As you set forth on the Infinite Adventure project there are several stumbling points that can arise. The questions on this week's section are geared to help smooth over and alleviate those potential problems!

1. Compound and nested data types can be incredibly flexible in the data they can hold, but they can be complicated or confusing to work with. You often need to track multiple indexes as things become nested, and it is imperative to track what data types are representing different parts of the data structure. To that end, this problem is designed to help you get some practice in working with these types of structures.

   In the starting repository is a file names `space_missions.json`. This file contain data on a host of fictional space missions. Each mission has a name, a crew, and a collection of hardware that was tested. Each crew member has a name, a role on that mission, and set list of certifications. In contrast, each piece of hardware has a unique identifier, and general category, and some diagnostics. Each diagnostic has a status associated with it and then an amount of power that was measured, in watts. Figure 1 below tries to give you a visual of the different nested pieces and their various types.

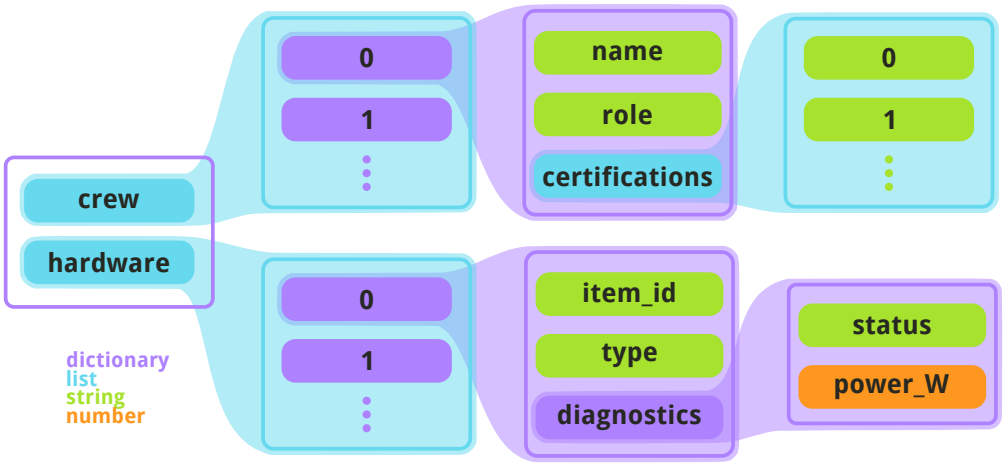

Figure 1: A color coordinated visual depiction of the internal structure of the space missions JSON data. Purple represents dictionaries, blue lists, green strings, and orange numbers.

   Your challenge in this problem will be three-fold: to load in the data and then use it to answer two questions.

   (a) Write a function called `load_data` that loads in the data from the JSON file and returns the corresponding Python dictionary. This should be straightforward and you just need to open the file (in read mode) and then utilize the `json` library.

   (b) Here you need to write a function called `eva_count` which counts (and returns) the number of *different* crew members who have an "EVA" certification and who participated in a mission that began with an "A". It is possible for the same crew member to serve on multiple missions, so make sure you don't double count! Your function should take the dictionary generated by Part A as its input.

(c) Write one last function called `power_missions` that takes the dictionary from Part A as its input. This function should return a *set* of all the mission names when the *Navigation* type of hardware tested during the mission averaged over 100 watts of power.

2. The Infinite Adventure project leverage having ChatGPT automatically generate rooms or scenes that a player tries to visit but which do not already exist. To do so, your Python code needs to "contact" and exchange information with a piece of software (ChatGPT) that is running on the web. To understand how that works, we need to understand the basics of web communication.

   The HTTP protocol defines a variety of ways in which software can communicate over the internet, but easily the most commonly are *GET* requests and *POST* requests. A GET request is what is used whenever a piece of software wants to simply **retrieve** some information from a computer that is connected to the internet. This is actually what your web browser is doing every time you visit a web page! Your browser makes a GET request to the web server located at the provided URL, and that web server returns a bunch of HTML, which your browser then renders. GET requests can be utilized to return other pieces of information as well, but the key idea behind them is that the client simply tries to access a given URL. In contrast, POST requests are used when a client needs to **send** information to some remote web server. In this case the client makes connection with the web server and then sends along what is frequently called the *payload*, where the payload is often a bit of JSON that contains the information to be conveyed. The server then usually acknowledges that it received the payload, and might return some further information as well.

   But how does this communication happen? Software that is present on other computer connected to the web essentially waits until a connection is made, at which point it strings into action. Imagine web servers as being similar to graphical software, but where the event listeners are listening for web connections instead of mouse clicks. Different software then responds in different ways. Web servers, which host webpages, might respond by sending along the HTML that makes up the webpage in question. Other servers might host what are called *Application Programming Interfaces*, or *API*s. These pieces of software are more flexible in what they can do, and they can generally handle all manner of inputs and return all manner of outputs. Think of them as functions that are defined on other computers that you can utilize. The NotOpenAI library connects to one such API. So long as we interact with the API properly, it will give us the desired information, in the same way that if we call a function correctly it will return our desired information.

   In the Infinite Adventure, we need to send our prompt to ChatGPT. Doing so requires that we make a POST request, and then interpret and parse the results. In this problem you will practice doing something similar. A special API has been constructed just for this Section, which you can find at https://section12api-production.up.railway.app/generate. This API only accepts POST requests, and it expects a payload with

a very specific format. In particular, it expects a payload dictionary with three keys defined:

- "name" - An individual's name
- "class_year" - An individual's class year. From Freshman to Senior.
- "favorite_animal" - An individual's favorite animal

When you send such a payload to the server, it will respond with a very simple JSON dictionary with a single "content" key. That key though corresponds to a string version of a dictionary that you need to decode. This is the *exact* same thing you need to do with the NotOpenAI request response, where you need to convert the text version of the scene dictionary back to an actual Python dictionary. You can accomplish this with `json.loads`.

All told then, you want to write some code that will:

1. Create your individual payload dictionary
2. Make the post request, providing your payload
3. Check the status of the returned response to ensure things went well
4. Extract the "content" key and convert its value back to a Python dictionary
5. Extract the "fun_fact" key from that dictionary, and print it out to the screen.

Despite the fact that there is a ton of text in this question, the code you need to accomplish the above is less than 10 lines long! So don't over think it!