17-480/780: Project Report

API Design - Fall 2020 - Team08

CDC Wonder API

Description / Documentation of original API | Project Repository

---

Section 1: Introduction and API Description

CDC WONDER is a comprehensive on-line public health information system of the Centers for Disease Control and Prevention (CDC) and developed to place timely, action-oriented information in the hands of public health professionals.[1] The CDC Wonder API is a query tool from the CDC that provides access to a collection of online databases for the analysis of public health data. These databases collect statistics from the United States such as births, detailed or compressed mortality rates, and causes of death.

To access the data in these databases, the API exposes an endpoint[2] which users of the API can access through an HTTP POST request which contains an XML document

[1] "CDC WONDER: a comprehensive on-line public health ...." https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1694976/. Accessed 15 Dec. 2020.
[2] "https://wonder.cdc.gov/controller/datarequest/[database ID]". "CDC WONDER." https://wonder.cdc.gov/. Accessed 15 Dec. 2020.

that specifies all of the parameters needed to define the specific data that the user wishes to receive from the database.

Section 2: Original API Shortcomings

The CDC Wonder API has some shortcomings that become readily apparent when formatting this XML parameter document and when parsing the XML response. These limitations take many forms: there is excessive, unnecessary verbosity both in specifying unnecessary parameters and in boilerplate code always necessary to parse and visualize the response data, there are hidden limitations and undocumented cases in which the API fails to return the expected data, there are dozens of parameters with cryptic names that are impossible to make sense of and make the API very unapproachable and the API's response requires a lot of extra processing to get the data into a usable format. We've outlined a few examples to expand on the above arguments. The sample code we are referring to for this section is available in the appendix section of this report.

Non-Essential Verbosity and Excessive Boilerplate

The CDC's Wonder API mandates a broadly unnecessary degree of unimportant work from any user who hopes to employ the CDC's vast swathes of data. The user has to deal with verbose request parameter definitions and add a lot of boilerplate necessary for processing the data returned in the server response, verbosity dominates the API and overwhelms users.

Evaluating the formatting of the CDC provided XML request file[3], we notice that significant sections of the expected parameter lists provide no additional information and serve only as clutter that lazily forces users to do simple work that the server receiving the API call ought to be handling. Provided below is a snippet of the sample code in python that sends the request to the API endpoint. The v_parameters header section below contains *over 20* parameters that are specified as their default values of either the empty string or *All*, while only a *single* parameter offers non-default data that will actually modify the response data.

Furthermore, there are some parameters that *must* be specified and cannot be

[3] "Example 1 Request - CDC Wonder." https://wonder.cdc.gov/wonder/help/API-Examples/D76_Example1-req.xml. Accessed 15 Dec. 2020.

changed, which is very hard to figure out unless users spend a significant amount of time debugging.

```
v_parameters = {
    "V_D76.V1": "*All*",         # Year/Month
    "V_D76.V10": "*All*",        # Census Regions
    "V_D76.V11": "*All*",    # 2006 Urbanization
    "V_D76.V12": "*All*",    # ICD-10 130 Cause List (Infants)
    "V_D76.V17": "*All*",    # Hispanic Origin
    "V_D76.V19": "*All*",    # 2013 Urbanization
    "V_D76.V2": "*All*",         # ICD-10 Codes
    "V_D76.V20": "*All*",    # Autopsy
    "V_D76.V21": "*All*",    # Place of Death
    "V_D76.V22": "*All*",    # Injury Intent
    "V_D76.V23": "*All*",    # Injury Mechanism
    "V_D76.V24": "*All*",    # Weekday
    "V_D76.V25": "*All*",    # Drug/Alcohol Induced Causes
    "V_D76.V27": "*All*",        # HHS Regions
    "V_D76.V4": "*All*",     # ICD-10 113 Cause List
    "V_D76.V5": ["15-24", "35-44"], # Ten-Year Age Groups
    "V_D76.V51": "*All*",    # Five-Year Age Groups
    "V_D76.V52": "*All*",    # Single-Year Ages
    "V_D76.V6": "00",        # Infant Age Groups
    "V_D76.V7": "*All*",     # Gender
    "V_D76.V8": "*All*",     # Race
    "V_D76.V9": ""           # State/County
}
```

Finally, the API requires users to both build an XML document to specify all the input parameters and parse an XML to get the data from the response, which leads to a lot of additional boilerplate code to access the data.

Unintuitive Parameter and Parameter Value Names

In addition to having to specify all of the parameters, the API also uses string parameters that are specified through confusingly-named codes rather than their actual names. Not only is using strings error prone since misspellings are easy to miss, but figuring out which codes correspond to which parameters is very hard, as can be seen in the example below.

```
f_parameters = {
    "F_D76.V1": ["*All*"], # year/month
    "F_D76.V10": ["*All*"], # Census Regions - dont change
# How is a user or a later viewer of this code supposed to
intuitively know what these codes are (without significant
Googling)?
    "F_D76.V2": ["K00-K92"], # ICD-10 Codes
    "F_D76.V27": ["*All*"], # HHS Regions - dont change
    "F_D76.V9": ["*All*"] # State County - dont change
}
```

Additionally, when filtering by options, like race for example, the user is expected to provide the codes for those values (which are often unintuitive alphanumeric IDs) instead of the race itself. So, if a user wanted to filter by Asian-American and Asian-Pacific Islander or Alaska Native races, they would have to change the "`V_D76.V8`" parameter from "`*All*`" to `["1002-5", "A-PI"]`. Easy enough, right?

