result = geocoder.location.from_batch(file_ = '/my-csv-file.csv',
                                      benchmark = 'Current',
                                      vintage = 'ACS2019')

result = geocoder.geography.from_batch(file_ = '/my-csv-file.csv',
                                       benchmark = 'Current',
                                       vintage = 'ACS2019')
```

See also:

- *Location.from_batch()*
- *GeographicArea.from_batch()*

Hint: Several important things to be aware of when it comes to benchmarks and vintages in the **Census Geocoder** library:

Unless over-ridden by the `CENSUS_GEOCODER_BENCHMARK` or `CENSUS_GEOCODER_VINTAGE` environment variables, the benchmark and vintage default to `'Current'` and `'Current'` respectively.

The benchmark and vintage are case-insensitive. This means that you can supply `'Current'`, `'CURRENT'`, or `'current'` and it will all work the same.

If you want to set a different default benchmark or vintage, you can do so by setting `CENSUS_GEOCODER_BENCHMARK` and `CENSUS_GEOCODER_VINTAGE` environment variables to the defaults you want to use.

3.2.2 Layers

When working with the Census Geocoder API (particularly when *getting geographic area data*), you have the ability to control which *types* of geographic area get returned. These types of geographic area are called "*layers*".

An example of two different "layers" might be "State" and "County". These are two different types of geographic area, one of which (County) may be encompassed by the other (State). In general, geographic areas within the same layer cannot and do not overlap. However different layers can and *do* overlap, where one layer (State) may contain multiple other layers (Counties), or one layer (Metropolitan Statistical Areas) may partially overlap multiple entities within a different layer (States).

When using the **Census Geocoder** you can easily specify the layers of data that you want returned. Unless overridden by the `CENSUS_GEOCODER_LAYERS` environment variable, the layers returned will always default to `'all'`.

Which layers are available is ultimately determined by the *vintage* of the data you are retrieving. The following represents the list of layers available in each vintage:

Current

- 2010 Census Public Use Microdata Areas
- 2010 Census PUMAs
- 2010 PUMAs
- Census Public Use Microdata Areas
- Census PUMAs
- PUMAs
- 2020 Census ZIP Code Tabulation Areas

- 2020 Census ZCTAs
- Census ZCTAs
- ZCTAs
- Tribal Census Tracts
- Tribal Block Groups
- Census Tracts
- Census Block Groups
- 2020 Census Blocks
- Census Blocks
- Blocks
- Unified School Districts
- Secondary School Districts
- Elementary School Districts
- Estates
- County Subdivisions
- Subbarrios
- Consolidated Cities
- Incorporated Places
- Census Designated Places
- CDPs
- Alaska Native Regional Corporations
- Tribal Subdivisions
- Federal American Indian Reservations
- Off-Reservation Trust Lands
- State American Indian Reservations
- Hawaiian Home Lands
- Alaska Native Village Statistical Areas
- Oklahoma Tribal Statistical Areas
- State Designated Tribal Stastical Areas
- Tribal Designated Statistical Areas
- American Indian Joint-Use Areas
- 116th Congressional Districts
- Congressional Districts
- 2018 State Legislative Districts - Upper
- State Legislative Districts - Upper
- 2018 State Legislative Districts - Lower

- State Legislative Districts - Lower
- Census Divisions
- Divisions
- Census Regions
- Regions
- Combined New England City and Town Areas
- Combined NECTAs
- New England City and Town Area Divisions
- NECTA Divisions
- Metropolitan New England City and Town Areas
- Metropolitan NECTAs
- Micropolitan New England City and Town Areas
- Micropolitan NECTAs
- Combined Statistical Areas
- CSAs
- Metropolitan Divisions
- Metropolitan Statistical Areas
- Micropolitan Statistical Areas
- States
- Counties

Census2020

- Urban Growth Areas
- Tribal Census Tracts
- Tribal Block Groups
- Census Tracts
- Census Block Groups
- Block Groups
- Census Blocks
- Blocks
- Unified School Districts
- Secondary School Districts
- Elementary School Districts
- Estates
- County Subdivisions
- Subbarrios
- Consolidated Cities

- Incorporated Places
- Census Designated Places
- CDPs
- Alaska Native Regional Corporations
- Tribal Subdivisions
- Federal American Indian Reservations
- Off-Reservation Trust Lands
- State American Indian Reservations
- Hawaiian Home Lands
- Alaska Native Village Statistical Areas
- Oklahoma Tribal Statistical Areas
- State Designated Tribal Stastical Areas
- Tribal Designated Statistical Areas
- American Indian Joint-Use Areas
- 116th Congressional Districts
- Congressional Districts
- 2018 State Legislative Districts - Upper
- State Legislative Districts - Upper
- 2018 State Legislative Districts - Lower
- State Legislative Districts - Lower
- Voting Districts
- Census Divisions
- Divisions
- Census Regions
- Regions
- Combined New England City and Town Areas
- Combined NECTAs
- New England City and Town Area Divisions
- NECTA Divisions
- Metropolitan New England City and Town Areas
- Metropolitan NECTAs
- Micropolitan New England City and Town Areas
- Micropolitan NECTAs
- Combined Statistical Areas
- CSAs
- Metropolitan Divisions

- Metropolitan Statistical Areas
- Micropolitan Statistical Areas
- States
- Counties
- Zip Code Tabulation Areas
- ZCTAs

ACS2019

- 2010 Census Public Use Microdata Areas
- 2010 Census PUMAs
- 2010 PUMAs
- Census Public Use Microdata Areas
- Census PUMAs
- PUMAs
- 2010 Census ZIP Code Tabulation Areas
- 2010 Census ZCTAs
- Census ZCTAs
- ZCTAs
- Tribal Census Tracts
- Tribal Block Groups
- Census Tracts
- Census Block Groups
- Unified School Districts
- Secondary School Districts
- Elementary School Districts
- Estates
- County Subdivisions
- Subbarrios
- Consolidated Cities
- Incorporated Places
- Census Designated Places
- CDPs
- Alaska Native Regional Corporations
- Tribal Subdivisions
- Federal American Indian Reservations
- Off-Reservation Trust Lands
- State American Indian Reservations

- Hawaiian Home Lands
- Alaska Native Village Statistical Areas
- Oklahoma Tribal Statistical Areas
- State Designated Tribal Stastical Areas
- Tribal Designated Statistical Areas
- American Indian Joint-Use Areas
- 116th Congressional Districts
- Congressional Districts
- 2018 State Legislative Districts - Upper
- State Legislative Districts - Upper
- 2018 State Legislative Districts - Lower
- State Legislative Districts - Lower
- Census Divisions
- Divisions
- Census Regions
- Regions
- 2010 Census Urbanized Areas
- Census Urbanized Areas
- Urbanized Areas
- 2010 Census Urban Clusters
- Census Urban Clusters
- Urban Clusters
- Combined New England City and Town Areas
- Combined NECTAs
- New England City and Town Area Divisions
- NECTA Divisions
- Metropolitan New England City and Town Areas
- Metropolitan NECTAs
- Micropolitan New England City and Town Areas
- Micropolitan NECTAs
- Combined Statistical Areas
- CSAs
- Metropolitan Divisions
- Metropolitan Statistical Areas
- Micropolitan Statistical Areas
- States

- Counties

ACS2018

- 2010 Census Public Use Microdata Areas
- 2010 Census PUMAs
- 2010 PUMAs
- Census Public Use Microdata Areas
- Census PUMAs
- PUMAs
- 2010 Census ZIP Code Tabulation Areas
- 2010 Census ZCTAs
- Census ZCTAs
- ZCTAs
- Tribal Census Tracts
- Tribal Block Groups
- Census Tracts
- Census Block Groups
- Unified School Districts
- Secondary School Districts
- Elementary School Districts
- Estates
- County Subdivisions
- Subbarrios
- Consolidated Cities
- Incorporated Places
- Census Designated Places
- CDPs
- Alaska Native Regional Corporations
- Tribal Subdivisions
- Federal American Indian Reservations
- Off-Reservation Trust Lands
- State American Indian Reservations
- Hawaiian Home Lands
- Alaska Native Village Statistical Areas
- Oklahoma Tribal Statistical Areas
- State Designated Tribal Stastical Areas
- Tribal Designated Statistical Areas

- American Indian Joint-Use Areas
- 116th Congressional Districts
- Congressional Districts
- 2018 State Legislative Districts - Upper
- State Legislative Districts - Upper
- 2018 State Legislative Districts - Lower
- State Legislative Districts - Lower
- Census Divisions
- Divisions
- Census Regions
- Regions
- 2010 Census Urbanized Areas
- Census Urbanized Areas
- Urbanized Areas
- 2010 Census Urban Clusters
- Census Urban Clusters
- Urban Clusters
- Combined New England City and Town Areas
- Combined NECTAs
- New England City and Town Area Divisions
- NECTA Divisions
- Metropolitan New England City and Town Areas
- Metropolitan NECTAs
- Micropolitan New England City and Town Areas
- Micropolitan NECTAs
- Combined Statistical Areas
- CSAs
- Metropolitan Divisions
- Metropolitan Statistical Areas
- Micropolitan Statistical Areas
- States
- Counties

ACS2017

- 2010 Census Public Use Microdata Areas
- 2010 Census PUMAs
- 2010 PUMAs

- Census Public Use Microdata Areas
- Census PUMAs
- PUMAs
- 2010 Census ZIP Code Tabulation Areas
- 2010 Census ZCTAs
- Census ZCTAs
- ZCTAs
- Tribal Census Tracts
- Tribal Block Groups
- Census Tracts
- Census Block Groups
- Unified School Districts
- Secondary School Districts
- Elementary School Districts
- Estates
- County Subdivisions
- Subbarrios
- Consolidated Cities
- Incorporated Places
- Census Designated Places
- CDPs
- Alaska Native Regional Corporations
- Tribal Subdivisions
- Federal American Indian Reservations
- Off-Reservation Trust Lands
- State American Indian Reservations
- Hawaiian Home Lands
- Alaska Native Village Statistical Areas
- Oklahoma Tribal Statistical Areas
- State Designated Tribal Stastical Areas
- Tribal Designated Statistical Areas
- American Indian Joint-Use Areas
- 115th Congressional Districts
- Congressional Districts
- 2016 State Legislative Districts - Upper
- State Legislative Districts - Upper

- 2016 State Legislative Districts - Lower
- State Legislative Districts - Lower
- Census Divisions
- Divisions
- Census Regions
- Regions
- 2010 Census Urbanized Areas
- Census Urbanized Areas
- Urbanized Areas
- 2010 Census Urban Clusters
- Census Urban Clusters
- Urban Clusters
- Combined New England City and Town Areas
- Combined NECTAs
- New England City and Town Area Divisions
- NECTA Divisions
- Metropolitan New England City and Town Areas
- Metropolitan NECTAs
- Micropolitan New England City and Town Areas
- Micropolitan NECTAs
- Combined Statistical Areas
- CSAs
- Metropolitan Divisions
- Metropolitan Statistical Areas
- Micropolitan Statistical Areas
- States
- Counties

Census2010

- Public Use Microdata Areas
- PUMAs
- Traffic Analysis Districts
- TADs
- Traffic Analysis Zones
- TAZs
- Urban Growth Areas
- ZIP Code Tabulation Areas

- Zip Code Tabulation Areas
- ZCTAs
- Tribal Census Tracts
- Tribal Block Groups
- Census Tracts
- Census Block Groups
- Census Blocks
- Blocks
- Unified School Districts
- Secondary School Districts
- Elementary School Districts
- Estates
- County Subdivisions
- Subbarrios
- Consolidated Cities
- Incorporated Places
- Census Designated Places
- CDPs
- Alaska Native Regional Corporations
- Tribal Subdivisions
- Federal American Indian Reservations
- Off-Reservation Trust Lands
- State American Indian Reservations
- Hawaiian Home Lands
- Alaska Native Village Statistical Areas
- Oklahoma Tribal Statistical Areas
- State Designated Tribal Stastical Areas
- Tribal Designated Statistical Areas
- American Indian Joint-Use Areas
- 113th Congressional Districts
- 111th Congressional Districts
- 2012 State Legislative Districts - Upper
- 2012 State Legislative Districts - Lower
- 2010 State Legislative Districts - Upper
- 2010 State Legislative Districts - Lower
- Voting Districts

- Census Divisions
- Divisions
- Census Regions
- Regions
- Urbanized Areas
- Urban Clusters
- Combined New England City and Town Areas
- Combined NECTAs
- New England City and Town Area Divisions
- NECTA Divisions
- Metropolitan New England City and Town Areas
- Metropolitan NECTAs
- Micropolitan New England City and Town Areas
- Micropolitan NECTAs
- Combined Statistical Areas
- CSAs
- Metropolitan Divisions
- Metropolitan Statistical Areas
- Micropolitan Statistical Areas
- States
- Counties

Note: You may notice that there are (logical) duplicate layers in the lists above, for example "2010 Census PUMAs" and "2010 Census Public Use Microdata Areas". This is because there are multiple ways that users of Census data may refer to particular layers in their work. This duplication is purely for the convenience of **Census Geocoder** users, since the Census Geocoder API actually uses numerical identifiers for the layers returned.

When geocoding data, you can simply supply the layers you want using the `layers` keyword argument as below:

Single-line Address

Parametrized Address

Coordinates

Batch File

