ICS H32 Fall 2024
Project 3: *From the Faraway Nearby*

Due date and time: *Wednesday, November 13, 11:59pm*

Background

We saw in the previous project that our Python programs are capable of connecting to the "outside world" around them — to other programs running on the same machine, or even to other programs running on different machines in faraway places. This is a powerful thing for a program to be able to do, because it is no longer limited to taking its input from a user or from a file stored locally; its input is now potentially anything that's accessible via the Internet, making it possible to solve a vast array of new problems and process a much broader collection of information. Once you have the ability to connect your programs to others, a whole new world opens up. Suddenly, the idea that you should be able to write a program that combines, say, Google search queries, the Internet Movie Database, and your favorite social network to find people who like movies similar to the out-of-the-ordinary ones you like doesn't seem so far-fetched.

But we also saw that getting programs to share information is tricky, for (at least) two reasons. Firstly, there's a software engineering problem: A protocol has to be designed that both programs can use to have their conversation. Secondly, there's a social problem: If the same person (or group of people) isn't writing both programs, it's necessary for them to agree on the protocol ahead of time, then to implement it. This second problem has a potentially catastrophic effect on our ability to make things work — how could you ever convince a large entity like Google to agree to use a bespoke protocol just to communicate with a program you wrote?

In practice, both of these problems are largely solved by the presence of *standards*, such as those defined by the World Wide Web Consortium and the Internet Engineering Task Force. Standards help by providing detailed communication protocols whose details have already been hammered out, with the intention of handling the most common set of needs that will arise in programs. This eliminates the need to design one's own protocol; where the standard protocols will suffice, which is more often than you might think, you can use them as-is. Further, using standards allows programs to be

combined in arbitrary ways; as long as two programs support the same protocol, they've taken a big step toward being able to interoperate with each other, so having many programs implementing the same standard protocol is a substantial improvement over having the same programs communicate using one-off techniques. What's more, standard protocols often have standard implementations, so that you won't have to implement the details yourself as you did in the previous project. For example, Python has built-in support for a number of standard Internet protocols, including HTTP (HyperText Transfer Protocol, the protocol that your browser uses to download web pages) among others.

At first blush, HTTP doesn't seem all that important. It appears to be a protocol that will allow you to write programs that download web pages (i.e., that allow you to write programs that play the same role that web browsers do). But it turns out that HTTP is a lot more important than that, since it is the protocol that underlies a wide variety of traffic on the Internet, limited not only to the conversation that browsers have with web servers in order to download a web page, but encompassing the way many applications — including those without any user interface at all — communicate. HTTP underlies a growing variety of program-to-program communications using web protocols, where web sites or other software systems communicate directly with what are broadly called *web services*, fetching data and also making changes to it. This is why you can post messages to a web site like Facebook using either their web site, a client application on your laptop, or a smartphone app; all of these applications use the same protocol to communicate with Facebook's service, differing only in the form of user interface they provide.

Fortunately, since HTTP support (as well as support for its more secure cousin, HTTPS) is built directly into Python, we can write programs that use these web services without having to handle low-level details of the protocol, though there are some details that you'll need to be familiar with if you want to use the provided implementation effectively. We'll be discussing some of these details in lecture soon, and these will be accompanied by a code example, which will give you some background in the tools you'll need to solve these kinds of problems in Python.

This project gives you the opportunity to explore a small part of the vast sea of possibilities presented by web APIs and web services. You'll likely find that you spend a fair amount of your time in this project understanding the web APIs you'll need — being able to navigate technical documentation, experiment with another system, and gradually build an understanding of it is a vital skill in building real software — and that the amount of code you need might not be as much as you expect when you first read the project write-up. As always, work incrementally rather than trying to work on the entire project all at once; there is partial credit available for a partial solution, as long as the portions that you've finished are stable and correct. When you're done, you'll have taken a valuable step toward being able to build Python programs that interact with web services, which opens up your ability to write programs for yourself that are real and useful.

Additionally, you'll get what might be your first experience with writing classes in Python, which will broaden your ability to write clean, expressive Python programs, a topic we'll continue revisiting and refining throughout the rest of this course. Along with that, you'll learn about why it can be a powerful technique to write multiple, similar classes in a way that leaves them intentionally identical in at least one aspect of how they behave.

The problem

Unless we're planning on spending some period of time indoors in a climate-controlled environment, part of how we'll plan that time involves knowing what kind of weather we expect wherever we plan to be. As a kid growing up in the 1980s, the most up-to-date way to get that kind of information was television or radio; for heavily populated areas, you might have been able to find an automated service that you could call using a (landline) phone. Nowadays, the Internet provides a valuable resource to help us to monitor weather forecasts on demand, obtaining forecasts for anywhere from anywhere. In your work on this project, you'll write a program that can answer a question similar to the following: What is the hottest it will feel near Bren Hall in Irvine, California over the next 48 hours?

