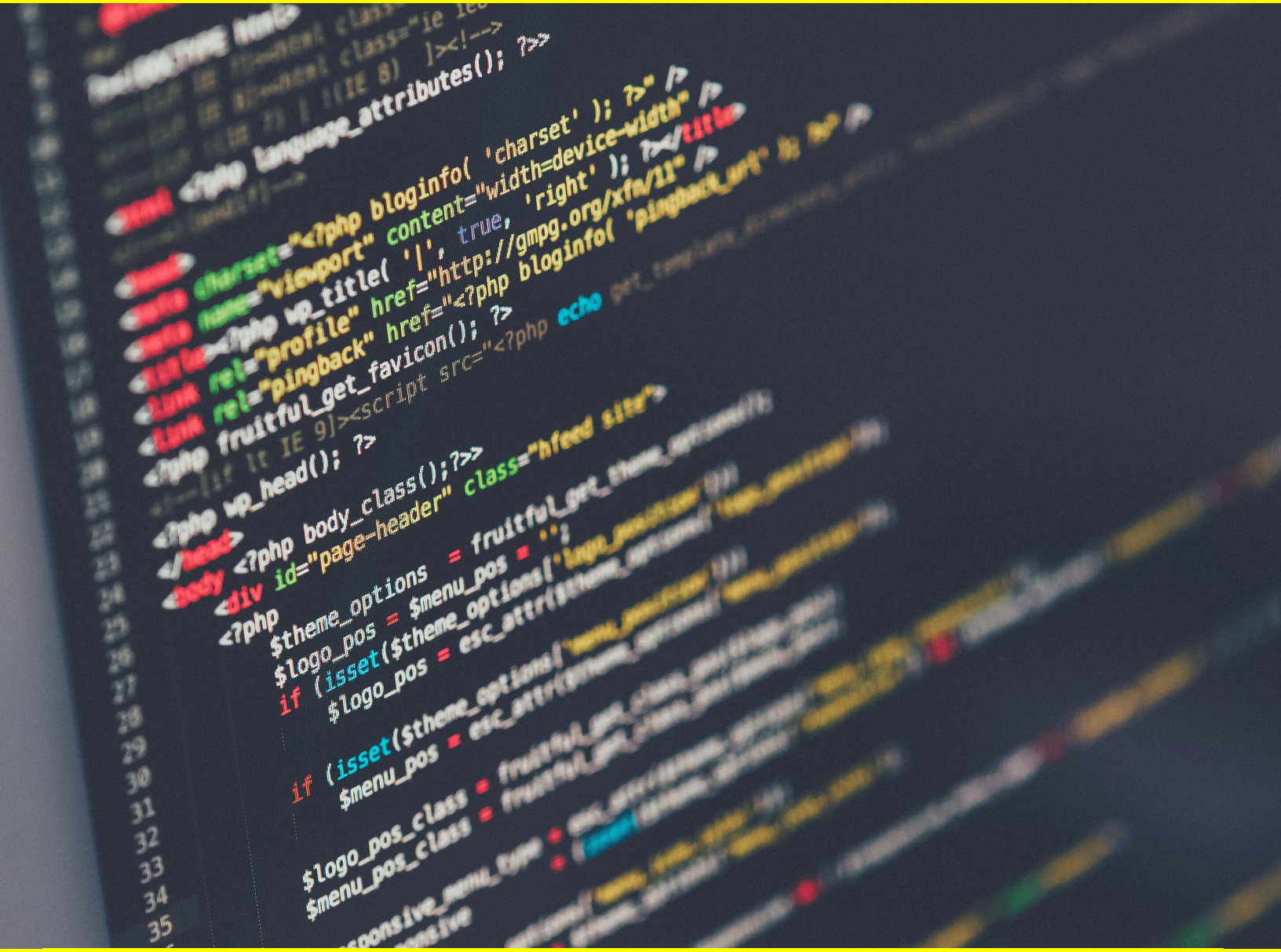
the fast lane to python

PART9



BY:

Arshadul Shaikh

Ref taken from Coursera + Udemy

The Internet: Behind the Scenes

What is the Internet?

The Internet is a global network of billions of computers and other electronic devices. With the Internet, it's possible to access almost any information, communicate with anyone else in the world, and do much more.

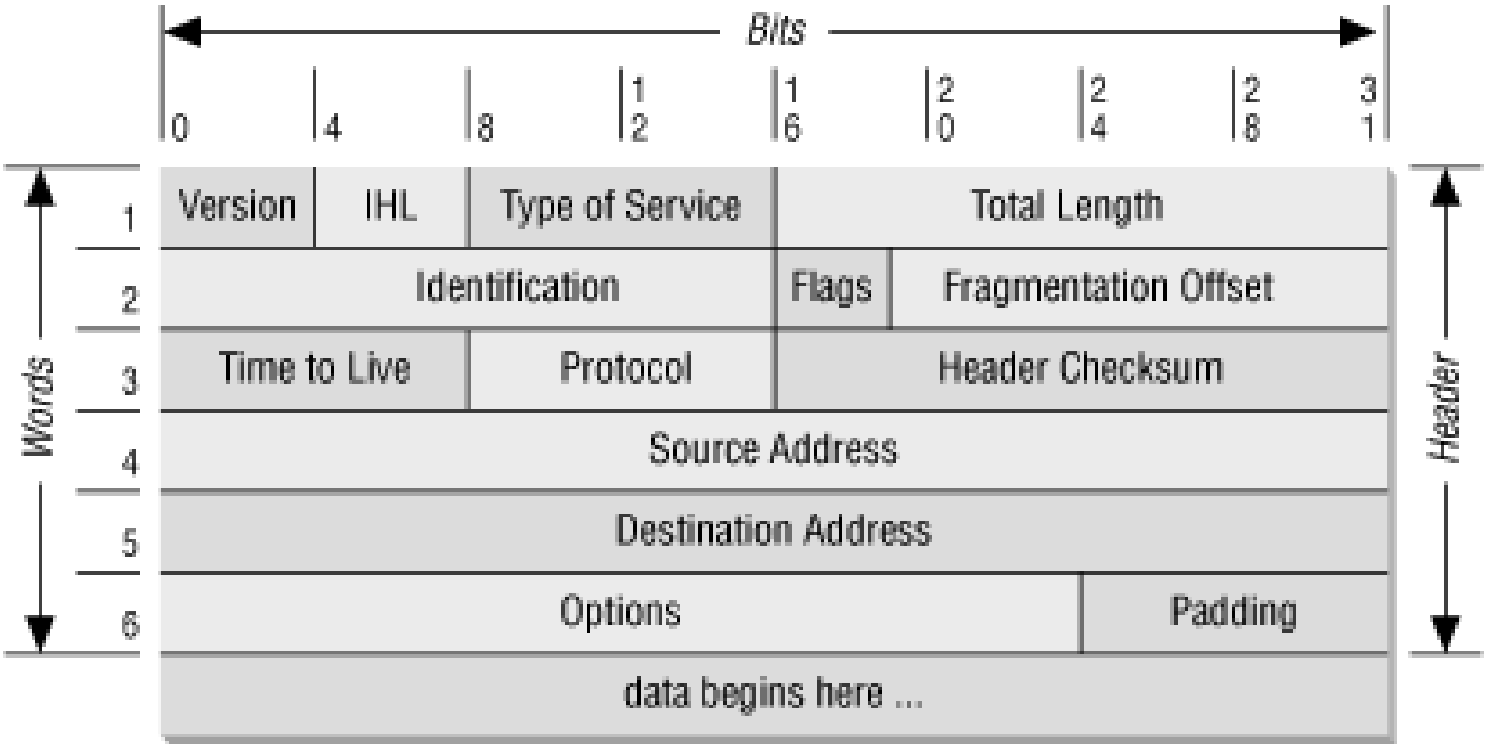
You can do all of this by connecting a computer to the Internet, which means you are online. When someone says a computer is online, it's just another way of saying it's connected to the Internet.