```
import census_geocoder as geocoder

result = geocoder.location.from_address('4600 Silver Hill Rd, Washington, DC 20233',
                                        benchmark = 'Current',
                                        vintage = 'ACS2019',
                                        layers = 'Census Tracts, States, CDPs, Divisions
↪')
```

(continues on next page)

(continued from previous page)

```
result = geocoder.geography.from_address('4600 Silver Hill Rd, Washington, DC 20233',
                                         benchmark = 'Current',
                                         vintage = 'ACS2019',
                                         layers = 'Census Tracts, States, CDPs, Divisions
↪')
```

See also:

- *Location.from_address()*
- *GeographicArea.from_address()*

```
import census_geocoder as geocoder

result = geocoder.location.from_address(street = '4600 Silver Hill Rd',
                                        city = 'Washington',
                                        state = 'DC',
                                        zip_code = '20233',
                                        benchmark = 'Current',
                                        vintage = 'ACS2019',
                                        layers = 'Census Tracts, States, CDPs, Divisions
↪')

result = geocoder.geography.from_address(street = '4600 Silver Hill Rd',
                                         city = 'Washington',
                                         state = 'DC',
                                         zip_code = '20233',
                                         benchmark = 'Current',
                                         vintage = 'ACS2019',
                                         layers = 'Census Tracts, States, CDPs, Divisions
↪')
```

See also:

- *Location.from_address()*
- *GeographicArea.from_address()*

```
import census_geocoder as geocoder

result = geocoder.location.from_coordinates(longitude = -76.92744,
                                            latitude = 38.845985,
                                            benchmark = 'Current',
                                            vintage = 'ACS2019',
                                            layers = 'Census Tracts, States, CDPs,␣
↪Divisions')

result = geocoder.geography.from_coordinates(longitude = -76.92744,
                                             latitude = 38.845985,
                                             benchmark = 'Current',
                                             vintage = 'ACS2019',
                                             layers = 'Census Tracts, States, CDPs,␣
↪Divisions')
```

See also:

- *Location.from_coordinates()*
- *GeographicArea.from_coordinates()*

```
import census_geocoder as geocoder

result = geocoder.location.from_batch(file_ = '/my-csv-file.csv',
                                      benchmark = 'Current',
                                      vintage = 'ACS2019')