Lack of Documentation

There is a lack of substantial official API documentation to apply to most use cases. For example, it isn't mentioned that *All* needs to be specified as a "filter" for a category, if you don't want the results to be filtered. The user also has no way to figure out what parameters the user needs to specify in the first place. The only way to figure out what structure of query works or doesn't is through trial and error.

Section 3: Introducing *CDCWonderPy!*

To address the issues brought to light in the previous section, we designed and implemented a new python API that serves as a cleaner interface to interact with the CDC Wonder API. Due to this project's timeframe, it would be impossible to address all the datasets that the CDC provides access to such as Birth databases, Compressed Mortality databases, etc. As such, we focus on improving the API for the CDC Wonder API Detailed Mortality database, as it had the most documentation on the CDC Wonder website, and we found more sample code that used this database than the others. By focusing on improving one database, we hope that this implementation provides a blueprint for how other Wonder API endpoints might be improved in the future.

Our API consists of two main classes. The `Request` class has many functions that set the required grouping and formatting options for the data. It also helps users filter the data by race, gender, cause of death, age groups, etc. The `Response` class is a wrapper around the Wonder API's response that provides utilities to format the response into more accessible and developer friendly formats such as a Pandas dataframe or a 2D list, as well as the raw XML text as before.

In addition to these two main classes, the API also provides various helper classes to facilitate the input of certain parameters. These include a `Dates` class that standardizes the use of both a range of dates and a single date so that they can be specified as a list of inputs to the request, various different enum classes that define all of the mappings from intuitive parameter names and values to the codes that the Wonder HTTP endpoint uses and an `ICD10Code` class that simplifies the specification of various granular or general causes of death[4] as inputs.

In general, we focused on reducing user boilerplate as much as possible. We iterated over the names of the functions and argument types, as well as the functionality that we offer to our users. Throughout our design process, we considered common use cases, while providing methods to deal with the uncommon use cases. We used enums extensively to help our user choose the right filters, and increase readability of client code. We also wrote client code after every iteration of our ideation and design process to figure out the best API we could, and that drove our design decisions into an end result that we're very happy with :)

In the following sections, we will take you through the evolution of each of the functions of our API and highlight significant design decisions along with our rationale for going with them.

Section 4: Dissecting *CDCWonderPy*

Because the Wonder API is a REST API, the first major design decision for our implementation was to wrap around this REST endpoint and provide a set of classes in Python that the user could use to define inputs and outputs easily, without having to worry about the way that the requests are sent out.

Request Class

The first of these classes would be one to represent a request, called `Request`. This

[4] We're sorry about the inherent morbidity of our API :'(

class represents a mutable instance of a request prior to being sent off to the Wonder database's HTTP endpoint. Upon creation, it defines all of the same default input parameters as the web-based GUI. This alone makes the API very powerful, since users can now simply instantiate the object and send it to get the most general version of the data possible, all in a single line: `response = Request().send()`

Furthermore, this `Request` class specifies various different methods to modify the input parameters by changing the instance's internal state. These include methods to change group-by parameters, demographics, weekdays, place of death, and more. Because all of these input parameters are optional, we felt that this was a perfect case to use a builder class to construct this request object. However, because the request object itself is only used to get a response, we figured that it was almost like it was the builder class and the send method acted as the actual building step, and the immutable object that it "built" is the response (discussed below). The result is that all of these state-modifying methods can be chained together, so that specifying some quick filters or formatting parameters can be quickly and intuitively done by users:

```
r = Request().gender(Gender.Male).group_by(Grouping.Age).send()
```

If users wanted to stick to their pythonic roots and break up each call on a separate line, that works too!

```
r = Request()
r.gender(Gender.Male)
r.group_by(Grouping.Age)
r.send()
```