If we deep dive in understanding it, the Internet is a transport mechanism that lets any connected device communicate with any other connected device. Let us see what is happening behind the scenes:

- Each active device has a **globally distinct IP address**, which is a **32 bit number**. Usually an IP address is represented as a sequence of four decimal numbers, each number in the range (0, 255).
    - For example, when I checked the IP address for my laptop just now, it was 103.272.172.164. this is my public address that i am assigned while using my current internet. These are dynamic IP addresses of a computer and are never fixed so sharing IP address is not dangerous because the IP address will map to your computer for a very short time as you reboot your router and switch the network, the IP address get's changed.

- Data is chopped up into reasonable sized packets (up to 65,535 bytes, but usually much smaller).

- Each data packet has a header that includes the destination IP address. (do not get confused , below figure is just to show how IP header looks like containing Destination IP)



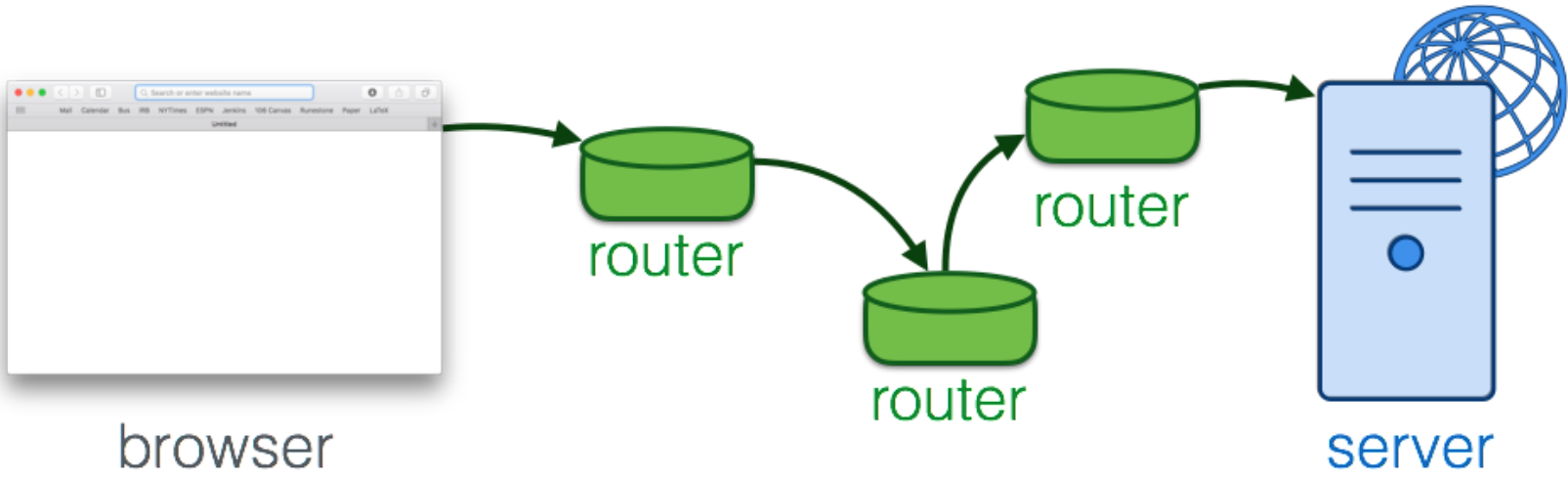
- **Each packet is routed independently**, getting passed on from one computing device to another until it reaches its destination. The **computing devices** that do that **packet forwarding** are called **routers**.

```
C:[redacted]>tracert www.google.com

Tracing route to www.google.com [172.217.166.164]
over a maximum of 30 hops:

  1     1 ms    <1 ms     1 ms  [redacted] [redacted]
  2     5 ms     4 ms     4 ms  [redacted]
  3    15 ms    16 ms    15 ms  as15169.bom.extreme-ix.net [redacted]
  4    18 ms    17 ms    21 ms  108.170.248.209
  5    80 ms   126 ms    16 ms  216.239.57.189
  6    16 ms    15 ms    16 ms  bom07s20-in-f4.1e100.net [172.217.166.164]

Trace complete.
```

    - I have shown with an example using tracert command to see how different routers were used hopping from one to another to finally reach the server from where Google was being served.

- Each router keeps an address table that says, when it gets a packet for some destination address, which of its neighbors should it pass the packet on to. The routers are constantly talking to each other passing information about how they should update their routing tables. The system was designed to be resistant to any local damage. If some of the routers stop working, the rest of the routers talk to each other and start routing packets around in a different way so that packets still reach their intended destination if there is some path to get there

- This is how the hopping looks like

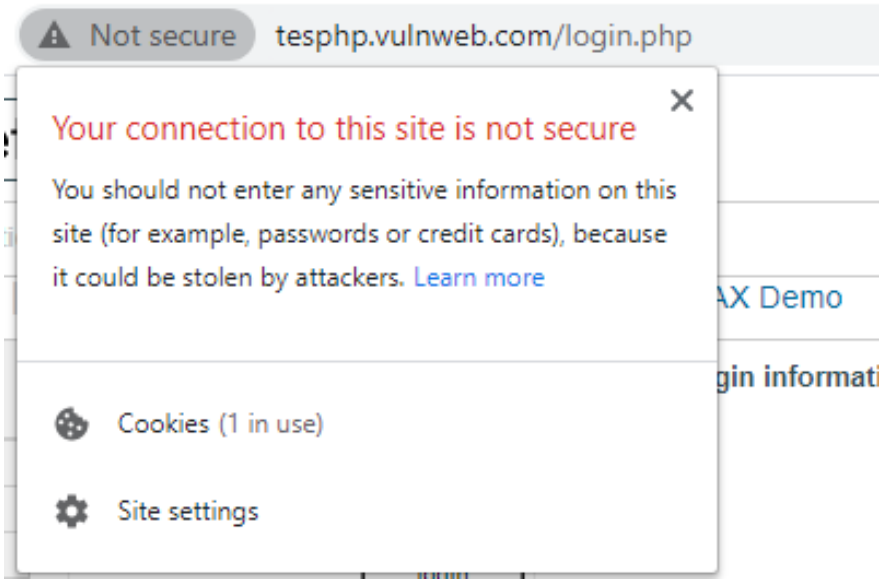


Anatomy of URLs

A URL is **used by a browser** or other program to **specify what server to connect** to and what page to ask for. Like other things that will be interpreted by computer programs, **URLs have a very specific formal structure**. If you put a colon in the wrong place, the URL won't work correctly. The overall structure of a URL is:

<scheme>://<host>:<port>/<path>

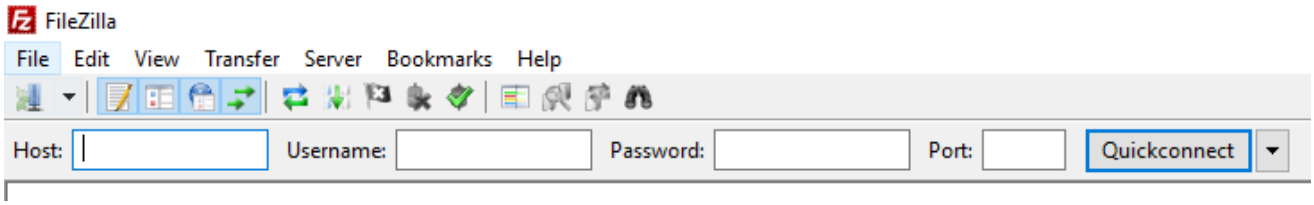
scheme
Usually, the scheme will be **http** or **https**(s stands for "secure"). When you use https, all of the communication between the two devices is encrypted. Any devices that intercepts some of the packets along the way will be unable to decrypt the contents and figure out what the data was.



This is a website to test all possible vulnerabilities. hackers use this website to practice and learn about some basic concepts of ethical hacking. And as you see, it clearly says not secure, meaning the data will never be encrypted which means if you feed some relevant data people can track it.

Other schemes that you will sometimes see include ftp (for file transfer) and mailto (for email addresses).

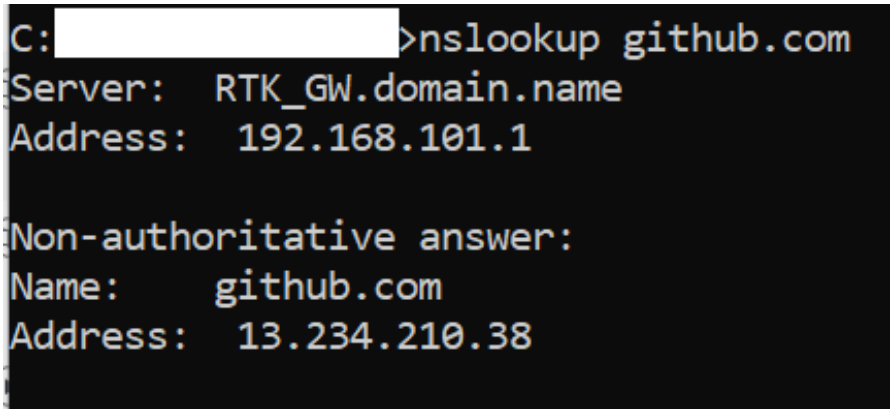
- FileZilla is a really good example for using ftp/sftp



host

The **host** will usually be a **domain name** like github.com or google.com. When the URL specifies a domain name, the first thing the computer program does is look up the domain name to find the **32-bit IP address.**

- For example, right now the IP adddress for github.com is 13.234.210.38. This could change if, for example, github moved its servers to a different location or contracted with a different Internet provider.
- Lookups use something called the Domain Name System, or DNS for short. Changes to the mapping from domain names to IP addresses can take a little while to propagate: if github.com announces a new IP address associated with its domain, it might take up to 24 hours for some computers to start translating github.com to the new IP address.



    - Alternatively, the host can be an IP address directly. This is less common, because IP addresses are harder to remember and because a URL containing a domain name will continue to work even if the remote server keeps its domain name but moves to a different IP address.

port

The :port is optional. If it is omitted, the default port number is 80. The port number is used on the receiving end to decide which computer program should get the data that has been received.

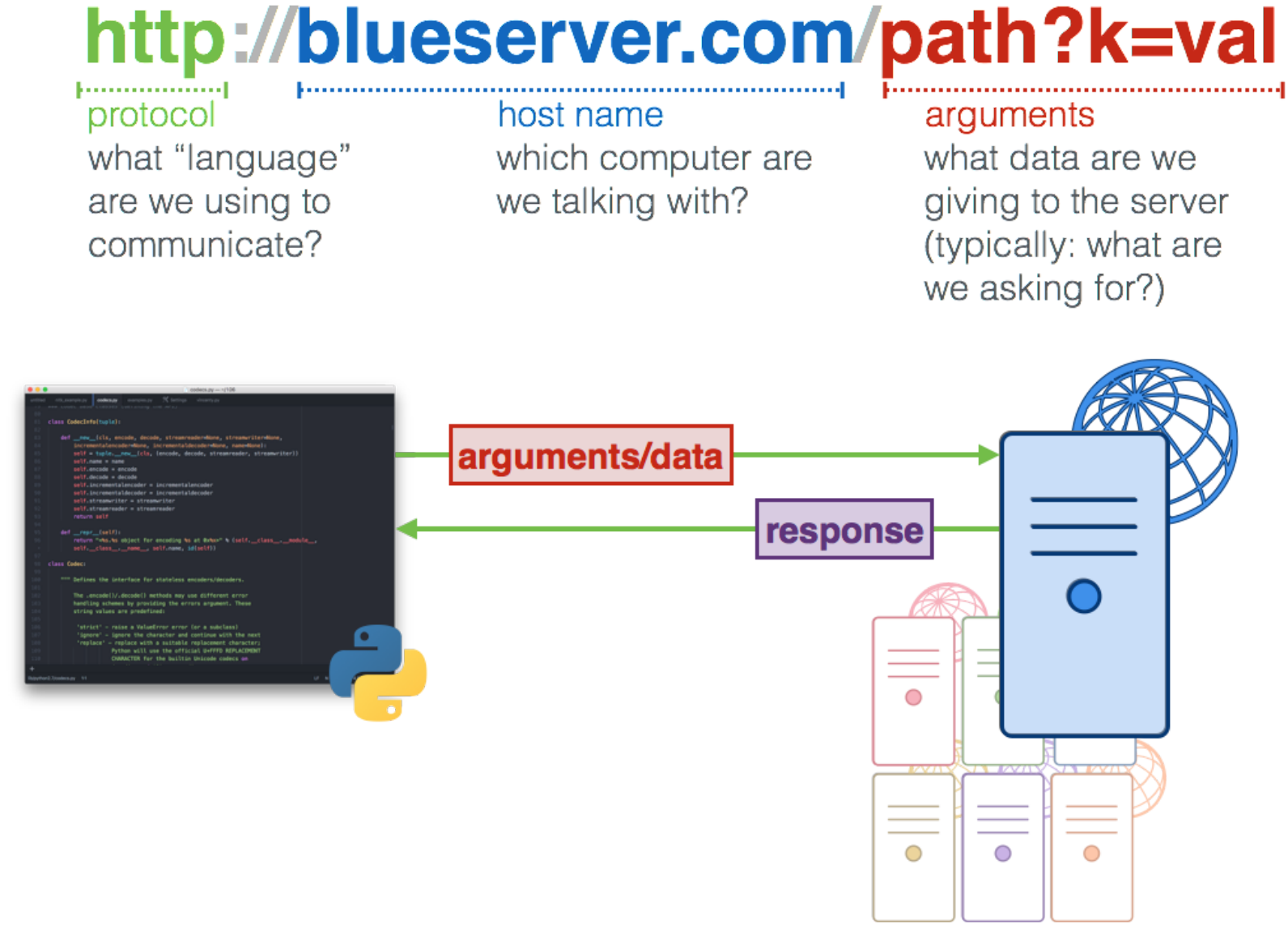
path

The /path is also optional. It specifies something about which page, or more generally which contents, are being requested.