To do that, we'll need some information that we won't have at our fingertips; it's not our ambition to build weather sensors and place them in the locations for which we'd like a forecast, after all. But thanks to the ubiquitous Internet of today, we'll be able to obtain and use (free of charge) information that will allow us to answer a question like this without ever leaving the house. What we'll need are two things.

- A collection of weather stations that gather weather-related information and produce accurate forecasts of upcoming weather all over the United States. We don't need the weather stations, of course, but we need their output.
- A geocoding service that can tell us things like "Where is Bren Hall in Irvine, California?" or "What's the street address at this latitude and longitude?"

Given the ability to obtain answers to those kinds of questions and use them as input to our program, the rest of the problem is reduced to interpreting that input appropriately and performing the right calculations on it.

Because we're building a program in a problem domain that's new to us, though, we'll need to know some things about it. We don't need to become experts in weather forecasting or the intricacies of geographic algorithms and mapping, but we need to know enough about those things to be able to build what we seek to build. When we build programs, we're in the automation business, but we have to know something about what we're automating, even if we don't have to know everything.

Temperature

Temperature scales

While temperature is a fairly universal concept, you've likely learned before that there are multiple *temperature scales* that are commonly used to report it. Depending on what problem you're solving or the preferences of the audience to whom you're communicating a result, you might choose one scale or another. When we measure air temperature as part of a weather forecast, there are two common choices, so if we're trying to report weather results to people, we might want to offer either one according to the preferences of those people.

- *Fahrenheit*, which is by far the most common choice in the United States.
- *Celsius*, which is by far the most common choice in most other parts of the world.

Fortunately, converting from a temperature in one of these scales to the other requires only a straightforward bit of arithmetic, which you may have learned at some point in the past.

- If you have a number of degrees Fahrenheit, you can obtain the number of degrees Celsius by subtracting 32, then multiplying the result by 5/9.
- If you have a number of degrees Celsius, you can obtain the number of degrees Fahrenheit by multiplying by 9/5, then adding 32 to the result.

The "feels like" temperature

When we check a weather forecast, it's usually because we want to know how to prepare ourselves for being outside in that weather. How will it feel? What kind of clothing should I wear, and what additional clothing should I bring with me? To answer these questions, we'll want to know the air temperature, though it's important to realize that this isn't enough information to answer them properly. At higher temperatures, humidity can make it feel a lot hotter than it might feel otherwise. At lower temperatures, wind can make it feel a lot colder than it might feel otherwise. So, what we really want to know is the combination of these effects: Given the air temperature, humidity, and wind speed, how will it feel to be outside? We often see this reported as a *"feels like" temperature*.

While there is not a single agreed-upon formula for calculating a "feels like" temperature, we'll need to agree on one for the purposes of this project, so we'll lean on science, which provides two approaches that we can combine to good effect.

- The *heat index* can be calculated for higher air temperatures. This tells us how humidity impacts us, which is mainly a consideration when temperatures are higher. For that reason, we'll only use this formula when the air temperature is at least 68°F.
- The *wind chill index* can be calculated for lower air temperatures. This tells us how wind makes it feel colder than it would otherwise, which is an effect that mainly impacts us when temperatures are low and at least a minimal amount of wind is blowing. Consequently, we'll only use this formula when the air temperature is no more than 50°F and the wind speed is over three miles per hour.

Putting these ideas together, the inputs to our formula are as follows.

- We say that T is the air temperature, measured in degrees Fahrenheit.
- We say that H is the relative humidity, measured as a percentage (i.e., we'd represent H as 55, as opposed to 0.55, if the relative humidity is 55%).
- We say that W is the wind speed, measured in miles per hour.

Given these inputs, you'll perform the following calculation to determine a "feels like" temperature, by summing up a sequence of values determined from these inputs.

When...	*The "feels like" temperature is...*	*Which is the sum of...*
$T \geq 68°F$	Heat Index	-42.379 $2.04901523T$ $10.14333127H$ $-0.22475541TH$ $-0.00683783T^2$ $-0.05481717H^2$ $0.00122874T^2H$ $0.00085282TH^2$ $-0.00000199T^2H^2$

$T \le 50°F$ and $W > 3$	Wind Chill	$35.74 + 0.6215T - 35.75W^{0.16} + 0.4275TW^{0.16}$
Otherwise	Air Temperature	T

Finding out more

If you're interested in reading more about these formulas and where they came from, you can follow the links below to articles that describe them in more detail, though these details aren't particularly important for our work here. (This is one of the things you have to decide when you're working in a new problem domain: How much time do you want to spend, especially early on, understanding the broader context in which your work fits? Spend too much time and you won't be able to become productive soon enough. Spend too little time and you won't be able to discuss your work intelligently with the non-technical people who are impacted by your work. So, you gauge your curiosity, apply your time management skills, and make your best decision.)

