

Day 22 Working with web data

Self-Study Guide · About 75 minutes · With answer key

By the end you will be able to

- Explain what happens in an HTTP request and response, and fetch JSON from a web API with the requests library.
- Check status codes with `raise_for_status()`, and walk the nested dictionaries and lists in a JSON response.
- Send query parameters with `params=`, set a timeout, and handle network errors without crashing.
- Keep an API key out of your code by reading it from an environment variable.

What to remember from earlier

- `json.loads(text)` turns JSON text into Python dictionaries and lists, and `json.dump(data, file, indent=2)` saves them to a file. ([Day 14: Files and data formats, 14.9 JSON Files with `json.dump` and `json.load`](#))
- `d.get(key, default)` gives back the default instead of raising `KeyError` when the key is missing. ([Day 8: Dictionaries, 8.4 Missing Keys: `KeyError` and `get\(\)`](#))
- With a virtual environment active, `python -m pip install name` installs a package into it. ([Day 13: Modules, packages and environments, 13.10 Installing Packages with `pip`](#))
- `except SomeError as error` catches only that type of exception and stores it in `error`. ([Day 15: Errors and exceptions, 15.4 Catching Specific Exceptions](#))
- An `except` clause catches the class it names and every child of that class. ([Day 17: Classes and objects, part 2, 17.1 Inheritance: Building on an Existing Class](#))
- The hint `X | None` means a value of type `X`, or `None`, so check for `None` before using it. ([Day 19: Type hints and code style, 19.3 Values That Might Be `None`](#))
- `sys.exit(message)` stops a program at once and prints the message as an error. ([Day 13: Modules, packages and environments, 13.8 The Computer and the Program: `os` and `sys`](#))

Make a `day22` folder for today's files, create and activate a virtual environment in it (Day 13: `python -m venv .venv`, then activate it), and open the folder in your editor. In 22.2 you will install the `requests` package into that environment. Programs that call a web API need an internet connection; the programs that read a saved response from a file do not.

Every program so far has worked with data you typed, wrote into the code, or kept in a file. Much of the data people care about lives on other computers and changes all the time: today's weather, train times, exchange rates, the scores of a game in progress. Today your programs learn to ask another computer for data, over the internet, and to cope calmly when the answer is slow, wrong, or never comes.

All of today's live examples use **Open-Meteo** (open-meteo.com), a free weather service that needs no account and no key for non-commercial use. Its answers change every hour, so output from a live call is marked **sample output**: yours will show different numbers. Output from a program that reads a saved answer from a file is exact.

22.1 Requests and Responses

When you open a web page, your browser sends a short message to another computer asking for the page, and that computer sends the page back. A Python program can send the same kind of message. Before writing one, you need a picture of what travels in each direction.

The computer that asks is the **client**: your browser, or your Python program. The computer that answers is the **server**. They talk using **HTTP** (Hypertext Transfer Protocol), the set of rules every website and web service follows. The conversation always has two halves:

- The client sends a **request**: a message that says what it wants.
- The server sends back a **response**: a message with the answer, or with a reason it cannot give one.

It works like asking at a library desk: you hand over a slip naming what you want, and the librarian hands back the book, or a note saying it is missing. Where the analogy stops: the server remembers nothing about you between requests, so every request must carry everything the server needs to know.

What you want is named by a **URL** (Uniform Resource Locator), a web address. Here is one that asks Open-Meteo for the temperature in London right now:

```
https://api.open-meteo.com/v1/forecast?latitude=51.51&longitude=-0.13&current=
temperature_2m
```

Part	In this URL	What it means
Scheme	https://	Use HTTP, encrypted (the s stands for secure)
Host	api.open-meteo.com	Which server to ask
Path	/v1/forecast	Which service on that server
Query string	?latitude=51.51&... (to the end)	Extra details for this request, after the ? (22.6)

A request also has a **method**, the verb that says what to do. **GET** means "send me this", and it is the only method you need today. (Others, such as POST, send data to the server, the way a form does.)

The response has three parts:

Part	Example	What it tells you
Status code	200	A three-digit number saying how it went (22.3)
Headers	Content-Type: application/json	Labeled facts about the response, one per line
Body	{"latitude":51.51147, ...}	The data itself

Here is that exchange, shortened (real messages carry more headers). The request:

```
GET /v1/forecast?latitude=51.51&longitude=-0.13&current=temperature_2m HTTP/1.1
Host: api.open-meteo.com
```

The response (sample output):

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"latitude":51.51147,"longitude":-0.13078308,"generationtime_ms":4.958868026733398,"utc_offset_seconds":0,"time":"iso8601","interval":"seconds","temperature_2m":"°C"},"current":{"time":"2026-10-15T18:30","interval":900,"temperature_2m":18.4}}
```

The body is JSON (Day 14), all on one line, because it is written for programs, not people. A URL that answers with data for programs instead of a page for people is part of a **web API** (Application Programming Interface): a service that other programs can call. Paste the URL into your browser's address bar and you will see this same JSON; the browser sends a GET request for you.

KEY IDEA

A client sends a request (a method such as GET and a URL) and the server sends back a response (a status code, headers, and a body). A web API answers with data for programs, usually as JSON.

PAUSE AND TRY

Question 1 Multiple choice

Which of these is part of the **response**, not the request?

- A. The method GET
- B. The host `api.open-meteo.com`
- C. The status code 200
- D. The query string `?latitude=51.51&longitude=-0.13`

Question 2 Short answer

Name the host, the path, and the query string in this URL, which searches Open-Meteo's place-name service:

```
https://geocoding-api.open-meteo.com/v1/search?name=Paris&count=2
```

22.2 Fetching Data with requests

Python's standard library can send HTTP requests (with a module called `urllib.request`), but it takes several steps for each one. Nearly every Python programmer uses the third-party package **requests** instead, which turns a request into one function call. Day 13 used it as the first example of installing a package. With your `day22` environment active:

```
python -m pip install requests
```

`requests.get(url, timeout=10)` sends a GET request to `url`, waits for the response, and returns a **response object** that holds everything the server sent back. (`timeout=10` means "give up after 10 seconds without an answer". 22.7 explains it; always include it.)