result = geocoder.geography.from_batch(file_ = '/my-csv-file.csv',
                                       benchmark = 'Current',
                                       vintage = 'ACS2019',
                                       layers = 'Census Tracts, States, CDPs, Divisions')
```

See also:

- *Location.from_batch()*
- *GeographicArea.from_batch()*

Hint: When using the **Census Geocoder** to return geographic area data, you can request multiple layers worth of data by passing them in a comma-delimited string. This will return separate data for each layer indicated. The comma-delimited string can include white-space for easy readability, which means that the following two values are considered identical:

- `layers = 'Census Tracts, States, CDPs, Divisions'`
- `layers = 'Census Tracts,States,CDPs,Divisions'`

To retrieve all available layers that have data for a given location, you can submit `'all'`. Unless you have set the `CENSUS_GEOCODER_LAYERS` environment variable to a different value, `'all'` is the default set of layers that will be returned.

Note that layer names in the **Census Geocoder** are case-insensitive.

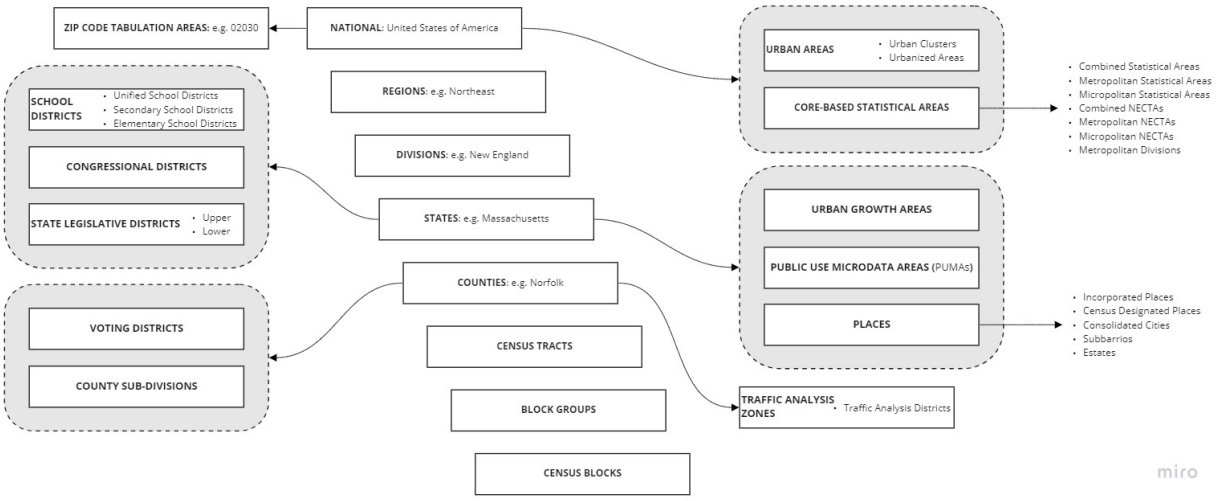
3.3 Census Geographic Hierarchies Explained

As you can tell from the list of layers above, there are lots of different types of geographic areas supported by the Census Geocoder API. These areas overlap in lots of different ways, and the US Census Bureau's documentation explaining this can be a little hard to find. Therefore, I've tried to explain the hierarchies' logic in straightforward language and diagrams below.

See also:

- `U.S. Census Bureau Geographic Entities and Concepts (PDF)`
- `The Standard Hierarchy of Census Geographic Entities (PDF)`
- `Hierarchy of American Indian, Alaska Native, and Native Hawaiian Areas (PDF)`
- `The Standard Hierarchy of Census Geographic Entities in Island Areas (PDF)`

3.3.1 Core Hierarchy



We should start by understanding the "core" of the US Census Bureau's hierarchy, and working our way "up" from the smallest section. This core hierarchy by definition does not overlap. Each area within a particular level of the hierarchy is precisely defined, with those definitions represented in the *Tigerline / Shapefile* data published by the US Census Bureau.

`Census Block`
: The single smallest element in the core hierarchy is the **Census Block**. This is the most granular geographical area for which the US Census Bureau reports data, and is the smallest geographic unit where data is available for 100% of its resident population.

`Block Groups`
: Collections of **Census Blocks**. In general, the population size for block groups are 600 - 3,000.

  This is the most granular geographical area for which the US Census Bureau reports *sampled data*.

`Census Tracts`
: Collections of **Block Groups**. They are considered small, permanent, and consistent statistical sections of their containing county.

  Optimally contains 4,000 people, and range from 1,200 - 8,000 people.

`Counties and County Equivalents`
: The first administrative (government administered) area defined in the core hierarchy. Counties have their own administrations, subordinate to the state administration. Defined as a collection of **Census Tracts**.

Note: In 48 states, "counties" in the data correspond to "counties" in the their legal administration.

In MD, MO, NV, and VA, Independent Cities are treated as counties.

In LA, parishes are treated as counties.

In Alaska, Cities, Boroughs, Municipalities, and Census Areas are treated as counties.

In Puerto Rico, municipios are treated as counties.

In American Samoa, islands and districts are treated as counties.

In the Northern Marianas, municipalities are treated as counties.

In the Virgin Islands, islands are treated as counties.

Guam and the District of Columbia are each treated as a county.

In addition to breaking down into census tracts, counties may also be broken down into:

- County Subdivisions
- Voting Districts

`States`
The federally-constituted state (or territory, as applicable). Defined as a collection of **Counties**.

In addition to breaking down into counties, states may also be broken down into:

- School Districts
- Congressional Districts
- State Legislative Districts

States also include **Places**, which are named entities in several types:

- **Incorporated Places**. Which are legally-bounded entities with some form of local governance recognized by the state. Typically they are referred to as cities, boroughs, towns, or villages.
- **Census Designated Places**. Which are statistical agglomerations of unincorporated areas that are still identifiable by name.
- **Consolidated Cities**. Which are statistical agglomerations of city-related places.

`Divisions`
Collections of states that comprise a division within the USGIS definition of divisions.

`Regions`
Collection of divisions that comprise a region, per the USGIS definition.

`National`
Collection of all regions, that in total makes up the United States of America.

In addition to breaking down into regions, the country can also be broken down into:

- Zip Code Tabulation Areas

Hint: It may be surprising that zip code tabulation areas are not defined at the state level. There are several important reasons for this fact:

- First, ZCTAs in the Census definition are only *approximate* matches for the US Postal Service's zip code definitions. They are *statistical* entities that are composed of Census Blocks, and so may not align perfectly to building zip codes.
- Zip codes in general are federally administered by the US Postal Service, and in some (very rare!) cases zip codes may actually straddle state lines.

The country also contains a number of standalone geographical areas, which while not comprising 100% of the nation, may represent significant sections of the country or its component parts. In particular, the country also includes:

- **Core-based Statistical Areas**. These are statistical areas that are composed of census blocks and which are used to represent different population agglomerations. Examples include Metropolitan Statistical Areas (which are statistical agglomerations for a given metro area), or NECTAs (New England City and Town Areas, which are division-specific agglomerations of New England communities).

- **Urban Areas**. These are statistical areas that are composed of census blocks, and which have two types: urban clusters (which contain 2,500 - 50,000 people) and urbanized areas (which contain 50,000 or more people).

3.3.2 Secondary Hierarchies

Budding off from the *core hierarchy*, specific geographic entities can either be broken down or contain other secondary hierarchies. Most secondary hierarchies are flat (i.e. they are themselves defined by a collection of *census blocks*), but they may be composed of different *types* of entities.

A good example of this pattern is the secondary-hierarchy concept of "School District". While school districts cannot be broken down further (they are defined by census blocks), there are three types of school district that are available within the US Census data: **Unified School Districts**, **Secondary School Districts**, and **Elementary School Districts**.

Places

Another major secondary hierarchy with similar "type-based" differentiation is the concept of "places". There are multiple types of place, including **Census Designated Places**, **Incorporated Places**, and **Consolidated Cities**. These are conceptual areas, which in turn can all be broken down into their component census blocks.

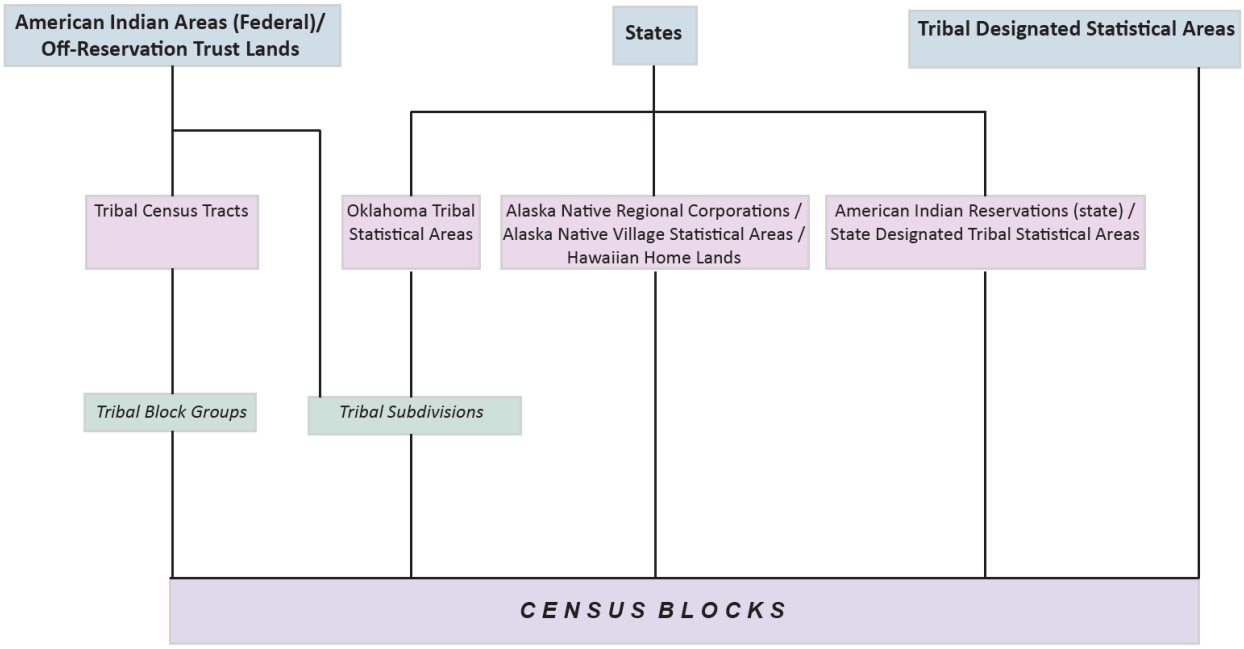
The most important types of places are:

- **Incorporated Places**. Which are legally-bounded entities with some form of local governance recognized by the state. Typically they are referred to as cities, boroughs, towns, or villages.
- **Census Designated Places**. Which are statistical agglomerations of unincorporated areas that are still identifiable by name.

3.3.3 AIANHH Hierarchy

Besides the *core hierarchy* described above, the US Census Bureau also reports data within an American Indian, Alaska Native, and Native Hawaiaan-oriented hierarchy.

This hierarchy is also built by rolling-up *Census Blocks*, however it does not conform to either the state or county-level definitions used in the core hierarchy. This is because tribal population groups, federally-designated American Indian areas, tribal-designated areas, etc. may often cross state, division, or regional lines.


American Indian Areas (Federal)/ Off-Reservation Trust Lands
States
Tribal Designated Statistical Areas
Tribal Census Tracts
Oklahoma Tribal Statistical Areas
Alaska Native Regional Corporations / Alaska Native Village Statistical Areas / Hawaiian Home Lands
American Indian Reservations (state) / State Designated Tribal Statistical Areas
Tribal Block Groups
Tribal Subdivisions
CENSUS BLOCKS

CHAPTER

FOUR

API REFERENCE

4.1 Locations

4.1.1 Location

class Location(***kwargs*)

Represents a specific location returned by the US Census Geocoder API.

classmethod from_address(**args*, ***kwargs*)

Return data from an adddress, supplied either as a single *one-line address* or a *parametrized address*.

Parameters

- **one_line** (str / None) – A single-line address, e.g. `'4600 Silver Hill Rd, Washington, DC 20233'`. Defaults to None.
- **street_1** (str / None) – A street address, e.g. `'4600 Silver Hill Rd'`. Defaults to None.
- **street_2** (str / None) – A secondary component of a street address, e.g. `'Floor 3'`. Defaults to None.

- **street_3** (str / None) – A tertiary component of a street address, e.g. 'Apt. B'. Defaults to None.
- **city** (str / None) – The city or town of a street address, e.g. 'Washington'. Defaults to None.
- **state** (str / None) – The state or territory of a street address, e.g. 'DC'. Defaults to None.
- **zip_code** (str / None) – The zip code (or zip code + 4) of a street address, e.g. '20233'. Defaults to None.
- **benchmark** (str) – The name of the *benchmark* of data to return. The default value is determined by the CENSUS_GEOCODER_BENCHMARK environment variable, and if that is not set defaults to 'Current' which represents the current default benchmark, per the Census Geocoder API.

  Accepts the following values:

  – 'Current' (default)

  – 'Census2020'

- **vintage** (str) – The vintage of Census data for which data should be returned. The default value is determined by the CENSUS_GEOCODER_VINTAGE environment variable, and if that is not set defaults to 'Current' which represents the default vintage per the Census Geocoder API.

  Acceptable values are dependent on the benchmark specified, as per the table below:

	BENCHMARKS	
	Current	Census2020
VINTAGES	Current	Census2020
	Census2020	Census2010
	ACS2019	
	ACS2018	
	ACS2017	
	Census2010	

- **layers** (str) – The set of geographic layers to return for the request. The default value is determined by the CENSUS_GEOCODER_LAYERS environment variable, and if that is not set defaults to 'all'.

  See also:

  – *Geographies Benchmarks, Vintages, and Layers*

Note: If more than one address-related parameter are supplied, this method will assume that a *parametrized address* is provided.

Returns A given geographic entity.

Return type GeographicEntity

Raises

- ***NoAddressError*** – if no address information is supplied

- **EntityNotFoundError** – if no geographic entity was found matching the address supplied
- *UnrecognizedBenchmarkError* – if the `benchmark` supplied is not recognized
- *UnrecognizedVintageError* – if the `vintage` supplied is not recognized

classmethod from_batch(**args*, ***kwargs*)

Return geographic entities for a batch collection of inputs.

Parameters

- **file** (`str`) – The name of a file in CSV, XLS/X, DAT, or TXT format. Expects the file to have the following columns *without a header row*:
  - Unique ID
  - Street Address
  - City
  - State
  - Zip Code
- **benchmark** (`str`) – The name of the *benchmark* of data to return. The default value is determined by the `CENSUS_GEOCODER_BENCHMARK` environment variable, and if that is not set defaults to `'Current'` which represents the current default benchmark, per the Census Geocoder API.

  Accepts the following values:
  - `'Current'` (default)
  - `'Census2020'`
- **vintage** (`str`) – The vintage of Census data for which data should be returned. The default value is determined by the `CENSUS_GEOCODER_VINTAGE` environment variable, and if that is not set defaults to `'Current'` which represents the default vintage per the Census Geocoder API.

  Acceptable values are dependent on the `benchmark` specified, as per the table below:

	BENCHMARKS	
	Current	Census2020
VINTAGES	Current	Census2020
	Census2020	Census2010
	ACS2019	
	ACS2018	
	ACS2017	
	Census2010	

- **layers** (`str`) – The set of geographic layers to return for the request. The default value is determined by the `CENSUS_GEOCODER_LAYERS` environment variable, and if that is not set defaults to `'all'`.

  See also:

  - *Geographies Benchmarks, Vintages, and Layers*

Returns A collection of geographic entities.

Return type list of GeographicEntity

Raises

- *NoFileProvidedError* – if no `file_` is provided
- FileNotFoundError – if `file_` does not exist on the filesystem
- *BatchSizeTooLargeError* – if `file_` contains more than 10,000 records
- **EntityNotFoundError** – if no geographic entity was found matching the address supplied
- *UnrecognizedBenchmarkError* – if the `benchmark` supplied is not recognized
- *UnrecognizedVintageError* – if the `vintage` supplied is not recognized

classmethod from_coordinates(**args*, ***kwargs*)

Return data from a pair of geographic coordinates (longitude and latitude).

Parameters

- **longitude** (*numeric*) – The longitude coordinate.
- **latitude** (*numeric*) – The latitude coordinate.
- **benchmark** (str) – The name of the *benchmark* of data to return. The default value is determined by the `CENSUS_GEOCODER_BENCHMARK` environment variable, and if that is not set defaults to `'Current'` which represents the current default benchmark, per the Census Geocoder API.

  Accepts the following values:

  – `'Current'` (default)

  – `'Census2020'`

- **vintage** (str) – The vintage of Census data for which data should be returned. The default value is determined by the `CENSUS_GEOCODER_VINTAGE` environment variable, and if that is not set defaults to `'Current'` which represents the default vintage per the Census Geocoder API.

  Acceptable values are dependent on the `benchmark` specified, as per the table below:

	BENCHMARKS	
	Current	Census2020
VINTAGES	Current	Census2020
	Census2020	Census2010
	ACS2019	
	ACS2018	
	ACS2017	
	Census2010	

- **layers** (str) – The set of geographic layers to return for the request. The default value is determined by the `CENSUS_GEOCODER_LAYERS` environment variable, and if that is not set defaults to `'all'`.

  See also:

  – *Geographies Benchmarks, Vintages, and Layers*

Note: If more than one address-related parameter are supplied, this method will assume that a *parametrized address* is provided.

Returns A given geographic entity.

Return type `GeographicEntity`

Raises

- ***NoAddressError*** – if no address information is supplied
- **EntityNotFound** – if no geographic entity was found matching the address supplied
- ***UnrecognizedBenchmarkError*** – if the `benchmark` supplied is not recognized
- ***UnrecognizedVintageError*** – if the `vintage` supplied is not recognized

classmethod from_csv_record(*csv_record*)

Create an instance of the geographic entity from its CSV record.

Parameters **csv_record** (`list` of `str`) – The list of columns for the CSV record.

Returns An instance of the geographic entity.

Return type `GeographicEntity`

classmethod from_dict(*as_dict*)

Create an instance of the geographic entity from its `dict` representation.

Parameters **as_dict** (`dict`) – The `dict` representation of the geographic entity.

Returns An instance of the geographic entity.

Return type `GeographicEntity`

classmethod from_json(*as_json*)

Create an instance of the geographic entity from its JSON representation.

Parameters **as_json** (`str`, `dict`, or `list`) – The JSON representation of the geographic entity.

Returns An instance of the geographic entity.

Return type `GeographicEntity`

inspect(*as_census_fields=False*)

Produce a list of the location's properties that have values.

Parameters **as_census_fields** (`bool`) – If `True`, return property names as they appear in Census databases or the output of the Census Geocoder API. If `False`, return properties as they are defined on the **Census Geocoder** objects. Defaults to `False`.

Return type `list` of `str`

to_dict()

Returns a `dict` representation of the geographic entity.

Note: The `dict` representation matches the JSON structure for the US Census Geocoder API. This is a not-very-pythonic `dict` structure, but at least this ensures idempotency.

Returns `dict` representation of the entity.

Return type dict

to_json()
Returns a JSON representation of the geographic entity.

Note: The JSON representation matches the JSON structure for the US Census Geocoder API. This is a not-very-pythonic structure, but at least this ensures idempotency.

Returns str representation of the entity.

Return type str

property benchmark
The short-hand value of the *benchmark* for which this *Location* was calculated.

Return type str / None

property benchmark_description
The description of the *benchmark* for which this data was returned.

Return type str

property benchmark_id
The name of the *benchmark* for which this data was returned.

Return type str

property benchmark_is_default
If True, indicates that the default *benchmark* has been applied.

Return type bool

property benchmark_name
The name of the *benchmark* for which this data was returned.

Return type str

property entity_type
The type of geographic entity that the object represents. Supports either: locations or geographies.

Return type str

property input_address
Returns a dict with the input address provided.

Return type dict

property input_city
The city that was provided as input to get this *Location*.

Return type str or None

property input_one_line
The one-line address that was provided as input to get this *Location*.

Return type str or None

property input_state
The state that was provided as input to get this *Location*.

Return type str or None

property input_street
The street address that was provided as input to get this *Location*.

Return type str or None

property input_zip_code
The zip code that was provided as input to get this *Location*.

Return type str or None

property matched_addresses
Collection of addresses that have been matched to the *Location*.

Return type list of *MatchedAddress* / None

property vintage
The short-hand value of the *vintage* for which this *Location* was calculated.

Return type str / None

property vintage_description
The description of the *vintage* for which this data was returned.

Return type str

property vintage_id
The name of the *vintage* for which this data was returned.

Return type str

property vintage_is_default
If True, indicates that the default *vintage* has been applied.

Return type bool

property vintage_name
The name of the *vintage* for which this data was returned.

Return type str

4.1.2 MatchedAddress

class MatchedAddress(***kwargs*)
Represents a matched address returned by the US Census GeoCoder API.

classmethod from_csv_record(*csv_record*)
Create an instance of the geographic entity from its CSV record.

Parameters **csv_record** (list of str) – The list of columns for the CSV record.

Returns An instance of the geographic entity.

Return type GeographicEntity

classmethod from_dict(*as_dict*)
Create an instance of the geographic entity from its dict representation.

Parameters **as_dict** (dict) – The dict representation of the geographic entity.

Returns An instance of the geographic entity.

Return type GeographicEntity

classmethod from_json(*as_json*)
Create an instance of the geographic entity from its JSON representation.

Parameters **as_json** (str, dict, or list) – The JSON representation of the geographic entity.

Returns An instance of the geographic entity.

Return type GeographicEntity

inspect(*as_census_fields=False*)
Produce a list of the matched address properties that have values.

Parameters **as_census_fields** (bool) – If True, return property names as they appear in Census databases or the output of the Census Geocoder API. If False, return properties as they are defined on the **Census Geocoder** objects. Defaults to False.

Return type list of str

to_dict()
Returns a dict representation of the geographic entity.

Note: The dict representation matches the JSON structure for the US Census Geocoder API. This is a not-very-pythonic dict structure, but at least this ensures idempotency.

Returns dict representation of the entity.

Return type dict

to_json()
Returns a JSON representation of the geographic entity.

Note: The JSON representation matches the JSON structure for the US Census Geocoder API. This is a not-very-pythonic structure, but at least this ensures idempotency.

Returns str representation of the entity.