- Fahrenheit (Wikipedia)
- Celsius (Wikipedia)
- Heat Index (National Weather Service)
- Wind Chill (Wikipedia)

It's worth noting that you may find other formulas online for solving similar kinds of problems — for example, the question of how a particular combination of temperature, humidity, and wind "feels" is one whose answer has some gray areas — though you'll want to implement the formulas described above regardless of what other research you do or what other formulas you find, so that the output of your program will match what is expected.

Latitudes, longitudes, and geocoding

Before you get too much farther, if you don't about how the latitude and longitude system works — don't feel bad if you don't, but you do need to understand this in order to finish this project! — take a look at the link below:

- Geographic coordinate system (Latitude and longitude) (Wikipedia)

In paticular, note the limits on allowable latitudes and longitudes, as well as the difference between North and South latitude and between West and East longitude. And note, too, that latitude and longitude, generally, don't work the same way, so once you've understood one, you'll still need to be sure you've wrapped your mind around the other. There aren't a lot of details, but if you haven't thought about them in a while — or if you've never seen them before — it's worth taking a few minutes to get your understanding sorted out before continuing.

What is geocoding?

The word *geocoding* sounds like some kind of programming technique, but it's actually something else: It's a process for converting the descriptions of places on the Earth into their locations and back again. In other words, it allows us to answer questions such as these.

- What is the latitude and longitude where Bren Hall in Irvine, California is located?
- What is located at latitude 33.674381°N and longitude 117.865975°W?

The first of those questions is what we'd call *forward geocoding* (i.e., taking the description of a location and turning it into geographic coordinates). The second is what we'd instead call *reverse geocoding* (i.e., taking geographic coordinates and describing what's there).

Of course, answering questions like these requires an enormous amount of data that we don't have, so it won't be up to us to determine these answers; instead, we'll obtain them online as we need them.

Where will we get our data?

While we'll be implementing some calculations of our own, the most meaningful input to our program will need to be obtained online, which raises the question of where we're going to get the information and how we're going to make sense out of it.

Hourly weather forecasts from the National Weather Service

The National Weather Service is a United States government agency that provides weather-related information, such as historical data, forecasts, and warnings. Among its services is a web API that offers real-time weather forecasts, including the *hourly forecasts* that we'll be using in our work in this project, which specify a handful of data points that forecast the upcoming weather on an hour-by-hour basis. Forecasts are only available in areas covered by the National Weather Service, which means we won't expect to find any forecasts available outside the United States; consequently, we won't expect our program to be able to answer questions about those locations not covered by the National Weather Service. (Our program is only as good as the APIs it relies on, in other words.)

The National Weather Service API includes documentation that describes its use, so your first step is taking a look through it. Based on what you find there, see if you can construct URLs that allow you to find an hourly forecast as near to 33.64324045°N and 117.84185686276017°W as possible. Don't worry if it takes a little while, but do spend some time working on that problem before you try to reach out to the National Weather Service API from your program; you can't use tools that you don't understand how to use, and you especially don't want to use services belonging to others until you understand them well enough to do so within the boundaries of what they permit.

- National Web Service API documentation

The National Web Service API is capable of returning information in a variety of formats, but we'll need to agree on what format we'll be using — because, as you'll see in the next section of the write-up, we'll need to know what format your program can handle, so we can test it properly — so we'll need to agree to always ask for an answer in the `application/geo+json` format. (As of this writing, that format is the default, which will help when you do browser-based experimentation, but it's still best for your program not to rely on that not changing.)

Geocoding via Nominatim's API

Nominatim is a web API that provides geocoding services using an open set of map data called OpenStreetMap. Nominatim requires no API key, though there are still some restrictions on its use, which we'll discuss a little later in this write-up.

Specifically, we'll be interested in using Nominatim for two things:

- Forward geocoding, which means that we want to take a description such as `Bren Hall, Irvine, CA` and find out its latitude and longitude.

- Reverse geocoding, which means that we have a latitude and longitude, such as 33.5935341°N and 117.874846°W, and we want a description of what's there.

Nominatim's API has fairly extensive documentation that describes its use, so you'll want to take a look through that to understand the services it provides and how to access them. Similar to how you experimented with the National Weather Service API, see if you can construct URLs that find the answers to the two examples above.