```

1 import requests
2
3 url = (
4     "https://api.open-meteo.com/v1/forecast"
5     + "?latitude=51.51&longitude=-0.13&current=temperature_2m"
6 )
7 response = requests.get(url, timeout=10)
8 print(response.status_code)
9 print(response.headers["Content-Type"])
10 print(response.text[:60])
11 data = response.json()
12 print(data["current"]["temperature_2m"])

```

Sample output:

```

200
application/json; charset=utf-8
{"latitude":51.51147,"longitude":-0.13078308,"generationtime
18.4

```

The URL is long, so it is built from two strings inside parentheses (Day 19's line continuation). Each `print` shows a different part of the response object:

Attribute or method	Gives
<code>response.status_code</code>	The status code, as an int: 200
<code>response.headers</code>	The headers, used like a dictionary: <code>response.headers["Content-Type"]</code>
<code>response.text</code>	The body as one string
<code>response.json()</code>	The body turned from JSON into Python dictionaries and lists
<code>response.url</code>	The URL that was actually requested

`response.text[:60]` slices the first 60 characters of the body (Day 3), which shows that `text` is just a string. `response.json()` does the useful work: it reads that string as JSON and gives back a dictionary, so `data["current"]["temperature_2m"]` reaches into it the way you reached into nested dictionaries on Day 8. 22.5 looks at that in detail.

KEY IDEA

Install `requests` with `pip`, then `response = requests.get(url, timeout=10)`. The response object has `status_code`, `headers`, `text` (a string), and `json()`, which turns a JSON body into dictionaries and lists.

WATCH OUT

Running without the package, or naming your file `requests.py`. If `import requests` fails with `ModuleNotFoundError: No module named 'requests'`, the package is not installed in the Python you are running: activate the environment and install it in that same terminal (Day 13). And never name your own file `requests.py`. Python looks in your folder first, so `import requests` imports your file instead of the package, and `requests.get` fails with an `AttributeError`.

(recent versions of Python suggest renaming the file). It is the same trap as a file named `random.py` on Day 13. Rename it, for example to `weather_check.py`.

PAUSE AND TRY

Question 3 Multiple choice

After `response = requests.get(url, timeout=10)` returns a JSON body that starts with `{`, what are the types of `response.text` and `response.json()`?

- A. Both are `str`
- B. `str` and `dict`
- C. `dict` and `str`
- D. Both are `dict`

22.3 Status Codes

A response always comes back, even when something went wrong. The URL might name a service that does not exist, a value in the query string might be impossible, or the server might be broken. The **status code** at the top of every response says which. Its first digit gives the kind of answer:

Range	Kind	Meaning
200–299	Success	The request worked; the body holds what you asked for
300–399	Redirect	The data is at another URL; requests follows the redirect for you
400–499	Client error	Something is wrong with the request: fix it before trying again
500–599	Server error	The request may be fine, but the server failed; trying later may work

Each code also has a short name, its **reason phrase**. These are the ones you will meet most:

Code	Reason phrase	Usual cause
200	OK	Success
301	Moved Permanently	The address changed
400	Bad Request	A value in the request is impossible or missing
401	Unauthorized	No API key, or a wrong one (22.8)
403	Forbidden	The key is fine, but it may not use this service
404	Not Found	No such path, often a typo in the URL
429	Too Many Requests	You called too often; wait before calling again
500	Internal Server Error	A bug on the server
503	Service Unavailable	The server is overloaded or down for maintenance

You met a few of these in Day 21's web server log. A program decides what to do with a response by its code, so it helps to write that decision as a plain function:

status_meaning.py

```
1 def status_meaning(code: int) -> str:
2     """Return what a status code's range says about a response."""
3     if 200 <= code <= 299:
4         return "success"
5     if 300 <= code <= 399:
6         return "redirect: the data is somewhere else"
7     if 400 <= code <= 499:
8         return "client error: something is wrong with the request"
9     if 500 <= code <= 599:
10        return "server error: the server failed"
11    return "not a status code"
12
13
14 for code in [200, 301, 404, 429, 503]:
15     print(code, status_meaning(code))
```

```
200 success
301 redirect: the data is somewhere else
404 client error: something is wrong with the request
429 client error: something is wrong with the request
503 server error: the server failed
```

Each if uses a chained comparison (Day 4) to test one range, and returns as soon as one matches. 429 is a client error even though the request itself was fine: the client did something wrong by calling too often.

An error response usually still has a body, which often explains the problem. Ask Open-Meteo for a latitude of 951.51 (latitudes only go from -90 to 90) and it answers with status 400 and this body (sample output):

```
{"reason": "Latitude must be in range of -90 to 90°. Given: 951.51.", "error": true}
```

That is not a forecast at all. A program that assumes every response is the data it asked for will go wrong on this one, which is why the next section matters. (`response.ok` is a shortcut that is `True` when the code is below 400.)

KEY IDEA

The status code says how a request went: 2xx success, 3xx redirect, 4xx a problem with the request, 5xx a problem on the server. An error response still has a body, but not the data you asked for.

PAUSE AND TRY**Question 4** Tracing

What does this program print?

next_step.py