For example:
https://github.com/IntriguedCuriosity/Complete-Python-3-Bootcamp

- https:// says to use the secure http protocol
- github.com says to connect to the server at github.com, which currently maps to the IP address 13.234.210.38 .
- The connection will be made on the default port, which is 443 for https
- "/IntriguedCuriosity/Complete-Python-3-Bootcamp" says to ask the remote server for the page . It is up to the remote server to decide how to map that to the contents of a file it has access to, or to some content that it generates on the fly.

The url **http://blueserver.com/path?k=val** is another example that we can consider. The path here a bit different from what we saw above because **it includes** what are **called "query parameters", the information after the ?.**



The HTTP protocol

A protocol specifies the order in which parties will speak and the format of what they say and the content of appropriate responses.

HTTP is part of the Application layer protocol from OSI model. **HTTP is the protocol that specifies how web browsers or other programs communicate with web servers.** One version of the formal specification, before it was later split into multiple documents, was IETF RFC 2616. It is 176 pages long! Fortunately, the basics are pretty easy to understand.

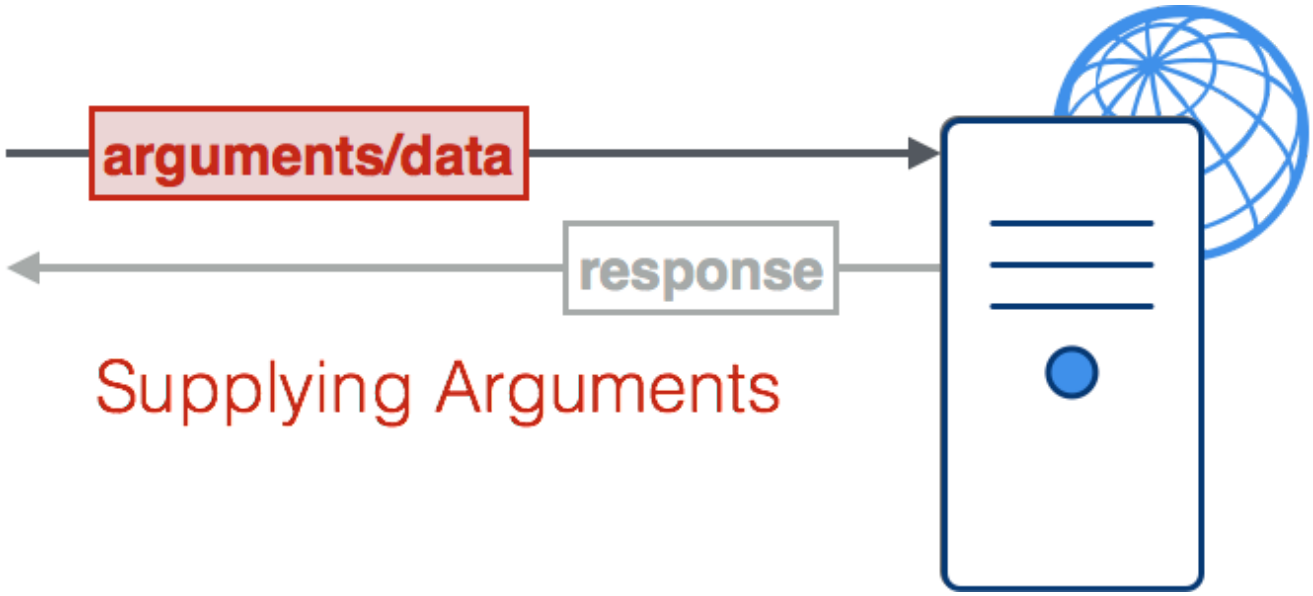
Step 1: the client makes a request to the server.

- If the request only involves fetching data, the client sends a message of the form GET <path>, where <path> is the path part of the URL like www.google.com
- If the request involves sending some data (e.g., a file upload, or some authentication information), the message starts with POST

In either case, the client sends some HTTP headers. These include:

- The type of client program. This allows the server to send back different things to small mobile devices than desktop browsers (a "responsive" website)
- Any cookies that the server previously asked the client to hold onto. This allows the server to continue previous interactions, rather than treating every request as stand-alone. It also allows ad networks to place personalized ads.

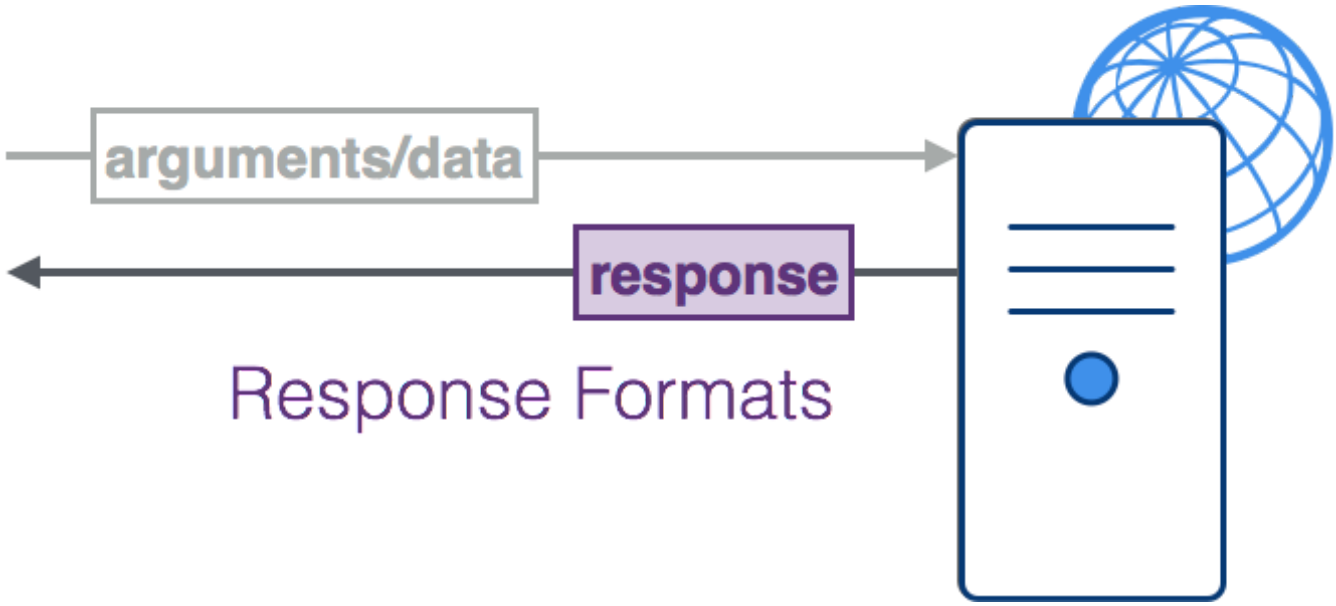
After the HTTP headers, for a POST type communication, there is some data (the body of the request).