- Nominatim API documentation

Nominatim's API is capable of returning information in a variety of formats, but we'll need to agree on what format we're using — because, as you'll see in the next section of the write-up, we'll need to know what format your program can handle, so we can test it properly — so we'll need to agree to always pass this query parameter in the URLs given to Nominatim's API, even if there are other options available:

- `format=jsonv2`

Attribution requirements

Nominatim has an *attribution* requirement, which is to say that they require us to generate output in our programs that specify that we've taken some of our data from their APIs. Not only for reasons of legality, but also for educational reasons (i.e., learning how to take seriously the legal aspects of the work that we're doing), we'll follow those requirements.

The National Weather Service does not have such a requirement specifically, but we'll nonetheless respect their work enough to print an attribution message crediting them for their data, as well.

Any time our program uses any data from Nominatim or the National Weather Service, we'll say so in our program's output. And we'll only generate that attribution when we actually used one or both of the APIs.

Testing without the APIs

Working on a project that depends on an external API, such as this one, you'll face a couple of challenges that you may not have faced before.

- Your ability to test the program — or even to run it and see its output — is at least partly dependent on the performance of the API. If the API isn't functioning properly, or if you aren't connected to the Internet, your program won't function properly either. But when you're building a program, it's good to be able to tell the difference between a program that isn't working because it's broken in some way and one that's working fine but dependent on something outside of it that's not working.
- Testing a program requires knowing what the output of a program is supposed to be. But if you're testing a program using real-time weather data as its input, your program's output changes every time the weather does, and your ability to test is only as varied as the weather (e.g., you may not be able to test summertime conditions in the dead of winter).

For these reasons, your program will need a way to obtain its information from files stored locally, instead of reaching out to APIs. This will allow you to test your program with known-good data, which you'll mostly want to do, except when you're specifically working on the parts of the program where you're reaching out to the APIs. In the next sections of this write-up, you'll see how we'll make that possible.

The program

Your program will read a sequence of lines of input from the Python shell that configure its behavior, then generate and print some output consistent with that configuration. The general goal of the program is this: Given a *target location* and weather-related queries, show a description of the closest location for which weather data is available, then display the answer to those weather-related queries for that location. (That's a mouthful, so you'll want to read that sentence a few times; there's a lot going on there. Read further, too, and you'll see an example that will help to clarify.)

The input

The first thing your program does is read several lines of input that describe the job you want it to do. Your program should not print any prompts to the user; it should just blindly read this input, expecting that the user understands how to use the program already.

- The first line of input will be in one of two formats:
  - `TARGET NOMINATIM location`, where **location** is any arbitrary, non-empty string describing the target location. For example, if this line of input said `TARGET NOMINATIM Bren Hall, Irvine, CA`, the target of

our analysis is Bren Hall on the campus of UC Irvine. The word `NOMINATIM` indicates that we'll use Nominatim's API to determine the precise location (i.e., the latitude and longitude) of our target point.
    - `TARGET FILE path`, where **path** is the path to a file stored locally, containing the result of a previous call to Nominatim. The file needs to exist. The expectation is the file will contain data in the same format that Nominatim would have given you, but will allow you to test your work without having to call the API every time — important, because Nominatim imposes limitations on how often you can call into it, and because this could allow you to make large parts of the program work without having hooked up the APIs at all.
- The second line of input will be in one of two formats:
    - `WEATHER NWS`, which specifies that we'll use the National Weather Service API to obtain hourly weather forecasts.
    - `WEATHER FILE path`, where **path** is the path to a file stored locally, containing the result of a previous call to the National Weather Service's API for obtaining an hourly weather forecast. The file needs to exist, but will allow you to test your work without depending on the National Weather Service API (or to see your expected results change as the weather does).
- The third line of input specifies the first *weather query* that the user would like answered, and will be in one of the following formats:
    - `TEMPERATURE AIR scale length limit`, which means that the user would like to see the *air temperature*.
        - **scale** indicates the *temperature scale* in which to report the result, with `F` meaning Fahrenheit and `C` meaning Celsius.
        - **length** is a positive integer indicating the number of hours into the future for which the query is being made (e.g., `24` means "over the next 24 hours").
        - **limit** indicates whether the user is interested in seeing a maximum (by specifying `MAX`) or a minimum (by specifying `MIN`) over the specified number of hours.
        - So, for example, `TEMPERATURE AIR F 12 MAX` means that the user would like to see the maximum air temperature over the next 12 hours, reported in degrees Fahrenheit.
    - `TEMPERATURE FEELS scale length limit`, which means that the user would like to see the *"feels like" air temperature*.
        - **scale** indicates the *temperature scale* in which to report the result, with `F` meaning Fahrenheit and `C` meaning Celsius.