```
1 def next_step(code: int) -> str:
2     """Say what a program should do after a response with this code."""
3     if code == 200:
4         return "use the data"
5     if code == 429:
6         return "wait, then try again"
7     if code in (401, 403):
8         return "check the API key"
9     if code >= 500:
10        return "try again later"
11    return "fix the request"
12
13
14 for code in [403, 200, 404, 502, 429]:
15     print(code, next_step(code))
```

22.4 Stopping on Errors with `raise_for_status()`

Here is a surprise: `requests.get` does **not** raise an exception for a 404 or a 500. As far as `requests` is concerned, the server answered, so it hands you the response and lets you decide. If your program carries straight on to `response.json()` and `data["current"]`, the failure shows up later, as a `KeyError` about a missing "current" key, far from its real cause.

The fix is one line. `response.raise_for_status()` does nothing if the code is a success, and raises `requests.exceptions.HTTPError` if the code is from 400 to 599. Here is what it does, written by hand with Day 15's tools so you can run it without a network:

raise_by_hand.py

```
1 class HTTPError(Exception):
2     pass
3
4
5 def raise_for_status(code: int, url: str) -> None:
6     """Raise HTTPError for a 4xx or 5xx code, as requests does."""
7     if 400 <= code <= 499:
8         raise HTTPError(f"{code} Client Error for url: {url}")
9     if 500 <= code <= 599:
10        raise HTTPError(f"{code} Server Error for url: {url}")
11
12
13 raise_for_status(200, "https://api.open-meteo.com/v1/forecast")
14 print("200 passed the check")
15 raise_for_status(404, "https://api.open-meteo.com/v1/forecast")
16 print("This line never runs")
```

```
200 passed the check
Traceback (most recent call last):
...
HTTPError: 404 Client Error for url: https://api.open-meteo.com/v1/forecast
```

The first call returns `None` without raising, so line 14 prints. The second URL misspells `forecast`, the server would answer 404, and line 8 raises. Line 16 never runs. The real method works the same way, and its message also includes the reason phrase (sample output):

```
requests.exceptions.HTTPError: 404 Client Error: Not Found for url: https://api.open-meteo.com/v1/forecast
```

So the pattern for every request is three lines, in this order:

```
1 response = requests.get(url, timeout=10)
2 response.raise_for_status()
3 data = response.json()
```

Now a bad response stops the program at the line that checks it, with a message naming the code and the URL, instead of producing a confusing error three lines later. 22.7 shows how to catch the `HTTPError` instead of crashing.

KEY IDEA

`requests` does not raise for error codes on its own. Call `response.raise_for_status()` right after `requests.get`: it does nothing for a success and raises `HTTPError` for any code from 400 to 599.

PAUSE AND TRY

Question 5 Multiple choice

This program has a typo in the path, so the server answers 404 with the body `{"reason": "Not Found", "error": true}`. Which line stops the program?

```
1 response = requests.get(
2     "https://api.open-meteo.com/v1/forecast?latitude=51.51&longitude=-0.13",
3     timeout=10,
4 )
5 data = response.json()
6 print("Got a response")
7 print(data["current"])
```

- A. The `requests.get` call, with an `HTTPError`
- B. `data = response.json()`, because the body is not JSON
- C. `print(data["current"])`, with a `KeyError`
- D. None: it prints `Got a response` and then `None`

22.5 Reading a JSON Response

`response.json()` is not magic. It does exactly what `json.loads(response.text)` does (Day 14): it reads JSON text and builds the matching dictionaries, lists, strings, numbers, booleans, and `None`. You can watch it happen without a network:

```
>>> import json
>>> body = '{"latitude": 51.5, "current": {"temperature_2m": 18.4}}'
>>> data = json.loads(body)
>>> type(data)
<class 'dict'>
>>> data["current"]
{'temperature_2m': 18.4}
>>> data["current"]["temperature_2m"]
18.4
>>> json.loads("<html>Service Unavailable</html>")
Traceback (most recent call last):
...
json.decoder.JSONDecodeError: Expecting value: line 1 column 1 (char 0)
```

Each level of `{...}` becomes a dictionary, so `data["current"]` is itself a dictionary, and a second pair of brackets reaches inside it. The last line shows what happens when the body is not JSON at all, such as the error page a broken server sometimes sends: a `JSONDecodeError` saying the first character was not the start of a JSON value. With requests, `response.json()` raises its own version, `requests.exceptions.JSONDecodeError`. (`raise_for_status()` before `json()` catches most of these, because error pages usually come with error codes.)

Saving a response to work with offline

Because a body is only text, you can save it in a file once and then work on your code all day without calling the API again. That is faster, does not use up the service's goodwill, and gives the same answer every run. The file `forecast.json` holds a response to this request (the extra query parameters ask for three days of highs, lows, and rain; 22.6 explains them):

```
https://api.open-meteo.com/v1/forecast?latitude=51.51&longitude=-0.13&current=
temperature_2m,wind_speed_10m&daily=temperature_2m_max,temperature_2m_min,precipitation_
sum&timezone=Europe/London&forecast_days=3
```

Here is part of the file, saved with `indent=2` so a person can read it:

```
{
  "latitude": 51.51147,
  "longitude": -0.13078308,
  ...
  "current_units": {
    "time": "iso8601",
    "interval": "seconds",
    "temperature_2m": "°C",
    "wind_speed_10m": "km/h"
  },
  "current": {
    "time": "2026-10-15T19:30",
    "interval": 900,
    "temperature_2m": 18.4,
    "wind_speed_10m": 12.2
  },
  ...
  "daily": {
    "time": ["2026-10-15", "2026-10-16", "2026-10-17"],
    "temperature_2m_max": [20.9, 17.2, 13.5],
    "temperature_2m_min": [14.6, 10.0, 8.3],
    "precipitation_sum": [0.0, 8.7, 0.0]
  }
}
```

WORKED EXAMPLE Reaching into nested dictionaries

read_forecast.py

```
1 import json
2 from pathlib import Path
3
4 # The text a web API sent back, saved earlier in a file.
5 text = Path("forecast.json").read_text(encoding="utf-8")
6 data = json.loads(text)
7
8 current = data["current"]
9 units = data["current_units"]
10 print("Time:", current["time"])
11 print("Temperature:", current["temperature_2m"], units["temperature_2m"])
12 print("Wind:", current["wind_speed_10m"], units["wind_speed_10m"])
```

```
Time: 2026-10-15T19:30
Temperature: 18.4 °C
Wind: 12.2 km/h
```

- Line 5 reads the whole file as one string, exactly what `response.text` would hold, and line 6 turns it into a dictionary, exactly what `response.json()` would do.
- Line 8 gives the inner dictionary a short name. `current["time"]` is then the same as `data["current"]["time"]`, but easier to read when you use it three times.
- Open-Meteo sends the units separately, in `"current_units"`, with the same keys as `"current"`. So `units["temperature_2m"]` is the unit for `current["temperature_2m"]`.

Expression	Value
<code>data</code>	The whole dictionary
<code>data["current"]</code>	A dictionary of four keys
<code>data["current"]["temperature_2m"]</code>	18.4, a float
<code>data["current_units"]["temperature_2m"]</code>	'°C', a string

The "daily" block has a different shape: a dictionary of **lists**, where position 0 in every list belongs to the first day, position 1 to the second, and so on. Lists that line up like that are walked together with `zip` (Day 11):

daily_table.py

```

1 import json
2 from pathlib import Path
3
4 data = json.loads(Path("forecast.json").read_text(encoding="utf-8"))
5 daily = data["daily"]
6
7 print(len(daily["time"]), "days")
8 for date, high, rain in zip(
9     daily["time"], daily["temperature_2m_max"], daily["precipitation_sum"]
10 ):
11     print(f"{date}: high {high:.1f}, rain {rain:.1f} mm")

```

```

3 days
2026-10-15: high 20.9, rain 0.0 mm
2026-10-16: high 17.2, rain 8.7 mm
2026-10-17: high 13.5, rain 0.0 mm

```

The other common shape is a **list of dictionaries**, one per record (Day 8). Open-Meteo's place-name search answers that way. `places.json` holds a shortened response to <https://geocoding-api.open-meteo.com/v1/search?name=Paris&count=2>, which asks for at most two places called Paris:

find_city.py

```

1 import json
2 from pathlib import Path
3
4 data = json.loads(Path("places.json").read_text(encoding="utf-8"))
5
6 print(len(data["results"]), "places")
7 for place in data["results"]:
8     print(f"{place['name']}, {place['admin1']}, {place['country']}")
9     print(f"  at {place['latitude']}, {place['longitude']}")

```

```

2 places
Paris, Île-de-France Region, France
  at 48.85341, 2.3488
Paris, Texas, United States
  at 33.66094, -95.55551

```

`data["results"]` is a list, so `len` counts the places and the `for` loop visits each dictionary in it. The f-strings use single quotes for the keys because the whole string is in double quotes.

To read a response you have not seen before, work from the outside in, as with a nested type hint (Day 19): print `data.keys()` to see the top level, then print one value to see what is inside it. And when a function receives parsed JSON, the honest hint for it is `dict[str, Any]`. `Any`, imported with `from typing import Any`, means "a value of any type", which is true of JSON values: one key holds a float, another a list.

KEY IDEA

`response.json()` is `json.loads(response.text)`. Each JSON object becomes a dictionary and each array a list, so chain brackets to reach inside: `data["current"]["temperature_2m"]`. Save a response to a file to work on it offline.

WATCH OUT

Assuming a key is always there. When the place search finds nothing, Open-Meteo leaves the `"results"` key out completely: the whole body is `{"generationtime_ms": 0.42927265}`, saved in `no_places.json`.

`no_places.py`

```
1 import json
2 from pathlib import Path
3
4 data = json.loads(Path("no_places.json").read_text(encoding="utf-8"))
5
6 for place in data["results"]:
7     print(place["name"])
8 print("Done")
```

Traceback (most recent call last):

```
...
KeyError: 'results'
```

An API's documentation says which keys are optional. For those, use `.get()` with a sensible default (Day 8), here an empty list:

`no_places_fixed.py`

```
1 import json
2 from pathlib import Path
3
4 data = json.loads(Path("no_places.json").read_text(encoding="utf-8"))
5
6 places = data.get("results", [])
7 if not places:
8     print("No places found")
9 for place in places:
10     print(place["name"])
11 print("Done")
```

No places found
Done

An empty list is falsy (Day 4), so `if not places:` reports the empty search, and the loop simply runs zero times.

PAUSE AND TRY

Question 6 Tracing

`places.json` is the Paris search from above. What does this program print?

`places_trace.py`