Group By

In order to format and provide structure to a query's response data, users have the ability to group their results by a series of data groupings corresponding to columns within the server DB. In the original CDC WONDER API, these grouping methods existed as "B Parameters" that, without proper documentation, required the input of at least one such parameter and at most 5. These parameters took the form of XML-formatted string values whose meanings were obfuscated from user with incoherent formatting (e.g. setting `"B_1": "D76.V2-level2"` was how users could group data by ICD-10 Code Sub Chapters). First, we hid this unnecessary complexity from users by creating an intuitive set of public-facing Enums (`Grouping`), each aptly named to clarify their intended grouping purpose and removing the unnecessary burden of researching the parameter values corresponding to the grouping column of their choice. Each call to this `group_by` method overrides all previous calls and gracefully

enables users to reset any groupings they do not want included in the actual sent request. Although we additionally intended to eliminate the almost arbitrarily selected upper limit of five group_by parameters, this was a server-side limitation to which we had no access to modify. Furthermore, calling this method allows for immediate feedback on potential errors that would arise from making the request such as asking for grouping by Ten Year Age Groups when only Single Year Age Groups were inputted.

```
request.group_by(Grouping.Gender, Grouping.Weekday)
```

```
>> Gender        HispanicOrigin      Deaths        Population   Crude Rate Per 100,000
0  Female      Hispanic or Latino   1305299.0     4.69351e+08                278.107
1  Female  Not Hispanic or Latino  23962179.0     2.62584e+09                912.552
2  Female              Not Stated     57157.0  Not Applicable         Not Applicable
3    Male      Hispanic or Latino   1614037.0     4.86044e+08                332.076
4    Male  Not Hispanic or Latino  23546322.0      2.5074e+09                939.075
5    Male              Not Stated     82780.0  Not Applicable         Not Applicable
```

Setters, Getters, & Attributes

We had initially named our functions set_race, set_gender, set_XYZ. This was when we were still considering adding getters for users to be able to obtain the state of the request object that the user was working on. However, we later decided to have an attributes object instead on which the user could call various functions, and allow the request object to have methods such as “race”, “gender”. The difference in the readability of client code was evident! Consider the following versions.

Before	After

`r = Request().set_gender(Gender.Male).set_hispanic_origin(HispanicOrigin.All)`
 	
`r.get_gender()`
	`r = Request().gender(Gender.Male).hispanic_origin(HispanicOrigin.All`
	
	`r.attributes().gender()`

However, we then moved onto interesting discussions regarding the state of the attributes object returned by `r.attributes()` (above). Did we want an object returned that would be a “snapshot” of the current state, or did we want an object that “tracked” the request object, by updating its state everytime the request state was updated? Before we could come to a consensus, we eliminated the attributes object entirely because we realized we didn’t have as many use cases for this functionality, and we’d be adding unnecessary conceptual weight. If there is a demand for this functionality, we can always add it in in subsequent versions of this API.

Request Filtering Features

Moving on to some of the filtering options provided by the API, we decided to have one function per filtering option. Since the api provides filtering by demographic information including race, gender, weekday, hispanic origin, age groups, but also by other information such as cause of death, autopsy present, and many more, we have a good number of functions to filter the response. We believe this was better than having one function that did a bunch of filtering, as it allows for clear understandable client code, and easy to understand functionality and documentation. We decided to have enums to specify each of the options a user could filter by. For example, a user can choose to filter by “Male”, “Female” or “All” gender types. The most convenient options to represent these 3 choices is an enum. We followed the same pattern to filter by hispanic origin, race and weekday. This makes the client code more readable. Earlier in this report we showed you sample code that filtered by race in an unclear and unintuitive fashion. To refresh your memory, if a user wanted to filter by Asian-American and Asian-Pacific Islander or Alaska Native races, they would have to change the “`V_D76.V8`” parameter from “`*All*`” to `["1002-5", "A-PI"].*` :(((* . Now however, users can do something as simple as

```
r = Request().race(Race.asian_or_pacific_islander,
black_or_african_american)
```

We considered taking in a list of enum options for each of these filters instead of variable arguments (*args in python). In other words, we were debating between

```
r = Request().hispanic_origin(HispanicOrigin.not_stated,
HispanicOrigin.hispanic_latino)
```

and

```
r = Request().hispanic_origin([HispanicOrigin.not_stated,
HispanicOrigin.hispanic_or_latino])
```

We decided to stick with the first approach because it made client code look nicer and we used the “pythonic way” of passing in arguments… by making use of *args when appropriate!

One downside to python compared to strongly typed languages is the possibility of the user providing incorrect types to functions at runtime. To ward against this, we raise TypeError exceptions whenever possible, to inform the user of incorrect types being passed in. Additionally, we also raise ValueExceptions if we don’t get the right number

of arguments.

Initially, we also allowed users to filter by “region” and “urbanization”. There were a couple of interesting design decisions behind choosing to go with our resulting function declaration.
Since users can filter by HHS region, census region, or state, we initially had an enum to represent the “type of region filtering” they wanted to perform. This made our code look like this

```
r = Request().region(RegionType.state, State.WA, State.PA)
```

However, we realized that we would want the onus of figuring out the type of region filtering performed to fall on the implementation, and not on the API users. Therefore, the API then evolved to -

```
r = Request().region(State.WA, State.PA)
```

If the users pass in regions of conflicting region types for filtering, that is, if they pass in State.WA and CensusRegion.EastRegion, then the API would’ve shower them with a type error!

Unfortunately, we later realized that the api endpoint doesn’t allow filtering by location and urbanization, even though the GUI version does. Therefore, we obliterated these functions and documented their doom in the class documentation.