        - **length** is a positive integer indicating the number of hours into the future for which the query is being made (e.g., `24` means "over the next 24 hours").
        - **limit** indicates whether the user is interested in seeing a maximum (by specifying `MAX`) or a minimum (by specifying `MIN`) over the specified number of hours.
    - `HUMIDITY length limit`, which means that the user would like to see the *relative humidity*, reported as a percentage.
        - **length** is a positive integer indicating the number of hours into the future for which the query is being made (e.g., `24` means "over the next 24 hours").
        - **limit** indicates whether the user is interested in seeing a maximum (by specifying `MAX`) or a minimum (by specifying `MIN`) over the specified number hours.
    - `WIND length limit`, which means that the user would like to see the *wind speed*, reported in miles per hour.
        - **length** is a positive integer indicating the number of hours into the future for which the query is being made (e.g., `24` means "over the next 24 hours").
        - **limit** indicates whether the user is interested in seeing a maximum (by specifying `MAX`) or a minimum (by specifying `MIN`) over the specified number hours.
    - `PRECIPITATION length limit`, which means that the user would like to see the *hourly chance of precipitation*, reported as a percentage.
        - **length** is a positive integer indicating the number of hours into the future for which the query is being made (e.g., `24` means "over the next 24 hours").
        - **limit** indicates whether the user is interested in seeing a maximum (by specifying `MAX`) or a minimum (by specifying `MIN`) over the specified number hours.
- Subsequent lines of input will specify additional weather queries, each in one of the same formats described above. You can assume that there will always be at least one weather query, but there is no limit on how many there might be. There are no rules restricting the order in which they might appear, nor are there rules preventing their duplication. Continue reading lines of input and treating them as weather queries until you read a line of input in the following format:
    - `NO MORE QUERIES`
- The final line of input will be in one of two formats:

- `REVERSE NOMINATIM`, which means that we want to use the Nominatim API to do reverse geocoding, i.e., to determine a description of where the nearest weather station is located.
- `REVERSE FILE path`, which means that we want to use a file stored locally, containing the results of previous calls to Nominatim's reverse geocoding API instead.

You can freely assume that the input will match the specification described above; we will not be testing your program on any inputs that don't match the specification.

The interplay between geocoding and weather data

There are two forms of data we gather, either from web APIs or files stored locally, and one thing we'll need to be sure we understand is how they relate to one another.

- We'll use Nominatim to take the description of the target location and decide where it is (i.e., we "forward geocode" it). Among other things, this will result in a latitude and longitude.
- We'll ask the National Weather Service for an hourly forecast for that latitude and longitude, but it's worth noting that it'll offer a forecast for a slightly different location than we asked for. The reason is simple: There are a limited number of weather stations, each offering a forecast for an area described by a *polygon* made up of latitudes and longitudes.
- We'll use Nominatim to take the actual location of the hourly forecast and determine its description (i.e., we "reverse geocode" it). This requires a latitude and longitude, which will need to be derived from the polygon reported by the National Weather Service.

When the National Web Service API reports an hourly forecast within an area described by a polygon, we'll agree to use the following approach to boil that polygon down to a single location (i.e., a single latitude and longitude).

- Keep only the unique points in that polygon. (The same point is often reported more than once.)
- Take the average of the latitudes and the average of the longitudes. That, for our purposes, is the *forecast location*.

The output

After reading all of the input, you'd first display the latitude and longitude of the target location, with latitudes and longitudes shown in the following format.

```
TARGET 33.64324045/N 117.84185686276017/W
```

Then, you'd use the information that's either stored in the specified files or downloaded from the specified APIs to determine the results of the weather queries, displaying the result of the queries in the order they were specified in the input. For example, suppose that the input was as follows:

```
TARGET NOMINATIM Bren Hall, Irvine, CA
WEATHER NWS
TEMPERATURE AIR F 12 MAX
HUMIDITY 24 MIN
NO MORE QUERIES
REVERSE NOMINATIM
```

This means we're looking up an hourly weather forecast from the National Weather Service as near to Bren Hall at UC Irvine as we can get, using Nominatim to describe the forecast location, then reporting two things:

- The maximum air temperature over the next 12 hours, in degrees Fahrenheit.
- The minimum relative humidity over the next 24 hours, reported as a percentage.