```
1 import json
2 from pathlib import Path
3
4 data = json.loads(Path("places.json").read_text(encoding="utf-8"))
5
6 second = data["results"][1]
7 print(second["country_code"])
8 print(data["results"][0]["population"] > second["population"])
9 print([place["timezone"] for place in data["results"]])
10 print(data.get("total", 0))
```

Question 7 Short answer

Using data loaded from `forecast.json`, write an expression for each: (a) the unit of wind speed, (b) the second day's lowest temperature, (c) the number of days in the forecast.

22.6 Query Parameters

Most APIs take extra details: which place, which measurements, how many results. They go in the query string, after the `?`, as **query parameters**: `name=value` pairs joined with `&`. In

`?latitude=51.51&longitude=-0.13`, there are two parameters, `latitude` with the value `51.51` and `longitude` with `-0.13`. Open-Meteo's documentation lists the ones it understands; here are the ones today uses:

Parameter	Example value	Asks for
<code>latitude</code>	<code>51.51</code>	The place, north or south
<code>longitude</code>	<code>-0.13</code>	The place, east or west
<code>current</code>	<code>temperature_2m,wind_speed_10m</code>	These measurements for right now
<code>daily</code>	<code>temperature_2m_max</code>	These values for each day, separated by commas
<code>timezone</code>	<code>auto</code>	Dates and times in the place's own time zone
<code>forecast_days</code>	<code>3</code>	How many days of forecast

Writing these by hand gets awkward fast, and some characters cannot appear in a URL as they are: a space would end the URL, and an `&` inside a value would look like the start of the next

parameter. So they are replaced by codes, a process called **URL encoding**: a space becomes `+` (or `%20`), and `&` becomes `%26`. The standard library's `urllib.parse.urlencode` shows what that looks like:

```
>>> from urllib.parse import urlencode
>>> urlencode({"latitude": 51.51, "longitude": -0.13})
'latitude=51.51&longitude=-0.13'
>>> urlencode({"name": "New York", "count": 1})
'name=New+York&count=1'
>>> urlencode({"name": "Fish & Chips"})
'name=Fish+%26+Chips'
```

Numbers are turned into text, pairs are joined with `&`, and the space and `&` inside values are encoded so they cannot be mistaken for anything else.

With requests, you never build the query string yourself. Pass a dictionary as `params=`, and requests encodes it and adds it to the URL:

```
1 import requests
2
3 params = {
4     "latitude": 51.51,
5     "longitude": -0.13,
6     "current": "temperature_2m,wind_speed_10m",
7 }
8 response = requests.get(
9     "https://api.open-meteo.com/v1/forecast", params=params, timeout=10
10 )
11 response.raise_for_status()
12 print(response.url)
13 print(response.json()["current"])
```

Sample output:

```
https://api.open-meteo.com/v1/forecast?latitude=51.51&longitude=-0.13&current=
temperature_2m%2Cwind_speed_10m
{'time': '2026-10-15T18:30', 'interval': 900, 'temperature_2m': 18.4, 'wind_speed_10m':
12.2}
```

`response.url` shows the URL requests really sent. The commas were encoded as `%2C`, which the server decodes back into commas. The base URL now has no `?` at all, and the details live in a dictionary that is easy to read and to change: asking for Paris instead means changing two numbers, not editing a long string.

KEY IDEA

Query parameters are `name=value` pairs after the `?`, joined with `&`. Pass them to requests as a dictionary with `params=`, and it URL-encodes them for you, so spaces and `&` inside values cannot break the URL.

WATCH OUT

Gluing user input into the URL with an f-string. A recipe search at an imagined `api.example.com` built like this:

```
1 url = f"https://api.example.com/v1/recipes?q={search}&page=1"
```

works for "pancakes", but when the user types Fish & Chips, the URL becomes `...?q=Fish & Chips&page=1`. The `&` ends the `q` parameter early, so the server searches for "Fish " and sees a strange extra parameter, " Chips". Nothing crashes; the results are just wrong. Write `params={"q": search, "page": 1}` instead, and the value is sent as `q=Fish+%26+Chips`.

PAUSE AND TRY**Question 8** Multiple choice

Which call sends a request to `https://api.example.com/v1/teas?q=green+tea&page=2?`

- A. `requests.get("https://api.example.com/v1/teas", params={"q": "green tea", "page": 2}, timeout=10)`
- B. `requests.get("https://api.example.com/v1/teas", params={"q=green tea", "page=2"}, timeout=10)`
- C. `requests.get("https://api.example.com/v1/teas?", params=["q", "green tea", "page", 2], timeout=10)`
- D. `requests.get("https://api.example.com/v1/teas", params={"green tea": "q", 2: "page"}, timeout=10)`

22.7 Timeouts and Network Errors

Reading a file on your own computer almost always works. A request crosses the internet, and many things can go wrong that have nothing to do with your code: the Wi-Fi drops, the server is down, the server is so busy it never answers, or a train goes into a tunnel. A program that talks to the network has to expect failure as a normal event, not a rare one.

Timeouts

Without a limit, requests may wait for a silent server for a very long time, and your program sits there looking frozen. The **timeout** is the longest, in seconds, that requests will wait: `timeout=10` means "if the server takes more than 10 seconds to connect, or more than 10 seconds to send the next part of its answer, give up". Giving up raises an exception you can handle. It does not limit the total time of a long download that keeps arriving, only each wait. Five or ten seconds suits most APIs. The requests documentation recommends a timeout on nearly every request in real code, and so does this course: every request on this page has one.

The exceptions requests raises

When a request fails, requests raises one of its own exception types, all in `requests.exceptions`:

Exception	Raised when
<code>ConnectionError</code>	No connection could be made: no internet, a misspelled host, the server is down

Timeout	The server took longer than the timeout
HTTPError	<code>raise_for_status()</code> found a code from 400 to 599 (22.4)
RequestException	The parent of all of them (and of <code>JSONDecodeError</code>)

They form a family: every one of them is a child class of `RequestException` (Day 17). So `except requests.exceptions.RequestException:` catches any failure from requests, and an `except` clause for a child catches only that kind. (`requests.exceptions.ConnectionError` is a different class from Python's built-in `ConnectionError`, so write the full name.)

WORKED EXAMPLE A request that never crashes the program

This function returns the parsed body, or `None` if anything went wrong, with a message saying what. The `dict[str, Any] | None` hint (Day 19) warns every caller to check for `None`.