Age Groups

Users can also filter the database based on the various age groups affected by each incident data point. Previously, these age groups were structured in Ten Year, Five Year, and Single Year buckets where Ten Year groupings followed format (5-14, 15-24, …, 75-84), Five Year groupings followed format (5-9, 10-14, …, 95-99), and Single Year groupings followed format (1,2,3, …, 98, 99). If you wanted to request all data on users from ages 35-64, inclusive, you would need to individually input each age grouping manually as a parameter for the tag corresponding to Ten Year Age Groupings.

```
“V_D76.V5”: “35-44”,
“V_D76.V5”: “45-54”,
“V_D76.V5”: “55-64”
```

However, if you additionally wanted to group results by a more granular Five Year Age

Grouping interval, you would need to input the same set of data in the form of 5-year granularity and input that as fields to “V_D76.V51” instead. There is no carry over of parameters. There is no flexibility. Moreover, the values of only one category of age parameters actually is taken into account by the API, a selection that is made by the value passed into the “O_age” parameter.

Overall, this full sequence of age group parameter debacles is too complex for users to quickly pick up and comfortably use. For these reasons, we built several structures to wrap this functionality. Users now have the ability to input a series of Ages parameters that can either consist of single ages or a range of ages.

```
request.age_groups(Ages.Single(40), Ages.Range(41,54),
Ages.Range(64-69))
```

We had initially considered continuing with Enums for each of these age ranges, but found it incredibly cumbersome for a user to input each Enum for, for example, 5 year age ranges between ages 5 and 64 (`5-9, 10-14, …, 55-59,60-64`) as opposed to simply passing in a single input for the range from 5-64 (`Ages.Range(5,64)`). At this point, without inputting any additional parameters, users are able to flexibly include all ages they care to consider in their query, with only slight restrictions. Due to the deep complexity involved in wrapping such a terribly designed portion of the WONDER API, we were unable to allow for ages that fell within edge cases of <1 years and 100+ years. Moreover, due to limitations brought on by the server-side implementation of the WONDER API, users are asked to more carefully consider the age data they input when attempting to additionally group data results by age group sizes greater than the Single Year option. If users wish to group their data in terms of Ten Year or Five Year Age Groupings, they must format their age_groups inputs to contain only age ranges covered by the WONDER API’s string parameters.

```
request.age_groups(Age.Range(5, 64))
request.group_by(Grouping.FiveYearAgeGroups) # Successful Call

request.age_groups(Age.Range(7, 66))
request.group_by(Grouping.FiveYearAgeGroups)) # ValueError
raised
```

Overall, we have built a very intuitive medium for representing the wide majority of user requests that would make their way through this API. However, due to the inherently problematic nature of the WONDER API’s own design, we were unable to support all possible edge cases allowed by the original API while still maintaining our primary

priority of facilitating API usage and code creation. Although there are several easy to implement API designs that could have accounted for these rare edge cases, we opted for a design that significantly improved the quality of the wide majority of use cases while raising exceptions and properly documenting edge case restrictions.

Dates

Let's address filtering of data by Dates or Date Ranges. The api is flexible in that it allows you to specify disjoint ranges of dates, or specific dates upto the granularity of a month, to filter by. This posed some interesting challenges when it came to designing a method that would take into account both these kinds of inputs for filtering. We initially considered taking any number of disjoint dates (with the granularity of a month or a year) either as a tuple of "dates" (dates being represented by a year and optionally a month) symbolizing a range of dates, or a single date, showing all days in that month or year. This would make the client code look like this….

```
r = Request().date(("6/2010", "7/2011"), 2013)
```

The first argument represents a range of dates starting from and including June 2010, upto and including July 2011. The second argument represents all days in the year 2013.

While this worked, we realized that we wanted to make client code look better. We then decided to make an object that could represent single months, single years as well as a range of months. This is what our final client code looks like!

```
r = Request().dates(Dates.range(Year(2001), Year(2003)),
Dates.single(YearAndMonth(2005, 4)),
Dates.range(YearAndMonth(2003, 6), Year(2004)))
```

We added a `Dates` object that allows users to specify a range through `Dates.range()` and specify a single month through `Dates.single(YearAndMonth(<year>, <month>))`, as well as a single year through `Dates.single(Year(<year>))`.

ICD-10 & Cause of Death

Finally, we address filtering by cause of death. Cause of death options can be specified by providing ICD10 codes[5]. ICD10 codes are complicated in that you have chapters of ICD10 codes, that have subchapters, which inturn have more layers of depth until you

[5] "Conversion - ICD-10 Codes: Lookup." https://icdcodelookup.com/icd-10/codes. Accessed 15 Dec. 2020.

reach a single code like A00.0 (from A00 - B99). In short, there are multiple levels of ICD10 hierarchy that we weren't sure we'd be able to cover, so we started off with allowing just the broadest ranges outlined in the gui. However, after much debate, we decided that it was worth our project time to come up with a fleshed out ICD10 api. We now allow filtering based on ICD10 codes to varying levels of granularity, by providing the `ICD10Code` class. For users that know the exact ICD10 codes they want to filter by, we've provided them (Example, A00.0). What's cool is that we also allow users to filter by A00 (which contains A00.0, A00.1, A00.9) and A00-A09! We also provide an additional layer of abstraction, allowing users to filter by A00_B99. This way, we account for a vast majority of our users, who want to cover causes of death to varying levels of specificity.