Step 2: the server responds to the client.

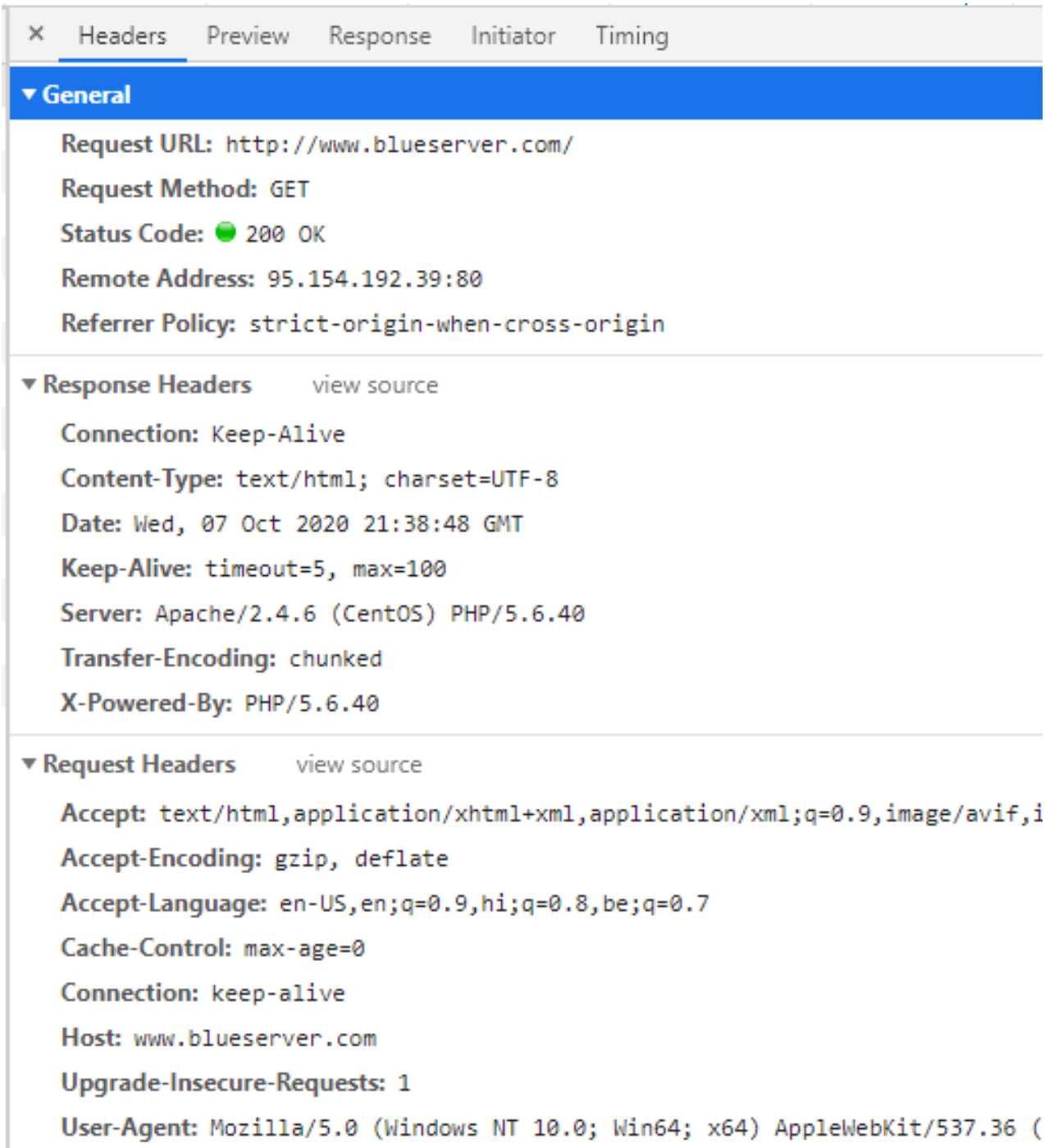
- a response code indicating whether the server thinks it has fulfilled the request or not.
- a description of the type of content it is sending back (e.g., text/html when it is sending html-formatted text).
- any cookies it would like the client to hold onto and send back the next time it communicates with the server.

After the headers come the contents. This is the stuff that you would see if you ask to “View Source” in a browser.



To check this in real time you follow below steps:

1) Use Chrome, visit a URL, right click, select Inspect to open the developer tools.
2) Select Network tab.
3) Reload the page, select any HTTP request on the left panel, and the HTTP headers will be displayed on the right panel.



Using REST APIs

REST stands for REpresentational State Transfer. It originally had a more abstract meaning, but has come to be a shorthand name for web sites that act a bit like python functions, taking as inputs values for certain parameters and producing outputs in the form of a long text string.

API stands for Application Programming Interface. An API specifies how an external program (an application program) can request that a program perform certain computations.

Heard of web service ? Not sure? but you would have definitely heard about a website!

when we put the two together, **a REST API** is what comes into picture. It **specifies how external programs can make HTTP requests** to a **web site** in order to request that some computation be carried out and data returned as output.

- When a website is designed to accept requests generated by other computer programs, and produce outputs to be consumed by other programs, it is sometimes called a **web service**, as **opposed** to a **web site which produces output** meant for humans to consume in a web browser

A browser requests data from a web site in order to display it directly to a human user of the browser.

URL Structure for REST APIs

the URL has a standard structure:

- the base URL
- a ? character
- one or more key-value pairs, formatted as **key=value pairs** and **separated** by the **&** character.

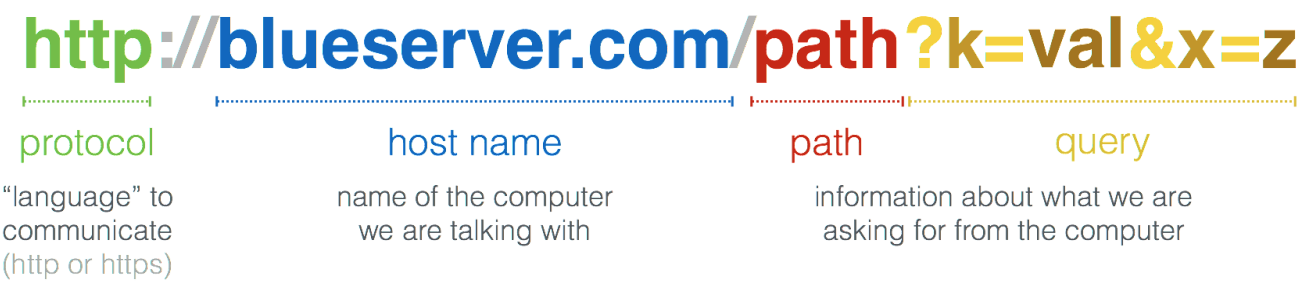
URL=https://itunes.apple.com/search?term=Ann+Arbor&entity=podcast.