```

1  from typing import Any
2
3  import requests
4
5
6  def get_json(url: str, params: dict[str, Any]) -> dict[str, Any] | None:
7      """Return the JSON body of a GET request, or None if it failed."""
8      try:
9          response = requests.get(url, params=params, timeout=10)
10         response.raise_for_status()
11         return response.json()
12     except requests.exceptions.Timeout:
13         print("The server took too long to answer.")
14     except requests.exceptions.ConnectionError:
15         print("Could not connect. Check your internet connection.")
16     except requests.exceptions.HTTPError as error:
17         print(f"The server refused the request: {error}")
18     except requests.exceptions.RequestException as error:
19         print(f"The request failed: {error}")
20     return None
21
22
23  data = get_json(
24      "https://api.open-meteo.com/v1/forecast",
25      {"latitude": 951.51, "longitude": -0.13, "current": "temperature_2m"},
26  )
27  if data is None:
28      print("No forecast today.")

```

Sample output:

```

The server refused the request: 400 Client Error: Bad Request for url: https://api.
open-meteo.com/v1/forecast?latitude=951.51&longitude=-0.13&current=temperature_2m
No forecast today.

```

- The `try` block holds all three steps, because each can fail: the request (no connection, timeout), the status check (`HTTPError`), and `json()` (a body that is not JSON).

- The impossible latitude gets a 400, so `raise_for_status()` raises `HTTPError`, its clause prints the message, and the function reaches `return None`.
- With the Wi-Fi switched off, the same call would print `Could not connect. Check your internet connection.` instead. With a working connection and a real latitude, `return response.json()` hands back the dictionary and no `except` clause runs.
- Python tries the clauses from the top and runs the first one that matches. `Timeout` comes before `ConnectionError` because a timeout while connecting is a child of both, and "took too long" is the more useful message. The parent, `RequestException`, comes last, to catch whatever the others did not.

Notice what is **not** caught: a `KeyError` from your own code, or a typo that raises `NameError`. Those are bugs, and you want to see their tracebacks (Day 15). Catch the network's failures, not your own.

KEY IDEA

Always pass `timeout=`. Put the request, `raise_for_status()`, and `json()` in a `try` block, and catch requests' exceptions from most specific to least: `Timeout`, `ConnectionError`, `HTTPError`, then `RequestException`.

WATCH OUT

Putting the parent first. If `except requests.exceptions.RequestException:` is the first clause, it matches every requests failure, because each one is a child of it. The clauses below it never run, so every failure gets the same vague message. Python does not warn you. Put the most specific clauses first and the parent last.

PAUSE AND TRY

Question 9 Multiple choice

`get_json` from the worked example is called while the laptop has no internet connection at all. What happens?

- A. The program crashes with a traceback ending in `ConnectionError`
- B. It prints `Could not connect. Check your internet connection.` and returns `None`
- C. It prints `The request failed: ...` and returns `None`
- D. It waits forever, because there is no server to answer

Question 10 Short answer

A program checks a train-times API once a minute, all day. It calls `requests.get(url)` with no timeout. One afternoon the server stops responding without closing the connection. Describe what the program does, and what changes with `timeout=10`.

22.8 Keeping API Keys Out of Your Code

Open-Meteo lets anyone call it, but many APIs do not. They give each user an **API key**: a long secret string, a bit like a password, that you send with every request so the service knows who is calling. It lets the service limit how often each user calls, and charge for heavy use. That makes a key valuable. Anyone who has yours can use up your limit, or run up your bill, in your name.

So a key must never be written into your code. Code gets shared: you send a file to a friend, post it on a forum to ask for help, or upload a project online. A key written in the file goes with it, and people do search online code for keys. Instead, keep the key outside the program, where only your computer has it, in an environment variable.

An **environment variable** is a named piece of text that the terminal hands to every program it starts. Your computer already has many, such as `PATH`, the list of folders where it looks for commands. You set your own in the terminal, before running the program. With a made-up service called Study Facts and a key `your-key-here` standing in for a real one:

```
macOS or Linux:      export STUDY_API_KEY="your-key-here"
Windows PowerShell:  $env:STUDY_API_KEY = "your-key-here"
Windows cmd:         set STUDY_API_KEY=your-key-here
```

Like activating a virtual environment, this lasts only for that one terminal window, until you close it. (Your editor or system settings can store it permanently; the variable's name is what matters to your program.)

Inside Python, `os.environ` holds all the environment variables, and it works like a dictionary of strings. Here it is in a terminal where `STUDY_API_KEY` has not been set:

```
>>> import os
>>> "STUDY_API_KEY" in os.environ
False
>>> print(os.environ.get("STUDY_API_KEY"))
None
>>> os.environ.get("STUDY_API_KEY", "not set")
'not set'
>>> os.environ["STUDY_API_KEY"]
Traceback (most recent call last):
...
KeyError: 'STUDY_API_KEY'
```

Square brackets raise `KeyError` for a missing variable, as with any dictionary.

`os.environ.get(name)` returns `None` instead, so the program can check and give a helpful message:

`api_key.py`

```
1 import os
2 import sys
3
4 api_key = os.environ.get("STUDY_API_KEY")
5 if api_key is None:
6     sys.exit("Set the STUDY_API_KEY environment variable first.")
7
8 print("Key found, ending in", api_key[-4:])
```

Run in a terminal without the variable:

Set the `STUDY_API_KEY` environment variable first.

`api_key` is `None`, so `sys.exit` (Day 13) stops the program with a message that says exactly what to do. Once the variable is set in that terminal, line 8 runs instead, and prints only the last four characters: enough to tell which key is in use, without showing the secret.

Each API says how to send its key, in its documentation. Two common ways are a query parameter or a header. The requests calls look like this, for the imagined Study Facts API:

```
1 response = requests.get(  
2     "https://api.example.com/v1/facts",  
3     headers={"Authorization": f"Bearer {api_key}"},  
4     timeout=10,  
5 )
```

`headers=` takes a dictionary of extra request headers, just as `params=` takes query parameters. The key is read from the environment at run time, so the file holds only the variable's name, never the secret. A missing or wrong key typically gets a 401 response (22.3).

KEY IDEA

Never write an API key in your code. Set it as an environment variable in the terminal, read it with `os.environ.get("NAME")`, and stop with a clear message if it is `None`.

WATCH OUT

Putting the key back in as a default. `os.environ.get("STUDY_API_KEY", "a1b2c3...")` looks tidy, but the real key is now in the file again, and it travels wherever the file goes. A default makes sense only for settings that are not secret. If a key reaches a shared file even once, assume someone has it: create a new key on the service's website and delete the old one.

PAUSE AND TRY

Question 11 Multiple choice

Where should a program get its API key from?

- A. A variable at the top of the program: `API_KEY = "a1b2c3..."`
- B. A comment in the program, so Python ignores it
- C. An environment variable, read with `os.environ.get`
- D. The query string, typed into the URL in the program

Question 12 Short answer

Ana types `export STUDY_API_KEY="your-key-here"` in one terminal window, then runs `python api_key.py` in a second window, and sees
Set the `STUDY_API_KEY` environment variable first. Explain why, and what she should do.

Putting it together

This command-line tool takes a city name, looks up where it is, and prints the weather there now and for the next three days. It is split into two modules on purpose. `report.py` turns a parsed response into lines of text and knows nothing about the network:

weather/report.py

```
1 """Turn an Open-Meteo forecast response into lines of text."""
2
3 from typing import Any
4
5
6 def current_line(data: dict[str, Any]) -> str:
7     """Return one line describing the weather right now."""
8     current = data["current"]
9     units = data["current_units"]
10    temperature = f"{current['temperature_2m']} {units['temperature_2m']}"
11    wind = f"{current['wind_speed_10m']} {units['wind_speed_10m']}"
12    return f"Now: {temperature}, wind {wind}"
13
14
15 def daily_lines(data: dict[str, Any]) -> list[str]:
16     """Return one line per day: the date, the low and high, and any rain."""
17     daily = data["daily"]
18     lines = []
19     for date, low, high, rain in zip(
20         daily["time"],
21         daily["temperature_2m_min"],
22         daily["temperature_2m_max"],
23         daily["precipitation_sum"],
24     ):
25         line = f"{date}: {low:.1f} to {high:.1f}"
26         if rain > 0:
27             line += f", rain {rain:.1f} mm"
28         lines.append(line)
29     return lines
```

Because it needs no network, you can try it, and test it (Day 20), on the saved `forecast.json`:

weather/show_report.py

```
1 import json
2 from pathlib import Path
3
4 from report import current_line, daily_lines
5
6 data = json.loads(Path("forecast.json").read_text(encoding="utf-8"))
7 print(current_line(data))
8 for line in daily_lines(data):
9     print(line)
```

```
Now: 18.4 °C, wind 12.2 km/h
2026-10-15: 14.6 to 20.9
2026-10-16: 10.0 to 17.2, rain 8.7 mm
2026-10-17: 8.3 to 13.5
```

`weather.py` does the network part, using `get_json` from 22.7 (copy it in above `find_place`, with its imports):

```
1  """Print the weather for a city: python weather.py CITY"""
2
3  import sys
4  from typing import Any
5
6  import requests
7
8  from report import current_line, daily_lines
9
10 GEOCODE_URL = "https://geocoding-api.open-meteo.com/v1/search"
11 FORECAST_URL = "https://api.open-meteo.com/v1/forecast"
12
13 # get_json from 22.7 goes here.
14
15
16 def find_place(name: str) -> dict[str, Any] | None:
17     """Return the best match for a place name, or None."""
18     data = get_json(GEOCODE_URL, {"name": name, "count": 1})
19     if data is None:
20         return None
21     results = data.get("results", [])
22     if not results:
23         print(f"No place called {name}.")
24         return None
25     return results[0]
26
27
28 def main() -> None:
29     if len(sys.argv) != 2:
30         sys.exit("Usage: python weather.py CITY")
31     place = find_place(sys.argv[1])
32     if place is None:
33         return
34     params = {
35         "latitude": place["latitude"],
36         "longitude": place["longitude"],
37         "current": "temperature_2m,wind_speed_10m",
38         "daily": "temperature_2m_max,temperature_2m_min,precipitation_sum",
39         "timezone": "auto",
40         "forecast_days": 3,
41     }
42     data = get_json(FORECAST_URL, params)
43     if data is None:
44         return
45     print(f"{place['name']}, {place['country']}")
46     print(current_line(data))
47     for line in daily_lines(data):
48         print(line)
49
50
51 if __name__ == "__main__":
52     main()
```

Run with `python weather.py Tokyo` (sample output):