In addition to that, the `ICD10Code` class also provides a few utility methods, such as fuzzy string description matching and description best match. While we allow filtering by code names as described in the previous paragraph, we also allow filtering by the description of the code! For example, users can search for the ICD10 code corresponding to "ear" infections, or can search for exactly "Cholera due to vibrio cholerae".

Exception Handling

The request is then sent using the instance's `send` method, which hangs until the server responds. If the response contains an error, the API parses the response for the exception reason and throws a custom exception to the user, letting them know why the server responded with an error.

One thing to note is that some of these state-changing methods have certain known exceptions that can occur. For example, you cannot specify both "All" and any additional specific option (e.g. "All" and "Female" for gender filtering) in any of these methods. As such, for any of these known exceptions that we were able to identify, we raise them locally so that the user can easily pinpoint the issue without waiting to hear from the server.

Response Class

If there was no issue from the server, the send method returns an instance of the `Response` class. This class is an immutable representation of the response returned from the Wonder HTTP endpoint and as such is not meant to be instantiated by the user. Instead, the Response class serves as a tool for formatting the response data in a variety of high utility formats. The API currently includes methods to return the response data as the raw XML in string format, as a 2D list, and as a Pandas dataframe with

labeled columns. The user can also specify a custom parsing function to parse the XML string. This provides a huge amount of extensibility since the user can now cleanly parse it into whatever form they want! The built-in parsing methods serve as an excellent blueprint for how the user could create their own data transformation functions. Additionally, this class could easily be extended to support many other built-in data formats if demand arises.

Thread Safety

In the design process for the Response class, there was a deliberate decision to enforce its immutability, even while the Request class was mutable. Fundamentally, API designers ought to maintain immutability unless beneficial otherwise, for the purposes of improving code quality, supporting concurrent operation, and minimizing error-prone user behavior. The Response object is a likely target for concurrent functionality, with common use-cases involving multiple threads operating on its response data upon generation. Moreover, the Response object would offer no sizable benefit to users if it were mutable instead. On the other hand, the Request object is more suited for repeated use with small changes made between calls. For example, a user may want to make multiple queries to the CDC database, the first query filtering results by a specific date range, a second query filtering by a particular ethnic background, and a third query grouping the results by a different set of columns. To support functionality like this without forcing the user to extensively copy-paste their functionality, the Request object was implemented with some degree of mutability, while the Response object did not merit such freedom.

For this reason, we notify our users that while we cannot promise thread safety on `Request`, we can do for our `Response` class.

Section 5: Testing

We implemented two kinds of testing. We had end to end testing implemented, for which we used the GUI as our golden file reference. We constructed a request through our api, and submitted the same request on the gui. We exported the result from the gui into a test file, and programmatically extracted the result from the api response on our object in python. We compared the two versions to see if they were equal! We repeated this process twice, with different configurations of filters and group by options.

We also added testing for our Response formatting functionality. We “mock” a response from the server, and ask the response object to generate the corresponding data_frame and 2d list objects. We verify that they are constructed as expected!

Section 6: Future Goals for this API

The CDC databases are queried by thousands of people around the world. Now more than ever, we believe that a refactoring of its public API endpoints is necessary. We hope that the work we've done here can serve as a blueprint for how the rest of the API and the other databases can be improved.

That being said, there are a few issues that we didn't get the chance to address, given this project's time constraints:

- The GUI version of this api allows for specifying a few advanced data aggregation options, covered by its measures section. We didn't incorporate some of these advanced options, such as choosing to display 95% confidence interval vs standard error on data. We also don't allow users to specify any additional "rate-options".

Section 7: Life Before and After our API!

We're very happy overall with how the API turned out. Say that you wanted to see how many of the database records were of people who had an autopsy. Here's how your coding session would play out if you were using the original API:

1. You find out that the CDC has a public API endpoint that could give you all the data you're looking for. Life is good.
2. You look at the documentation, you see hundreds of weirdly-named parameters until you see that you need to set "V_D76.V20". Life isn't as good.
3. You spend hours figuring out the inputs and formatting the input XML, nothing works so you keep debugging the server's error responses, you see that the issue was that you weren't sending the mandatory parameter that you can't change. Life sucks.
4. You spend a lot more time debugging until you finally get some useful data. Hurray! Oh wait, you still have to parse the response…
5. You parse the response and you finally get your data, but at that point you've lost all interest in what you originally set out to do and are now questioning your life decisions.