We might see something like this as a result — though, of course, the results are entirely dependent on the data returned by the APIs, so you may find that you receive a different result if you run your program with the same input.

```
FORECAST 33.654532225/N 117.83296842499999/W
1 Sunnyhill, Irvine, CA
2024-11-07T23:00:00Z 77.0000
2024-11-07T22:00:00Z 6.0000%
```

There are a couple of things worth noting in the example above:

- The word `FORECAST` precedes the forecast location, which you'll have determined from the National Weather Service API data, as described previously.
- The description of the location (i.e., what's shown as `1 Sunnyhill, Irvine, CA` above) is what arises from the reverse geocoding of the forecast location.
- Each weather query is displayed as a date and time, followed by a space, followed by a value. The date and time specify the *start time* of the applicable hour-long block of time from the hourly forecast (e.g., the hour in which the temperature was maximum, the humidity was minimum, or whatever).
  - In the event of a tie (e.g., when the same maximum temperature occurs during more than one hour), only report the earliest of these hours.
- The date and time are displayed in the UTC time zone, expressed in ISO 8601 format. (You'll likely find the `datetime` module in Python's standard library helpful in handling this requirement.)
- The values of weather queries are always displayed to four decimal places, even if those decimal places are zeroes. Percentages are always followed by a `%` character.

After information about all of the locations has been printed, you will wrap up the output by printing *attribution messages* for any of the sources of data (National Weather Service and/or Nominatim) that was actually used when looking up forward geocoding, reverse geocoding, or real-time weather data. For example, you would only print National Weather Service's attribution if `WEATHER API` was in the input rather than `WEATHER FILE`.

```
**Forward geocoding data from OpenStreetMap
**Reverse geocoding data from OpenStreetMap
**Real-time weather data from National Weather Service, United States Department of C
```

If all three attributions are to be printed, they should be shown in the order above. If fewer than three are printed, they should be shown in the relative order above (e.g., forwarding geocoding would be shown before reverse geocoding or weather data).

A complete example that uses locally stored data

I recommend that you do the majority of your testing with locally stored data. Testing requires not only running a program, but also knowing what the output is supposed to be; only then can you know whether you've got the correct output. But when you're writing a program that reads data from an API that will quite possibly give you different data every time you call it, it becomes difficult to know what the right answer is.

So, as a first step in this direction, you'll find some example data below. Download these files and store them in the same directory as your program's code.

- nominatim_target.json
- nominatim_reverse.json
- nws_hourly.json

Once you've finished with your program, you should be able to run the following test and see the results shown below.

```
TARGET FILE nominatim_target.json
WEATHER FILE nws_hourly.json
TEMPERATURE AIR C 24 MAX
NO MORE QUERIES
REVERSE FILE nominatim_reverse.json
TARGET 33.6480612/N 117.8469736/W
FORECAST 33.654532225/N 117.83296842499999/W
Auburn Aisle, Oxford Court, University Town Center, Irvine, Orange County, Californic
2024-09-17T19:00:00Z 22.7778
```

What to do in the case of API failure

In this project, we face the problem that our program may be written perfectly, yet still might fail in some circumstances. This is because we're dependent on two APIs sending us the data we need, in the format we expect, without which our program can't generate its output. Yet, the APIs are themselves software, and software fails; our communication with the

APIs is done via a computer network, and computer networks fail, too. So, we'll need to account for these possibilities in our design, and also have a mechanism for testing them.

First, we'll need to decide what it means for the APIs to have failed. To do that, we'll attack the problem from the opposite angle: What does success look like?

- The HTTP status code in the response to all of our API requests was 200. Any other status code is considered a failure, regardless of the data that sent in the response.
- The content of all of our API requests was formatted as we expected (e.g., it was in JSON format if that's what we expected).
- If we used a file stored locally in place of a call to an API, the file existed and the contents of the file were formatted as we expected (e.g., it was in JSON format if that's what we expected).

Notably, this means that we're not considering it to be failure if the National Weather Service API returns fewer hours of forecast data than the user asked for. For example, if the user asked for a 24-hour forecast, but only 12 hours were available, we'll simply use the 12 hours that are available and report the answer accordingly.

In any other case, we'll say that our program has failed, and we'll print an alternatively formatted set of output — entirely separate from the normal one — that briefly describes the first failure you encountered.