Try copying that URL into a browser, or just clicking on it. It retrieves data about podcasts posted from Ann Arbor, MI. Depending on your browser, it may put the contents into a file attachment that you have to open up to see the contents, or it may just show the contents in a browser window.


```
{
 "resultCount":42,
 "results": [
{"wrapperType":"track", "kind":"podcast", "collectionId"
Media Ann Arbor", "collectionName":"Michigan Insider", "
"collectionCensoredName":"Michigan Insider", "trackCenso
"collectionViewUrl":"https://podcasts.apple.com/us/podca
"feedUrl":"https://www.omnycontent.com/d/playlist/a7b0bd
a88d0118334a/6d8e6f96-7456-4af2-b00a-a88d0118334f/podcas
"trackViewUrl":"https://podcasts.apple.com/us/podcast/mi
"artworkUrl30":"https://is3-ssl.mzstatic.com/image/thumb
d674ae9b5c72/mza_5443611793607478813.jpg/30x30bb.jpg", "
ssl.mzstatic.com/image/thumb/Podcasts123/v4/3b/d2/56/3bd
```


Let's pull apart that URL.

- the base URL: https://itunes.apple.com/search
- a ? character
- key=value pairs. In this case, there are two pairs. The keys are term and entity. An & separates the two pairs.



Encoding URL Parameters

Here's another URL that has a similar format.

https://www.google.com/search?q=%22violins+and+guitars%22&tbm=isch.

It's a search on Google for images that match the string "violins and guitars". It's not actually based on a REST API, because the contents that come back are meant to be displayed in a browser. But the URL has the same structure we have been exploring above and introduces the idea of "encoding" URL parameters.

- The base URL is https://www.google.com/search
- ?
- Two key=value parameters, separated by
  &q=%22violins+and+guitars%22
  says that the query to **search** for is "**violins and guitars**".**tbm=isch**
  says to go to the tab for image search

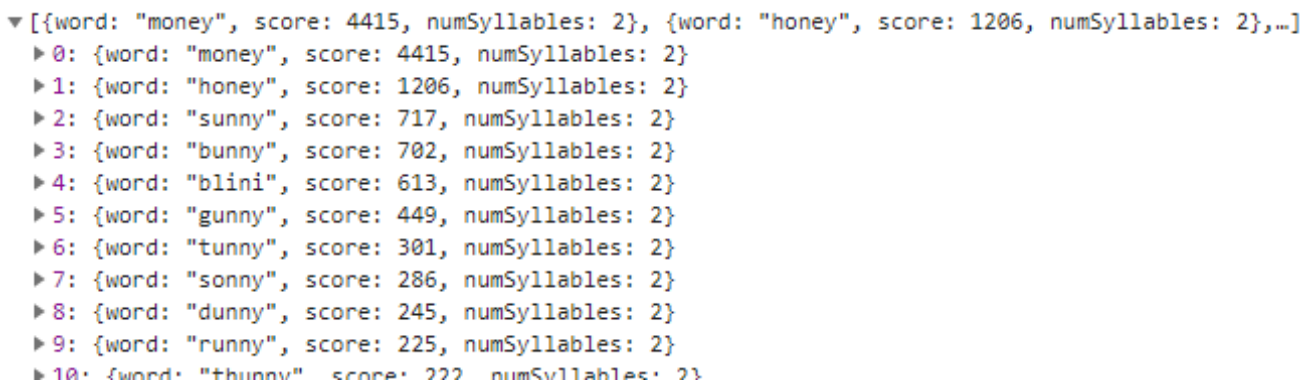
Now why is "violins and guitars" represented in the URL as %22violins+and+guitars%22?
The answer is that some characters are not safe to include, as is, in URLs. like a **URL path is not allowed** to include the **double -quote character**. It also can't include **a : or / or a space.** Whenever we want to include one of those characters in a URL, we have to encode them with other characters. **A space is encoded as +. " is encoded as %22.** : would be encoded as %3A. And so on.

Fetching a page using requests.get

You don't need to use a browser to fetch the contents of a page, though. In Python, there's a module available, called requests. You can use the get function in the requests module to fetch the contents of a page.

For illustration purposes, try visiting

https://api.datamuse.com/words?rel_rhy=funny
in your browser.



- It returns data in **JSON format**, not in HTML.
- Then try running the code below to fetch the same text string in a python program. Try changing "funny" to some other word, both in the browser, and in the code below. You'll see that, either way, you are retrieving the same thing, the datamuse API's response to your request for words that rhyme with some word that you are sending as a query parameter.

```python
#importing requests module to use REST API
#json for aligning the output in readable format
import requests, json

#using get function to call the web service linked to the url
#to get the data we want
page = requests.get("https://api.datamuse.com/words?rel_rhy=funny")

#it will the output  we got is of which type
print(type(page))

print(page.text[:150]) # print the first 150 characters
print(page.url) # print the url that was fetched

print("------")
x = page.json() # turn page.text into a python object

#post conversion of page.json the data would turn into
#a list of dictionaries
print(type(x))

print("---first item in the list---")
print(x[0])

print("---the whole list, pretty printed---")
print(json.dumps(x, indent=2)) # pretty print the results

```

```
<class 'requests.models.Response'>
[{"word":"money","score":4415,"numSyllables":2},{"word":"honey","score":1
umSyllables":2},{"word":"
https://api.datamuse.com/words?rel_rhy=funny
------
<class 'list'>
---first item in the list---
{'word': 'money', 'score': 4415, 'numSyllables': 2}
---the whole list, pretty printed---
```

- Once we run **requests.get**, **a python object** is **returned**. It's an **instance** of a **class called Response** that is **defined** in the **requests module**.
- Each instance of the class will have some attributes; different instances will have different values for the same attributes as the requests would be different. All instances can also invoke certain methods that are defined for the class.

the 3 major attribute that we saw in above code were:

- The **.text attribute** that you saw on line 12, **contains** the **contents of the file** or other information available from the url (or **sometimes an error message**).

- The **.url attribute**. We will see soon that **requests.get** takes an **optional second parameter** that is **used to add** some characters to the end of the base url that is the first parameter. The .url attribute **displays the full url** that was **generated from the input parameters**. It can be helpful for debugging purposes; you can print out the URL, paste it into a browser, and see exactly what was returned.

- The **.json()** method converts the text into a python list or dictionary, by passing the contents of the .text attribute to the jsons.loads function.

- The **.status_code attribute**
  - When a server thinks that it is sending back what was requested, it sends the code 200.
  - When the requested page doesn't exist, it sends back code 404, which is sometimes described as "File Not Found".
  - When the page has moved to a different location, it sends back code 301 and a different URL where the client is supposed to retrieve from. In the full implementation of the requests module, the get function is so smart that when it gets a 301, it looks at the new url and fetches it. Example github.