Now, with our API, things look very different:

1. You import our API.
2. You look at the documentation. You see that everything is set by default, life is great.
3. You write a single line of code to get your request:

```
Request().autopsy(Autopsy.Yes).send()
```

4. You get your response object, you print it as a pandas dataframe, and you go on with your day!

Jokes aside, we hope this api is a decent upgrade to python users as compared to directly using the CDC WONDER API.

Section 8: Lessons Learned

Designing an API is easy. Designing a good API is hard. Designing a great API is very hard, and takes time and many many iterations. Having four people designing an API gives good perspective on the different options available to support a certain functionality, and the best of the discovered options to proceed with :). Designing with client code in mind ensures that we are better serving the users of the API… which is the entire point of an API to begin with.

We also realized the importance of the iterative design process behind developing the different components of an api may it be naming, function parameters, classes used, functionality provided and client code quality.

Once you design an API poorly, you make it incredibly difficult for anyone, including yourself, to fix your mistakes. More than likely, by the time the designers of this API had realized they had made a series of terrible API design decisions, if they cared so at all, it would have been far too late for them to be able to fix their mistakes. After you deploy your API to the public and people begin to utilize it, you cannot simply go back and redesign the exact same API because there exists code out there that depends on your previous design.

For this reason, at some point all you can do is create wrappers for your terrible API. However, if said API is terrible enough, the process of creating these wrappers can be far more difficult and complex than designing the API properly in the first place! If the designers had spent just another few days thinking through their decisions (or hiring a CMU 17-780 Alumnus to do so), there would not need to be a brigade of students and professors alike dedicating their time toward breaking down and building this back up. However, it was certainly one fine learning experience. And for this, we thank you, CDC developers, Josh, and our incredibly hard-working course staff.

Section 9: Attribution

All of us worked very closely with each other and Josh and Spencer to complete this project. We all worked on the documentation for each of our parts as well as the class documentation. As for each of the functions of the API, we *roughly* split the responsibilities as follows:

Request Internal State & Request handling - Diego
Demographics, Location, & Chronology - Pratyusha
Group By, Age Grouping, & Response Handling - Ike
Chronology & Cause of Death - Joel
Testing, Project Report, & Documentation - All

Our “issues list” is essentially our Meeting Notes document. We documented our design decisions, todos, meeting updates and changes based on ideation sessions and meetings with Josh and Spencer.

We would also like to sincerely thank Josh and Spencer for meeting with us *multiple times* to address our concerns and guide us throughout this semester.

Section 9: *Appendix*

Sample Code[6]

```
# One of many examples where users must needlessly append
"*None*" or "*All*" to every XML request field. Cumbersome.
b_parameters = {
    "B_1": "D76.V1-level1",
    "B_2": "D76.V8",
    "B_3": "*None*",
    "B_4": "*None*",
    "B_5": "*None*"
}

m_parameters = {
    "M_1": "D76.M1",   # Deaths, must be included
    "M_2": "D76.M2",   # Population, must be included
    "M_3": "D76.M3",   # Crude rate, must be included
    #"M_31": "D76.M31",        # Standard error (crude rate)
    #"M_32": "D76.M32"         # 95% confidence interval (crude
rate)
    "M_41": "D76.M41", # Standard error (age-adjusted rate)
    "M_42": "D76.M42"  # 95% confidence interval (age-adjusted
rate)
}

i_parameters = {
    "I_D76.V1": "*All* (All Dates)",  # year/month
    "I_D76.V10": "*All* (The United States)", # Census Regions -
dont change
    "I_D76.V2": "K00-K92 (Diseases of the digestive system)", #
ICD-10 Codes
    "I_D76.V27": "*All* (The United States)", # HHS Regions -
dont change
    "I_D76.V9": "*All* (The United States)" # State County -
dont change
}

v_parameters = {
```

[6] "alipphardt/cdc-wonder-api: This repo provides ... - GitHub." https://github.com/alipphardt/cdc-wonder-api. Accessed 15 Dec. 2020.

```
    "V_D76.V1": "",         # Year/Month
    "V_D76.V10": "",        # Census Regions
    "V_D76.V11": "*All*",   # 2006 Urbanization
    "V_D76.V12": "*All*",   # ICD-10 130 Cause List (Infants)
    "V_D76.V17": "*All*",   # Hispanic Origin
    "V_D76.V19": "*All*",   # 2013 Urbanization
    "V_D76.V2": "",         # ICD-10 Codes
    "V_D76.V20": "*All*",   # Autopsy
    "V_D76.V21": "*All*",   # Place of Death
    "V_D76.V22": "*All*",   # Injury Intent
    "V_D76.V23": "*All*",   # Injury Mechanism and All Other
Leading Causes
    "V_D76.V24": "*All*",   # Weekday
    "V_D76.V25": "*All*",   # Drug/Alcohol Induced Causes
    "V_D76.V27": "",        # HHS Regions
    "V_D76.V4": "*All*",    # ICD-10 113 Cause List
    "V_D76.V5": ["15-24", "25-34", "35-44"], # Ten-Year Age
Groups
    "V_D76.V5": "*All*",
    "V_D76.V51": "*All*",   # Five-Year Age Groups
    "V_D76.V51": ["45-49", "50-54", "55-59", "60-64"],
    "V_D76.V52": "*All*",   # Single-Year Ages
    "V_D76.V6": "00",       # Infant Age Groups
    "V_D76.V7": "*All*",    # Gender
    "V_D76.V8": "*All*",    # Race
    "V_D76.V9": ""          # State/County
}

o_parameters = {
    "O_V10_fmode": "freq",    # Use regular finder and ignore v
parameter value
    "O_V1_fmode": "freq",     # Use regular finder and ignore v
parameter value
    "O_V27_fmode": "freq",    # Use regular finder and ignore v
parameter value
    "O_V2_fmode": "freq",     # Use regular finder and ignore v
parameter value
    "O_V9_fmode": "freq",     # Use regular finder and ignore v
parameter value
    "O_aar": "aar_std",       # age-adjusted rates
    "O_aar_pop": "0000",      # population selection for
```

