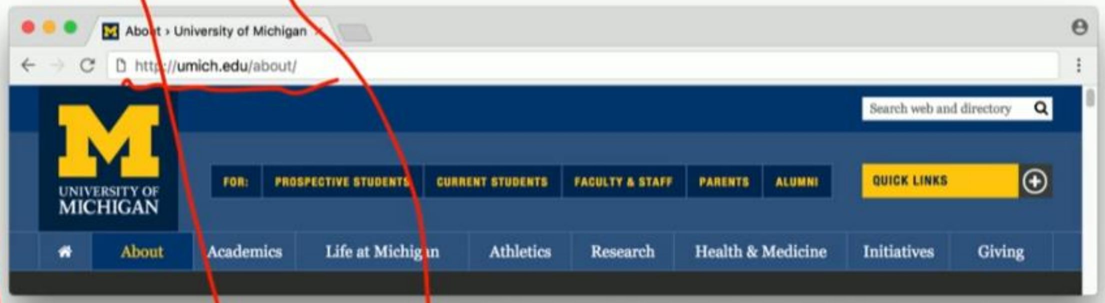
When you go to a website, you might see something like http://umich.edu/about in the URL bar:

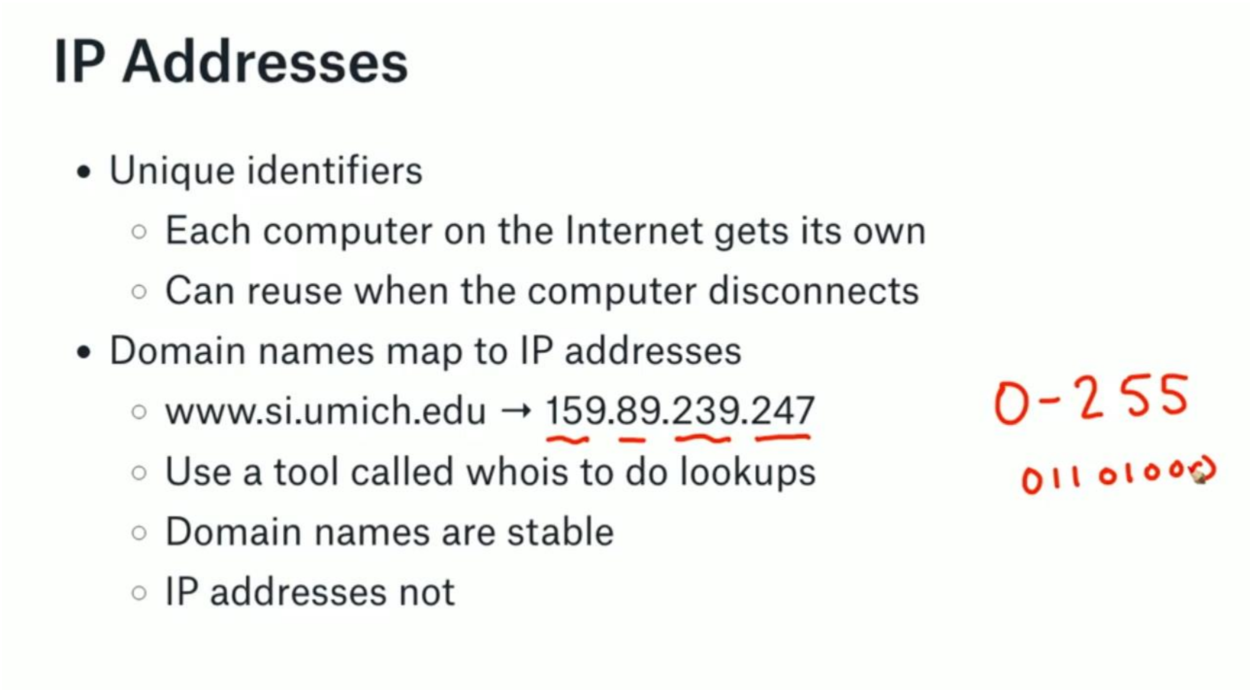


```
{protocol}://{server}/{arguments}
```

Host (server) names

The **host name** is a name for the specific computer you're communicating with. Some example host names:

- si.umich.edu
- www.theride.org
- www.google.com



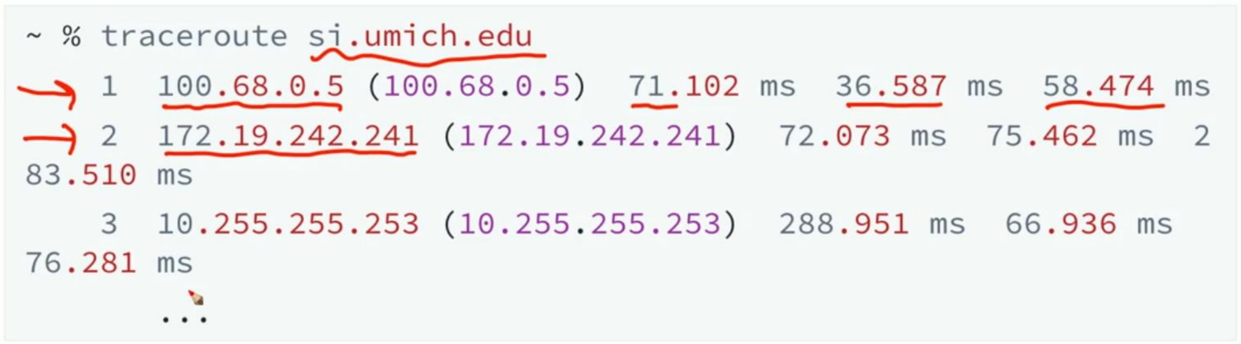
IP Addresses

- Unique identifiers
  - Each computer on the Internet gets its own
  - Can reuse when the computer disconnects
- Domain names map to IP addresses
  - www.si.umich.edu → 159.89.239.247
  - Use a tool called whois to do lookups
  - Domain names are stable
  - IP addresses not

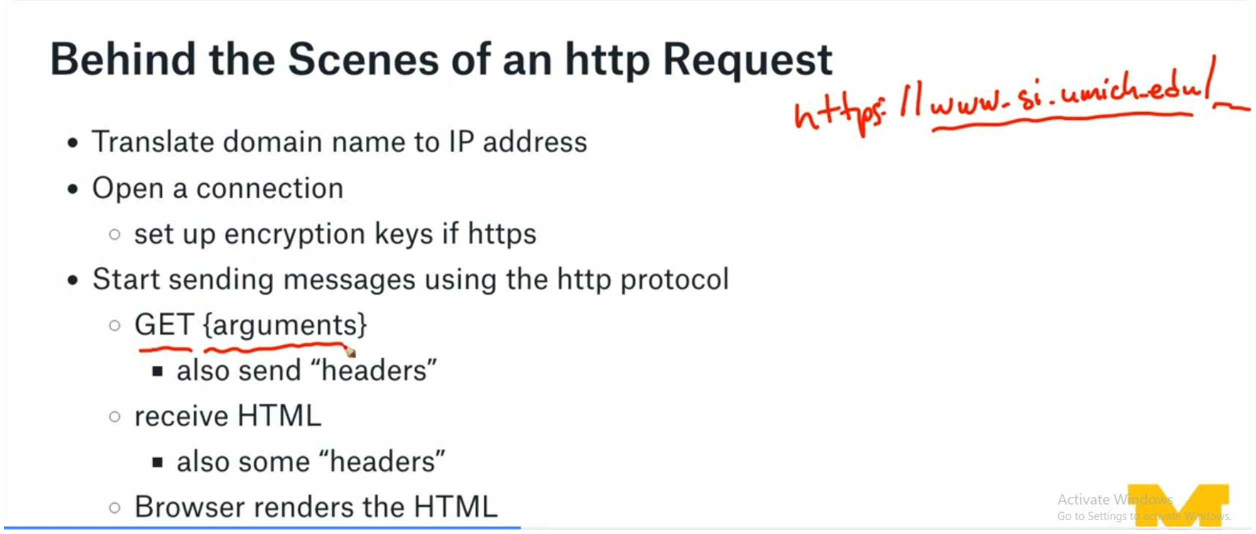
0-255

01101000

To reach si.umich.edu the packets hop from one router to another. Here 1 to 2 to 3 and after 8-9 hops it reaches the website.



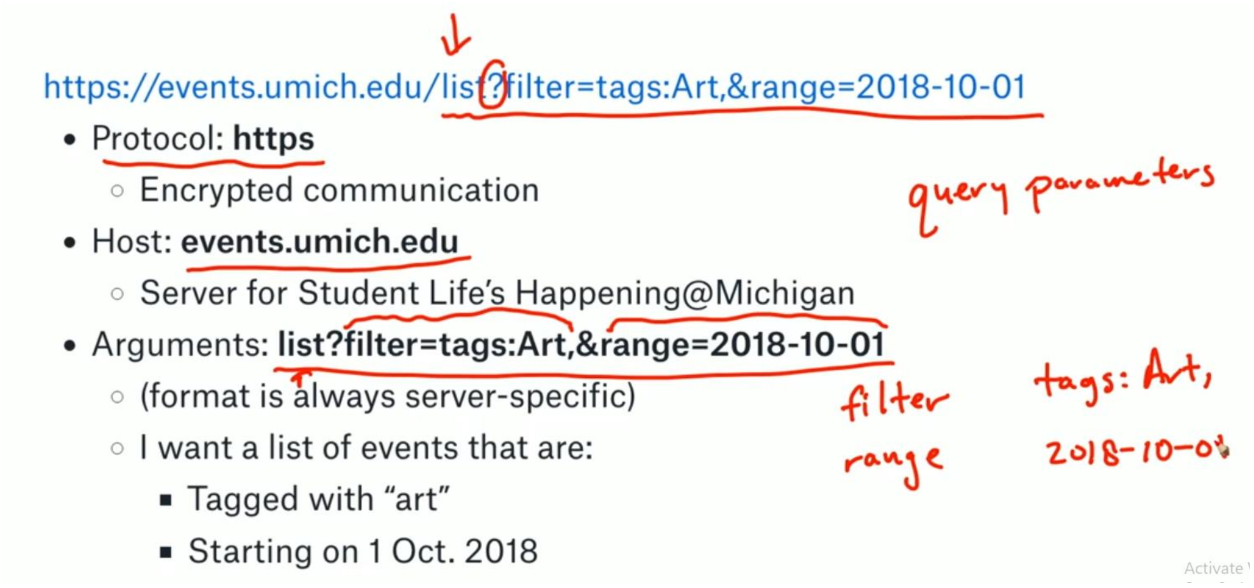
```
~ % traceroute si.umich.edu
    1   100.68.0.5 (100.68.0.5)  71.102 ms  36.587 ms  58.474 ms
    2   172.19.242.241 (172.19.242.241)  72.073 ms  75.462 ms  2
83.510 ms
    3   10.255.255.253 (10.255.255.253)  288.951 ms  66.936 ms
76.281 ms
        ...
```



Behind the Scenes of an http Request

https://www.si.umich.edu/

- Translate domain name to IP address
- Open a connection
  - set up encryption keys if https
- Start sending messages using the http protocol
  - GET {arguments}
    - also send "headers"
  - receive HTML
    - also some "headers"
  - Browser renders the HTML



https://events.umich.edu/list?filter=tags:Art,&range=2018-10-01

- Protocol: **https**
  - Encrypted communication
- Host: **events.umich.edu**
  - Server for Student Life's Happening@Michigan
- Arguments: **list?filter=tags:Art,&range=2018-10-01**
  - (format is always server-specific)
  - I want a list of events that are:
    - Tagged with "art"
    - Starting on 1 Oct. 2018

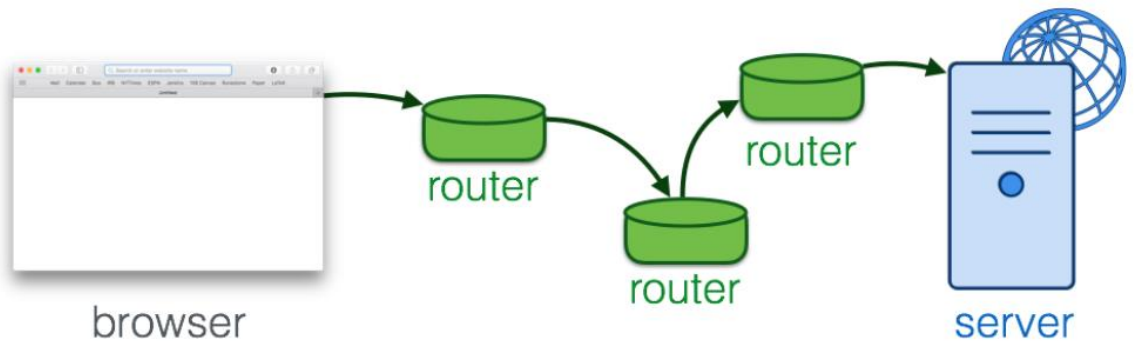
query parameters

filter	tags: Art,
range	2018-10-01

24.2. The Internet: Behind the Scenes

The Internet is a transport mechanism that lets any connected device communicate with any other connected device. Behind the scenes:

- Each 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 141.211.203.248. Any IP address beginning with 141.211 is for a device at the University of Michigan. When I take my laptop home and connect to a network there, my laptop gets a different IP address that it uses there.
- 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.
- 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. 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. It is this technical capability that has spawned metaphoric quotes like this one from John Gilmore: "The Net interprets censorship as damage and routes around it."
- At the destination, the packets are reassembled into the original data message.