- **.headers attribute** has as its value a dictionary consisting of keys and values. To **find out all the headers,** you can run the code and add a statement **print(p.headers.keys())**

- The **.history attribute** contains a list of previous responses, if there were redirects.

To summarize, a Response object, in the full implementation of the requests module has the following useful attributes that can be accessed in your program:

  - .text
  - .url
  - .json()
  - .status_code
  - .headers
  - .history

Using requests.get to encode URL parameters

Fortunately, when you want to pass information as a URL parameter value, you don't have to remember all the substitutions that are required to encode special characters. Instead, that capability is built into the requests module.

The **get** function in the requests module takes an **optional parameter called params**. If a value is specified for that parameter, it should be a dictionary. The keys and values in that dictionary are used to append something to the URL that is requested from the remote site.

For example,as we saw earlier
https://www.google.com/search?q=%22violins+and+guitars%22&tbm=isch.
A dictionary with two parameters were passed.

d = {'q': '"violins and guitars"', 'tbm': 'isch'}

d = {'q': '"violins and guitars"', 'tbm': 'isch'}
results = requests.get("https://google.com/search", params=d)
print(results.url)

```
d = {'q': '"violins and guitars"', 'tbm': 'isch'}
results = requests.get("https://google.com/search", params=d)
print(results.url)
print(results.text[:150])
```

```
https://www.google.com/search?q=%22violins+and+guitars%22&tbm=isch
<!DOCTYPE html PUBLIC "-//WAPFORUM//DTD XHTML Mobile 1.0//EN" "http://www.w
p://www.w3.org/1999/xht
```

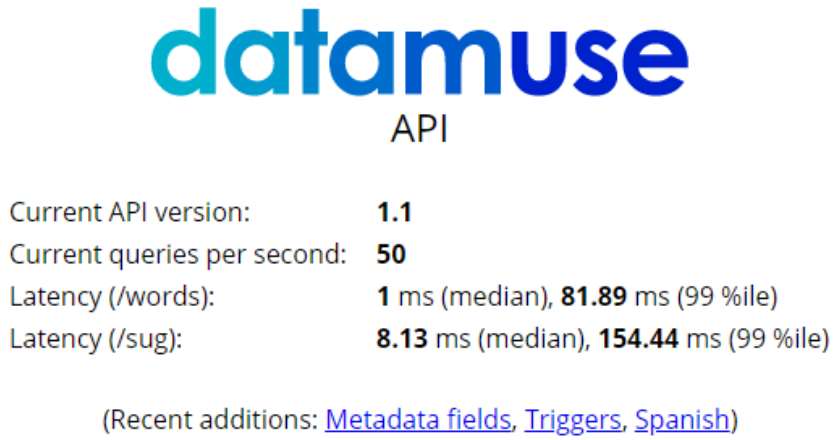
Figuring Out How to Use a REST API

There are five questions that you'll need to answer in order to use REST API:

- What is the baseurl?
- What keys should you provide in the dictionary you pass for the params parameter?
- What values should you provide associated with those keys?
- Do you need to authenticate yourself as a licensed user of the API and, if so, how?
- What is the structure and meaning of the data that will be provided?

The **answers to these questions** always **depend on design choices** made **by** the **service provider who is running the server**. Thus, the official documentation they provide will usually be the most helpful. It may also be helpful to find example code snippets or full URLs and responses; if you don't find that in the documentation, you may want to search for it on Google or StackOverflow.

Example: the datamuse API



we will be using datamuse API as an example:
https://www.datamuse.com/api/

will try to find answers one by one that we read above:

What is the baseurl?

As soon as you land on the URL, you will see details about **"what is Datamuse API"** and "**What is it good for?"** where you will read how even Amazon Echo/Alexa enabled devices are also using this API

Below that there is a column header titled,
"…use https://api.datamuse.com…"

That specifies the first part of the URL:
"https://api.datamuse.com/".
However, below that all of the examples include some additional characters after the **/ and before the ?: either words or sug**. These are called **endpoints**.
Thus, the **baseurl will be** one of the **two endpoints**, either **https://api.datamuse.com/words** or **https://api.datamuse.com/sug.**

As we know the answers of 1 which is baseurl, let's try to find out answers to question 2 & 3:

- What keys should you provide in the dictionary you pass for the params parameter?
- What values should you provide associated with those keys?

The answers to above questions, about the contents of the value of the params dictionary, can be found in the section of the documentation that describes the particular endpoint.

- When you take a look at the documentation for the **“words” endpoint**. The entire request will return some words, and all of the params contents will specify constraints that restrict the search.
- If the url includes **ml=funny** (**ml as you** would read is a **"Means like" constraint**: require that the results have a meaning related to this string value, which can be any word or sequence of words), then all the **words** that will be **returned** will **“have a meaning related to” to the word funny**.
- If the url includes **rel_cns=book (rel_[code]** is related_words and here cns is consonant match), then all the words returned will have “Consonant match” to “book”. It’s not clear exactly what that means, but it includes words like bike and back: you can try it by visiting https://api.datamuse.com/words?rel_cns=book

```python
import requests, json

#using get function to call the web service linked to the url
#to get the data we want
page = requests.get("https://api.datamuse.com/words?rel_cns=book")
x = page.json()
print("---the whole list, pretty printed---")
print(json.dumps(x, indent=2))
```

```
---the whole list, pretty printed---
[
  {
    "word": "back",
    "score": 9938,
    "numSyllables": 1
  },
```

The **words to the left of the =,** like **ml** and **rel_cns** and **rel_rhy**, will be keys in the dictionary that you pass as the value of params in the call to requests.**get.** The values associated with those keys will be words, like book and funny

Many **providers of APIs require** you to **register in advance** to make use of an **API**, and then **authenticate yourself with each request.** That way they can charge money, or restrict usage in some way. A popular form of authentication is to have a personal **“api_key”** that you pass as one of the **key=value pairs** in the URL.

- For example, the flickr API requires that, we will see later
- Some services, such as Facebook and Twitter, require an even more complex, and secure, form of authentication, where a credential is used to cryptographically sign requests. We will not cover the use of that more complex authentication, as it is considerably harder to debug.

Datamuse is free until now, snippet frmo the website

Usage limits

You can use this service without restriction and without an API key for up to 100,000 requests per day. Please be aware that beyond that limit, keyless requests may be rate-limited without notice. If you'd like to use this in a customer-facing application, or if you need a custom vocabulary, or if you plan to make more than 100,000 requests per day, you can get an API key and a stable version to use with a greater availability guarantee. API keys are free for noncommercial use.