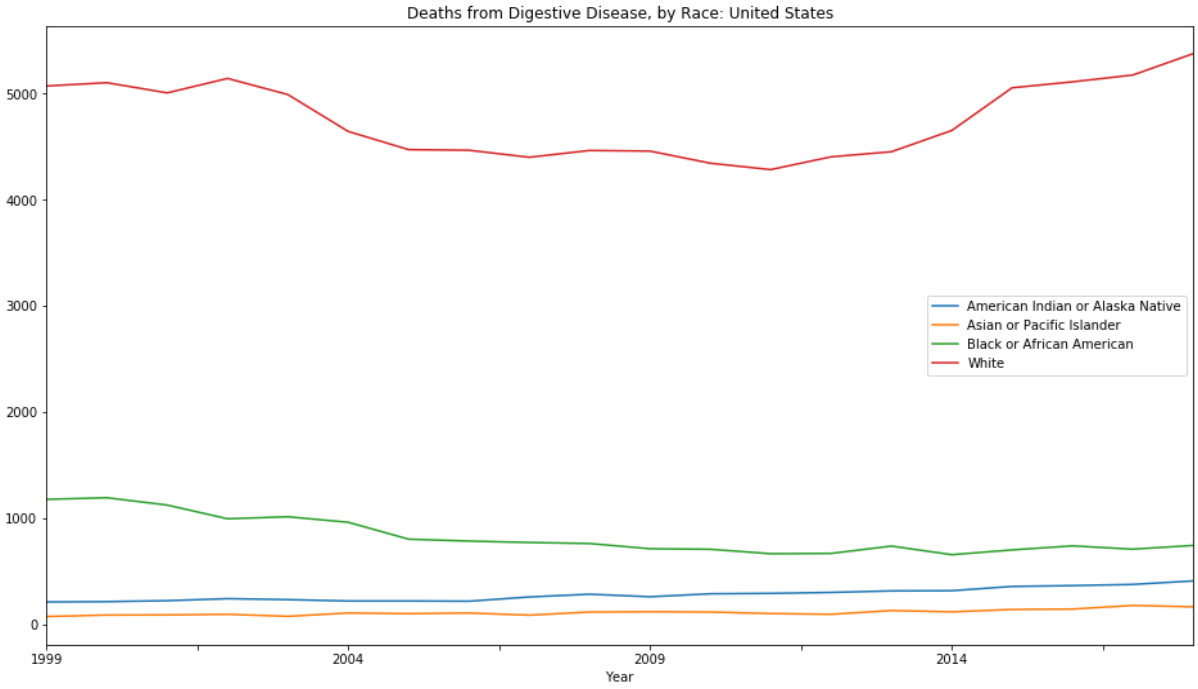
```
age-adjusted rates
    "O_age": "D76.V5",        # select age-group (e.g. ten-year,
five-year, single-year, infant groups)
    "O_javascript": "on",     # Set to on by default
    "O_location": "D76.V9",   # select location variable to use
(e.g. state/county, census, hhs regions)
    "O_precision": "1",       # decimal places
    "O_rate_per": "100000",   # rates calculated per X persons
    "O_show_totals": "false",  # Show totals for
    "O_timeout": "300",
    "O_title": "Digestive Disease Deaths, by Age Group",    #
title for data run
    "O_ucd": "D76.V2",        # select underlying cause of death
category
    "O_urban": "D76.V19"      # select urbanization category
}

vm_parameters = {
    "VM_D76.M6_D76.V10": "",        # Location
    "VM_D76.M6_D76.V17": "*All*",   # Hispanic-Origin
    "VM_D76.M6_D76.V1_S": "*All*",  # Year
    "VM_D76.M6_D76.V7": "*All*",    # Gender
    "VM_D76.M6_D76.V8": "*All*"     # Race
}

# These never change, but must always be included...
misc_parameters = {
    "action-Send": "Send",
    "finder-stage-D76.V1": "codeset",
    "finder-stage-D76.V1": "codeset",
    "finder-stage-D76.V2": "codeset",
    "finder-stage-D76.V27": "codeset",
    "finder-stage-D76.V9": "codeset",
    "stage": "request"
}



# Far too much boilerplate to simply format the request.
def createParameterList(parameterList):
    """Helper function to create a parameter list from a
```

```
dictionary object"""

    parameterString = ""

    for key in parameterList:
        parameterString += "<parameter>\n"
        parameterString += "<name>" + key + "</name>\n"

        if isinstance(parameterList[key], list):
            for value in parameterList[key]:
                parameterString += "<value>" + value +
"</value>\n"
        else:
            parameterString += "<value>" + parameterList[key] +
"</value>\n"

        parameterString += "</parameter>\n"

    return parameterString

xml_request = "<request-parameters>\n"
xml_request += createParameterList(b_parameters)
xml_request += createParameterList(m_parameters)
xml_request += createParameterList(f_parameters)
xml_request += createParameterList(i_parameters)
xml_request += createParameterList(o_parameters)
xml_request += createParameterList(vm_parameters)
xml_request += createParameterList(v_parameters)
xml_request += createParameterList(misc_parameters)
xml_request += "</request-parameters>"

import requests

# Step out of the 1950's, a single API call for this breadth of
information is unnecessary and amalgamates the numerous purposes
for requesting the data into one convoluted hodgepodge.
url = "https://wonder.cdc.gov/controller/datarequest/D76"
response = requests.post(url, data={"request_xml": xml_request,
"accept_datause_restrictions": "true"})

# There is no understanding of WHY an error took place, only
```

```
that it SOME arbitrary error was present.
if response.status_code == 200:
    data = response.text
else:
    print("something went wrong")

# BeautifulSoup library facilitates parsing of XML response
import bs4 as bs

# This library facilitates 2-dimensional array operations and
visualization
import pandas as pd