Return type str

property address
The canonical address that was matched for the *Location*.

Return type str / None

property block
Census Block Code

Return type str / None

property city
The canonical city name that was matched for the *Location*.

Rdirection str / None

property county_fips_code
County FIPS Code

Return type str

property entity_type
The type of geographic entity that the object represents. Supports either: `locations` or `geographies`.

Return type str

property from_address
The canonical lower-bound street number that was matched for the *Location*.

Rdirection str / None

property geographies
Collection of geographical areas that this address is part of.

Return type GeographyCollection / None

property latitude
The latitude coordinate for the location.

Return type decimal

property longitude
The longitude coordinate for the location.

Return type decimal

property pre_direction
The canonical pre-direction that was matched for the *Location*.

Rdirection str / None

property pre_qualifier
The canonical pre-qualifier that was matched for the *Location*.

Rqualifier str / None

property pre_type
The canonical pre-type that was matched for the *Location*.

Return type str / None

property state
The canonical state that was matched for the *Location*.

Rdirection str / None

property state_fips_code
State FIPS Code

Return type str

property street
The canonical street name that was matched for the *Location*.

Rdirection str / None

property suffix_direction
The canonical suffix-direction that was matched for the *Location*.

Rdirection str / None

property suffix_qualifier
The canonical suffix-qualifier that was matched for the *Location*.

Rqualifier str / None

property suffix_type
The canonical suffix-type that was matched for the *Location*.

Return type str / None

property tigerline_id
The TigerLine ID for the matched address.

Return type str / None

property tigerline_side
The TigerLine side of the street for the matched address. Accepts either 'L' or 'R'.

Return type str / None

property to_address
The canonical upper-bound street number that was matched for the *Location*.

Rdirection str / None

property tract
Census Tract Code

Return type str

property zip_code
The canonical zip code that was matched for the *Location*.

Rdirection str / None

4.2 Geographies

4.2.1 GeographyCollection

class GeographyCollection(***kwargs*)
Collection of *GeographicArea* objects.

from_csv_record(*csv_record*)
Create an instance of the geographic entity from its CSV record.

Parameters **csv_record** (list of str) – The list of columns for the CSV record.

Returns An instance of the geographic entity.

Return type GeographicEntity

classmethod from_dict(*as_dict*)
Create an instance of the geographic entity from its dict representation.

Parameters **as_dict** (dict) – The dict representation of the geographic entity.

Returns An instance of the geographic entity.

Return type GeographicEntity

classmethod from_json(*as_json*)
Create an instance of the geographic entity from its JSON representation.

Parameters **as_json** (str, dict, or list) – The JSON representation of the geographic entity.

Returns An instance of the geographic entity.

Return type GeographicEntity

to_dict()
Returns a dict representation of the geographic entity.

Note: The dict representation matches the JSON structure for the US Census Geocoder API. This is a not-very-pythonic dict structure, but at least this ensures idempotency.

Returns dict representation of the entity.

Return type dict

to_json()
Returns a JSON representation of the geographic entity.

Note: The JSON representation matches the JSON structure for the US Census Geocoder API. This is a not-very-pythonic structure, but at least this ensures idempotency.

Returns str representation of the entity.

Return type str

property american_indian_joint_use_areas
American Indian Joint-Use Areas

Return type list of *AIJUA*

property anrc
Alaska Native Regional Corporations

Return type list of *ANRC*

property anvsa
Alaska Native Village Statistical Area

Return type list of *ANVSA*

property block_groups
Census Block Groups

Return type list of *CensusBlockGroup*

property blocks
Census Blocks

Return type list of *CensusBlock*

property blocks_2020
2020 Census Blocks

Return type list of *CensusBlock_2020*

property combined_nectas
Combined New England City and Town Areas

Return type list of *CombinedNECTA*

property congressional_districts_111
111th Congressional Districts

Return type list of *CongressionalDistrict_111*

property congressional_districts_113
113th Congressional Districts

Return type list of *CongressionalDistrict_113*

property congressional_districts_115
115th Congressional Districts

Return type list of *CongressionalDistrict_115*

property congressional_districts_116
116th Congressional Districts

Return type list of *CongressionalDistrict_116*

property consolidated_cities
Consolidated Cities

Return type list of *ConsolidatedCity*

property counties
Census Counties

Return type list of *County*

property county_subdivisions
County Sub-division

Return type list of *CountySubDivision*

property csa
Combined Statistical Areas

Return type list of *CombinedStatisticalArea*

property divisions
Census Divisions

Return type list of *CensusDivision*

property elementary_school_districts
Elementary School Districts

Return type list of *ElementarySchoolDistrict*

property entity_type
The type of geographic entity that the object represents. Supports either: `locations` or `geographies`.

Return type str

property estates
Estates

Return type list of *Estate*

property federal_american_indian_reservations
Federal American Indian Reservations

Return type list of *FederalAmericanIndianReservation*

property hawaiian_home_lands
Hawaiian Home Lands

Return type list of *HawaiianHomeLand*

property incorporated_places
Incorporated Places

Return type list of *IncorporatedPlace*

property metropolitan_divisions
Metropolitan Divisions

Return type list of *MetropolitanDivision*

property metrpolitan_nectas
Metropolitan New England City and Town Areas

Return type list of *MetropolitanNECTA*

property micropolitan_nectas
Micropolitan New England City and Town Areas

Return type list of *MicropolitanNECTA*

property msa
Metropolitan Statistical Area

Return type list of *MetropolitanStatisticalArea*

property necta_divisions
New England City and Town Area Divisions

Return type list of *NECTADivision*

property off_reservation_trust_lands
Off-Reservation Trust Lands

Return type list of *OffReservationTrustLand*

property otsa
Oklahoma Tribal Statistical Areas

Return type list of *OTSA*

property pumas
Public Use Microdata Areas

Return type list of *PUMA*

property pumas_2010
2010 Census Public Use Microdata Areas

Return type list of *PUMA_2010*

property regions
Census Regions

Return type list of *CensusRegion*

property sdtsa
State Designated Tribal Statistical Areas

Return type list of *SDTSA*

property secondary_school_districts
Secondary School Districts

Return type list of *SecondarySchoolDistrict*

property state_american_indian_reservations
State American Indian Reservation

Return type list of *StateAmericanIndianReservation*

property state_legislative_districts_lower
State Legislative Districts - Lower

Return type list of *StateLegislativeDistrictLower*

property state_legislative_districts_lower_2010
2010 State Legislative Districts - Lower

Return type list of StateLegislativeDistrictLower_2010

property state_legislative_districts_lower_2012
2012 State Legislative Districts - Lower

Return type list of StateLegislativeDistrictLower_2012

property state_legislative_districts_lower_2016
2016 State Legislative Districts - Lower

Return type list of StateLegislativeDistrictLower_2016

property state_legislative_districts_lower_2018
2018 State Legislative Districts - Lower

Return type list of StateLegislativeDistrictLower_2018

property state_legislative_districts_upper
2010 State Legislative Districts - Upper

Return type list of StateLegislativeDistrictUpper_2010

property state_legislative_districts_upper_2010
2010 State Legislative Districts - Upper

Return type list of StateLegislativeDistrictUpper_2010

property state_legislative_districts_upper_2012
2012 State Legislative Districts - Upper

Return type list of StateLegislativeDistrictUpper_2012

property state_legislative_districts_upper_2016
2016 State Legislative Districts - Upper

Return type list of StateLegislativeDistrictUpper_2016

property state_legislative_districts_upper_2018
2018 State Legislative Districts - Upper

Return type list of StateLegislativeDistrictUpper_2018

property states
States

Return type list of *State*

property subbarrios
Sub-barrios

Return type list of *Subbarrio*

property tdsa
Tribal Designated Statistical Areas

Return type list of *TDSA*

property tracts
Census Tracts

Return type list of *CensusTract*

property traffic_analysis_districts
Traffic Analysis Districts

Return type list of *TrafficAnalysisDistrict*

property traffic_analysis_zones
Traffic Analysis Zones

Return type list of *TrafficAnalysisZone*

property tribal_block_groups
Tribal Census Block Groups

Return type list of *TribalCensusBlockGroup*

property tribal_subdivisions
Tribal Sub-divisions

Return type list of *TribalSubDivision*

property tribal_tracts
Tribal Census Tracts

Return type list of *TribalCensusTract*

property unified_school_districts
Unified School Districts

Return type list of *UnifiedSchoolDistrict*

property urban_clusters
Urban Clusters

Return type list of *UrbanCluster*

property urban_clusters_2010
2010 Census Urban Clusters

Return type list of *urban_clusters_2010*

property urban_growth_areas
Urban Growth Areas

Return type list of *UrbanGrowthArea*

property urbanized_areas
Urbanized Areas

Return type list of *UrbanizedArea*

property urbanized_areas_2010
2010 Census Urbanized Areas

Return type list of *UrbanizedArea_2010*

property voting_districts
Voting Districts

Return type list of *VotingDistrict*

property zcta5
Zip Code Tabulation Area

Return type list of *ZCTA5*

property zcta_2010
2010 Census ZIP Code Tabulation Areas

Return type list of *ZCTA_2010*

property zcta_2020
2020 Census ZIP Code Tabulation Areas

Return type list of *ZCTA_2020*

4.2.2 GeographicArea

class GeographicArea(***kwargs*)
Base class for a given *geography* as supported by the US government.

classmethod from_address(**args*, ***kwargs*)
Return data from an adddress, supplied either as a single *one-line address* or a *parametrized address*.

Parameters

- **one_line** (str / None) – A single-line address, e.g. '4600 Silver Hill Rd, Washington, DC 20233'. Defaults to None.
- **street_1** (str / None) – A street address, e.g. '4600 Silver Hill Rd'. Defaults to None.
- **street_2** (str / None) – A secondary component of a street address, e.g. 'Floor 3'. Defaults to None.
- **street_3** (str / None) – A tertiary component of a street address, e.g. 'Apt. B'. Defaults to None.
- **city** (str / None) – The city or town of a street address, e.g. 'Washington'. Defaults to None.
- **state** (str / None) – The state or territory of a street address, e.g. 'DC'. Defaults to None.
- **zip_code** (str / None) – The zip code (or zip code + 4) of a street address, e.g. '20233'. Defaults to None.
- **benchmark** (str) – The name of the *benchmark* of data to return. The default value is determined by the CENSUS_GEOCODER_BENCHMARK environment variable, and if that is not set defaults to 'Current' which represents the current default benchmark, per the Census Geocoder API.

  Accepts the following values:

  – 'Current' (default)

  – 'Census2020'

- **vintage** (str) – The vintage of Census data for which data should be returned. The default value is determined by the CENSUS_GEOCODER_VINTAGE environment variable, and if that is not set defaults to 'Current' which represents the default vintage per the Census Geocoder API.

  Acceptable values are dependent on the benchmark specified, as per the table below:

	BENCHMARKS	
	Current	Census2020
VINTAGES	Current	Census2020
	Census2020	Census2010
	ACS2019	
	ACS2018	
	ACS2017	
	Census2010	

- **layers** (str) – The set of geographic layers to return for the request. The default value is determined by the CENSUS_GEOCODER_LAYERS environment variable, and if that is not set defaults to 'all'.

  See also:

  – *Geographies Benchmarks, Vintages, and Layers*

Note: If more than one address-related parameter are supplied, this method will assume that a *parametrized address* is provided.

Returns A given geographic entity.

Return type GeographicEntity

Raises

- ***NoAddressError*** – if no address information is supplied
- **EntityNotFoundError** – if no geographic entity was found matching the address supplied
- ***UnrecognizedBenchmarkError*** – if the benchmark supplied is not recognized
- ***UnrecognizedVintageError*** – if the vintage supplied is not recognized

classmethod from_batch(**args*, ***kwargs*)

Return geographic entities for a batch collection of inputs.

Parameters

- **file** (str) – The name of a file in CSV, XLS/X, DAT, or TXT format. Expects the file to have the following columns *without a header row*:
  - Unique ID
  - Street Address
  - City
  - State
  - Zip Code
- **benchmark** (str) – The name of the *benchmark* of data to return. The default value is determined by the CENSUS_GEOCODER_BENCHMARK environment variable, and if that is not set defaults to 'Current' which represents the current default benchmark, per the Census Geocoder API.

  Accepts the following values:

  - `'Current'` (default)
  - `'Census2020'`

- **vintage** (`str`) – The vintage of Census data for which data should be returned. The default value is determined by the `CENSUS_GEOCODER_VINTAGE` environment variable, and if that is not set defaults to `'Current'` which represents the default vintage per the Census Geocoder API.

  Acceptable values are dependent on the `benchmark` specified, as per the table below:

	BENCHMARKS	
	Current	Census2020
VINTAGES	Current	Census2020
	Census2020	Census2010
	ACS2019	
	ACS2018	
	ACS2017	
	Census2010	

- **layers** (`str`) – The set of geographic layers to return for the request. The default value is determined by the `CENSUS_GEOCODER_LAYERS` environment variable, and if that is not set defaults to `'all'`.

  See also:

  - *Geographies Benchmarks, Vintages, and Layers*

Returns A collection of geographic entities.

Return type `list` of `GeographicEntity`

Raises

- ***NoFileProvidedError*** – if no `file_` is provided
- **FileNotFoundError** – if `file_` does not exist on the filesystem
- ***BatchSizeTooLargeError*** – if `file_` contains more than 10,000 records
- **EntityNotFoundError** – if no geographic entity was found matching the address supplied
- ***UnrecognizedBenchmarkError*** – if the `benchmark` supplied is not recognized
- ***UnrecognizedVintageError*** – if the `vintage` supplied is not recognized

classmethod from_coordinates(**args*, ***kwargs*)

Return data from a pair of geographic coordinates (longitude and latitude).

Parameters

- **longitude** (*numeric*) – The longitude coordinate.
- **latitude** (*numeric*) – The latitude coordinate.
- **benchmark** (`str`) – The name of the *benchmark* of data to return. The default value is determined by the `CENSUS_GEOCODER_BENCHMARK` environment variable, and if that is not set defaults to `'Current'` which represents the current default benchmark, per the Census Geocoder API.

  Accepts the following values:

  - `'Current'` (default)
  - `'Census2020'`

- **vintage** (`str`) – The vintage of Census data for which data should be returned. The default value is determined by the `CENSUS_GEOCODER_VINTAGE` environment variable, and if that is not set defaults to `'Current'` which represents the default vintage per the Census Geocoder API.

  Acceptable values are dependent on the `benchmark` specified, as per the table below:

	BENCHMARKS	
	Current	Census2020
VINTAGES	Current	Census2020
	Census2020	Census2010
	ACS2019	
	ACS2018	
	ACS2017	
	Census2010	

- **layers** (`str`) – The set of geographic layers to return for the request. The default value is determined by the `CENSUS_GEOCODER_LAYERS` environment variable, and if that is not set defaults to `'all'`.

  See also:

  - *Geographies Benchmarks, Vintages, and Layers*

Note: If more than one address-related parameter are supplied, this method will assume that a *parametrized address* is provided.

Returns A given geographic entity.

Return type `GeographicEntity`

Raises

- ***NoAddressError*** – if no address information is supplied
- **EntityNotFound** – if no geographic entity was found matching the address supplied
- ***UnrecognizedBenchmarkError*** – if the `benchmark` supplied is not recognized
- ***UnrecognizedVintageError*** – if the `vintage` supplied is not recognized

classmethod from_csv_record(*csv_record*)

Create an instance of the geographic entity from its CSV record.

Parameters **csv_record** (`list` of `str`) – The list of columns for the CSV record.

Returns An instance of the geographic entity.

Return type `Geography`

classmethod from_dict(*as_dict*)

Create an instance of the geographic entity from its `dict` representation.

Parameters **as_dict** (`dict`) – The `dict` representation of the geographic entity.

Returns An instance of the geographic entity.

Return type GeographicEntity

classmethod from_json(*as_json*)
Create an instance of the geographic entity from its JSON representation.

Parameters **as_json** (str, dict, or list) – The JSON representation of the geographic entity.

Returns An instance of the geographic entity.

Return type GeographicEntity

inspect(*as_census_fields=False*)
Produce a list of the geographic area's properties that have values.

Parameters **as_census_fields** (bool) – If True, return property names as they appear in Census databases or the output of the Census Geocoder API. If False, return properties as they are defined on the **Census Geocoder** objects. Defaults to False.

Return type list of str

to_dict()
Returns a dict representation of the geographic entity.

Note: The dict representation matches the JSON structure for the US Census Geocoder API. This is a not-very-pythonic dict structure, but at least this ensures idempotency.

Warning: Note that certain geography types only use a subset of the properties returned. Unused or unavailable properties will be returned as None which will be converted to null if serialized to JSON.

Returns dict representation of the entity.

Return type dict

to_json()
Returns a JSON representation of the geographic entity.

Note: The JSON representation matches the JSON structure for the US Census Geocoder API. This is a not-very-pythonic structure, but at least this ensures idempotency.

Returns str representation of the entity.

Return type str

property basename
The human-readable basename of the geography.

Return type str / None

property block
Census Block Code

Return type str / None

property block_group
Census Block Group Code

Return type str

property cbsa
Census CBSA Code

Return type str / None

property cbsa_pci
CBSA Principal City Indciator

Return type str / None

property congressional_session_code
Congressional Session Code

Return type str / None

property county_cc
County Class Code

Return type str / None

property county_fips_code
County FIPS Code

Return type str

property county_ns
County ANSI Feature Code

Return type str / None

property csa
Census CSA Code

Return type str / None

property division_fips_code
State FIPS Code

Return type str

property entity_type
The type of geographic entity that the object represents. Supports either: `locations` or `geographies`.

Return type str

property funcstat
The functional status code of the geography.

See also:

- Functional Status Codes and Definitions

Return type str

property functional_status
The functional status of the geography.

See also:

- Functional Status Codes and Definitions

Return type str

property geography_type
Returns the Geography Type for the given geography.

property geoid
The Geographic Identifier.

Note: Fully concatenated geographic code (State FIPS and component numbers).

Return type str / None

property high_school_grade
School District - Highest Grade

Return type str / None

property is_principal_city
If True, indicates that the geography is the principal city of its surrounding entity.

Return type bool

property land_area
The area of the geography that is on solid land, expressed in square meters.

Return type int / None

property latitude
The *centroid latitude* for the geographic area.

Return type Decimal / None

property latitude_internal_point
The *internal point latitude* for the geographic area.

Return type Decimal / None

property legal_statistical_area
Legal/Statistical Area Descriptor

See also:

- Legal/Statistical Area Descriptor Codes and Definitions

Return type str / None

property legislative_session_year
Legislative Session Year (LSY)

Return type str / None

property longitude
The *centroid longitude* for the geographic area.

Return type Decimal / None

property longitude_internal_point
The *internal point longitude* for the geographic area.