For Data interpretation, go through the ***Interpreting the results part***

Defining a function to make repeated invocations:

Suppose you were writing a computer program that was going to automatically translate paragraphs of text into paragraphs with similar meanings but with more rhymes. You would want to contact the datamuse API repeatedly, passing different values associated with the key **rel_rhy**. Let's make a python function to do that. You can think of it as a wrapper for the call to **requests.get.**

```python
import json, requests

def get_rhymes(word):
    #step 1 defining the baseurl
    #we have used 'words' for endpoint
    #you can try using sug too
    baseurl="https://api.datamuse.com/words"

    #step 2&3 defining the keys for params
    params_dict={}
    #as we saw rel_rhy i.e relates rhyme and word
    #becomes a pair and max,3 would be another
    #so 2 parasm with and operation will be sent as RESPT API
    params_dict['rel_rhy']=word
    params_dict['max']='3'

    #here we are recording the response
    resp=requests.get(baseurl,params=params_dict)

    #checking what exactly url was sent for retreiving data
    print(resp.url)

    #converting the response in json format
    words=resp.json()

    #printing with pretty_print to see what was the result
    print(json.dumps(words,indent=2))

    #returning all words returning using list comprehencsion
    return [rhyme['word'] for rhyme in words]

print(get_rhymes('hey'))
print(get_rhymes('sunny'))
```

```
https://api.datamuse.com/words?rel_rhy=hey&max=3
[
  {
    "word": "day",
    "score": 9228,
    "numSyllables": 1
  },
  {
    "word": "away",
    "score": 9015,
    "numSyllables": 2
  },
  {
    "word": "way",
    "score": 6428,
    "numSyllables": 1
  }
]
['day', 'away', 'way']
https://api.datamuse.com/words?rel_rhy=sunny&max=3
[
  {
    "word": "money",
    "score": 4415,
    "numSyllables": 2
  },
  {
    "word": "funny",
    "score": 1265,
    "numSyllables": 2
  },
  {
    "word": "honey",
    "score": 1206,
    "numSyllables": 2
  }
]
['money', 'funny', 'honey']
```

Debugging calls to requests.get()

you will not always get a Response object back from a call to --> requests.get.

It will be an informative message telling us where exactly it failed.

First thing that might go wrong:

when you call requests.get(dest_url). There are two possibilities for what's gone wrong in that case:

- One possibility is that the value provided for the params parameter is not a valid dictionary or doesn't have key-value pairs that can be converted into text strings suitable for putting into a URL.
    - For example, if you execute requests.get("http://github.com", params = [0,1]), [0,1] is a list rather than a dictionary and the python interpreter generates the error, TypeError: 'int' object is not iterable.

- The second possibility is that the variable dest_url is either not bound to a string, or is bound to a string that isn't a valid URL.
    - For example, it might be bound to the string "http://foo.bar/bat". foo.bar is not a valid domain name that can be resolved to an ip address, so there's no server to contact. That will yield an error of type requests.exceptions.ConnectionError.

ConnectionError: HTTPSConnectionPool(host='api.datamuse.com', port=443): Max retries exceeded with url: /words?rel_cns=book (Caused by NewConnectionError('<urllib3.connection.VerifiedHTTPSConnection object at 0x000001FFB7379AF0>: Failed to establish a new connection: [Errno 11001] getaddrinfo failed'))

The best approach is to look at the URL that is produced, eyeball it, and plug it into a browser to see what happens. Unfortunately, if the call to requests.get

```python
import requests
def requestURL(baseurl, params = {}):
    # This function accepts a URL path and a params diction as inputs.
    # It calls requests.get() with those inputs,
    # and returns the full URL of the data you want to get.
    req = requests.Request(method = 'GET', url = baseurl, params = params)
    prepped = req.prepare()
    return prepped.url

print(requestURL(some_base_url, some_params_dictionary))
```

Assuming requestURL() returns a URL, match up what you see from the printout of the params dictionary to what you see in the URL that was printed out. If you have a sample of a URL from the API documentation, see if the structure of your URL matches what's there. Perhaps you have misspelled one of the API parameter names or you misspelled the base url.

Fortunately, the response object returned by **requests.get()** has the **.url** attribute, which will help you with debugging. It's a good practice during program development to have your program print it out. This is easier than calling **requestURL()** but is only available to you if **requests.get()** succeeds in returning a Response object.

More importantly, you'll want to print out the contents. Sometimes the text that's retrieved is an error message that you can read, such as *{"request empty": "There is no data that corresponds to your search."}.*

In other cases, it's just obviously the wrong data. Print out the first couple hundred characters of the response text to see if it makes sense.

```
import requests
dest_url = <some expr>
d = <some dictionary>
resp = requests.get(dest_url, params = d)
print(resp.url)
print(resp.text[:200])
```

Assuming requestURL() returns a URL, match up what you see from the printout of the params dictionary to what you see in the URL that was printed out. If you have a sample of a URL from the API documentation, see if the structure of your URL matches what's there. Perhaps you have misspelled one of the API parameter names or you misspelled the base url.

Fortunately, the response object returned by **requests.get()** has the **.url** attribute, which will help you with debugging. It's a good practice during program development to have your program print it out. This is easier than calling **requestURL()** but is only available to you if **requests.get()** succeeds in returning a Response object.

More importantly, you'll want to print out the contents. Sometimes the text that's retrieved is an error message that you can read, such as *{"request empty": "There is no data that corresponds to your search."}.*

In other cases, it's just obviously the wrong data. Print out the first couple hundred characters of the response text to see if it makes sense.

Caching Response Content

For now we have seen how we can retrieve the data by using REST API. But lets give it a thought, not all data that you get will be easy, some will be too complicated data and it might have taken you several tries to compose it at the first placed, you would probably need to save it.

Or else debugging the same again and again would become a pain point. It is a good practice, for many reasons, not to keep contacting a REST API to re-request the same data every time you run your program.

To **avoid re-requesting** the same data, **we will use** a programming pattern **known as caching**. It works like this:

- Before doing some expensive operation (like calling requests.get to get data from a REST API), check whether you have already saved ("cached") the results that would be generated by making that request.
- If so, return that same data.
- If not, perform the expensive operation and save ("cache") the results (e.g. the complicated data) in your cache so you won't have to perform it again the next time.

If you go on to learn about web development, you'll find that you encounter caching all the time – if you've ever had the experience of seeing old data when you go to a website and thinking, *"Huh, that's weird, it should really be different now… why am I still seeing that?"* that happens because the browser has accessed a cached version of the site.

Why is caching a good idea during your software development using REST API?

- It reduces load on the website that is providing you data. It is always nice to be courteous when using other people's resources. Moreover, some websites impose rate limits: for example, after 15 requests in a 15 minute period, the site may start sending error responses. That will be confusing and annoying for you

- It will make your program run faster. Connections over the Internet can take a few seconds, or even tens of seconds, if you are requesting a lot of data. It might not seem like much, but debugging is a lot easier when you can make a change in your program, run it, and get an almost instant response.

- It is harder to debug the code that processes complicated data if the content that is coming back can change on each run of your code. It's amazing to be able to write programs that fetch real-time data like the available iTunes podcasts or the latest tweets from Twitter. But it can be hard to debug that code if you are having problems that only occur on certain Tweets (e.g. those in foreign languages). When you encounter problematic data, it's helpful if you save a copy and can debug your program working on that saved, static copy of the data.

- It is easier to run automated tests on code that retrieves data if the data can never change, for the same reasons it is helpful for debugging. In fact, we rely on use of cached data in the automated tests that check your code in exercises.

PART 10