# All of this boilerplate just to format the response data into
a Pandas Dataframe?? A user should not need to do all this work
that the API ~absolutely~ could and should be doing for them.
def xml2df(xml_data):
    """ This function grabs the root of the XML document and
iterates over
        the 'r' (row) and 'c' (column) tags of the data-table
        Rows with a 'v' attribute contain a numerical value
        Rows with a 'l attribute contain a text label and may
contain an
        additional 'r' (rowspan) tag which identifies how many
rows the value
        should be added. If present, that label will be added to
the following
        rows of the data table.

        Function returns a two-dimensional array or data frame
that may be
        used by the pandas library."""

    root = bs.BeautifulSoup(xml_data,"lxml")
    all_records = []
    row_number = 0
    rows = root.find_all("r")

    for row in rows:
```

```
        if row_number >= len(all_records):
            all_records.append([])

        for cell in row.find_all("c"):
            if 's' in cell.attrs:
          try:

     all_records[row_number].append(float(cell.attrs["v"].replac
     e(',','')))
          except ValueError:
            all_records[row_number].append(cell.attrs["v"])

          else:
            if 'r' not in cell.attrs:

all_records[row_number].append(cell.attrs["l"])
            else:
                    for row_index in
range(int(cell.attrs["r"])):
                  if (row_number + row_index) >=
               len(all_records):
                     all_records.append([])
                     all_records[row_number +
               row_index].append(cell.attrs["l"])
                  else:
                     all_records[row_number +
               row_index].append(cell.attrs["l"])
        row_number += 1
    return all_records

data_frame = xml2df(data)

df = pd.DataFrame(data=data_frame, columns=["Year", "Race",
"Deaths", "Population", "Crude Rate", "Age-adjusted Rate",
"Age-adjusted Rate Standard Error"])

df.head()

# Load matplotlib for plotting and instruct jupyter to display
figures inline
from matplotlib import pyplot as plt
```

```
%matplotlib inline

# Group total number of deaths by year
df.groupby(by=['Year']).sum()['Deaths'].plot(title='Deaths from
Digestive Disease: United States');
fig = plt.gcf()
fig.set_size_inches(16, 9)

# Create mult-line chart for deaths by race

# Store figure and axis for shared plot
fig, ax = plt.subplots()

# Store labels for all race groups
labels = []

# For each group in the groupby object, grab the 'Race' label
and create a line plot for it
for key, grp in df.groupby(['Race']):
    ax = grp.plot(ax=ax, kind='line', x='Year', y='Deaths')
    labels.append(key)

# Set the labels for each line using the group labels
lines, _ = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')

# Configure chart size and title
fig = plt.gcf()
fig.set_size_inches(16, 9)
plt.title("Deaths from Digestive Disease, by Race: United
States");
```


Deaths from Digestive Disease, by Race: United States
5000
4000
3000
2000
1000
0
1999
2004
2009
2014
Year
American Indian or Alaska Native
Asian or Pacific Islander
Black or African American
White