Return type Decimal / None

property low_school_grade
School District - Lowest Grade

Return type str / None

property lsad
Legal/Statisical Area Descriptor (LSAD) Code

See also:

- Legal/Statistical Area Descriptor Codes and Definitions

Return type str / None

property lsad_category
Indicates the category of the LSAD for the geography. Returns either:

- Unspecified
- Prefix
- Suffix
- Balance

Return type str

property name
The human-readable name of the geography.

Return type str / None

property necta_pci
NECTA Principal City Indciator

Return type str / None

property object_id
The Object Identifier.

Return type str / None

property oid
The OID.

Return type str / None

property place
Census Place Code

Return type str / None

property place_cc
Place Class Code

Return type str / None

property place_ns
Place ANSI Feature Code

Return type str / None

property region_fips_code
Region FIPS Code

Return type str

property school_district_type
School District Type

Return type str / None

property state_abbreviation
State Abbreviation

Return type str

property state_fips_code
State FIPS Code

Return type str

property state_ns
State ANSI Feature Code

Return type str

property tract
Census Tract Code

Return type str

property water_area
The area of the geography that is covered in water, expressed in square meters.

Note: Water area calculations in this table include only perennial water. All other water (intermittent, glacier, and marsh/swamp) is included in this table as part of `land_area` calculations.

Return type int / None

property zcta5
ZCTA-5 Zip Code Value

Return type str / None

property zcta5_cc
ZCTA5 Class Code

Return type str / None

4.2.3 Census Block and Related

class CensusBlock(***kwargs*)
Census Block

class CensusBlock_2020(***kwargs*)
2020 Census Blocks

4.2.4 Census Block Group

class CensusBlockGroup(***kwargs*)
Census Block Group

4.2.5 Tribal Census Block Group

class TribalCensusBlockGroup(***kwargs*)
Tribal Census Block Group

4.2.6 Census Tract

class CensusTract(***kwargs*)
Census Tract

4.2.7 Tribal Census Tract

class TribalCensusTract(***kwargs*)
Tribal Census Tract

4.2.8 County and Related

class County(***kwargs*)

class CountySubDivision(***kwargs*)
County Sub-division

4.2.9 State

class State(***kwargs*)

4.2.10 PUMA and Related

class PUMA(***kwargs*)
Public Use Microdata Area

class PUMA_2010(***kwargs*)
2010 Census Public Use Microdata Area

4.2.11 State Legislative District and Related

class StateLegislativeDistrictLower(***kwargs*)
State Legislative District - Lower

class StateLegislativeDistrictLower_2010(***kwargs*)
2010 State Legislative District - Lower

class StateLegislativeDistrictLower_2012(***kwargs*)
2012 State Legislative District - Lower

class StateLegislativeDistrictLower_2016(***kwargs*)
2016 State Legislative District - Lower

class StateLegislativeDistrictLower_2018(***kwargs*)
2018 State Legislative District - Lower

class StateLegislativeDistrictUpper(***kwargs*)
State Legislative District - Upper

class StateLegislativeDistrictUpper_2010(***kwargs*)
2010 State Legislative District - Upper

class StateLegislativeDistrictUpper_2012(***kwargs*)
2012 State Legislative District - Upper

class StateLegislativeDistrictUpper_2016(***kwargs*)
2016 State Legislative District - Upper

class StateLegislativeDistrictUpper_2018(***kwargs*)
2018 State Legislative District - Upper

4.2.12 ZCTA5 and Related

class ZCTA5(***kwargs*)

class ZCTA_2010(***kwargs*)
2010 Zip Code Tabulation Areas

class ZCTA_2020(***kwargs*)
2020 Zip Code Tabulation Areas

4.2.13 School District-Related

class UnifiedSchoolDistrict(***kwargs*)
Unified School District

class SecondarySchoolDistrict(***kwargs*)
Secondary School District

class ElementarySchoolDistrict(***kwargs*)
Elementary School District

4.2.14 Voting District

class VotingDistrict(***kwargs*)
Voting District

4.2.15 Metropolitan Division

class MetropolitanDivision(***kwargs*)
Metropolitan Division

4.2.16 Combined Statistical Area

class CombinedStatisticalArea(***kwargs*)
Combined Statistical Area

4.2.17 Tribal Subdivision

class TribalSubDivision(***kwargs*)
Tribal Sub-division

4.2.18 Census Designated Place

class CensusDesignatedPlace(***kwargs*)
Census Designated Place

4.2.19 Division

class CensusDivision(***kwargs*)
Census Division

4.2.20 Congressional District and Related

class CongressionalDistrict(***kwargs*)
Congressional District

class CongressionalDistrict_116(***kwargs*)
116th Congressional District

class CongressionalDistrict_115(***kwargs*)
115th Congressional District

class CongressionalDistrict_113(***kwargs*)
113th Congressional District

class CongressionalDistrict_111(***kwargs*)
111th Congressional District

4.2.21 Region

class CensusRegion(***kwargs*)
Census Region

4.2.22 Metropolitan Statistical Area

class MetropolitanStatisticalArea(***kwargs*)
Metropolitan Statistical Area

4.2.23 Micropolitan Statistical Area

class MicropolitanStatisticalArea(***kwargs*)
Micropolitan Statistical Area

4.2.24 Estate

class Estate(***kwargs*)

4.2.25 Subbarrio

class Subbarrio(***kwargs*)

4.2.26 Consolidated City

class ConsolidatedCity(***kwargs*)
Consolidated City

4.2.27 Incorporated Place

class IncorporatedPlace(***kwargs*)
Incorporated Place

4.2.28 Alaska Native Regional Corporation

class ANRC(***kwargs*)
Alaska Native Regional Corporation

4.2.29 Federal American Indian Reservation

class FederalAmericanIndianReservation(***kwargs*)
Federal American Indian Reservation

4.2.30 Off-Reservation Trust Land

class OffReservationTrustLand(***kwargs*)
Off-Reservation Trust Land

4.2.31 State American Indian Reservation

class StateAmericanIndianReservation(***kwargs*)
State American Indian Reservation

4.2.32 Hawaiian Home Land

class HawaiianHomeLand(***kwargs*)
Hawaiian Home Land

4.2.33 Alaska Native Village Statistical Area

class ANVSA(***kwargs*)
Alaska Native Village Statistical Area

4.2.34 Oklahoma Tribal Statistical Areas

class OTSA(***kwargs*)
Oklahoma Tribal Statistical Area

4.2.35 State Designated Tribal Statistical Areas

class SDTSA(***kwargs*)
State Designated Tribal Statistical Areas

4.2.36 Tribal Designated Statistical Areas

class TDSA(***kwargs*)
Tribal Designated Statistical Area

4.2.37 American Indian Joint-Use Areas

class AIJUA(***kwargs*)
American Indian Joint-Use Area

4.2.38 CombinedNECTA and Related

class CombinedNECTA(***kwargs*)
Combined New England City and Town Area

class NECTADivision(***kwargs*)
New England City and Town Area Division

class MetropolitanNECTA(***kwargs*)
Metropolitan New England City and Town Area

class MicropolitanNECTA(***kwargs*)
Micropolitan New England City and Town Area

4.2.39 Urban-related Geographical Areas

class UrbanGrowthArea(***kwargs*)
Urban Growth Area

class UrbanizedArea(***kwargs*)
Urbanized Area

class UrbanizedArea_2010(***kwargs*)
2010 Census Urbanized Area

class UrbanCluster(***kwargs*)
Urban Cluster

class UrbanCluster_2010(***kwargs*)
2010 Census Urban Cluster

4.2.40 Traffic Analysis Zone and Related

class TrafficAnalysisZone(***kwargs*)
Traffic Analysis Zone

class TrafficAnalysisDistrict(***kwargs*)
Traffic Analysis District

4.3 Census Geocoder Internals

4.3.1 Base Entity

class BaseEntity
Abstract base clase for geographic entities that may or may not be supported by the API.

abstract classmethod from_csv_record(*csv_record*)
Create an instance of the geographic entity from its CSV record.

Parameters **csv_record** (`list` of `str`) – The list of columns for the CSV record.

Returns An instance of the geographic entity.

Return type *GeographicEntity*

abstract classmethod from_dict(*as_dict*)
Create an instance of the geographic entity from its `dict` representation.

Parameters **as_dict** (`dict`) – The `dict` representation of the geographic entity.

Returns An instance of the geographic entity.

Return type *GeographicEntity*

classmethod from_json(*as_json*)
Create an instance of the geographic entity from its JSON representation.

Parameters **as_json** (`str`, `dict`, or `list`) – The JSON representation of the geographic entity.

Returns An instance of the geographic entity.

Return type *GeographicEntity*

abstract to_dict()
Returns a `dict` representation of the geographic entity.

Note: The `dict` representation matches the JSON structure for the US Census Geocoder API. This is a not-very-pythonic `dict` structure, but at least this ensures idempotency.

Returns `dict` representation of the entity.

Return type `dict`

to_json()
Returns a JSON representation of the geographic entity.

Note: The JSON representation matches the JSON structure for the US Census Geocoder API. This is a not-very-pythonic structure, but at least this ensures idempotency.

Returns `str` representation of the entity.

Return type `str`

abstract property entity_type
The type of geographic entity that the object represents. Supports either: `locations` or `geographies`.

Return type `str`

4.3.2 Geographic Entity

class GeographicEntity

Abstract base class for geographic entities that *are* supported by the API.

classmethod from_address(**args*, ***kwargs*)

Return data from an adddress, supplied either as a single *one-line address* or a *parametrized address*.

Parameters

- **one_line** (str / None) – A single-line address, e.g. `'4600 Silver Hill Rd, Washington, DC 20233'`. Defaults to None.
- **street_1** (str / None) – A street address, e.g. `'4600 Silver Hill Rd'`. Defaults to None.
- **street_2** (str / None) – A secondary component of a street address, e.g. `'Floor 3'`. Defaults to None.
- **street_3** (str / None) – A tertiary component of a street address, e.g. `'Apt. B'`. Defaults to None.
- **city** (str / None) – The city or town of a street address, e.g. `'Washington'`. Defaults to None.
- **state** (str / None) – The state or territory of a street address, e.g. `'DC'`. Defaults to None.
- **zip_code** (str / None) – The zip code (or zip code + 4) of a street address, e.g. `'20233'`. Defaults to None.
- **benchmark** (str) – The name of the *benchmark* of data to return. The default value is determined by the `CENSUS_GEOCODER_BENCHMARK` environment variable, and if that is not set defaults to `'Current'` which represents the current default benchmark, per the Census Geocoder API.

  Accepts the following values:

  – `'Current'` (default)

  – `'Census2020'`

- **vintage** (str) – The vintage of Census data for which data should be returned. The default value is determined by the `CENSUS_GEOCODER_VINTAGE` environment variable, and if that is not set defaults to `'Current'` which represents the default vintage per the Census Geocoder API.

  Acceptable values are dependent on the `benchmark` specified, as per the table below:

	BENCHMARKS	
	Current	Census2020
VINTAGES	Current	Census2020
	Census2020	Census2010
	ACS2019	
	ACS2018	
	ACS2017	
	Census2010	

- **layers** (str) – The set of geographic layers to return for the request. The default value is determined by the `CENSUS_GEOCODER_LAYERS` environment variable, and if that is not set defaults to `'all'`.

See also:

- *Geographies Benchmarks, Vintages, and Layers*

Note: If more than one address-related parameter are supplied, this method will assume that a *parametrized address* is provided.

Returns A given geographic entity.

Return type *GeographicEntity*

Raises

- ***NoAddressError*** – if no address information is supplied
- **EntityNotFoundError** – if no geographic entity was found matching the address supplied
- ***UnrecognizedBenchmarkError*** – if the `benchmark` supplied is not recognized
- ***UnrecognizedVintageError*** – if the `vintage` supplied is not recognized

classmethod from_batch(**args*, ***kwargs*)

Return geographic entities for a batch collection of inputs.

Parameters

- **file** (`str`) – The name of a file in CSV, XLS/X, DAT, or TXT format. Expects the file to have the following columns *without a header row*:
  - Unique ID
  - Street Address
  - City
  - State
  - Zip Code
- **benchmark** (`str`) – The name of the *benchmark* of data to return. The default value is determined by the `CENSUS_GEOCODER_BENCHMARK` environment variable, and if that is not set defaults to `'Current'` which represents the current default benchmark, per the Census Geocoder API.

  Accepts the following values:
  - `'Current'` (default)
  - `'Census2020'`
- **vintage** (`str`) – The vintage of Census data for which data should be returned. The default value is determined by the `CENSUS_GEOCODER_VINTAGE` environment variable, and if that is not set defaults to `'Current'` which represents the default vintage per the Census Geocoder API.

  Acceptable values are dependent on the `benchmark` specified, as per the table below:

	BENCHMARKS	
	Current	Census2020
VINTAGES	Current	Census2020
	Census2020	Census2010
	ACS2019	
	ACS2018	
	ACS2017	
	Census2010	

- **layers** (str) – The set of geographic layers to return for the request. The default value is determined by the CENSUS_GEOCODER_LAYERS environment variable, and if that is not set defaults to 'all'.

  See also:

  – *Geographies Benchmarks, Vintages, and Layers*

Returns A collection of geographic entities.

Return type list of *GeographicEntity*

Raises

- *NoFileProvidedError* – if no file_ is provided
- **FileNotFoundError** – if file_ does not exist on the filesystem
- *BatchSizeTooLargeError* – if file_ contains more than 10,000 records
- **EntityNotFoundError** – if no geographic entity was found matching the address supplied
- *UnrecognizedBenchmarkError* – if the benchmark supplied is not recognized
- *UnrecognizedVintageError* – if the vintage supplied is not recognized

classmethod from_coordinates(**args*, ***kwargs*)

Return data from a pair of geographic coordinates (longitude and latitude).

Parameters

- **longitude** (*numeric*) – The longitude coordinate.
- **latitude** (*numeric*) – The latitude coordinate.
- **benchmark** (str) – The name of the *benchmark* of data to return. The default value is determined by the CENSUS_GEOCODER_BENCHMARK environment variable, and if that is not set defaults to 'Current' which represents the current default benchmark, per the Census Geocoder API.

  Accepts the following values:

  – 'Current' (default)

  – 'Census2020'

- **vintage** (str) – The vintage of Census data for which data should be returned. The default value is determined by the CENSUS_GEOCODER_VINTAGE environment variable, and if that is not set defaults to 'Current' which represents the default vintage per the Census Geocoder API.

  Acceptable values are dependent on the benchmark specified, as per the table below:

	BENCHMARKS	
	Current	Census2020
VINTAGES	Current	Census2020
	Census2020	Census2010
	ACS2019	
	ACS2018	
	ACS2017	
	Census2010	

- **layers** (str) – The set of geographic layers to return for the request. The default value is determined by the CENSUS_GEOCODER_LAYERS environment variable, and if that is not set defaults to 'all'.

  See also:

  – *Geographies Benchmarks, Vintages, and Layers*

Note: If more than one address-related parameter are supplied, this method will assume that a *parametrized address* is provided.

Returns A given geographic entity.

Return type *GeographicEntity*

Raises

- ***NoAddressError*** – if no address information is supplied
- **EntityNotFound** – if no geographic entity was found matching the address supplied
- ***UnrecognizedBenchmarkError*** – if the benchmark supplied is not recognized
- ***UnrecognizedVintageError*** – if the vintage supplied is not recognized

abstract classmethod from_csv_record(*csv_record*)

Create an instance of the geographic entity from its CSV record.

Parameters **csv_record** (list of str) – The list of columns for the CSV record.

Returns An instance of the geographic entity.

Return type *GeographicEntity*

abstract classmethod from_dict(*as_dict*)

Create an instance of the geographic entity from its dict representation.

Parameters **as_dict** (dict) – The dict representation of the geographic entity.

Returns An instance of the geographic entity.

Return type *GeographicEntity*

classmethod from_json(*as_json*)

Create an instance of the geographic entity from its JSON representation.

Parameters **as_json** (str, dict, or list) – The JSON representation of the geographic entity.

Returns An instance of the geographic entity.

Return type *GeographicEntity*

inspect(*as_census_fields=False*)

Produce a list of the entity's properties that have values.

Parameters **as_census_fields** (bool) – If `True`, return property names as they appear in Census databases or the output of the Census Geocoder API. If `False`, return properties as they are defined on the **Census Geocoder** objects. Defaults to `False`.

Return type `list` of `str`

abstract to_dict()

Returns a `dict` representation of the geographic entity.

Note: The `dict` representation matches the JSON structure for the US Census Geocoder API. This is a not-very-pythonic `dict` structure, but at least this ensures idempotency.

Returns `dict` representation of the entity.

Return type `dict`

to_json()

Returns a JSON representation of the geographic entity.

Note: The JSON representation matches the JSON structure for the US Census Geocoder API. This is a not-very-pythonic structure, but at least this ensures idempotency.

Returns `str` representation of the entity.

Return type `str`

abstract property entity_type

The type of geographic entity that the object represents. Supports either: `locations` or `geographies`.

Return type `str`

CHAPTER

FIVE

ERROR REFERENCE

5.1 Handling Errors

5.1.1 Stack Traces

Because the **Census Geocoder** produces exceptions which inherit from the standard library, it leverages the same API for handling stack trace information. This means that it will be handled just like a normal exception in unit test frameworks, logging solutions, and other tools that might need that information.

5.2 Census Geocoder Errors

5.2.1 CensusGeocoderError (from ValueError)

class CensusGeocoderError
Base error raised by the **Census Geocoder**. Inherits from ValueError.

5.2.2 CensusAPIError (from CensusGeocoderError)

class CensusAPIError
Error raised when the Census Geocoder API returned an error.

5.2.3 ConfigurationError (from CensusGeocoderError)

class ConfigurationError
Error raised when a geocoding request was configured incorrectly.

5.2.4 UnrecognizedBenchmarkError (from ConfigurationError)

class UnrecognizedBenchmarkError
Error raised when a *benchmark* has been specified incorrectly.

5.2.5 UnrecognizedVintageError (from ConfigurationError)

class UnrecognizedVintageError
Error raised when a *vintage* has been specified incorrectly.

5.2.6 MalformedBatchFileError (from ConfigurationError)

class MalformedBatchFileError
Error raised when a batch file is structured improperly.

5.2.7 NoAddressError (from ConfigurationError)

class NoAddressError
Error raised when there was no address supplied with the request.

5.2.8 NoFileProvidedError (from ConfigurationError)

class NoFileProvidedError
Error raised when a batch file indicated in the request does not exist or cannot be read.

5.2.9 BatchSizeTooLargeError (from ConfigurationError)

class BatchSizeTooLargeError
Error raised when the size of a batch address file exceeds the limit of 10,000 imposed by the Census Geocoder API.

5.3 Census Geocoder Warnings

5.3.1 CensusGeocoderWarning (from UserWarning)

class CensusGeocoderWarning
Base warning raised by the **Census Geocoder**. Inherits from `UserWarning`.

CHAPTER

SIX

CONTRIBUTING TO THE CENSUS GEOCODER

Note: As a general rule of thumb, the **US Census Geocoder** applies **PEP 8** styling, with some important differences.

Branch	Unit Tests
latest	
v.0.5	
develop	

What makes an API idiomatic?

One of my favorite ways of thinking about idiomatic design comes from a talk given by Luciano Ramalho at Pycon 2016[5] where he listed traits of a Pythonic API as being:

- don't force [the user] to write boilerplate code
- provide ready to use functions and objects
- don't force [the user] to subclass unless there's a *very good* reason
- include the batteries: make easy tasks easy
- are simple to use but not simplistic: make hard tasks possible
- leverage the Python data model to:
  - provide objects that behave as you expect
  - avoid boilerplate through introspection (reflection) and metaprogramming.

Contents:

[5] https://www.youtube.com/watch?v=k55d3ZUF3ZQ

6.1 Design Philosophy

The **Census Geocoder** is meant to be a "beautiful" and "usable" library. That means that it should offer an idiomatic API that:

- works out of the box as intended,
- minimizes "bootstrapping" to produce meaningful output, and
- does not force users to understand how it does what it does.

In other words:

> Users should simply be able to drive the car without looking at the engine.

6.2 Style Guide

6.2.1 Basic Conventions

- Do not terminate lines with semicolons.
- Line length should have a maximum of *approximately* 90 characters. If in doubt, make a longer line or break the line between clear concepts.
- Each class should be contained in its own file.
- If a file runs longer than 2,000 lines…it should probably be refactored and split.
- All imports should occur at the top of the file.
- Do not use single-line conditions:

```
# GOOD
if x:
  do_something()

# BAD
if x: do_something()
```

- When testing if an object has a value, be sure to use `if x is None:` or `if x is not None`. Do **not** confuse this with `if x:` and `if not x:`.
- Use the `if x:` construction for testing truthiness, and `if not x:` for testing falsiness. This is **different** from testing:
  - `if x is True:`
  - `if x is False:`
  - `if x is None:`
- As of right now, because we feel that it negatively impacts readability and is less-widely used in the community, we are **not** using type annotations.

6.2.2 Naming Conventions

- `variable_name` and not `variableName` or `VariableName`. Should be a noun that describes what information is contained in the variable. If a `bool`, preface with `is_` or `has_` or similar question-word that can be answered with a yes-or-no.
- `function_name` and not `function_name` or `functionName`. Should be an imperative that describes what the function does (e.g. `get_next_page`).
- `CONSTANT_NAME` and not `constant_name` or `ConstantName`.
- `ClassName` and not `class_name` or `Class_Name`.

6.2.3 Design Conventions

- Functions at the module level can only be aware of objects either at a higher scope or singletons (which effectively have a higher scope).
- Functions and methods can use **one** positional argument (other than `self` or `cls`) without a default value. Any other arguments must be keyword arguments with default value given.

```
def do_some_function(argument):
  # rest of function...