24.3. 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>
```

Usually, the *scheme* will be http or https. The s in https 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.

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

The *host* will usually be a domain name, like si.umich.edu or 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 192.30.252.130. 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.

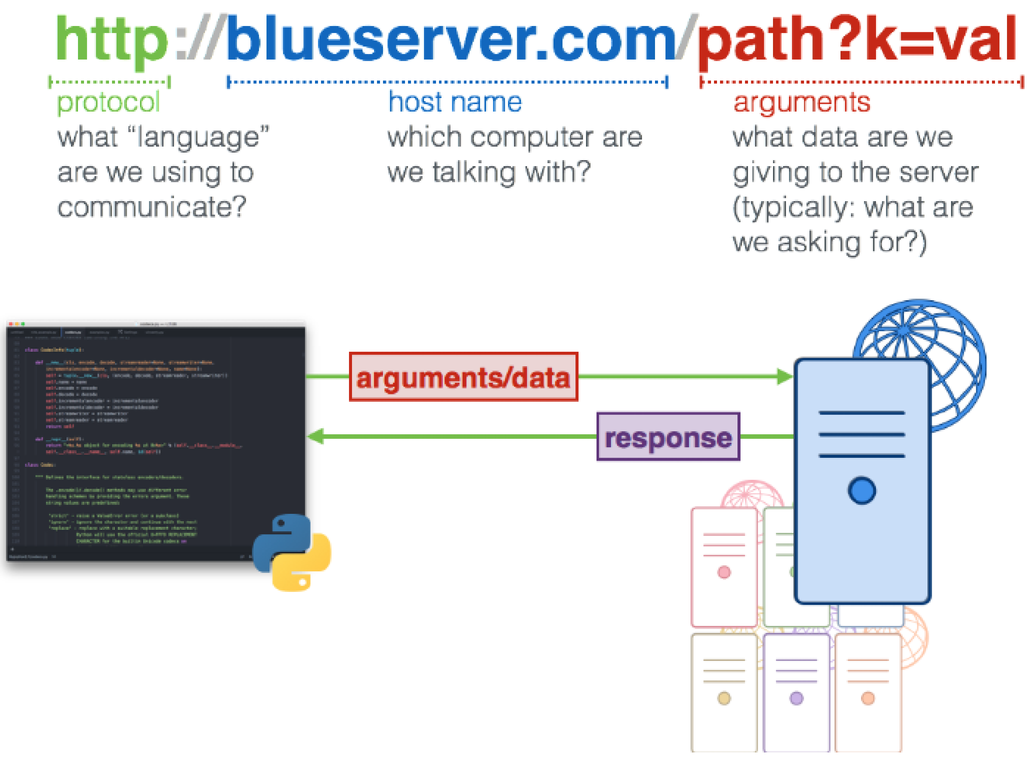
The *:port* is optional. If it is omitted, the default port number is 80. The port number is used on the receiving end to decide which computer program should get the data that has been received. We probably will not encounter any URLs that include the : and a port number in this course.

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

For example, consider the url https://github.com/presnick/runestone:

- 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 192.30.252.130. The connection will be made on the default port, which is 443 for https.
- /presnick/runestone says to ask the remote server for the page presnick/runestone. 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 https://github.com/presnick/runestone because it includes what are called “query parameters”, the information after the `?`.

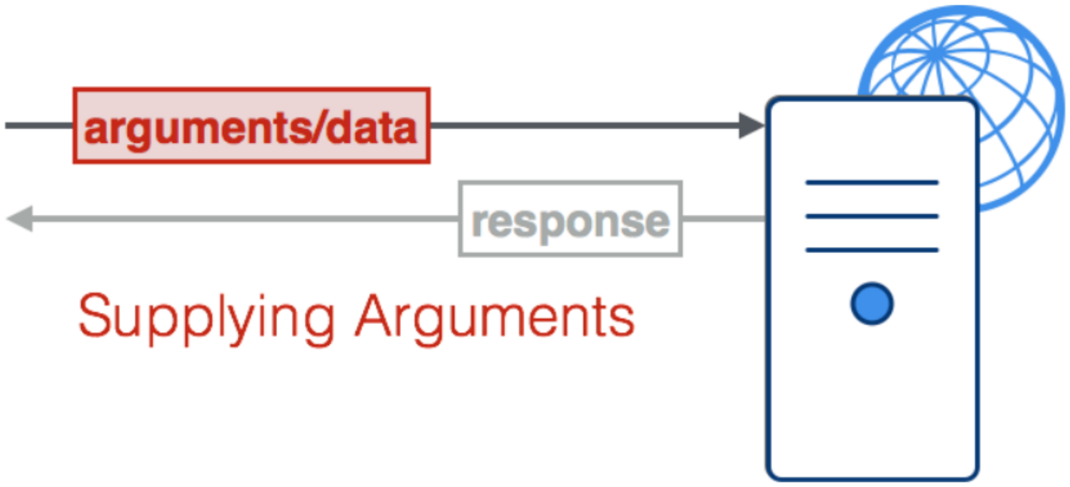

http://blueserver.com/path?k=val
protocol
what “language” are we using to communicate?
host name
which computer are we talking with?
arguments
what data are we giving to the server (typically: what are we asking for?)
arguments/data
response

24.4. 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.

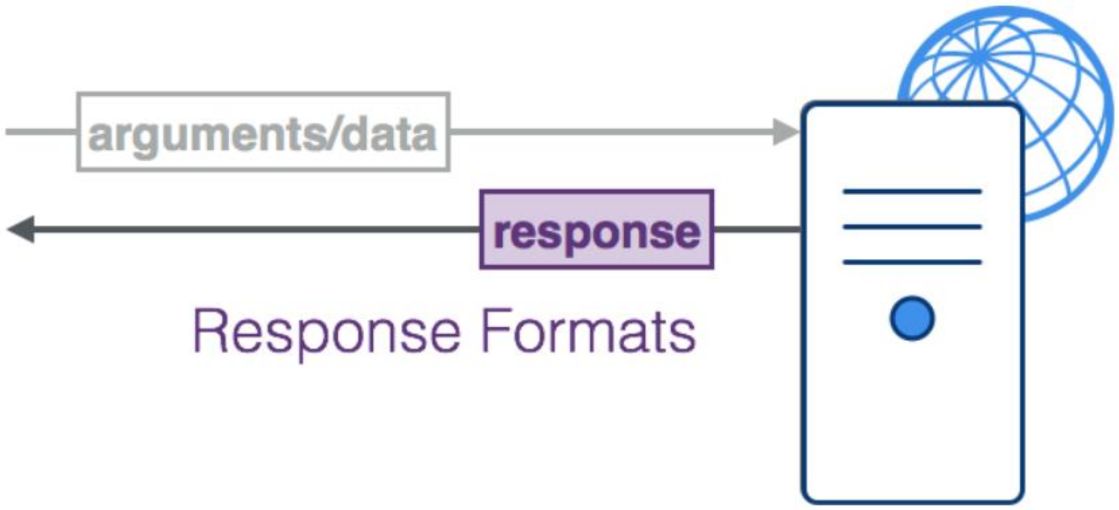
HTTP is the protocol that specifies how web browsers or other programs communicate with web servers. One version of the formal specification, before it was later split into multiple documents, was IETF RFC 2616. It is 176 pages long! Fortunately, the basics are pretty easy to understand.

- **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
    - 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.**
  - **The server first sends back some HTTP headers. These include:**
    - 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.



24.5. 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.

Putting the two together, a REST API 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.

Of course, a web browser is just a computer program, so all requests to web sites come from other computer programs. But usually a browser requests data from a web site in order to display it directly to a human user of the browser.

Prior to the development of REST APIs, there were other ways that computer programs made remote requests to other computers on a network, asking them to perform some computation or retrieve some data. Those other techniques are still in use. REST APIs are particularly convenient, however, both for students and for others, because it is easy to see what is going on in a request and a response between two computers, and thus it is easier to debug.

We will examine a common pattern used in REST APIs, where there is a base URL that defines an “endpoint”, and then additional information is appended to the URL as query parameters, and the response comes back not as HTML but as a format called JSON. Along the way, we will see some functions in python modules that are helpful for constructing the URLS and for processing the JSON formatted response.

24.5.1. URL Structure for REST APIs

In a REST API, the client or application program– the kind of program you will be writing– makes an HTTP request that includes information about what kind of request it is making. Web sites are free to define whatever format they want for how the request should be formatted. This chapter covers a particularly common and particularly simple format, where the request information is encoded right in the URL. This is convenient, because if something goes wrong, we can debug by copying the URL into a browser and see what happens when it tries to visit that URL.

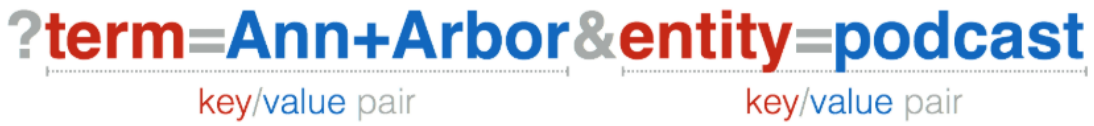
In this format, the URL has a standard structure:

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

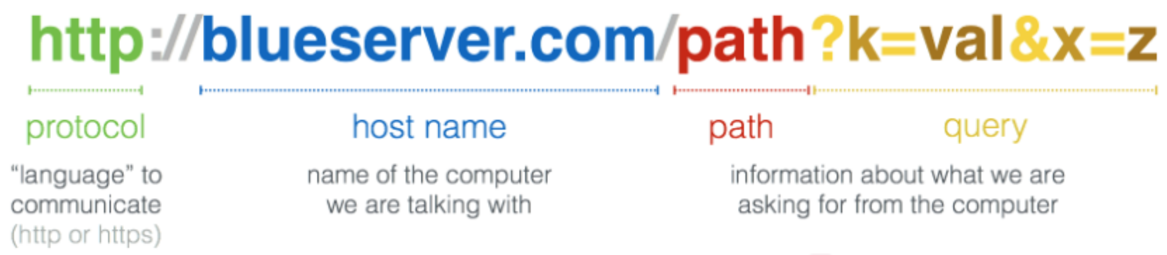
For example, consider the URL https://itunes.apple.com/search?term=Ann+Arbor&entity=podcast. Try copying that URL into a browser, or just clicking on it. It retrieves data about podcasts posted from Ann Arbor, MI. Depending on your browser, it may put the contents into a file attachment that you have to open up to see the contents, or it may just show the contents in a browser window.

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.



All those parts are concatenated together to form the full URL.


http://blueserver.com/path?k=val&x=z
protocol
"language" to communicate (http or https)
host name
name of the computer we are talking with
path
query
information about what we are asking for from the computer
Note that in the search term Ann Arbor, the space had to be "encoded" as `+`. More on that below.

24.5.2. 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. For example, 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.

24.6. Fetching a page

The web works with a metaphor of “pages”. When you put a URL into a browser, you see a “page” of content.

For example, if you visit https://github.com/RunestoneInteractive/RunestoneServer, you will see the home page for the open source project whose contents are used to run this online textbook.

The browser is just a computer program that fetches the contents and displays them in a nice way. If you want to see what the contents are, in plain text, right click your mouse on the page and select `View source`, or whatever the equivalent is in your browser.

24.6.1. Fetching in python with 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.

Note

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. Your browser will display the results, information about some words that rhyme with “funny”, but it won’t look like a normal web page. 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.

24.6.2. More Details of Response objects

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. We won’t look at it’s definition. Think of it as analogous to the Turtle class. Each instance of the class has some attributes; different instances have different values for the same attribute. All instances can also invoke certain methods that are defined for the class.

In the Runestone environment, we have a very limited version of the `requests` module available. The Response object has only two attributes that are set, and one method that can be invoked.

- The *.text* attribute. It contains the contents of the file or other information available from the url (or sometimes an error message).
- The *.url* attribute. We will see later 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. This converts the text into a python list or dictionary, by passing the contents of the *.text* attribute to the `jsons.loads` function.

The full Requests module provides some additional attributes in the Response object. These are not implemented in the Runestone environment.

- 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. For example, github redirects all requests using http to the corresponding page using https (the secure http protocol). Thus, when we ask for http://github.com/presnick/runestone, github sends back a 301 code and the url https://github.com/presnick/runestone. The requests.get function then fetches the other url. It reports a status of 200 and the updated url. We have to do further inquire to find out that a redirection occurred (see below).
- The *.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())`. One of the headers is 'Content-type'. Some possible values are `text/html; charset-utf-8` and `application/json; charset=utf-8`.
- 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 (not available in Runestone implementation)
- .headers (not available in Runestone implementation)
- .history (not available in Runestone implementation)

24.6.3. Using requests.get to encode URL parameters¶

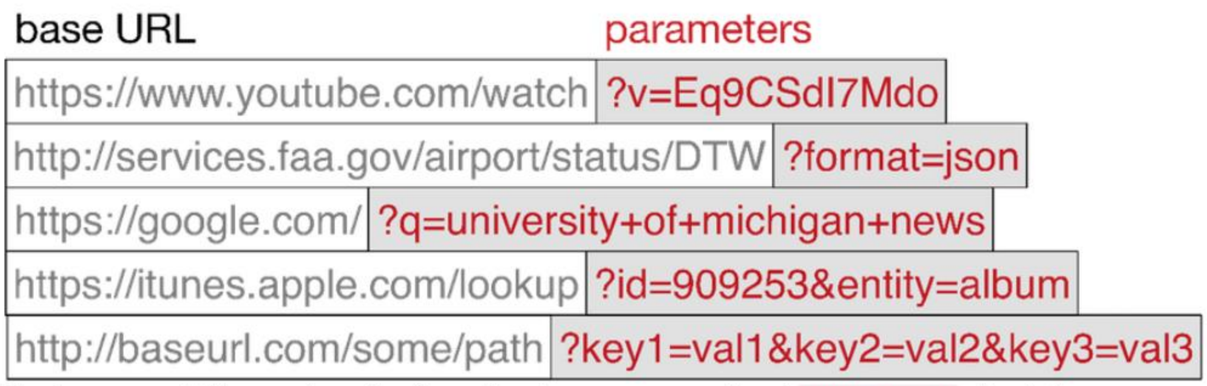
Fortunately, when you want to pass information as a URL parameter value, you don't have to remember all the substitutions that are required to encode special characters. Instead, that capability is built into the requests module.

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.

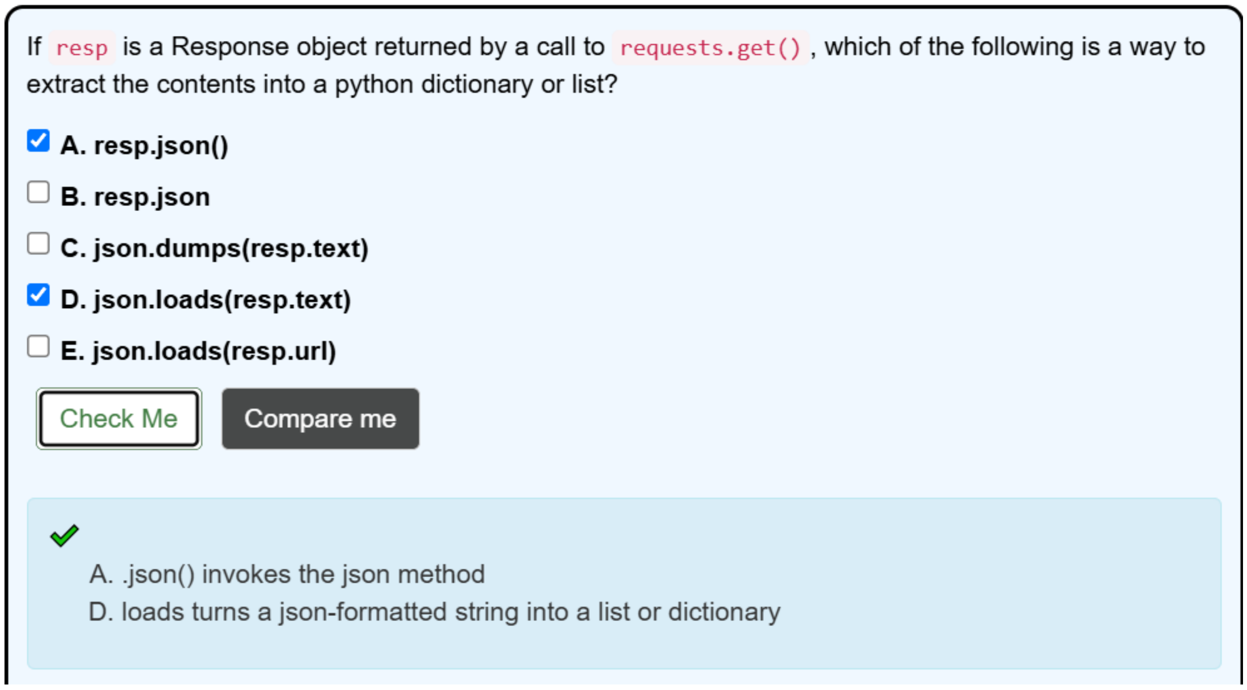
For example, in the following, the base url is https://google.com/search. A dictionary with two parameters is passed. Thus, the whole url is that base url, plus a question mark, "?", plus a "q=…" and a "tbm=…" separated by an "&". In other words, the final url that is visited is https://www.google.com/search?q=%22violins+and+guitars%22&tbm=isch. Actually, because dictionary keys are unordered in python, the final url might sometimes have the encoded key-value pairs in the other order: https://www.google.com/search?tbm=isch&q=%22violins+and+guitars%22. Fortunately, most websites that accept URL parameters in this form will accept the key-value pairs in any order.

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

Below are more examples of urls, outlining the base part of the url - which would be the first argument when calling `request.get()` - and the parameters - which would be written as a dictionary and passed into the params argument when calling `request.get()`.


base URL
parameters
https://www.youtube.com/watch
?v=Eq9CSdI7Mdo
http://services.faa.gov/airport/status/DTW
?format=json
https://google.com/
?q=university+of+michigan+news
https://itunes.apple.com/lookup
?id=909253&entity=album
http://baseurl.com/some/path
?key1=val1&key2=val2&key3=val3

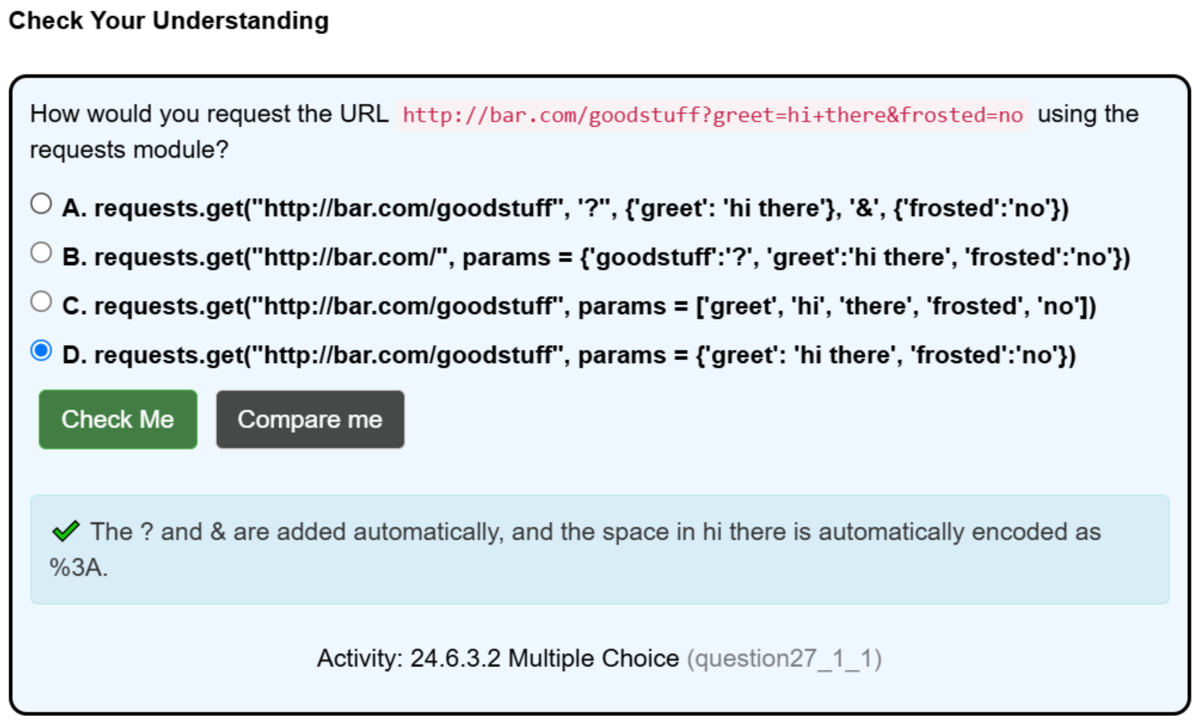
Here's an executable sample, using the optional params parameter of `requests.get`. It gets the same data from the datamus api that we saw previously. Here, however, the full url is built inside the call to `requests.get`; we can see what url was built by printing it out, on line 5.



If `resp` is a Response object returned by a call to `requests.get()`, which of the following is a way to extract the contents into a python dictionary or list?

- [✓ **A. resp.json()**]
- **B. resp.json**
- **C. json.dumps(resp.text)**
- [✓ **D. json.loads(resp.text)**]
- **E. json.loads(resp.url)**

✓
A. .json() invokes the json method
D. loads turns a json-formatted string into a list or dictionary



Check Your Understanding

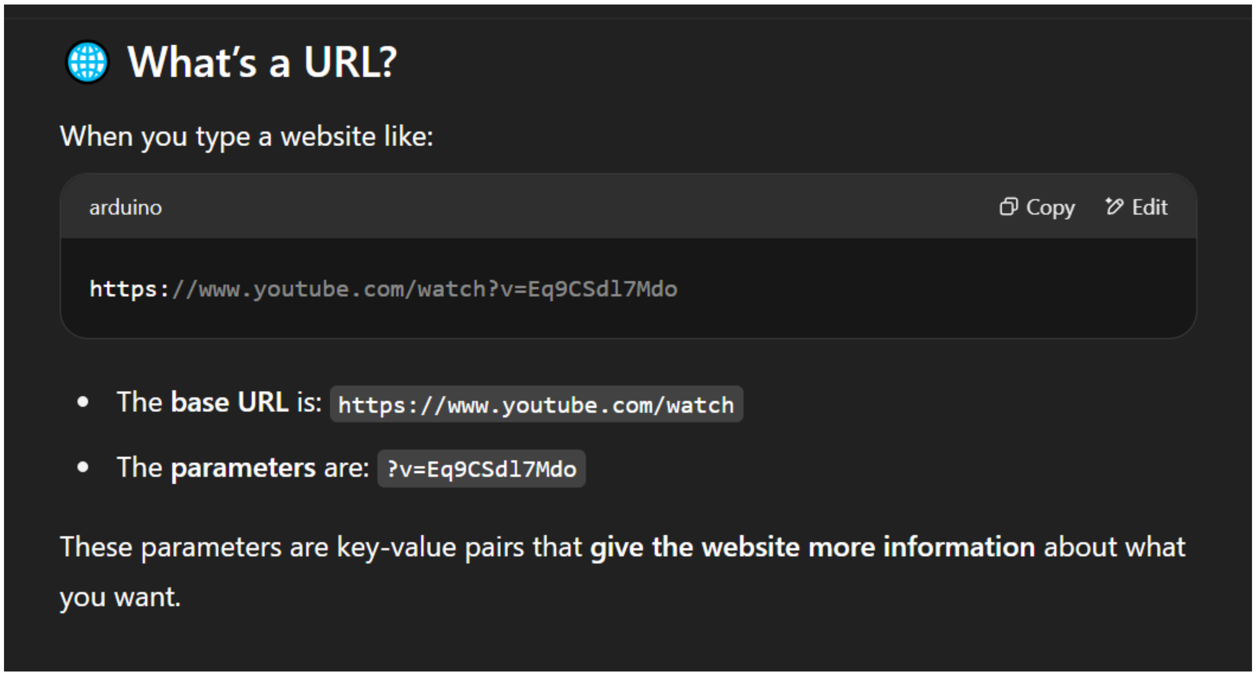
How would you request the URL `http://bar.com/goodstuff?greet=hi+there&frosted=no` using the requests module?

- **A. requests.get("http://bar.com/goodstuff", '?", {'greet': 'hi there'}, '&', {'frosted':'no'})**
- **B. requests.get("http://bar.com/", params = {'goodstuff':'?', 'greet':'hi there', 'frosted':'no'})**
- **C. requests.get("http://bar.com/goodstuff", params = ['greet', 'hi', 'there', 'frosted', 'no'])**
- [✓ **D. requests.get("http://bar.com/goodstuff", params = {'greet': 'hi there', 'frosted':'no'})**]

✓ The ? and & are added automatically, and the space in hi there is automatically encoded as %3A.

Activity: 24.6.3.2 Multiple Choice (question27_1_1)

Easier explanation:



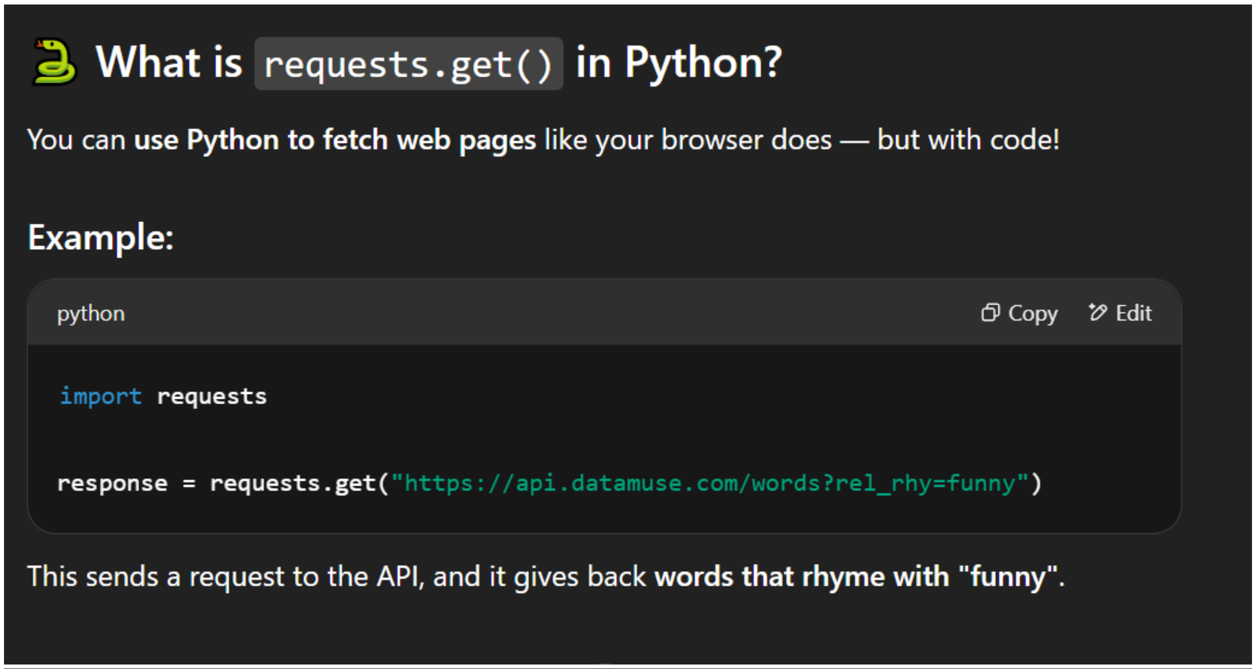
🌐 What's a URL?

When you type a website like:

```arduino
https://www.youtube.com/watch?v=Eq9CSdl7Mdo
```

- The **base URL** is: `https://www.youtube.com/watch`
- The **parameters** are: `?v=Eq9CSdl7Mdo`

These parameters are key-value pairs that **give the website more information** about what you want.



🐍 What is `requests.get()` in Python?

You can **use Python to fetch web pages** like your browser does — but with code!

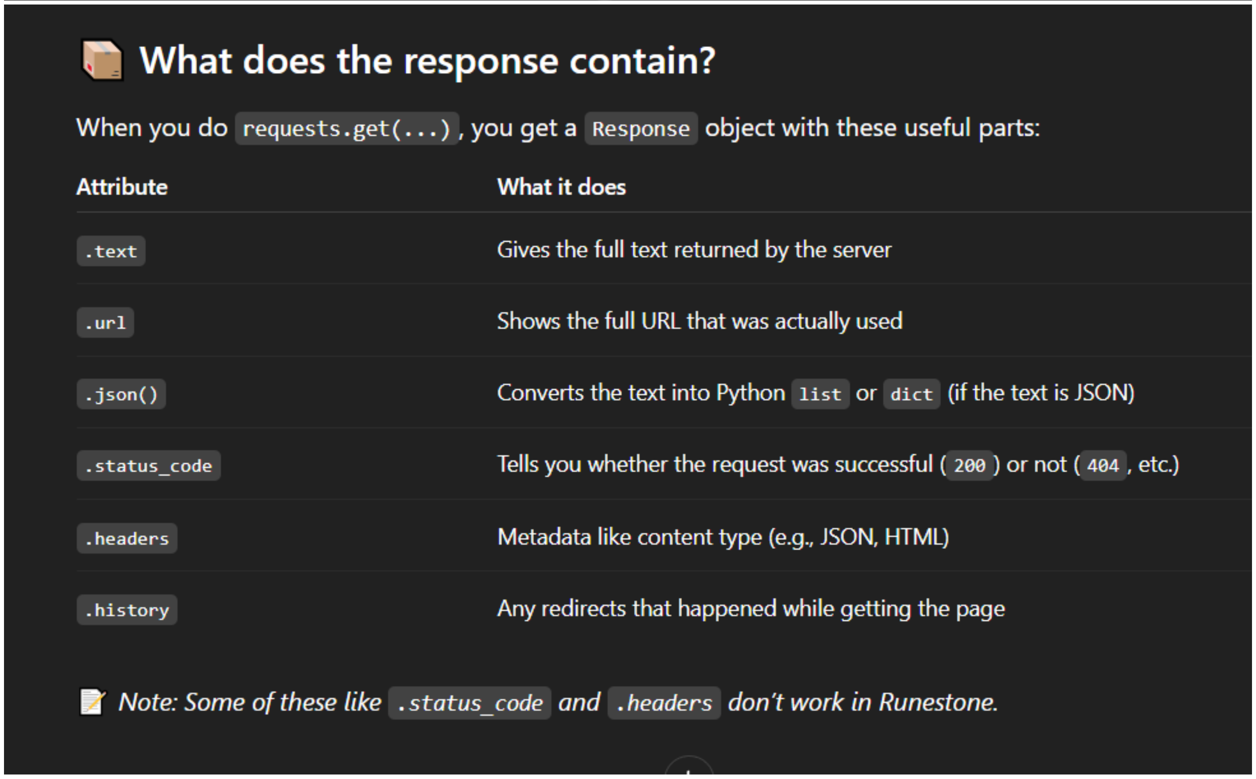
Example:

python   Copy   Edit

```python
import requests

response = requests.get("https://api.datamuse.com/words?rel_rhy=funny")
```

This sends a request to the API, and it gives back **words that rhyme with "funny"**.

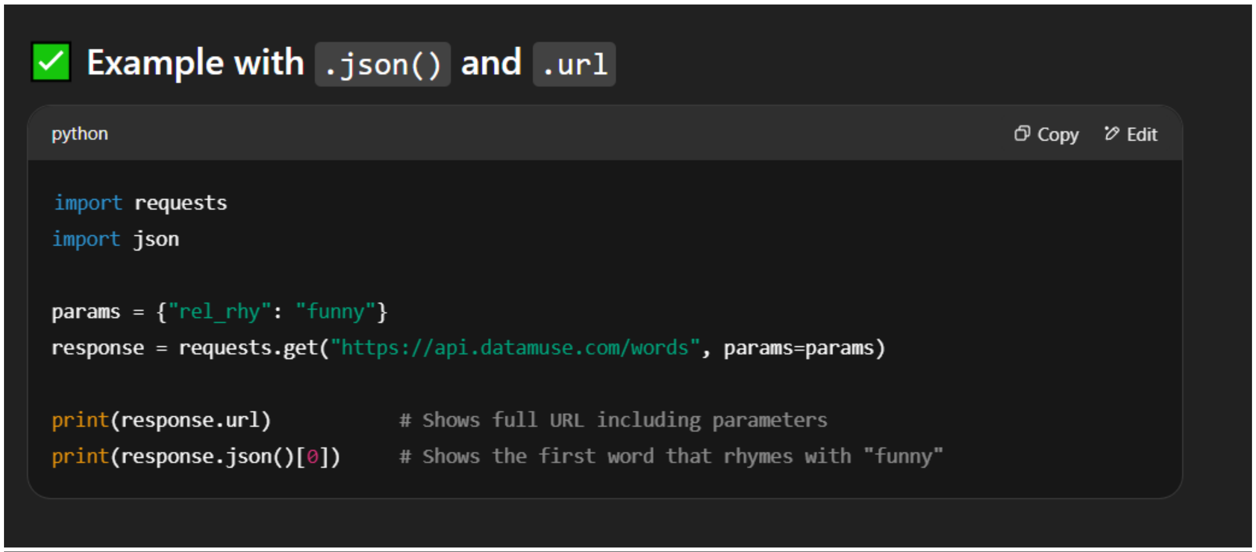


📦 What does the response contain?

When you do `requests.get(...)`, you get a `Response` object with these useful parts:

Attribute	What it does
`.text`	Gives the full text returned by the server
`.url`	Shows the full URL that was actually used
`.json()`	Converts the text into Python `list` or `dict` (if the text is JSON)
`.status_code`	Tells you whether the request was successful (`200`) or not (`404`, etc.)
`.headers`	Metadata like content type (e.g., JSON, HTML)
`.history`	Any redirects that happened while getting the page

📝 *Note: Some of these like `.status_code` and `.headers` don't work in Runestone.*

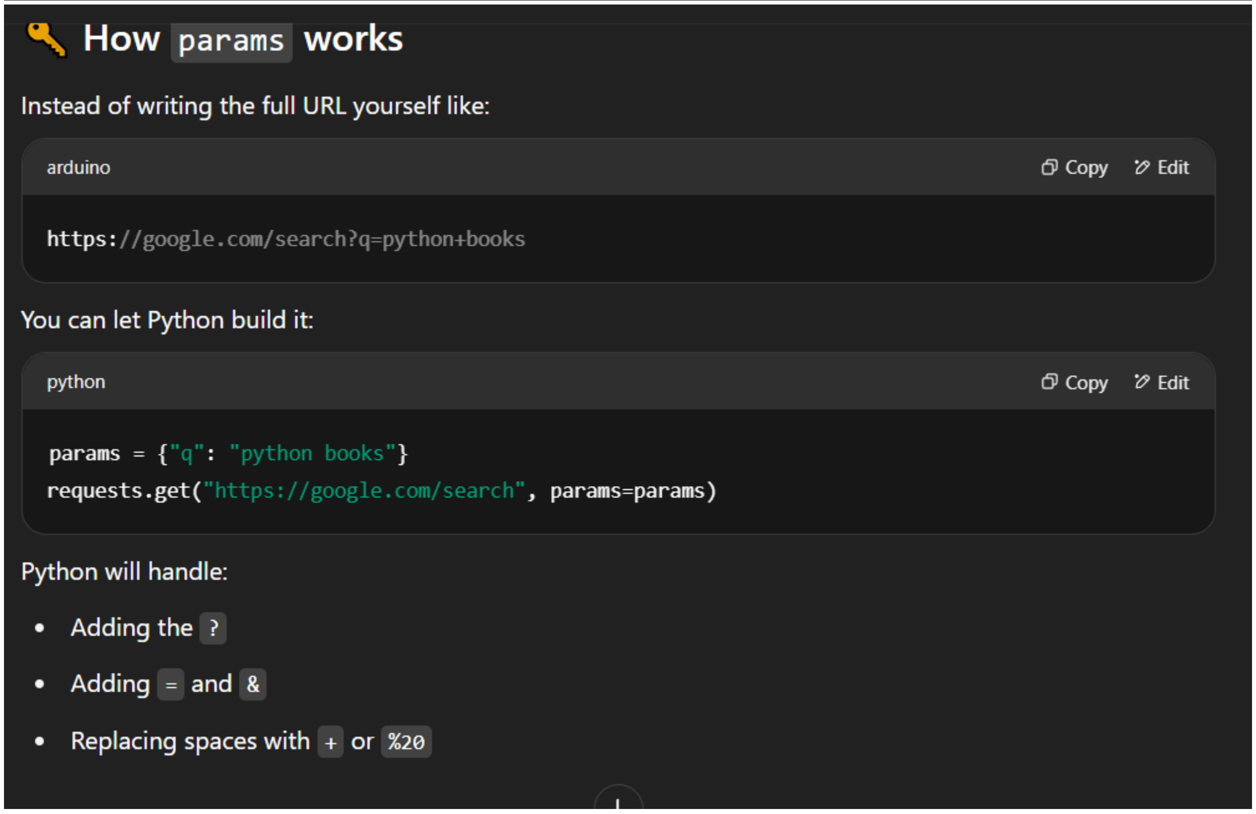


✅ Example with `.json()` and `.url`

```python
import requests
import json

params = {"rel_rhy": "funny"}
response = requests.get("https://api.datamuse.com/words", params=params)

print(response.url)            # Shows full URL including parameters
print(response.json()[0])      # Shows the first word that rhymes with "funny"
```



🔑 How `params` works

Instead of writing the full URL yourself like:

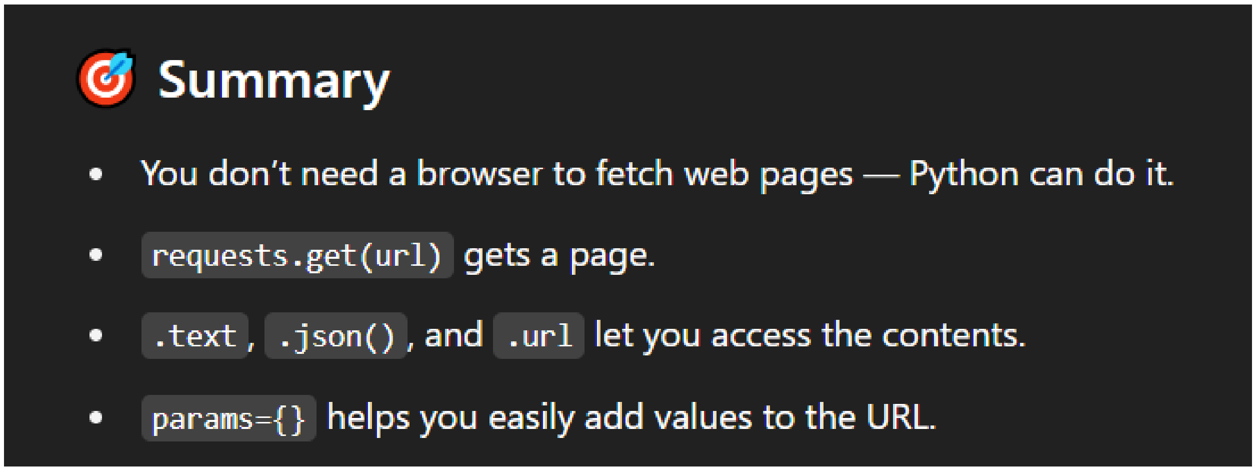
```arduino
https://google.com/search?q=python+books
```

You can let Python build it:

```python
params = {"q": "python books"}
requests.get("https://google.com/search", params=params)
```

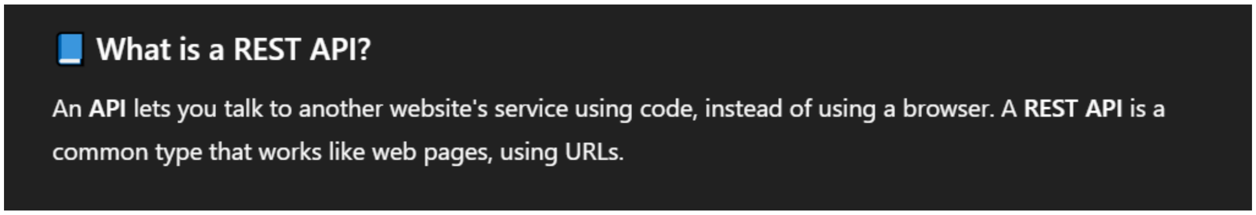
Python will handle:

- Adding the `?`
- Adding `=` and `&`
- Replacing spaces with `+` or `%20`



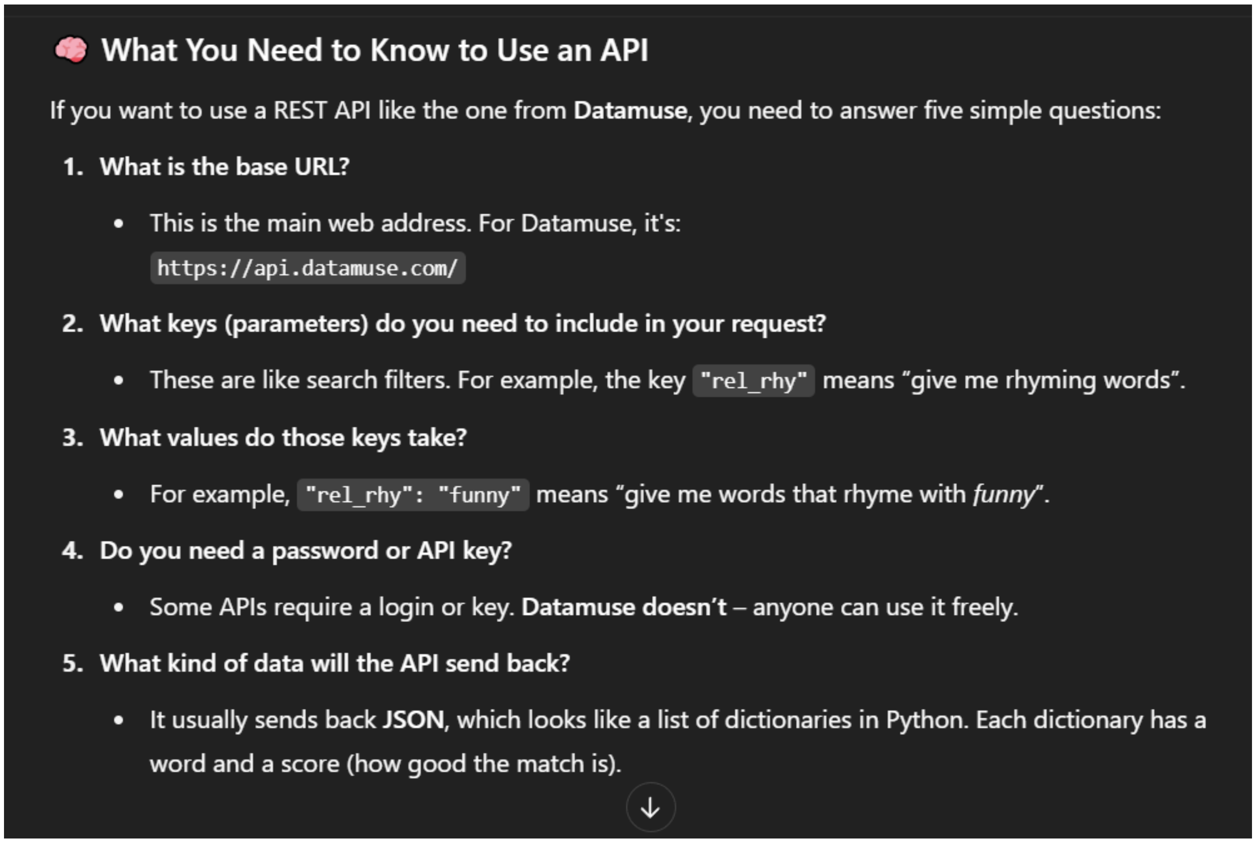
🎯 Summary

- You don't need a browser to fetch web pages — Python can do it.
- `requests.get(url)` gets a page.
- `.text`, `.json()`, and `.url` let you access the contents.
- `params={}` helps you easily add values to the URL.



📘 What is a REST API?

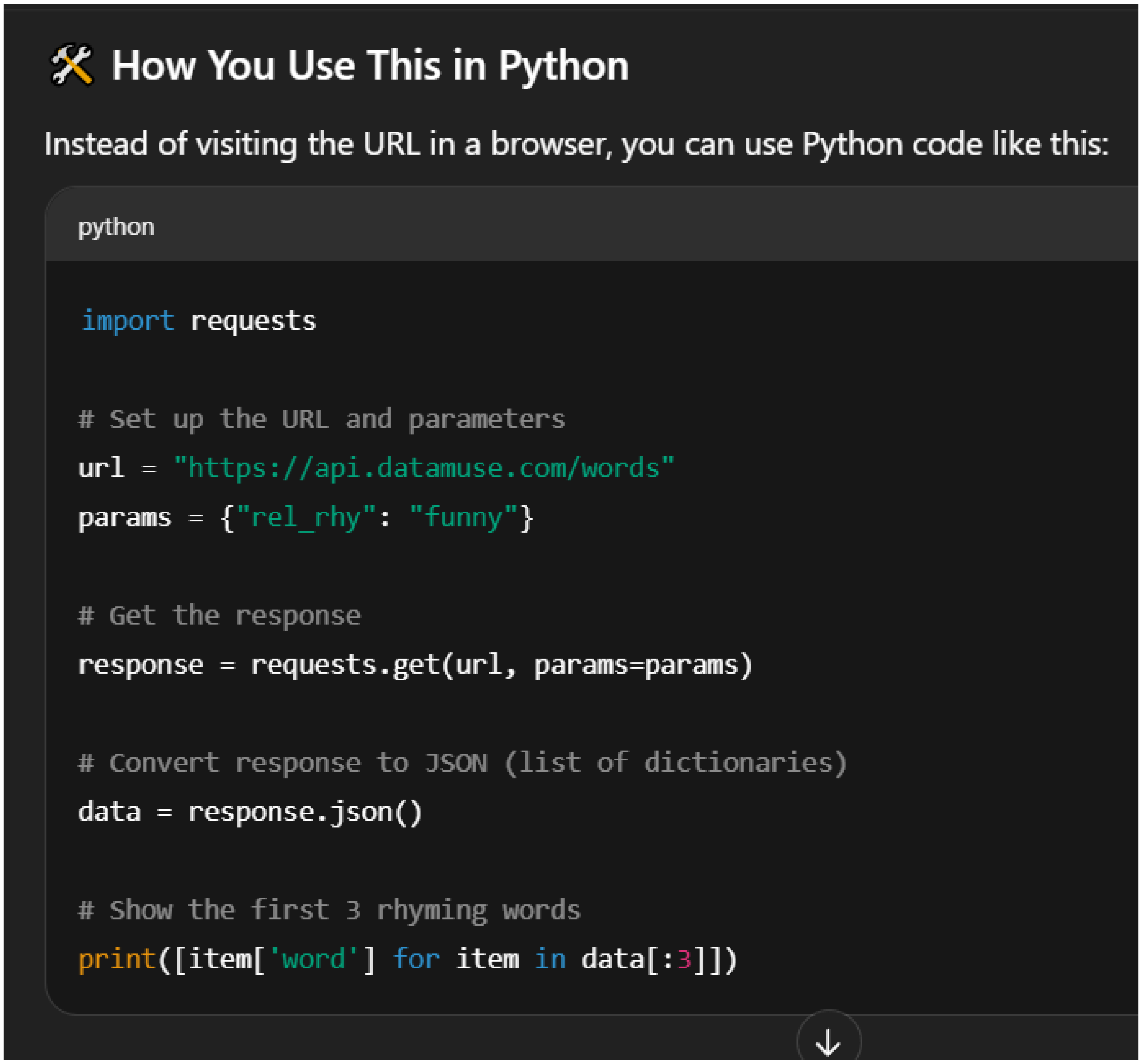
An **API** lets you talk to another website's service using code, instead of using a browser. A **REST API** is a common type that works like web pages, using URLs.



🧠 What You Need to Know to Use an API

If you want to use a REST API like the one from **Datamuse**, you need to answer five simple questions:

1. **What is the base URL?**
   - This is the main web address. For Datamuse, it's: `https://api.datamuse.com/`
2. **What keys (parameters) do you need to include in your request?**
   - These are like search filters. For example, the key `"rel_rhy"` means "give me rhyming words".
3. **What values do those keys take?**
   - For example, `"rel_rhy": "funny"` means "give me words that rhyme with *funny*".
4. **Do you need a password or API key?**
   - Some APIs require a login or key. **Datamuse doesn't** – anyone can use it freely.
5. **What kind of data will the API send back?**
   - It usually sends back **JSON**, which looks like a list of dictionaries in Python. Each dictionary has a word and a score (how good the match is).



🛠️ How You Use This in Python

Instead of visiting the URL in a browser, you can use Python code like this:

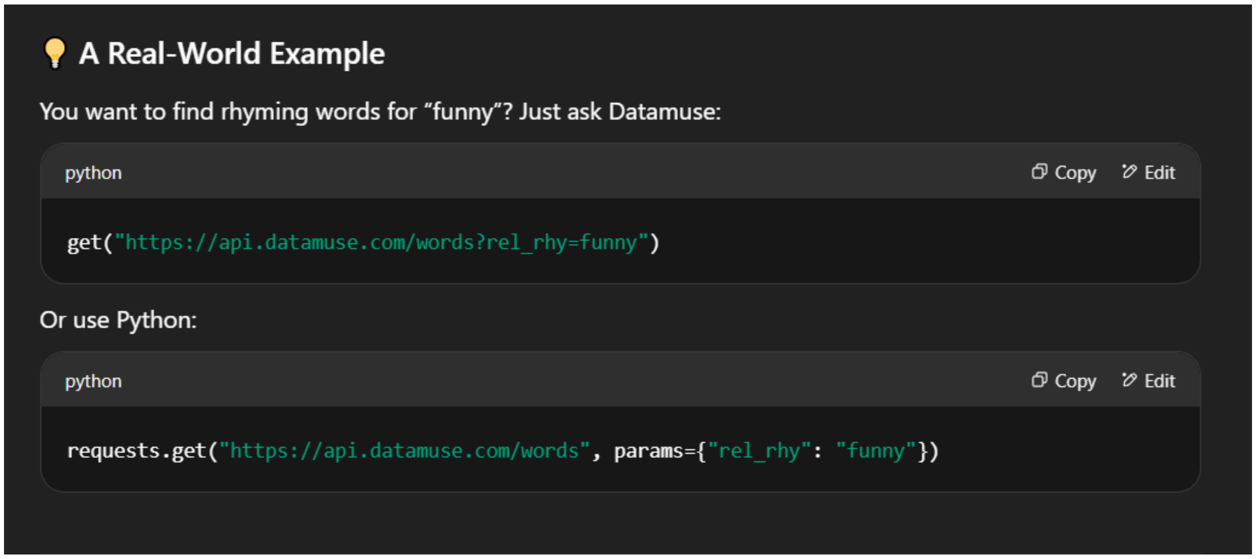
```python
import requests

# Set up the URL and parameters
url = "https://api.datamuse.com/words"
params = {"rel_rhy": "funny"}

# Get the response
response = requests.get(url, params=params)

# Convert response to JSON (list of dictionaries)
data = response.json()

# Show the first 3 rhyming words
print([item['word'] for item in data[:3]])
```



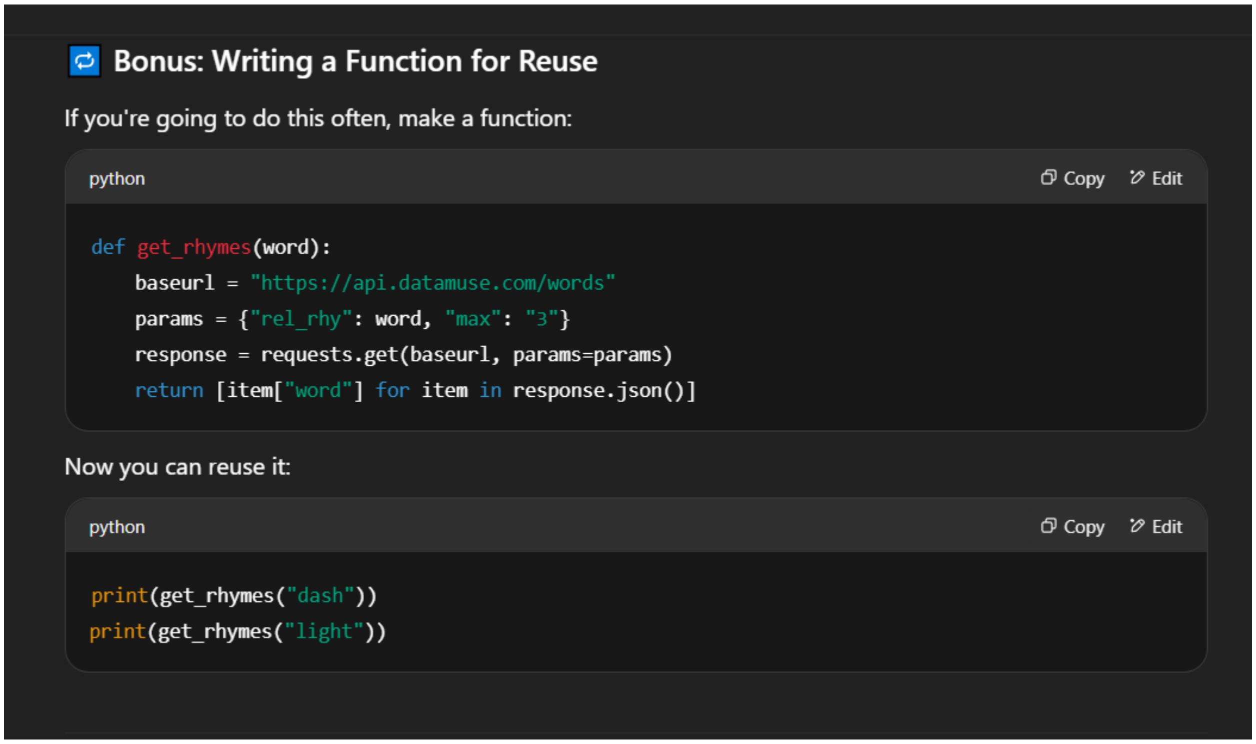
💡 A Real-World Example

You want to find rhyming words for "funny"? Just ask Datamuse:

```python
get("https://api.datamuse.com/words?rel_rhy=funny")
```

Or use Python:

```python
requests.get("https://api.datamuse.com/words", params={"rel_rhy": "funny"})
```



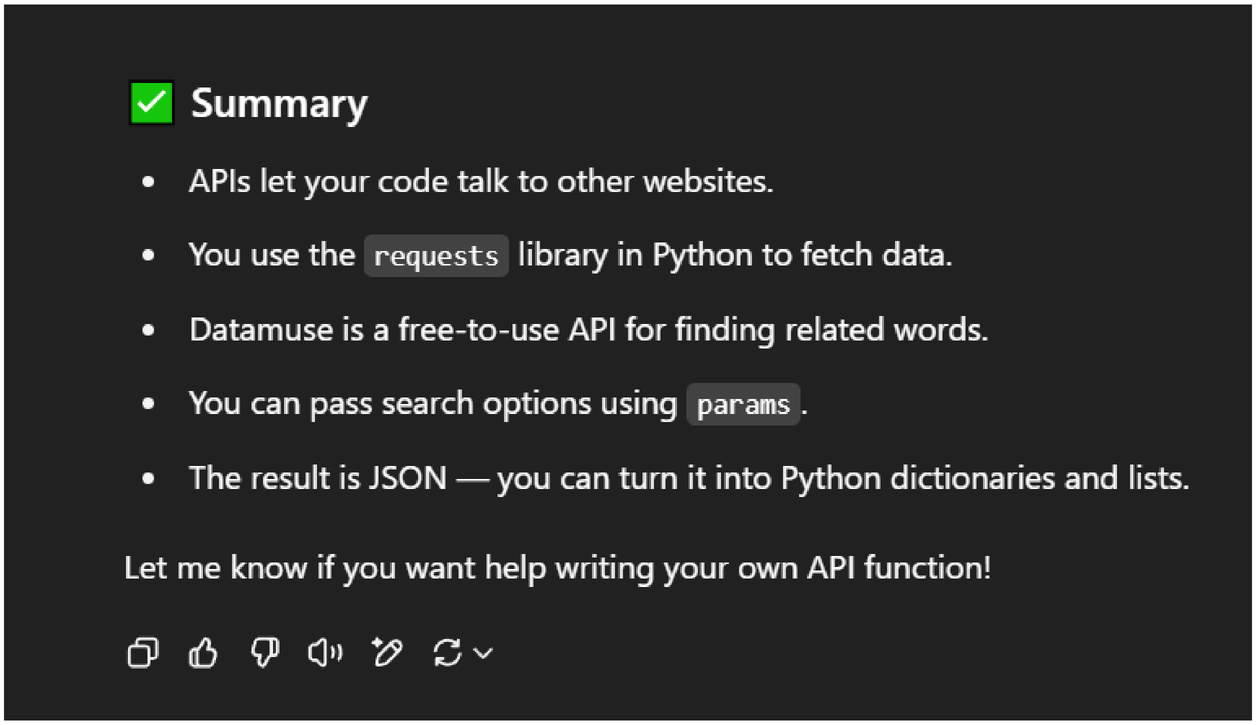
🔁 Bonus: Writing a Function for Reuse

If you're going to do this often, make a function:

```python
def get_rhymes(word):
    baseurl = "https://api.datamuse.com/words"
    params = {"rel_rhy": word, "max": "3"}
    response = requests.get(baseurl, params=params)
    return [item["word"] for item in response.json()]
```

Now you can reuse it:

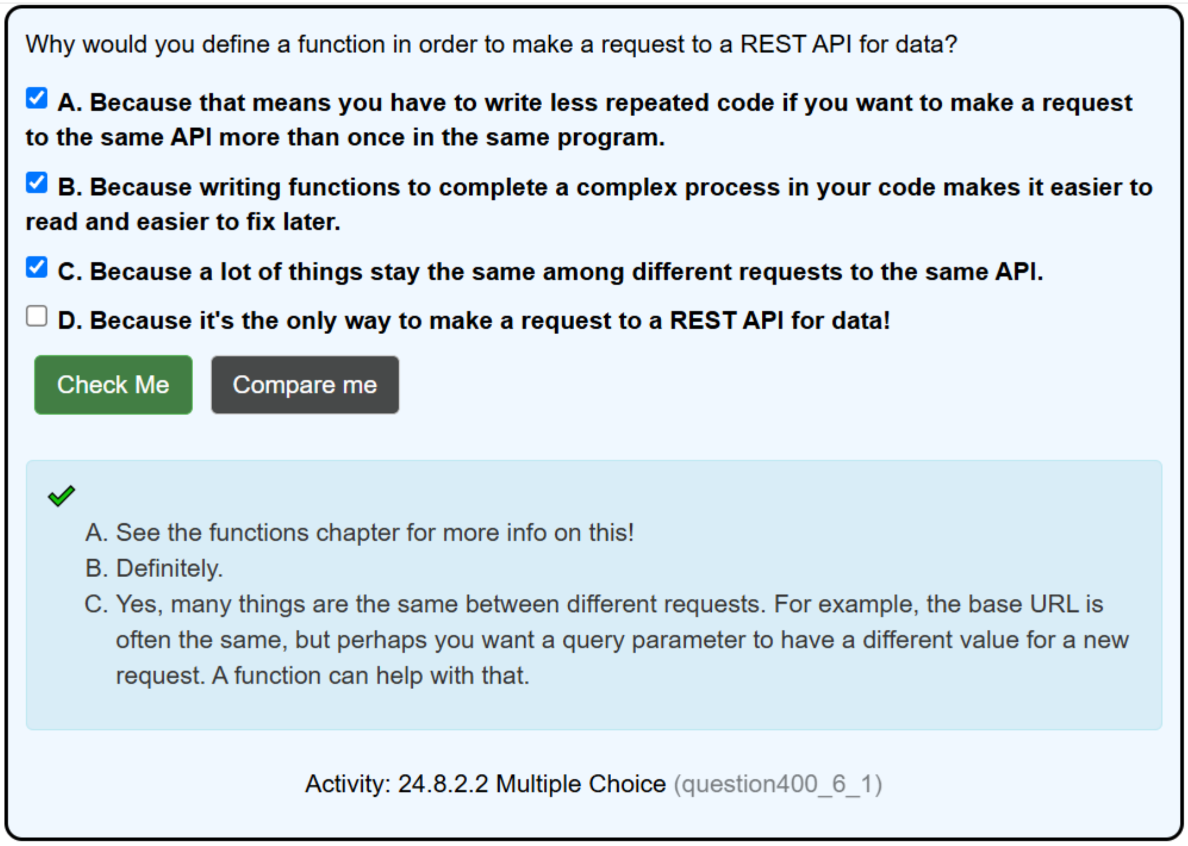
```python
print(get_rhymes("dash"))
print(get_rhymes("light"))
```



✅ **Summary**

- APIs let your code talk to other websites.
- You use the `requests` library in Python to fetch data.
- Datamuse is a free-to-use API for finding related words.
- You can pass search options using `params`.
- The result is JSON — you can turn it into Python dictionaries and lists.

Let me know if you want help writing your own API function!



Why would you define a function in order to make a request to a REST API for data?

☑ **A. Because that means you have to write less repeated code if you want to make a request to the same API more than once in the same program.**

☑ **B. Because writing functions to complete a complex process in your code makes it easier to read and easier to fix later.**

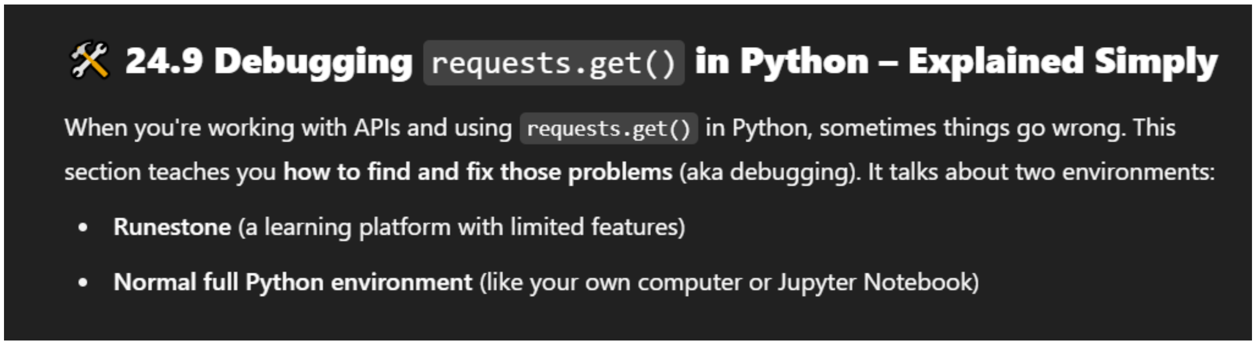
☑ **C. Because a lot of things stay the same among different requests to the same API.**

☐ **D. Because it's the only way to make a request to a REST API for data!**

✓

A. See the functions chapter for more info on this!
B. Definitely.
C. Yes, many things are the same between different requests. For example, the base URL is often the same, but perhaps you want a query parameter to have a different value for a new request. A function can help with that.

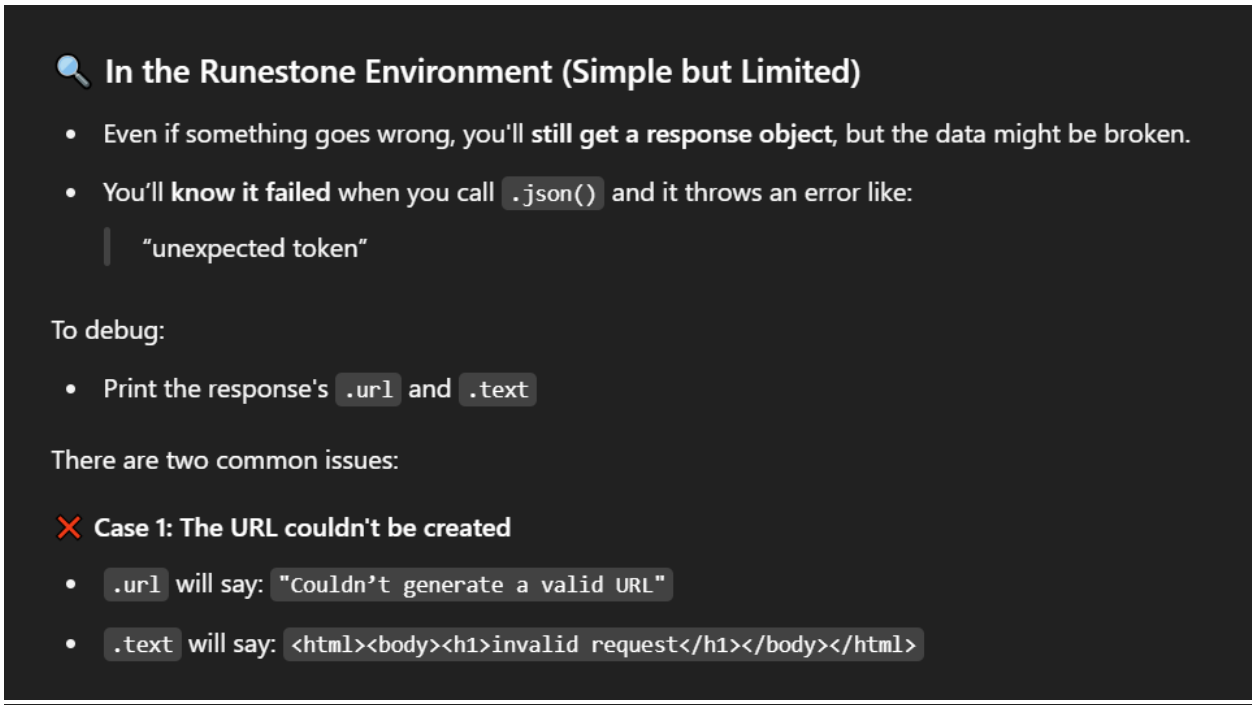
Activity: 24.8.2.2 Multiple Choice (question400_6_1)



🛠️ 24.9 Debugging `requests.get()` in Python – Explained Simply

When you're working with APIs and using `requests.get()` in Python, sometimes things go wrong. This section teaches you **how to find and fix those problems** (aka debugging). It talks about two environments:

- **Runestone** (a learning platform with limited features)
- **Normal full Python environment** (like your own computer or Jupyter Notebook)



🔍 In the Runestone Environment (Simple but Limited)

- Even if something goes wrong, you'll **still get a response object**, but the data might be broken.
- You'll **know it failed** when you call `.json()` and it throws an error like:

  > "unexpected token"

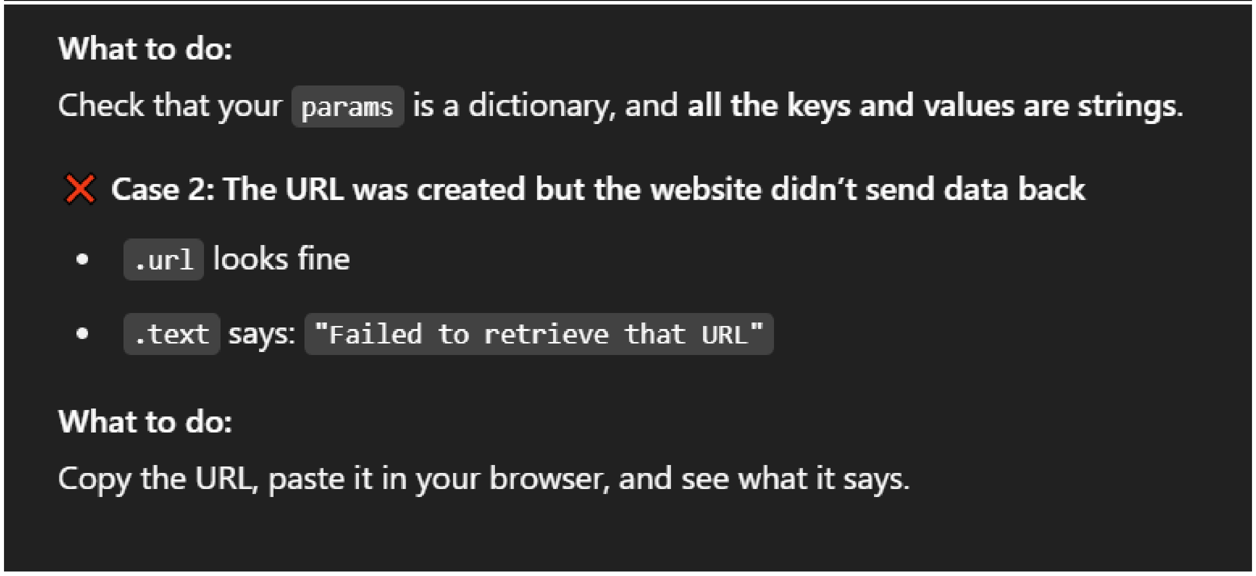
To debug:

- Print the response's `.url` and `.text`

There are two common issues:

❌ Case 1: The URL couldn't be created

- `.url` will say: `"Couldn’t generate a valid URL"`
- `.text` will say: `<html><body><h1>invalid request</h1></body></html>`



What to do:

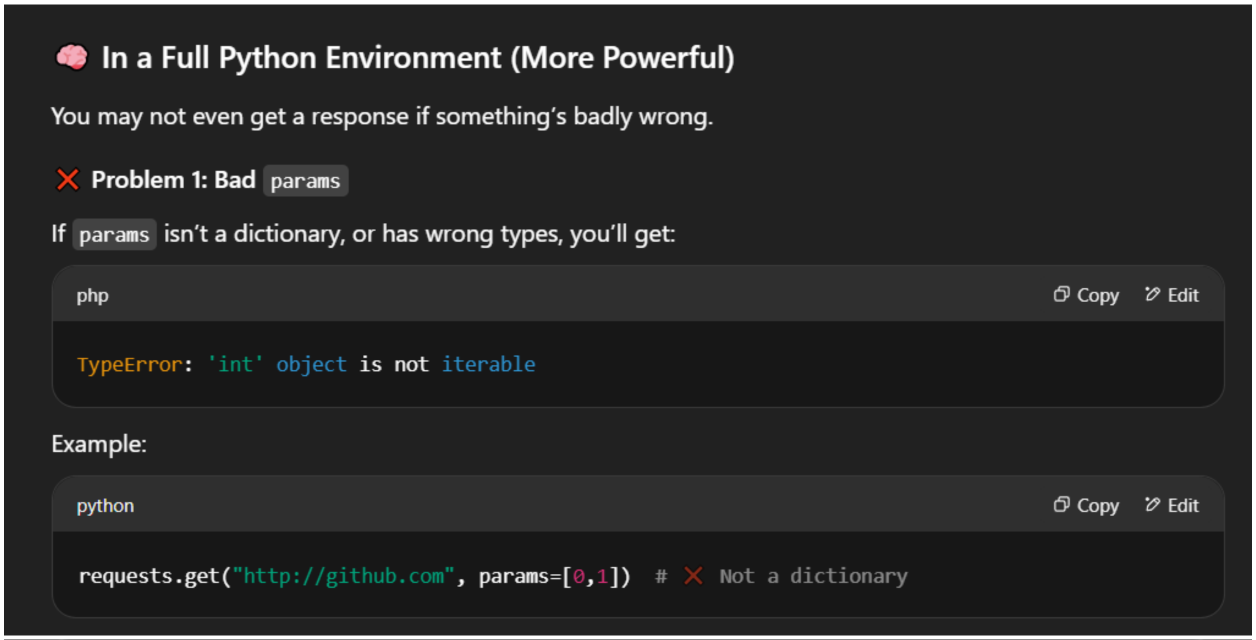
Check that your `params` is a dictionary, and **all the keys and values are strings.**

❌ Case 2: The URL was created but the website didn’t send data back

- `.url` looks fine
- `.text` says: `"Failed to retrieve that URL"`

What to do:

Copy the URL, paste it in your browser, and see what it says.



🧠 In a Full Python Environment (More Powerful)

You may not even get a response if something's badly wrong.

❌ Problem 1: Bad `params`

If `params` isn't a dictionary, or has wrong types, you'll get:

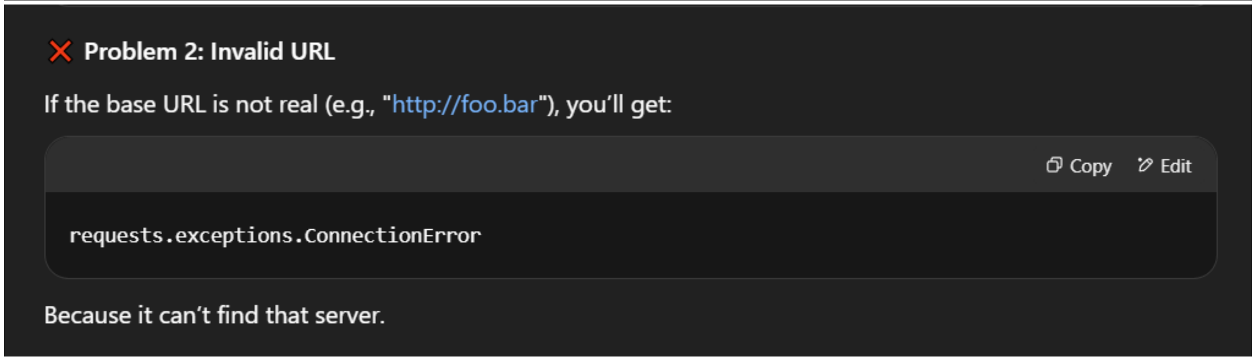
php Copy Edit

```php
TypeError: 'int' object is not iterable
```

Example:

python Copy Edit

```python
requests.get("http://github.com", params=[0,1])  # ❌ Not a dictionary
```



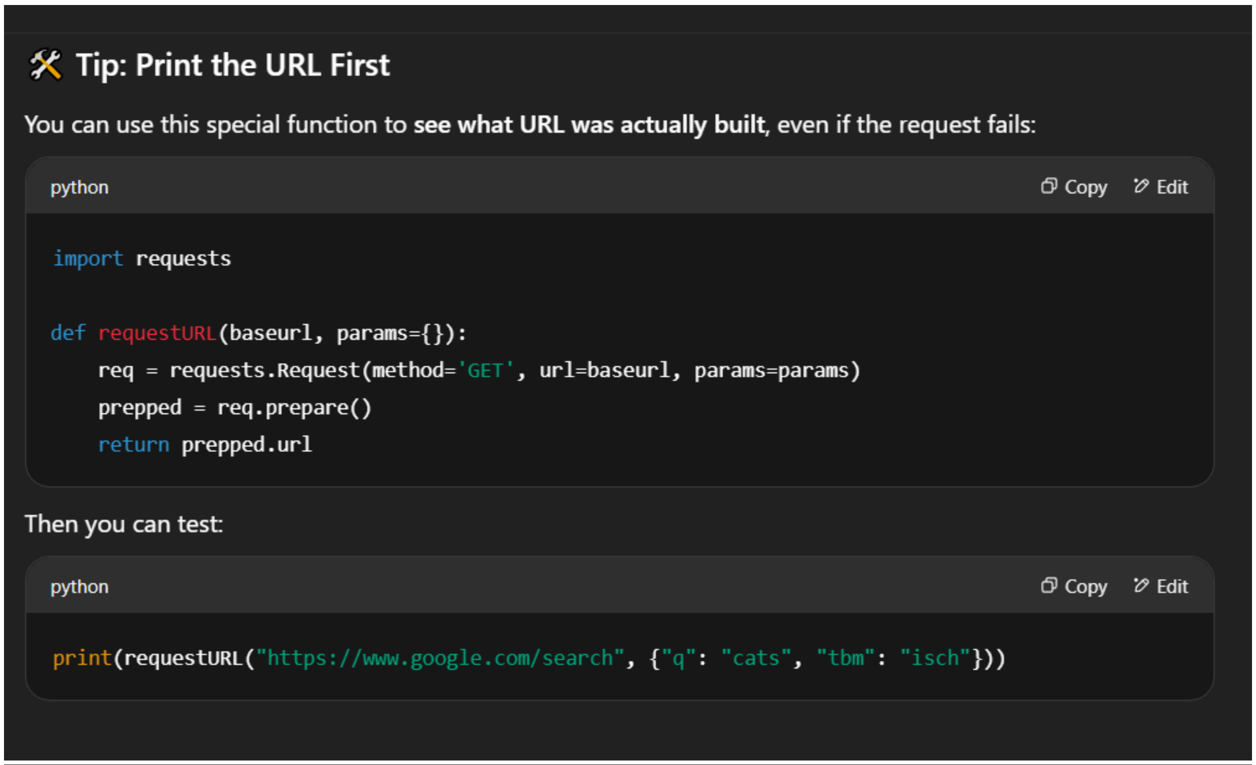
❌ Problem 2: Invalid URL

If the base URL is not real (e.g., "http://foo.bar"), you'll get:

Copy Edit

```
requests.exceptions.ConnectionError
```

Because it can't find that server.



🛠 Tip: Print the URL First

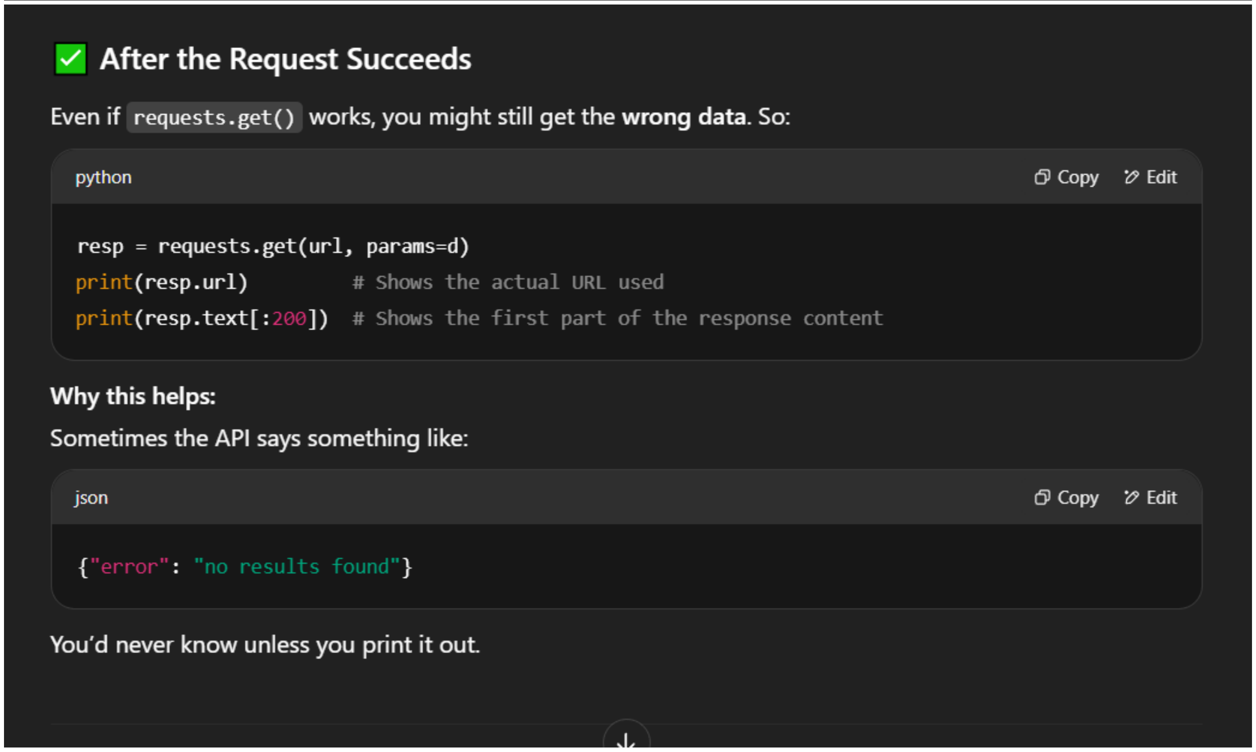
You can use this special function to **see what URL was actually built**, even if the request fails:

```python
import requests

def requestURL(baseurl, params={}):
    req = requests.Request(method='GET', url=baseurl, params=params)
    prepped = req.prepare()
    return prepped.url
```

Then you can test:

```python
print(requestURL("https://www.google.com/search", {"q": "cats", "tbm": "isch"}))
```



✅ After the Request Succeeds

Even if `requests.get()` works, you might still get the **wrong data**. So:

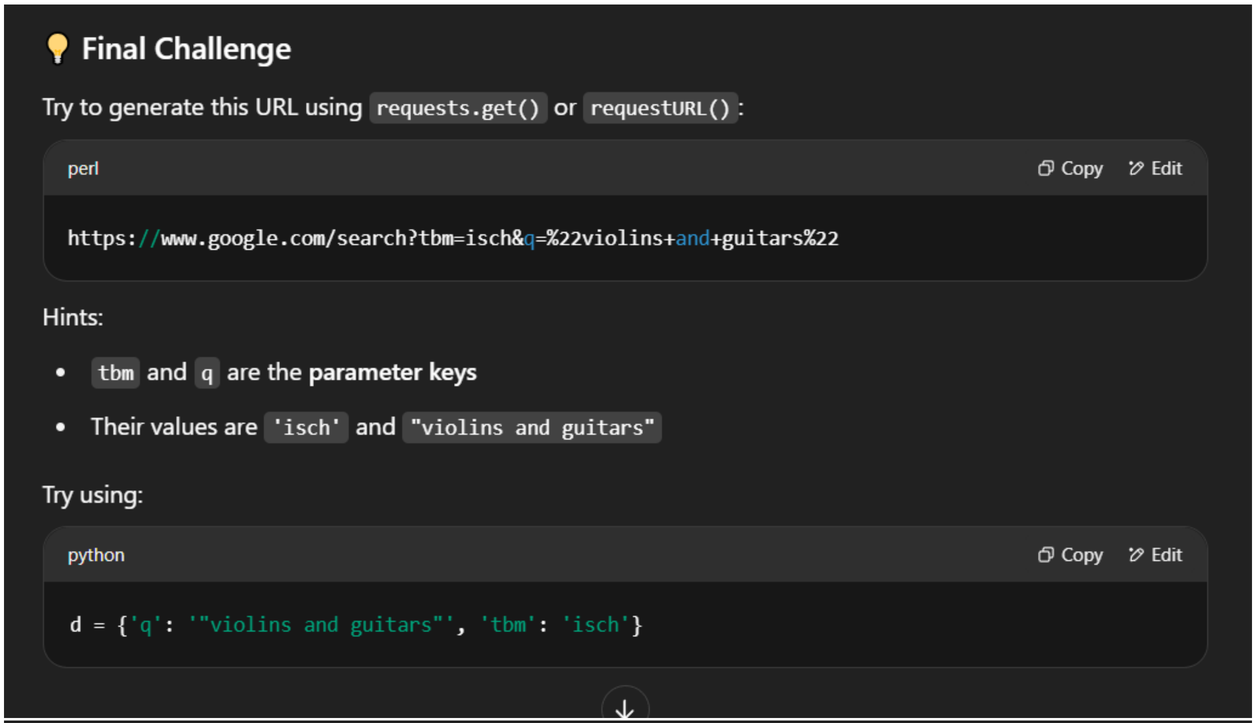
```python
resp = requests.get(url, params=d)
print(resp.url)         # Shows the actual URL used
print(resp.text[:200])  # Shows the first part of the response content
```

Why this helps:

Sometimes the API says something like:

```json
{"error": "no results found"}
```

You'd never know unless you print it out.


💡 Final Challenge

Try to generate this URL using `requests.get()` or `requestURL()`:

perl

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

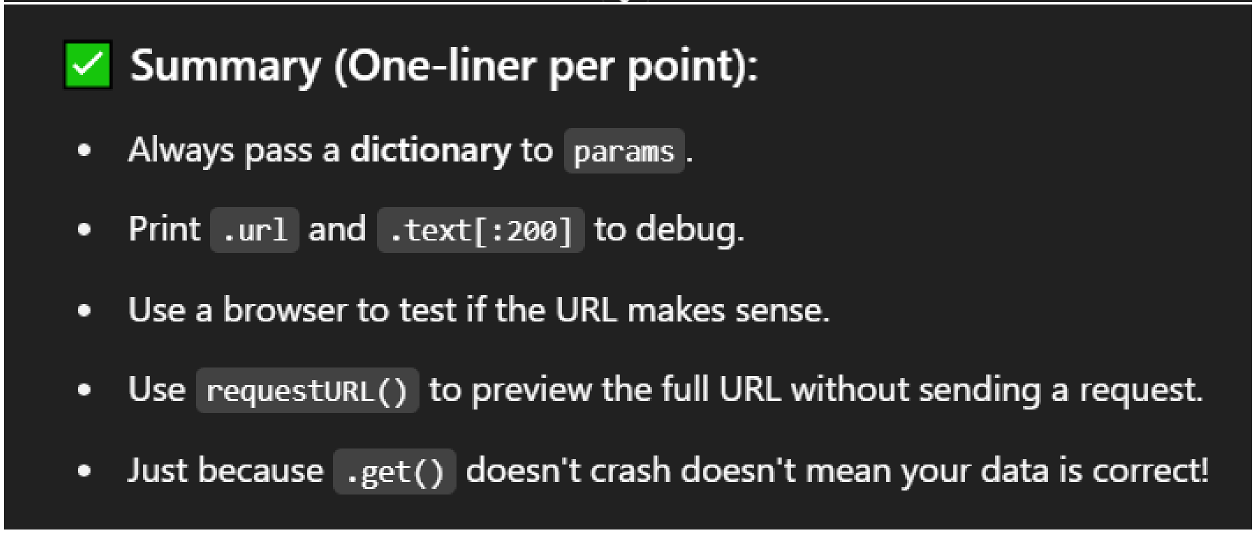
Hints:

- `tbm` and `q` are the **parameter keys**
- Their values are `'isch'` and `"violins and guitars"`

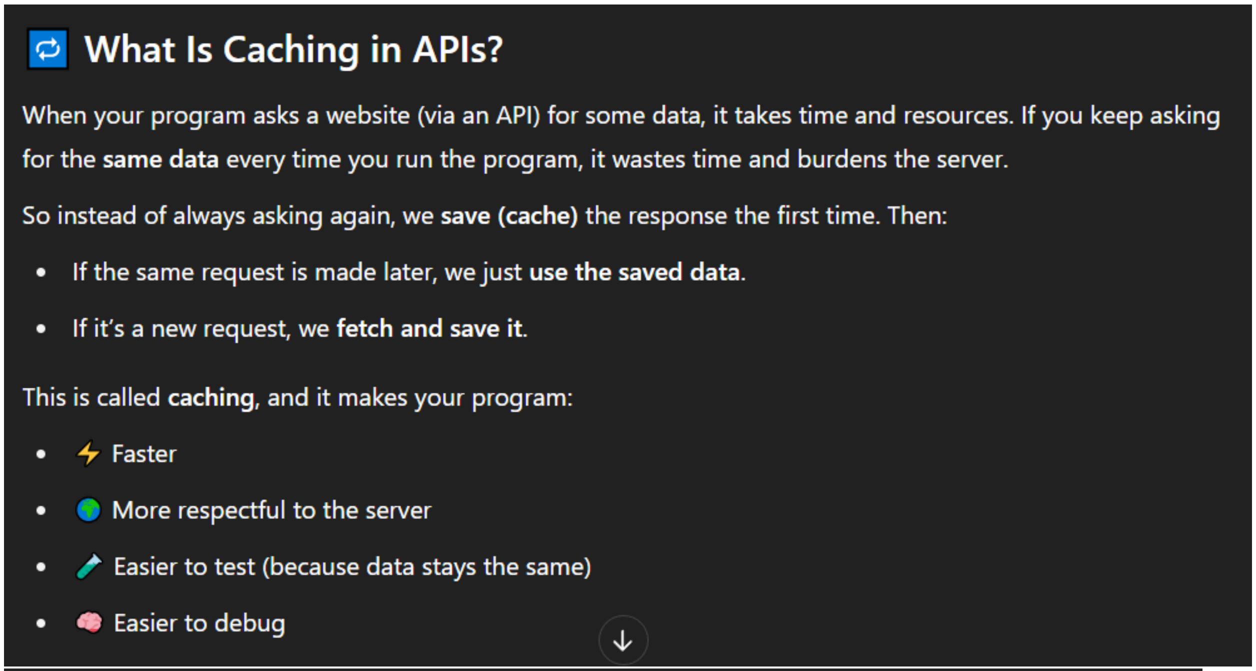
Try using:

python

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


✅ Summary (One-liner per point):

- Always pass a **dictionary** to `params`.
- Print `.url` and `.text[:200]` to debug.
- Use a browser to test if the URL makes sense.
- Use `requestURL()` to preview the full URL without sending a request.
- Just because `.get()` doesn't crash doesn't mean your data is correct!



🔁 What Is Caching in APIs?

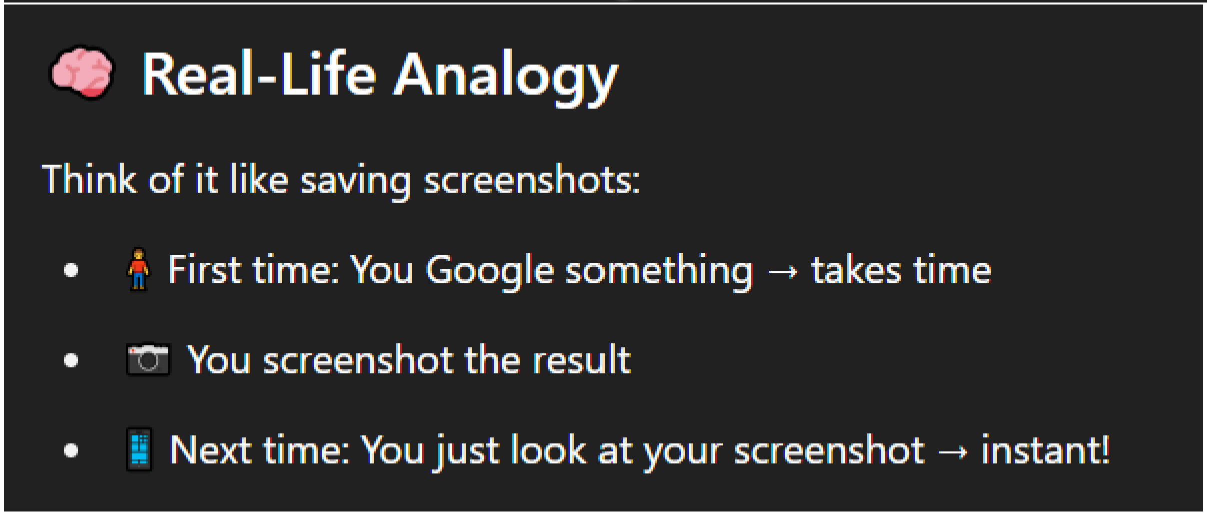
When your program asks a website (via an API) for some data, it takes time and resources. If you keep asking for the **same data** every time you run the program, it wastes time and burdens the server.

So instead of always asking again, we **save (cache)** the response the first time. Then:

- If the same request is made later, we just **use the saved data**.
- If it's a new request, we **fetch and save it**.

This is called **caching**, and it makes your program:

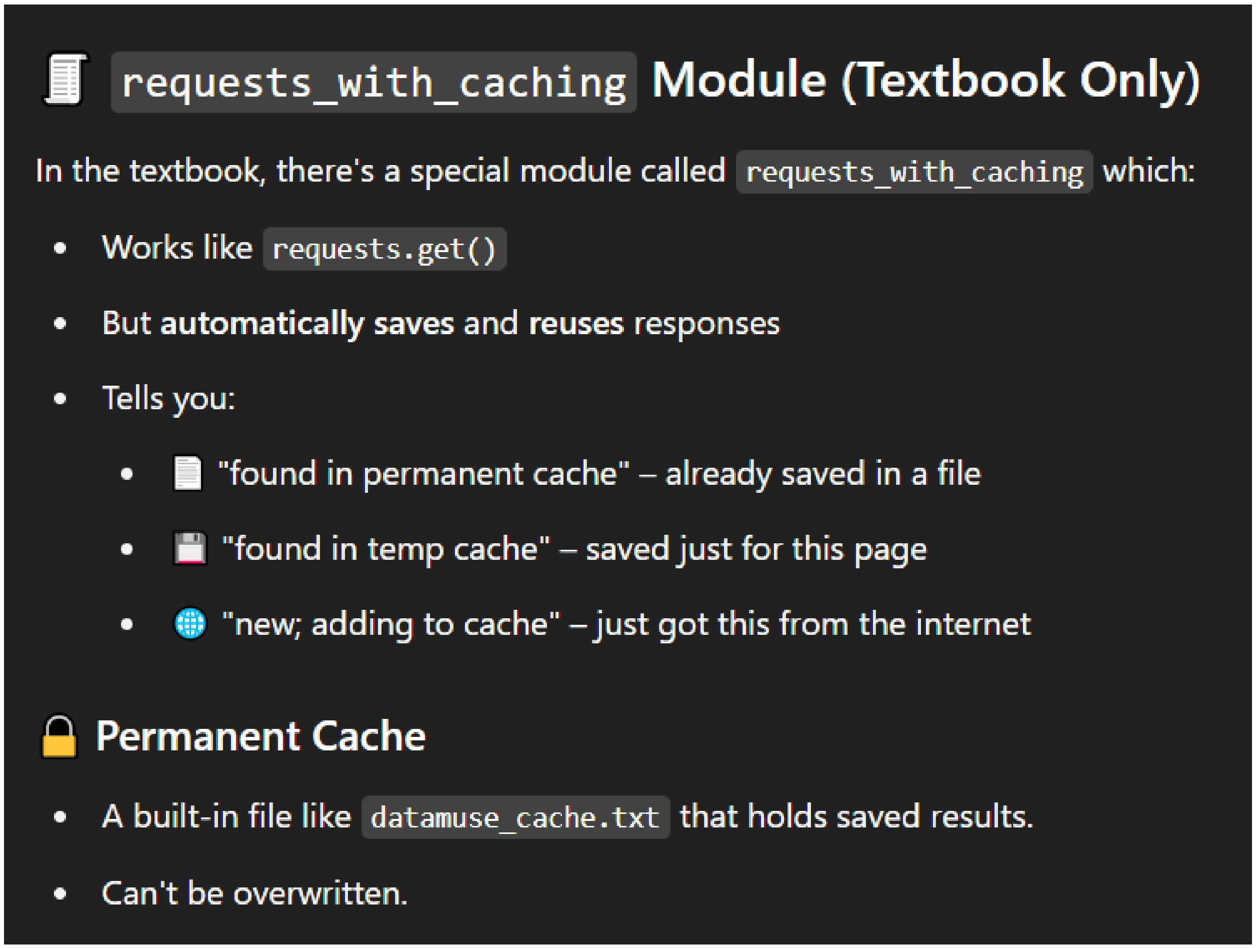
- ⚡ Faster
- 🌍 More respectful to the server
- 🧪 Easier to test (because data stays the same)
- 🧠 Easier to debug



🧠 Real-Life Analogy

Think of it like saving screenshots:

- 🧍 First time: You Google something → takes time
- 📷 You screenshot the result
- 📱 Next time: You just look at your screenshot → instant!



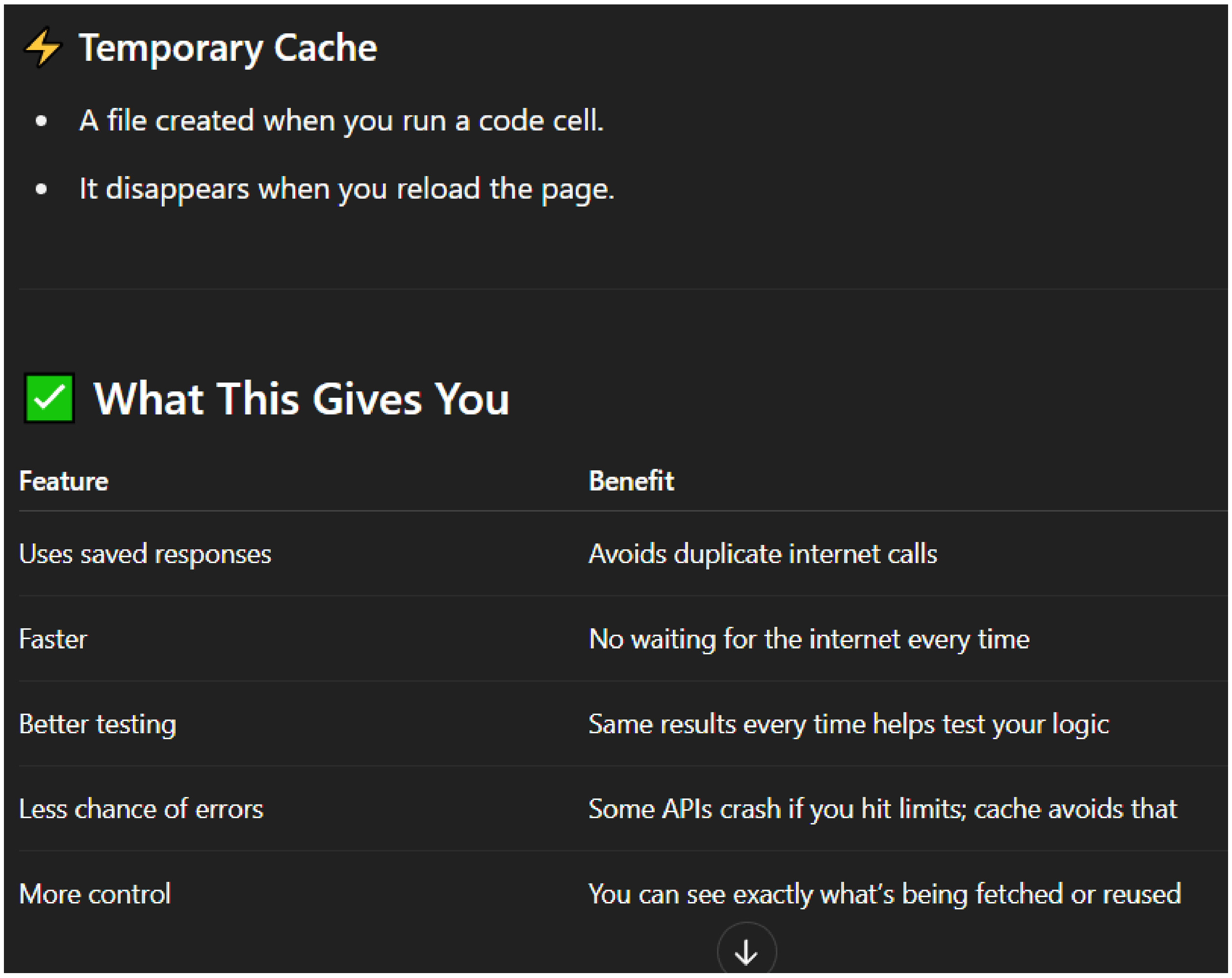
🧾 `requests_with_caching` Module (Textbook Only)

In the textbook, there's a special module called `requests_with_caching` which:

- Works like `requests.get()`
- But **automatically saves** and **reuses** responses
- Tells you:
  - 📄 "found in permanent cache" – already saved in a file
  - 💾 "found in temp cache" – saved just for this page
  - 🌐 "new; adding to cache" – just got this from the internet

🔒 Permanent Cache

- A built-in file like `datamuse_cache.txt` that holds saved results.
- Can't be overwritten.

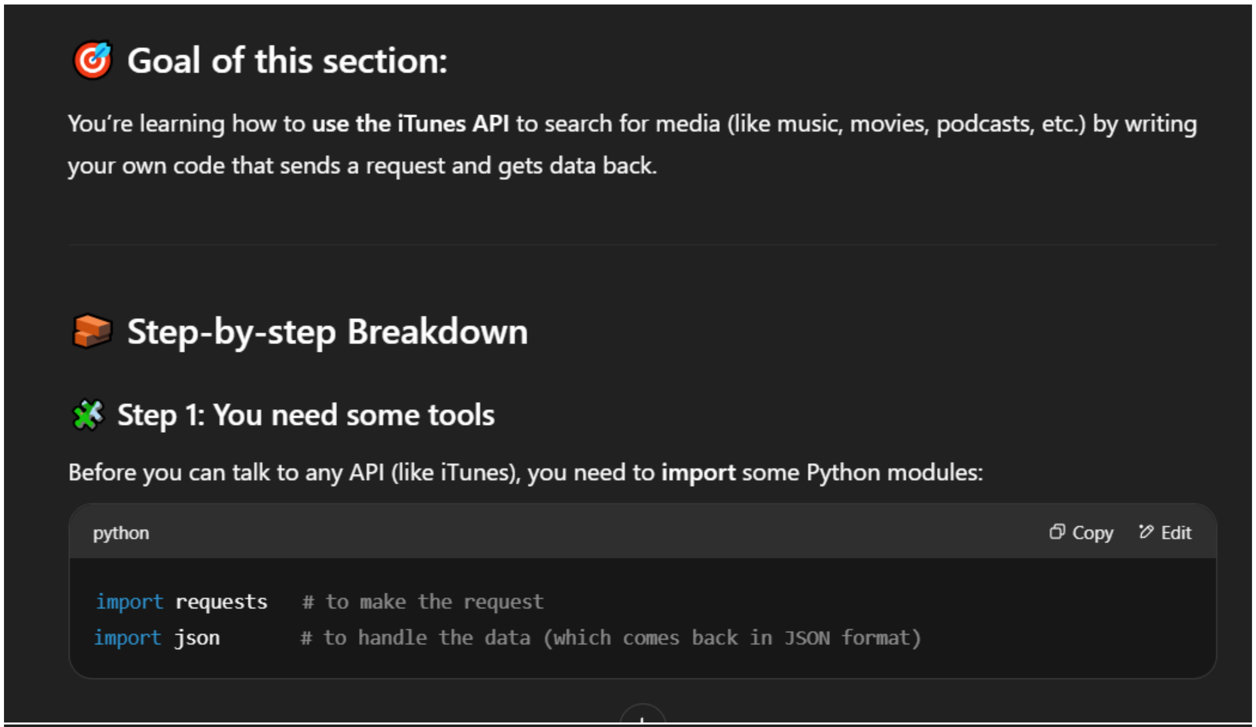


⚡ Temporary Cache

- A file created when you run a code cell.
- It disappears when you reload the page.

✅ What This Gives You

Feature	Benefit
Uses saved responses	Avoids duplicate internet calls
Faster	No waiting for the internet every time
Better testing	Same results every time helps test your logic
Less chance of errors	Some APIs crash if you hit limits; cache avoids that
More control	You can see exactly what's being fetched or reused



🎯 Goal of this section:

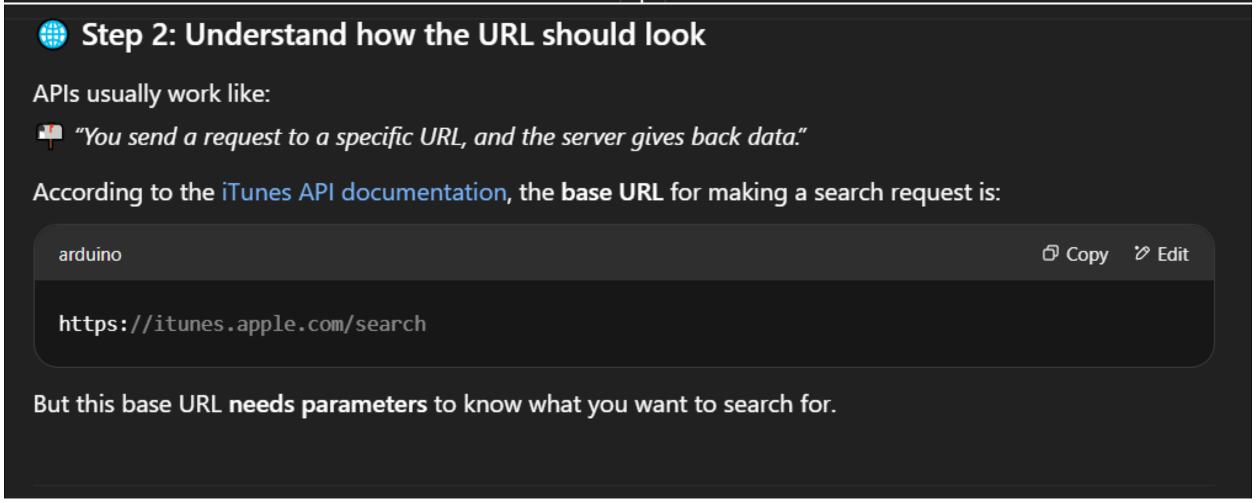
You're learning how to **use the iTunes API** to search for media (like music, movies, podcasts, etc.) by writing your own code that sends a request and gets data back.

🧱 Step-by-step Breakdown

🛠️ Step 1: You need some tools

Before you can talk to any API (like iTunes), you need to **import** some Python modules:

```python
import requests   # to make the request
import json       # to handle the data (which comes back in JSON format)
```



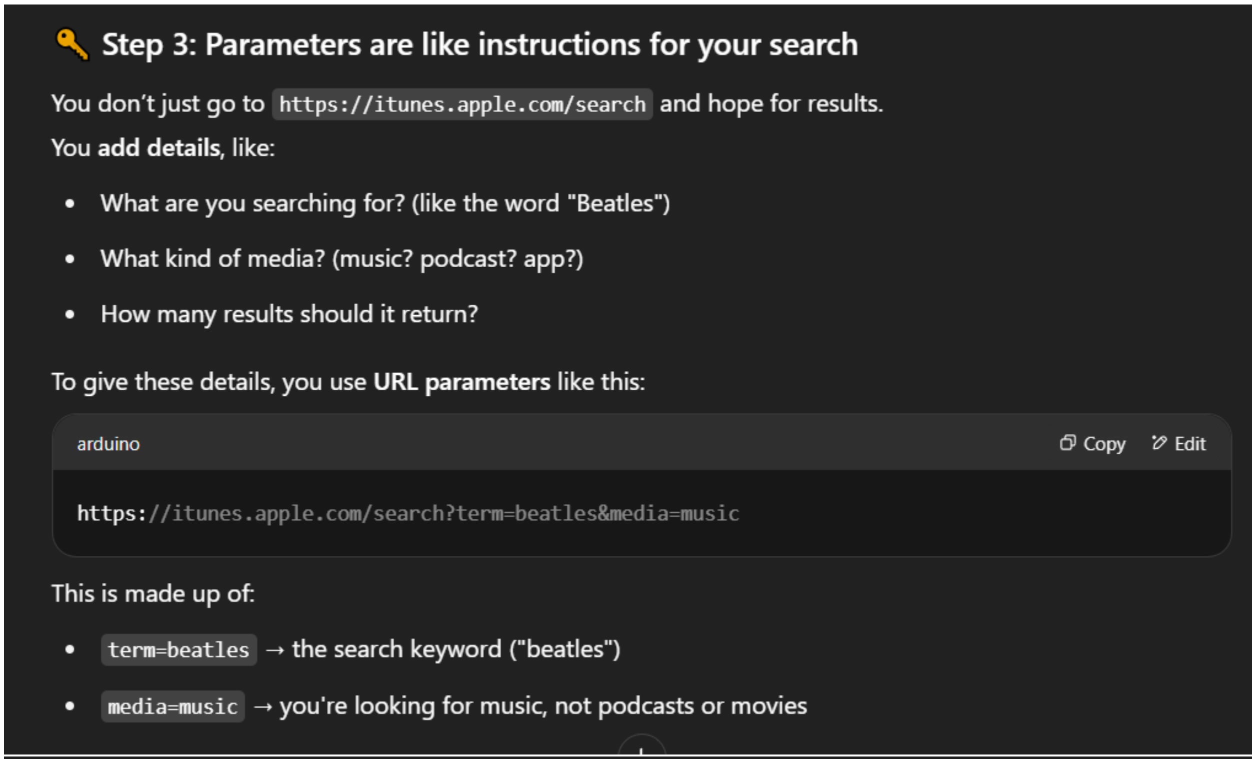
🌐 Step 2: Understand how the URL should look

APIs usually work like:
📫 *"You send a request to a specific URL, and the server gives back data."*

According to the iTunes API documentation, the **base URL** for making a search request is:

```arduino
https://itunes.apple.com/search
```

But this base URL **needs parameters** to know what you want to search for.



🔑 Step 3: Parameters are like instructions for your search

You don't just go to `https://itunes.apple.com/search` and hope for results.
You **add details**, like:

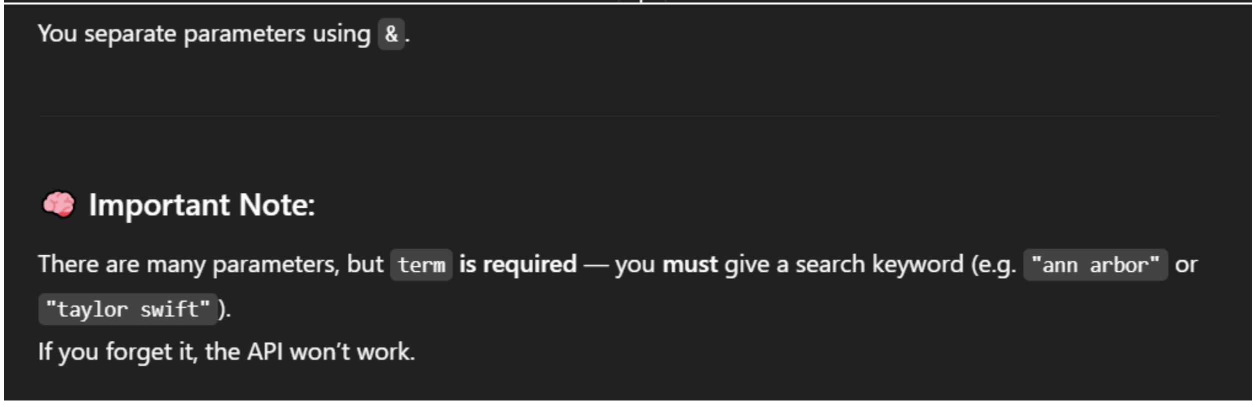
- What are you searching for? (like the word "Beatles")
- What kind of media? (music? podcast? app?)
- How many results should it return?

To give these details, you use **URL parameters** like this:

```arduino
https://itunes.apple.com/search?term=beatles&media=music
```

This is made up of:

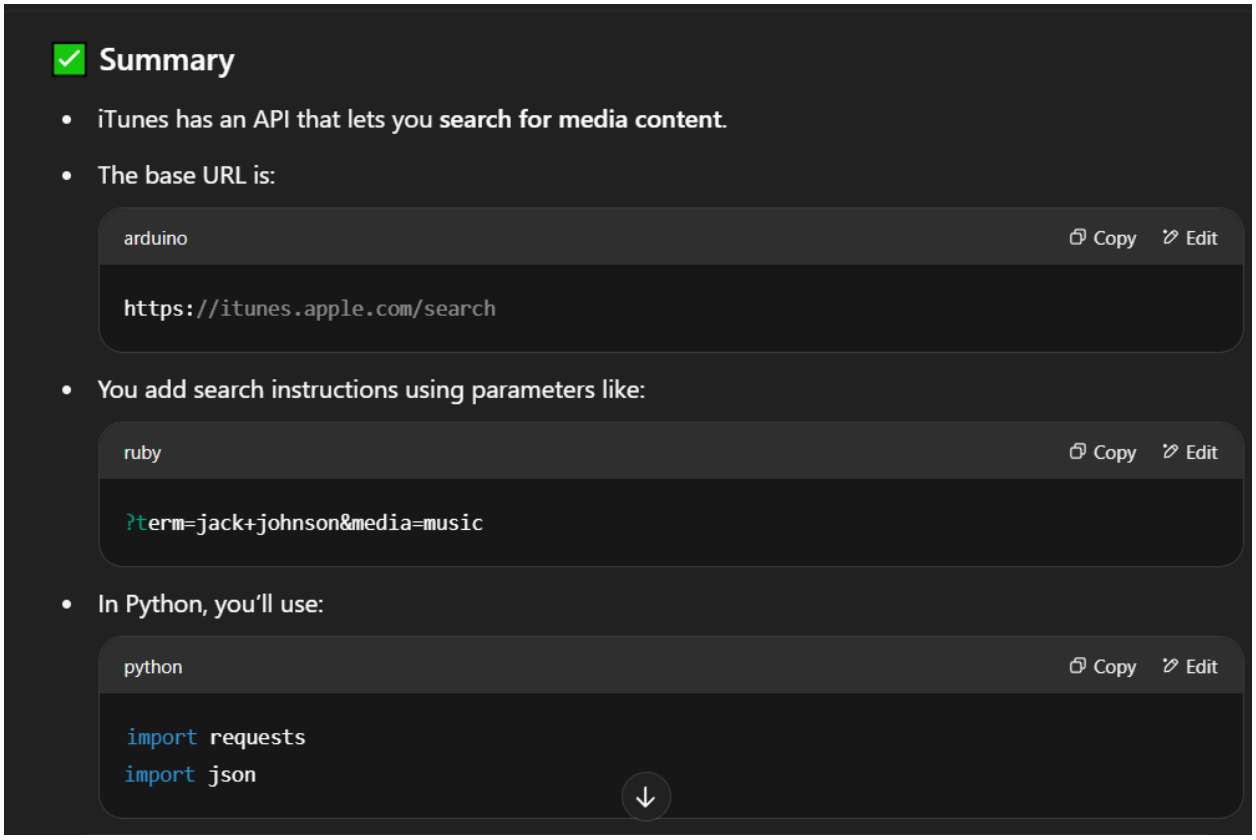
- `term=beatles` → the search keyword ("beatles")
- `media=music` → you're looking for music, not podcasts or movies



You separate parameters using `&`.

🧠 Important Note:

There are many parameters, but `term` **is required** — you **must** give a search keyword (e.g. `"ann arbor"` or `"taylor swift"`).
If you forget it, the API won't work.



✅ Summary

- iTunes has an API that lets you **search for media content**.
- The base URL is:

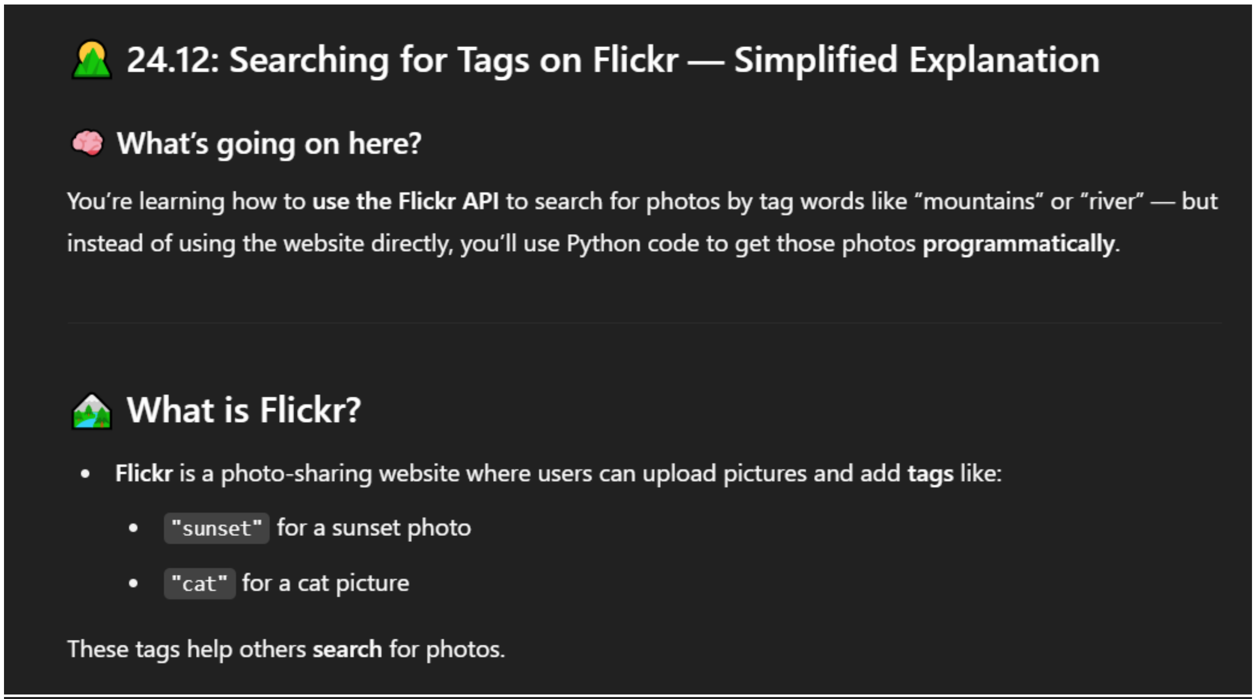
```arduino
https://itunes.apple.com/search
```

- You add search instructions using parameters like:

```ruby
?term=jack+johnson&media=music
```

- In Python, you'll use:

```python
import requests
import json
```



🌄 24.12: Searching for Tags on Flickr — Simplified Explanation

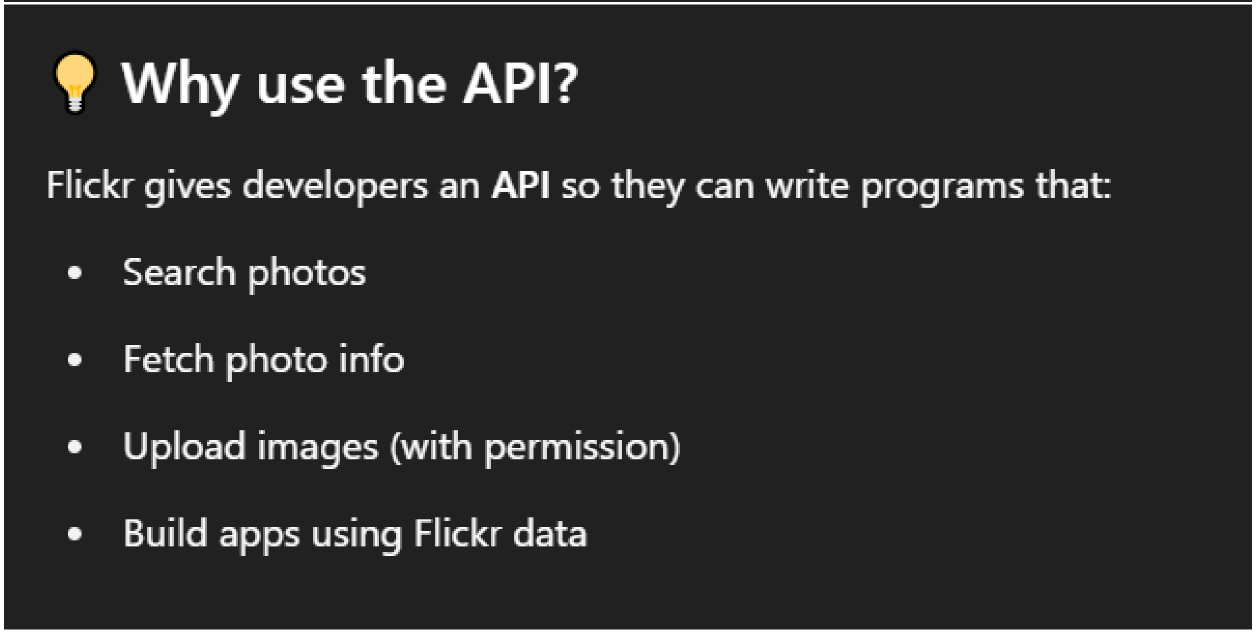
🧠 What's going on here?

You're learning how to **use the Flickr API** to search for photos by tag words like "mountains" or "river" — but instead of using the website directly, you'll use Python code to get those photos **programmatically**.

🏞️ What is Flickr?

- **Flickr** is a photo-sharing website where users can upload pictures and add **tags** like:
  - `"sunset"` for a sunset photo
  - `"cat"` for a cat picture

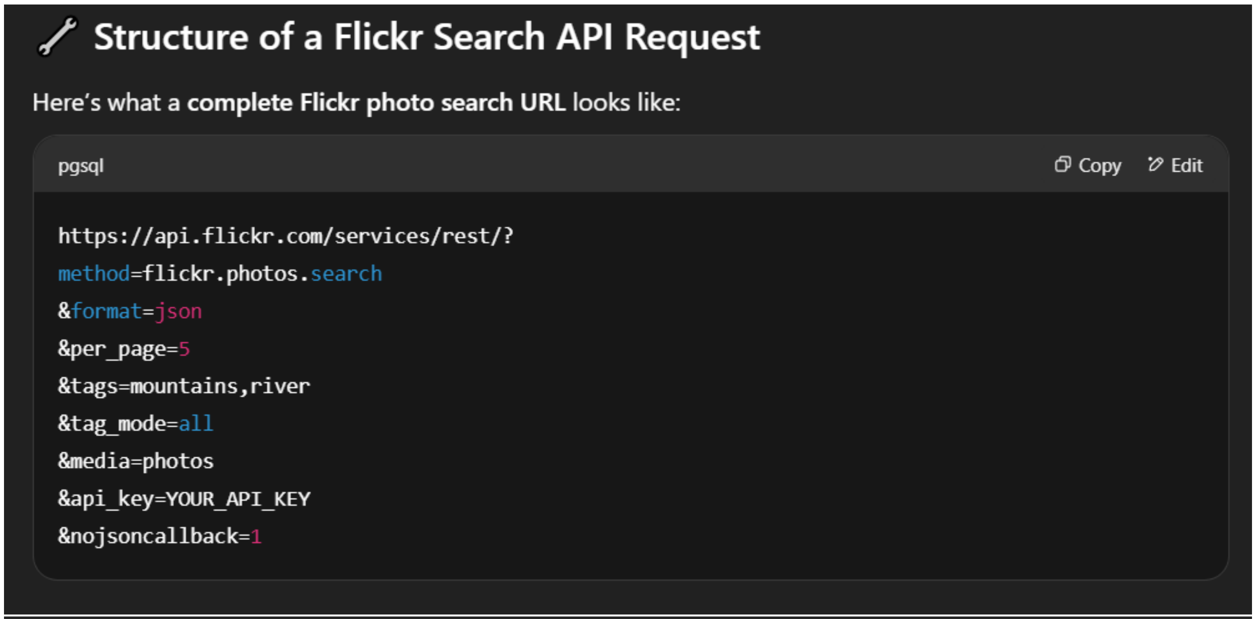
These tags help others **search** for photos.



💡 Why use the API?

Flickr gives developers an **API** so they can write programs that:

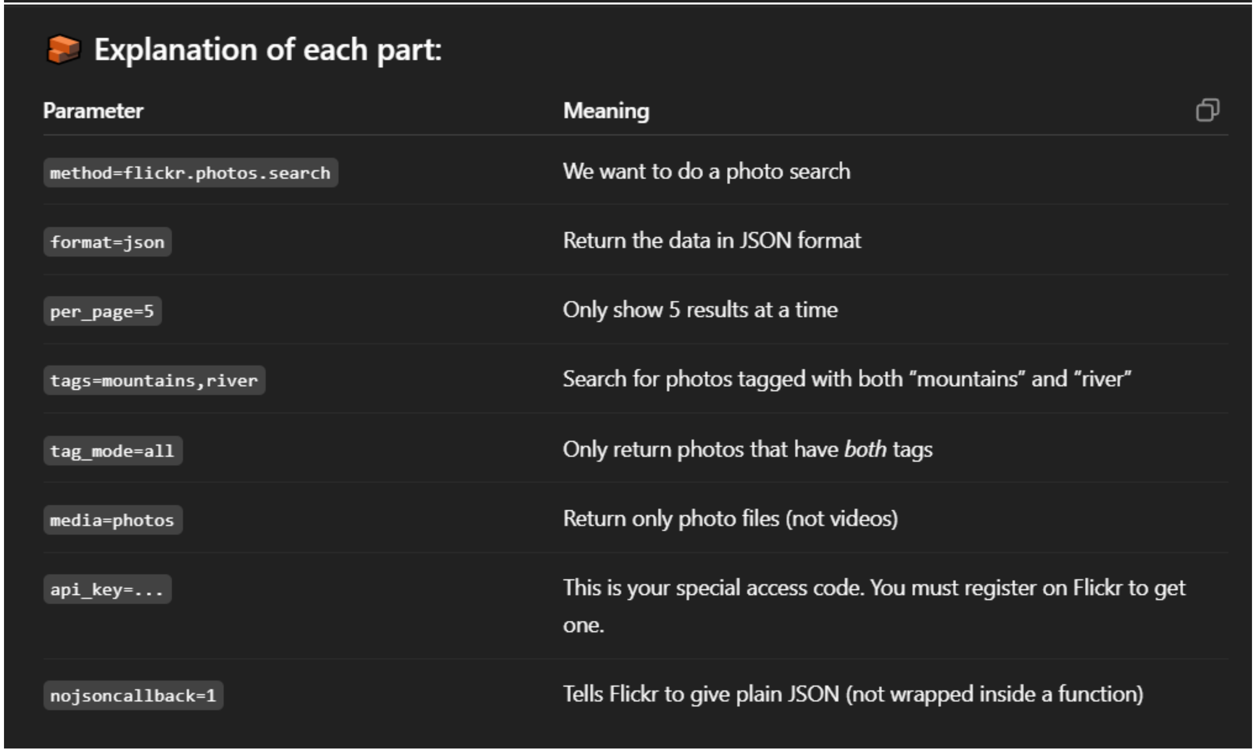
- Search photos
- Fetch photo info
- Upload images (with permission)
- Build apps using Flickr data



🔧 Structure of a Flickr Search API Request

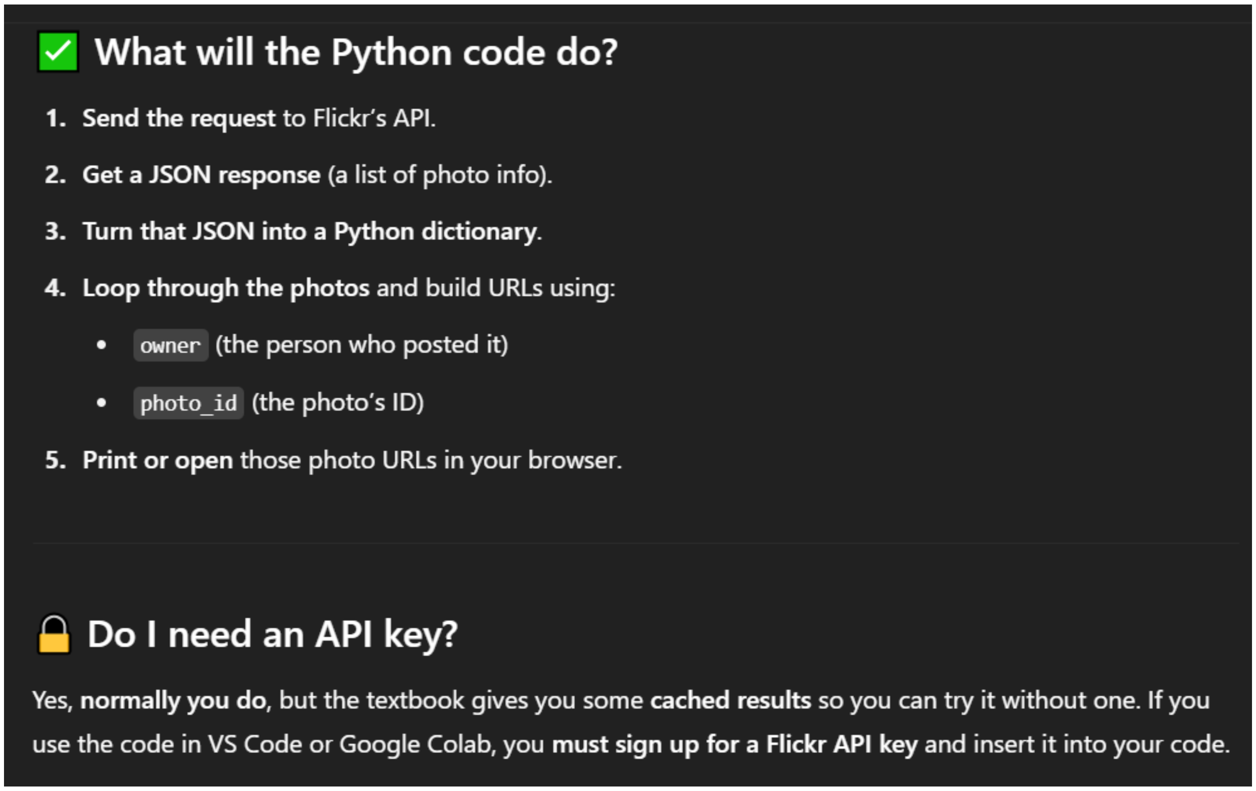
Here's what a **complete Flickr photo search URL** looks like:

```pgsql
https://api.flickr.com/services/rest/?
method=flickr.photos.search
&format=json
&per_page=5
&tags=mountains,river
&tag_mode=all
&media=photos
&api_key=YOUR_API_KEY
&nojsoncallback=1
```



📦 Explanation of each part:

Parameter	Meaning
`method=flickr.photos.search`	We want to do a photo search
`format=json`	Return the data in JSON format
`per_page=5`	Only show 5 results at a time
`tags=mountains,river`	Search for photos tagged with both "mountains" and "river"
`tag_mode=all`	Only return photos that have *both* tags
`media=photos`	Return only photo files (not videos)
`api_key=...`	This is your special access code. You must register on Flickr to get one.
`nojsoncallback=1`	Tells Flickr to give plain JSON (not wrapped inside a function)

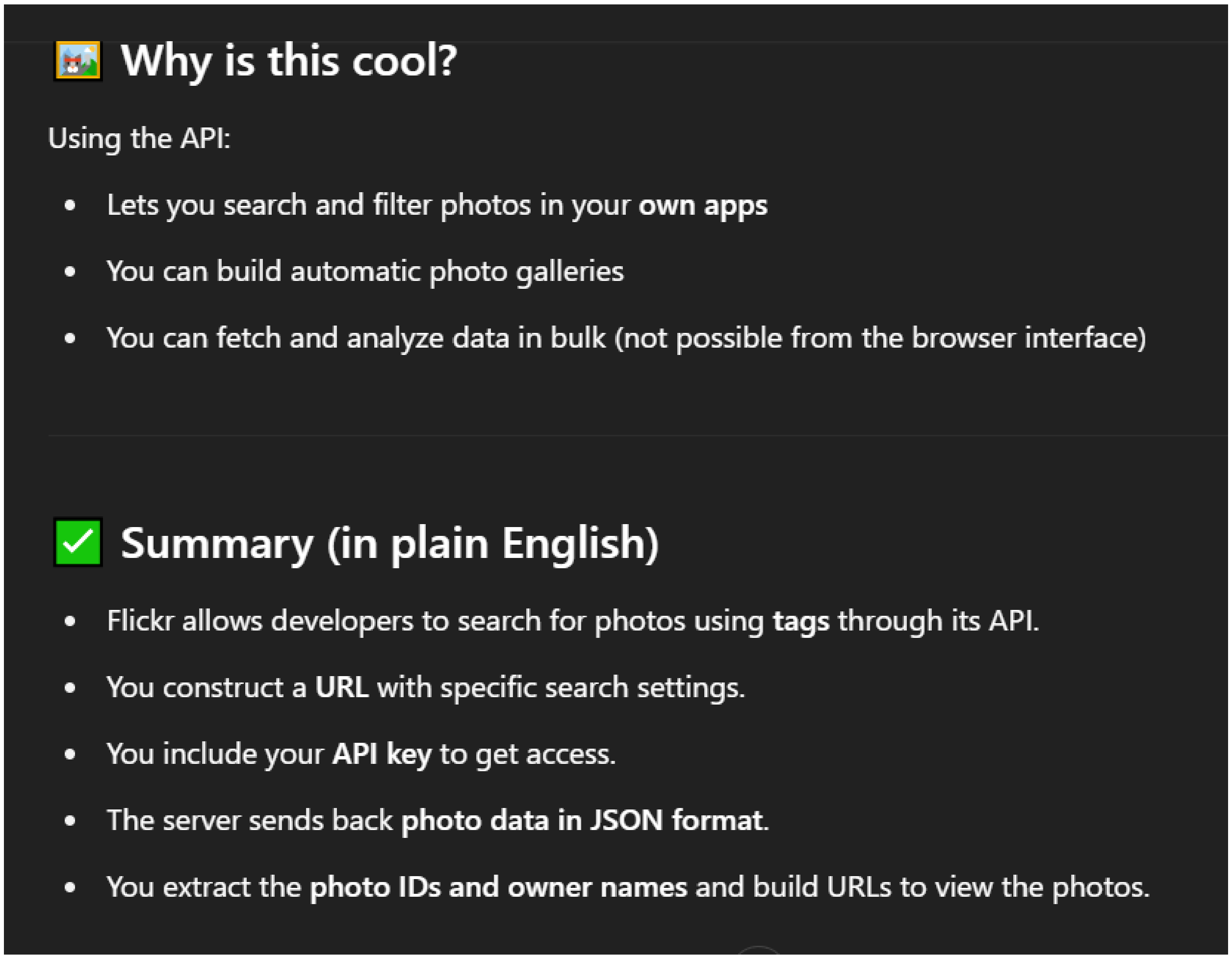


✅ What will the Python code do?

1. **Send the request** to Flickr's API.
2. **Get a JSON response** (a list of photo info).
3. **Turn that JSON into a Python dictionary.**
4. **Loop through the photos** and build URLs using:
   - `owner` (the person who posted it)
   - `photo_id` (the photo's ID)
5. **Print or open** those photo URLs in your browser.

🔒 Do I need an API key?

Yes, **normally you do**, but the textbook gives you some **cached results** so you can try it without one. If you use the code in VS Code or Google Colab, you **must sign up for a Flickr API key** and insert it into your code.



🖼️ Why is this cool?

Using the API:

- Lets you search and filter photos in your **own apps**
- You can build automatic photo galleries
- You can fetch and analyze data in bulk (not possible from the browser interface)

✅ Summary (in plain English)

- Flickr allows developers to search for photos using **tags** through its API.
- You construct a **URL** with specific search settings.
- You include your **API key** to get access.
- The server sends back **photo data in JSON format**.
- You extract the **photo IDs and owner names** and build URLs to view the photos.