- The first line of output will simply be the word `FAILED`.
- If the first failure you encountered was due to the use of an API...
  - The second line of output will contain the HTTP status code of the first API request that failed, as well as the URL that you connected to. (Note that the status code might still be 200, if the failure was due to missing or misformatted content.)
    - If there was no HTTP status code (e.g., because your computer is not connected to the Internet and can't contact the server at all), then you would print the URL here, but not the status code, since there would be no status code to print.
  - The third line of output will be exactly one of these three phrases:
    - `NOT 200` (if an API request returned a status code other than 200)
    - `FORMAT` (if an API request returned data that had missing or misformatted content)
    - `NETWORK` (if an API request couldn't be sent at all because, for example, there was no network connectivity)

- If the first failure you encountered was due to a file stored locally that you were using in place of a call to an API...
  - The second line of output will contain the path to the file that you attempted to use.
  - The third line of output will be exactly one of these two phrases:
    - `MISSING` (if the file does not exist or couldn't be opened)
    - `FORMAT` (if the file could be opened, but its contents had missing or misformatted content)

For example, if your program makes an API request whose response contains the HTTP status code 429, your output would be something like this (albeit with the actual URL that failed):

```
FAILED
429 https://whatever.the.url.that/failed/was?including=its&parameters=please
NOT 200
```

Or if your program tried to use the file `D:\Examples\Python\nws.json` but that file didn't exist, your output would be this instead:

```
FAILED
D:\Examples\Python\nws.json
MISSING
```

To be clear, you'll print this alternative output (and only this alternative output) if any of the API requests or usages of files fails; otherwise, you'll follow the requirements above and print output describing the target location and the results of any weather queries.

Design requirements and advice

As with the previous project, you'll be required to design your program using multiple Python modules (i.e., multiple `.py` files), each encapsulating a different major part of the program. We'll leave you some flexibility in determining where to

draw the line between what's in one module and what's in another, but the module that you'd execute to run your program must be named precisely `project3.py`.

Fetching our data with classes

There are three points in your program where you'll need to fetch data from either an API or a file:

1. When you use forward geocoding to determine the location of the target of your analysis.
2. When you need to obtain hourly weather forecasts for a given location.
3. When you use reverse geocoding to determine the description of where hourly weather forecasts are available.

In each of these three cases, there are two separate ways to solve the problem — one using an API and the other using a file. In each case, you'll be required to implement Python *classes*, which contain attributes that configure it, if necessary (e.g., the path to a file that should be read), and a method that obtains the data. Classes that obtain the same data must share an interface (i.e., they must have a method with the same name, the same parameters, and the same type of return value), so that you can build objects of these types when you read your program's input, then execute them later without knowing which types of objects they actually are.

This is one key benefit in using classes in Python; we can treat different kinds of objects with similar capabilities the same way, which avoids us having to use `if` statements to differentiate. We saw an example of this in lecture, when we talked about Duck Typing.

Where should I start?

There are lots of ways to start this project, but your goal, as always, is to find stable ground as often as possible. One problem you know you'll need to solve is generating the final report, so you could begin by generating a portion of it — maybe just some details of the output report that are formatted correctly, even if the data is hard-coded. Now you're on stable ground.

One problem you know you'll need to solve is the problem of calculating a "feels like" temperature, given the air temperature, relative humidity, and wind speed; you might consider continuing with that. You can test this using the Python shell or `assert`-based tests before proceeding, and then you're on stable ground. Temperature conversions aren't a bad idea as a next step; implement and test those, and now you're on stable ground again.

From there, you might continue by implementing a module that obtains the hourly weather forecasts from the National Weather Service API, perhaps first by implementing the class that reads that data from a file, then later implementing the class that loads it from the web instead. (You'll want the part that reads from a file pretty early on, because you'll want stable data you can test with, instead of receiving different weather data as the weather changes.)

Once you've got these implemented, you might continue with forward and reverse geocoding using Nominatim — again, first by implementing the classes that read this data from a file, then later implementing the classes that load them from Nominatim's API instead.

Now you'd have a lot of pieces in place, and you can start thinking about how to tie them together. At this point, you may feel like you don't have a program yet, but that's not so out of the ordinary when you work on a large project; it's often quite a while before you have something that runs an entire end-to-end process, because you first need to build and test a lot of smaller-scale tools. In that sense, this project is a pretty realistic view into what it takes to build realistic programs that interact with complex sets of inputs and outputs.

But, again, there are lots of sequences that could lead to a good solution, and you'll want to consider how you can achieve partial solutions that nonetheless meet the requirements partially, because partial credit is available for those. Still, if you find a way to approach this that's different than what I've suggested, but that leads you to a complete program that meets the design requirements, that's fine; we don't care what order you implement it in, ultimately, but we're happy to help you find an ordering if you're not sure what to work on next.

Limitations

Third-party libraries

Remember that, as stated in the Project Guide, third-party libraries — libraries that are not part of Python's standard library — are off-limits in your work unless they are explicitly permitted. This includes, for example, code you might find online that communicates with the Nominatim or National Weather Service APIs, or third-party libraries such as `requests` that are commonly used for HTTP-based communication. The intent here is that you be the one to write that code, because that's one of the learning objectives here.