def do_some_function(first_arg,
                     second_arg = None,
                     third_arg = True):
  # rest of function ...
```

- Functions and methods that accept values should start by validating their input, throwing exceptions as appropriate.
- When defining a class, define all attributes in `__init__`.

- When defining a class, start by defining its attributes and methods as private using a single-underscore prefix. Then, only once they're implemented, decide if they should be public.
- Don't be afraid of the private attribute/public property/public setter pattern:

```
class SomeClass(object):
  def __init__(*args, **kwargs):
    self._private_attribute = None

  @property
  def private_attribute(self):
    # custom logic which  may override the default return

    return self._private_attribute

  @setter.private_attribute
  def private_attribute(self, value):
    # custom logic that creates modified_value

    self._private_attribute = modified_value
```

- Separate a function or method's final (or default) `return` from the rest of the code with a blank line (except for single-line functions/methods).

6.2.4 Documentation Conventions

We are very big believers in documentation (maybe you can tell). To document the **US Census Geocoder** we rely on several tools:

Sphinx[1]

Sphinx[1] is used to organize the library's documentation into this lovely readable format (which is also published to ReadTheDocs[2]). This documentation is written in reStructuredText[3] files which are stored in `<project>/docs`.

Tip: As a general rule of thumb, we try to apply the ReadTheDocs[?] own Documentation Style Guide[4] to our RST documentation.

Hint: To build the HTML documentation locally:

1. In a terminal, navigate to `<project>/docs`.
2. Execute `make html`.

When built locally, the HTML output of the documentation will be available at `./docs/_build/index.html`.

---

[1] http://sphinx-doc.org
[2] https://readthedocs.org
[3] http://www.sphinx-doc.org/en/stable/rest.html
[4] http://documentation-style-guide-sphinx.readthedocs.io/en/latest/style-guide.html

Docstrings

- Docstrings are used to document the actual source code itself. When writing docstrings we adhere to the conventions outlined in **PEP 257**.

6.3 Dependencies

- Validator-Collection v1.5.0 or higher
- Backoff-Utils v1.0.1 or higher
- Requests v2.26 or higher

6.4 Preparing Your Development Environment

In order to prepare your local development environment, you should:

1. Fork the Git repository.
2. Clone your forked repository.
3. Set up a virtual environment (optional).
4. Install dependencies:

```
census-geocoder/ $ pip install -r requirements.txt
```

And you should be good to go!

6.5 Ideas and Feature Requests

Check for open issues or create a new issue to start a discussion around a bug or feature idea.

6.6 Testing

If you've added a new feature, we recommend you:

- create local unit tests to verify that your feature works as expected, and
- run local unit tests before you submit the pull request to make sure nothing else got broken by accident.

See also:

For more information about the **Census Geocoder** testing approach please see: *Testing the Census Geocoder*

6.7 Submitting Pull Requests

After you have made changes that you think are ready to be included in the main library, submit a pull request on Github and one of our developers will review your changes. If they're ready (meaning they're well documented, pass unit tests, etc.) then they'll be merged back into the main repository and slated for inclusion in the next release.

6.8 Building Documentation

In order to build documentation locally, you can do so from the command line using:

```
census-geocoder/ $ cd docs
census-geocoder/docs $ make html
```

When the build process has finished, the HTML documentation will be locally available at:

```
census-geocoder/docs/_build/html/index.html
```

Note: Built documentation (the HTML) is **not** included in the project's Git repository. If you need local documentation, you'll need to build it.

6.9 Contributors

Thanks to everyone who helps make the **Census Geocoder** useful:

- Chris Modzelewski (@insightindustry)

6.10 References

CHAPTER

SEVEN

TESTING THE CENSUS GEOCODER

Contents

7.1 Testing Philosophy

Note: Unit tests for the **Census Geocoder** are written using pytest[1] and a comprehensive set of test automation are provided by tox[2].

There are many schools of thought when it comes to test design. When building the **Census Geocoder**, we decided to focus on practicality. That means:

- **DRY is good, KISS is better.** To avoid repetition, our test suite makes extensive use of fixtures, parametrization, and decorator-driven behavior. This minimizes the number of test functions that are nearly-identical. However, there are certain elements of code that are repeated in almost all test functions, as doing so will make future readability and maintenance of the test suite easier.
- **Coverage matters…kind of.** We have documented the primary intended behavior of every function in the **SQLAthanor** library, and the most-likely failure modes that can be expected. At the time of writing, we have about 85% code coverage. Yes, yes: We know that is less than 100%. But there are edge cases which are almost impossible to bring about, based on confluences of factors in the wide world. Our goal is to test the key functionality, and as bugs are uncovered to add to the test functions as necessary.

[1] https://docs.pytest.org/en/latest/
[2] https://tox.readthedocs.io

7.2 Test Organization

Each individual test module (e.g. `test_validators.py`) corresponds to a conceptual grouping of functionality. For example:

- `test_validators.py` tests validator functions found in `census_geocoder/_validators.py`

Certain test modules are tightly coupled, as the behavior in one test module may have implications on the execution of tests in another. These test modules use a numbering convention to ensure that they are executed in their required order, so that `test_1_NAME.py` is always executed before `test_2_NAME.py`.

7.3 Configuring & Running Tests

7.3.1 Installing with the Test Suite

Installing via pip

From Local Development Environment

```
$ pip install census-geocoder[tests]
```

See also:

When you *create a local development environment*, all dependencies for running and extending the test suite are installed.

7.3.2 Command-line Options

The **Census Geocoder** does not use any custom command-line options in its test suite.

Tip: For a full list of the CLI options, including the defaults available, try:

```
census-geocoder $ cd tests/
census-geocoder/tests/ $ pytest --help
```

7.3.3 Running Tests

Entire Test Suite

Test Module

Test Function

```
tests/ $ pytest
```

```
tests/ $ pytest tests/test_module.py
```

```
tests/ $ pytest tests/test_module.py -k 'test_my_test_function'
```

7.4 Skipping Tests

Note: Because of the simplicity of the **Census Geocoder**, the test suite does not currently support any test skipping.

7.5 Incremental Tests

Note: The **Census Geocoder** test suite does support incremental testing, however at the moment none of the tests designed rely on this functionality.

A variety of test functions are designed to test related functionality. As a result, they are designed to execute incrementally. In order to execute tests incrementally, they need to be defined as methods within a class that you decorate with the `@pytest.mark.incremental` decorator as shown below:

```
@pytest.mark.incremental
class TestIncremental(object):
    def test_function1(self):
        pass
    def test_modification(self):
        assert 0
    def test_modification2(self):
        pass
```

This class will execute the `TestIncremental.test_function1()` test, execute and fail on the `TestIncremental.test_modification()` test, and automatically fail `TestIncremental.test_modification2()` because of the `.test_modification()` failure.

To pass state between incremental tests, add a `state` argument to their method definitions. For example:

```
@pytest.mark.incremental
class TestIncremental(object):
    def test_function(self, state):
        state.is_logged_in = True
        assert state.is_logged_in = True
    def test_modification1(self, state):
        assert state.is_logged_in is True
        state.is_logged_in = False
        assert state.is_logged_in is False
    def test_modification2(self, state):
        assert state.is_logged_in is True
```

Given the example above, the third test (`test_modification2`) will fail because `test_modification` updated the value of `state.is_logged_in`.

Note: `state` is instantiated at the level of the entire test session (one run of the test suite). As a result, it can be affected by tests in other test modules.

CHAPTER

EIGHT

RELEASE HISTORY

Contributors

- Chris Modzelewski (@insightindustry)

Contents

8.1 Release 0.1.0

- Initial public release.

CHAPTER

NINE

GLOSSARY

Benchmark The period in time when the geographic data was snapshotted for use / return by the Census Geocoder API.

Census Block The single smallest element in the *core geographic hierarchy* is the **Census Block**. This is the most granular geographical area for which the US Census Bureau reports data, and is the smallest geographic unit where data is available for 100% of its resident population.

Census Data This is information that is collected from the Constitutionally-mandated decennial census, which collects information from 100% of residents in the United States.

Centroid Latitude The latitude coordinate for the geometric center of a *geographic area*.

Centroid Longitude The longitude coordinate for the geometric center of a *geographic area*.

Internal Point Latitude The Census Bureau calculates an internal point (latitude and longitude coordinates) for each geographic entity. For many geographic entities, the internal point is at or near the geographic center of the entity. For some irregularly shaped entities (such as those shaped like a crescent), the calculated geographic center may be located outside the boundaries of the entity. In such instances, the internal point is identified as a point inside the entity boundaries nearest to the calculated geographic center and, if possible, within a land polygon.

Internal Point Longitude The Census Bureau calculates an internal point (latitude and longitude coordinates) for each geographic entity. For many geographic entities, the internal point is at or near the geographic center of the entity. For some irregularly shaped entities (such as those shaped like a crescent), the calculated geographic center may be located outside the boundaries of the entity. In such instances, the internal point is identified as a point inside the entity boundaries nearest to the calculated geographic center and, if possible, within a land polygon.

Forward Geocoding Also known as *geocoding*, a process that identifies a specific canonical location based on its street address.

Geocoding The act of determining a specific, canonical location based on some input data.

See also:

- *Forward Geocoding*
- *Reverse Geocoding*

Geography A geographical area. Corresponds to a *layer* and represented in the **Census Geocoder** as a `GeographicArea`.

Layer When working with the Census Geocoder API (particularly when *getting geographic area data*), you have the ability to control which *types* of geographic area get returned. These types of geographic area are called "*layers*". Which layers are available is ultimately determined by the *vintage* of the data you are retrieving.

See also:

- *Geographies in the Census Geocoder* > *Benchmarks, Vintages, and Layers*

One-line Address A physical / mailing address represented in a single line of text, like `'4600 Silver Hill Rd, Washington, DC 20233'`.

Parametrized Address An address that has been broken down into its component parts. Thus, a single-line address like `'4600 Silver Hill Rd, Washington, DC 20233'` gets broken down into:

- **STREET**: `'4600 Silver Hill Rd'`
- **CITY**: `'Washington'`
- **STATE**: `'DC'`
- **ZIP CODE**: `'20233'`

Reverse Geocoding A process that identifies a specific canonical location based on its precise geographic coordinates (typically expressed as latitude and longitude).

Sampled Data Data reported by the US Census Bureau that is derived from data collected from a subset of the resident population (i.e. from a surveyed sample of potential respondents).

Tigerline Tigerline and Shapefiles represent the GIS data that defines all of the features (places) and geographical areas (polygons) that comprise the mapping data for the Census Geocoder API.

Vintage The census or survey data that the geographic area meta-data returned by the Census Geocoder API is linked to, given that geographic area's *benchmark*.

CHAPTER

TEN

SQLATHANOR LICENSE

MIT License

Copyright (c) 2021 Insight Industry Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The **US Census Geocoder** is a Python library that provides Python bindings for the U.S. Census Geocoder API. It enables you to use simple Python function calls to retrieve Python object representations of geographic meta-data for the addresses or coordinates that you are searching for.

Warning: The **US Census Geocoder** is completely unofficial, and is in no way affiliated with the US Government or the US Census Bureau. We strongly recommend that you do business with them directly as needed, and simply provide this Python library as a facilitator for your programmatic interactions with the excellent services provided by the US Census Bureau.

Contents

CHAPTER

ELEVEN

INSTALLATION

To install the **US Census Geocoder**, just execute:

```
$ pip install census-geocoder
```

11.1 Dependencies

- Validator-Collection v1.5.0 or higher
- Backoff-Utils v1.0.1 or higher
- Requests v2.26 or higher

CHAPTER

TWELVE

WHY THE CENSUS GEOCODER?

In fulfilling its constitutional and statutory obligations, the US Census Bureau provides extensive data about the United States. They make this data available publicly through their website, through their raw data files, and through their APIs. However, while their public APIs provide great data, they are limited in both tooling and documentation. So to help with that, we've created the **US Census Geocoder** library.

The **Census Geocoder** library is designed to provide a Pythonic interface for interacting with the Census Bureau's Geocoder API. It is specifically designed to eliminate the scaffolding needed to query the API directly, and provides for simpler and cleaner function calls to return *forward geocoding* and *reverse geocoding* information. Furthermore, it exposes Python object representations of the outputs returned by the API making it easy to work with the API's data in your applications.

12.1 Key Census Geocoder Features

- **Easy to adopt**. Just install and import the library, and you can be *forward geocoding* and *reverse geocoding* with just two lines of code.
- **Extensive documentation**. One of the main limitations of the Geocoder API is that its documentation is scattered across the different datasets released by the Census Bureau, making it hard to navigate and understand. We've tried to fix that.
- Location Search
  - Using Geographic Coordinates (reverse geocoding)
  - Using a One-line Address
  - Using a Parametrized Address
  - Using Batched Addresses
- Geography Search
  - Using Geographic Coordinates (reverse geocoding)
  - Using a One-line Address
  - Using a Parametrized Address
  - Using Batched Addresses
- Supports all available *benchmarks*, *vintages*, and *layers*.
- Simplified syntax for indicating benchmarks, vintages, and layers.
- No more hard to interpret field names. The library uses simplified (read: human understandable) names for location and geography properties.

12.2 The US Census Geocoder vs Alternatives

While we're partial to the **US Census Geocoder** as our primary means of interacting with the Census Geocoder API, there are obviously alternatives for you to consider. Some might be better for your use specific use cases, so here's how we think about them:

Roll Your Own

Census Geocode

CensusBatchGeocoder

geocoder/geopy

The Census Geocoder API is a straightforward RESTful API. Which means that you can just execute your own HTTP requests against it, retrieve the JSON results, and work with the resulting data entirely yourself. This is what I did for years, until I got tired of repeating the same patterns over and over again, and decided to build the **Census Geocoder** instead.

For a super-simple use case, probably the most expedient way to do it. But of course, more robust use cases would require your own scaffolding with built-in retry-logic, object representation, error handling, etc. which becomes non-trivial.

Why not use a library with batteries included?

Tip: When to use it?

In practice, I find that rolling my own solution is great when it's an extremely simple use case, or a one-time operation (e.g. in a Jupyter Notebook) with no business logic to speak of. It's a "quick-and-dirty" solution, where I'm trading rapid implementation (yay!) for less flexibility/functionality (boo!).

Considering how easy the **Census Geocoder** is to use, however, I find that I never really roll my own scaffolding when working with the Census Geocoder API.

The Census Geocode library is fantastic, and it was what I had used before building the **Census Geocoder** library. However, it has a number of significant limitations when compared to the **US Census Geocoder**:

- Results are returned as-is from the Census Geocoder API. This means that:
  - Results are essentially JSON objects represented as `dict`, which makes interacting with them in Python a little more cumbersome (one has to navigate nested `dict` objects).
  - Property/field names are as in the original Census data. This means that if you do not have the documentation handy, it is hard to intuitively understand what the data represents.
- The library is licensed under GPL3, which may complicate or limit its utilization in commercial or closed-source software operating under different (non-GPL) licenses.
- The library requires you to remember / apply a lot of the internals of the Census Geocoder API as-is (e.g. benchmark vintages) which is complicated given the API's limited documentation.
- The library does not support custom *layers*, and only returns the default set of layers for any request.

The **Census Geocoder** explicitly addresses all of these concerns:

- The library uses native Python classes to represent results, providing a more pythonic syntax for interacting with those classes.
- Properties / fields have been renamed to more human-understandable names.
- The **Census Geocoder** is made available under the more flexible *MIT License*.

- The library streamlines the configuration of *benchmarks* and *vintages*, and provides extensive *documentation*.
- The library supports any and all layers supported by the Census Geocoder API.

Tip: When to use it?

Census Geocode has one advantage over the **US Census Geocoder**: It has a CLI.

I haven't found much use for a CLI in the work I've done with the Census Geocoder API, so have not implemented it in the **US Census Geocoder**. Might add it in the future, if there are enough feature requests for it.

Given the above, it may be worth using Census Geocode instead of the **Census Geocoder** if you expect to be using a CLI.

The CensusBatchGeocoder is a fantastic library produced by the team at the Los Angeles Times Data Desk. It is specifically designed to provide a fairly pythonic interface for doing bulk geocoding operations, with great pandas serialization / de-serialization support.

However, it does have a couple of limitations:

- **Stale / Unmaintained**? The library does not seem to have been updated since 2017, leading me to believe that it is stale and unmaintained. There are numerous open issues dating back to 2020, 2018, and 2017 that have seen no activity.
- **No benchmark/vintage/layer support**. The library does not support the configuration of *benchmarks*, *vintages*, or *layers*.
- **Limited error handling**. The library has somewhat limited error handling, judging by the issues that have been reported in the repository.
- **Optimized for bulk operations**. The design of the library has been optimized for geocoding in bulk, which makes transactional one-off requests cumbersome to execute.

The **Census Geocoder** is obviously fresh / maintained, and has explicitly implemented robust error handling, and support for *benchmarks*, *vintages*, and *layers*. It is also designed to support bulk operations *and* transactional one-off requests.

Tip: When to use it?

CensusBatchGeocoder has one advantage over the **US Census Geocoder**: It can serialize results to a pandas `DataFrame` seamlessly and simply.

This is a useful feature, and one that I have added/pinned for the **US Census Geocoder**. If there are enough requests / up-votes on the issue, I may extend the library with this support in the future.

Given all this, it may be worth using CensusBatchGeocoder instead of the **US Census Geocoder** if you expect to be doing a lot of bulk operations using the default benchmark/vintage/layers.

---

geocoder and geopy are two of my favorite geocoding libraries in the Python ecosystem. They are both inherently pythonic, elegant, easy to use, and support most of the major geocoding providers out there with a standardized / unified API.

So at first blush, one might think: Why not just use one of these great libraries to handle requests against the Census Geocoder API?

Well, the problem is that neither geocoder nor geopy supports the Census Geocoder API as a geocoding provider. So… you can't just use either of them if you specifically want US Census geocoding data.

Secondly, both the geocoder and geopy libraries are optimized around providing coordinates and feature information (e.g. matched address), which the Census Geocoder API results go beyond (and are not natively compatible with).

So really, if you want to interact with the Census Geocoder API, the **Census Geocoder** library is designed to do exactly that.

Tip: When to use them?

If you only need relatively simple, coordinate/feature-focused *forward* or *reverse* geocoding from a different provider than the US Census Bureau, and you specifically do not need data from the US Census Bureau.

CHAPTER

THIRTEEN

HELLO WORLD AND BASIC USAGE

13.1 1. Import the Census Geocoder

```
import census_geocoder as geocoder
```

13.2 2. Execute a Coding Request

13.2.1 Using a One-line Address

```
location = geocoder.location.from_address('4600 Silver Hill Rd, Washington, DC 20233')

geography = geocoder.geography.from_address('4600 Silver Hill Rd, Washington, DC 20233')
```

13.2.2 Using a Parametrized Address

```
location = geocoder.location.from_address(street_1 = '4600 Silver Hill Rd',
                                          city = 'Washington',
                                          state = 'DC',
                                          zip_code = '20233')

geography = geocoder.geography.from_address(street_1 = '4600 Silver Hill Rd',
                                            city = 'Washington',
                                            state = 'DC',
                                            zip_code = '20233')
```

13.2.3 Using Batched Addresses

```
# Via a CSV File
location = geocoder.location.from_batch('my-batched-address-file.csv')

geography = geocoder.geography.from_batch('my-batched-address-file.csv')
```

13.2.4 Using Coordinates

```
location = geocoder.location.from_coordinates(latitude = 38.845985,
                                              longitude = -76.92744)


geography = geocoder.geography.from_coordinates(latitude = 38.845985,
                                                longitude = -76.92744)
```

13.3 3. Work with the Results

13.3.1 Work with Python Objects

```
location.matched_addresses[0].address

>> 4600 SILVER HILL RD, WASHINGTON, DC 20233
```

CHAPTER

FOURTEEN

QUESTIONS AND ISSUES

You can ask questions and report issues on the project's Github Issues Page

CHAPTER

FIFTEEN

CONTRIBUTING

We welcome contributions and pull requests! For more information, please see the *Contributor Guide*. And thanks to all those who've already contributed:

- Chris Modzelewski (@insightindustry)

CHAPTER

SIXTEEN

TESTING

We use TravisCI for our build automation and ReadTheDocs for our documentation.

Detailed information about our test suite and how to run tests locally can be found in our *Testing Reference*.

CHAPTER

SEVENTEEN

LICENSE

The **Census Geocoder** is made available under an *MIT License*.

CHAPTER

EIGHTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

c

t

INDEX

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Z