Respecting the terms of service of the APIs we'll use

The APIs we're using in this project are subject to terms of service, which is to say that there are restrictions around how we're permitted to use them. In particular, we'll need to be cognizant of the following restrictions.

National Weather Service

The National Weather Service API requires that we set a header called `User-Agent`, as a way of letting them know who is connecting to their API. Web browsers generally set this header in a way that identifies themselves — the name of the browser and its version is commonly sent — but we'll be connecting from a Python program instead. So, we'll set the `User-Agent` header as follows (including the parentheses):

- `(https://www.ics.uci.edu/~thornton/icsh32/ProjectGuide/Project3/, YOUR_UCI_EMAIL)`
- Replace `YOUR_UCI_EMAIL` with your UCI email address, including the `@uci.edu` part. If, for example, your UCI email address is `boo@uci.edu`, you would send `(https://www.ics.uci.edu/~thornton/icsh32/ProjectGuide/Project3/, boo@uci.edu)`.

Nominatim

Nominatim has a rate limit of one request per second, which means that you'll need to be sure that your program "pauses" for one second between subsequent requests. Since there may be multiple requests being made to Nominatim during one run of your program, this is something you'll need to include in your program; if you're making multiple requests, you'll need to "pause" for one second between them.

Nominatim requires that we set a header called `Referer`, which specifies information about where the request came from. Set the `Referer` header as follows:

- `https://www.ics.uci.edu/~thornton/icsh32/ProjectGuide/Project3/YOUR_UCINETID`
- Replace `YOUR_UCINETID` with your UCInetID. If, for example, your UCI email address is `boo@uci.edu`, you would replace `YOUR_UCINETID` with `boo` instead, meaning you would send `https://www.ics.uci.edu/~thornton/icsh32/ProjectGuide/Project3/boo`.

More details on the usage policies governing the National Weather Service and Nominatim APIs can be found at the links below.

- National Weather Service API - Overview
- Nominatim API Usage Policy

If you do not respect these terms of use and your API usage is limited or revoked, we will not be offering an extension on the due date of this project, so you'll need to be respectful of these limitations to be sure that your usage of these APIs will be unimpeded throughout your work in this project cycle.

Deliverables

Gathering your files for submission

We've written automation tools to help us to manage your submissions and report your scores, but these tools require us to know that everyone's submission will be structured the same way. For this reason, we're providing you a tool that can gather your files into a single file whose format we can count on, which you'll then submit to Canvas.

To submit your work, follow these instructions:

1. Make sure that all of the `.py` files that make up your program are all in the same directory.
2. Download the Python script linked below, storing it in the same directory as your program:
    - make_project3_submission.py
3. Run the Python script your downloaded in the previous step. It will gather all of the `.py` files in the same directory (except for ones that it intentionally skips), verify that they're readable as text, and will then generate a *submission file* named `project3.zip` in the same directory.
    - If there are any issues — files in the wrong format, for example — they'll be reported to you.
    - The files included in the submission will be listed in the script's output; you'll want to read that output to ensure that all of the files you want to be submitted are included. (You might also want to unzip the file somewhere else, just to be sure it contains what you intend; the risk of your submission containing all of the files you want submitted is yours.)
4. Submit the submission file `project3.zip` (and only that file) to Canvas.

Note, too, that if you submit separate files, create your own `.zip` file arranged in your own way, or otherwise don't follow these instructions, we reserve the right to score your project as low as zero. There are no exceptions to this rule.

There are a few additional rules to be aware of.

- You're responsible for submitting the version of your project that you want graded prior to the deadline. Contacting us afterward and telling us that you accidentally submitted the wrong version will not be grounds for a resubmission under any circumstances.
- You're responsible for making a submission in order to receive credit, which means you'll want to be sure that you've remembered to submit your work and verified in Canvas that it's been received. A later claim of having forgotten to submit your work or having misremembered the due date will not be grounds for a resubmission under any circumstances.
- The determination of whether your work has been submitted before the deadline is the time it was submitted to Canvas. Neither timestamps on local copies of your files nor timestamps in other places (e.g., emails or online storage) are considered evidence of completion prior to the deadline under any circumstances.

Can I submit after the deadline?

Yes, it is possible, subject to the late work policy for this course, which is described in the section titled *Late work* at this link.

What do I do if Canvas adjusts my filename?

Canvas will sometimes modify your filenames when you submit them (e.g., by adding a numbering scheme like **-1** or a long sequence of hexadecimal digits to its name). In general, this is fine; as long as the file you submitted has the correct name prior to submission, we'll be able to obtain it with that same name, even if Canvas adjusts it.