```
Tokyo, Japan
Now: 15.9 °C, wind 4.7 km/h
2026-10-16: 15.9 to 25.9
2026-10-17: 15.8 to 24.1
2026-10-18: 12.8 to 22.6
```

Where	What to notice	Idea
<code>find_place</code>	The first request turns a name into a latitude and longitude, through <code>params=</code>	22.6
<code>data.get("results", [])</code>	A search with no matches has no <code>"results"</code> key, so <code>.get</code> avoids a <code>KeyError</code>	22.5
<code>get_json</code>	Timeout, <code>raise_for_status()</code> , and every requests failure handled in one place, returning <code>None</code>	22.4, 22.7
<code>if place is None: return</code>	Each caller checks for <code>None</code> before using the result, as the hint warns	22.7
<code>"timezone": "auto"</code>	Dates come back in the city's own time zone, so Tokyo's "today" is already the 16th	22.6
<code>report.py</code> , zip over daily lists	The daily block is a dictionary of lists that line up by position	22.5
Two modules	Only <code>weather.py</code> touches the network; <code>report.py</code> can be tested on saved data	22.5

Keeping the network code thin and the rest of the program in plain functions of plain data is the habit to take away: the part that is hard to test (the internet) is small, and the part with the logic can be tested with saved responses, as many times as you like, with exact answers. A city name with a space needs quotes, as on Day 13: `python weather.py "New York"`.

Common mistakes

Use this table to go from what you see to its usual cause. Messages from requests are samples; their wording varies by version.

What you see	Usual cause	Fix
<code>ModuleNotFoundError: No module named 'requests'</code>	<code>requests</code> is not installed in the Python you are running	Activate the environment, then <code>python -m pip install requests</code>
<code>AttributeError on requests.get</code>	Your own file is named <code>requests.py</code>	Rename it
<code>KeyError</code> on a key the documentation promises	An error response, read without checking the status	Call <code>raise_for_status()</code> before <code>json()</code>
<code>KeyError</code> on an optional key	The API left the key out of this response	Use <code>.get(key, default)</code>
<code>HTTPError: 404 Client Error</code>	A typo in the URL's path	Compare the URL with the documentation

<code>HTTPError: 400 Client Error</code>	A missing or impossible query parameter	Read the body, which often says which one
<code>HTTPError: 401 Client Error</code>	No API key, or a wrong one	Check the environment variable in this terminal
<code>HTTPError: 429 Client Error</code>	Too many requests	Wait, call less often, and save responses to reuse
<code>ConnectionError</code>	No internet, or a misspelled host	Check the connection and the host name
<code>Timeout</code>	The server did not answer in time	Try again later; catch it so the program carries on
<code>JSONDecodeError</code>	The body is not JSON, such as an error page	Check the status first; print <code>response.text</code> to see it

Vocabulary

Term	Definition	Example
client	The program or computer that sends a request	Your Python program
server	The computer that answers requests	<code>api.open-meteo.com</code>
HTTP	The rules for requests and responses on the web	<code>GET /v1/forecast</code>
request	A message from a client: a method, a URL, and headers	<code>requests.get(url, timeout=10)</code>
response	A server's answer: a status code, headers, and a body	200, then JSON
URL	A web address: scheme, host, path, and query string	<code>https://api.open-meteo.com/v1/search</code>
method	The verb of a request; GET asks for data	<code>GET</code>
status code	A three-digit number saying how a request went	404
reason phrase	The short name of a status code	Not Found
headers	Labeled facts sent with a request or response	<code>Content-Type: application/json</code>
body	The data part of a response	<code>{"latitude": 51.51, ...}</code>
web API	A service that answers requests with data for programs	Open-Meteo
requests	A third-party package for sending HTTP requests	<code>import requests</code>
response object	What <code>requests.get</code> returns: everything the server sent back	<code>response.status_code</code>

<code>raise_for_status()</code>	Raises <code>HTTPError</code> if the status code is from 400 to 599	<code>response.raise_for_status()</code>
<code>Any</code>	A type hint meaning a value of any type	<code>dict[str, Any]</code>
query parameter	A <code>name=value</code> pair in the query string	<code>count=2</code>
URL encoding	Replacing characters that cannot appear in a URL with codes	<code>space → +, & → %26</code>
timeout	The longest to wait for the server before giving up	<code>timeout=10</code>
API key	A secret string that identifies you to an API	Sent in a header
environment variable	A named text value that the terminal gives to programs it starts	<code>STUDY_API_KEY</code>

Summary

- A **client** sends an HTTP **request** (a method such as GET and a **URL**), and the **server** sends a **response** (a **status code**, **headers**, and a **body**). A **web API** answers with JSON for programs to use.
- `requests.get(url, params=..., timeout=10)` sends a request. `params=` takes a dictionary and URL-encodes it into the query string; `headers=` adds headers.
- Status codes: 2xx success, 3xx redirect, 4xx a problem with the request, 5xx a problem on the server. `requests` does not raise for them on its own, so call `raise_for_status()` before `response.json()`.
- `response.json()` is `json.loads(response.text)`: chain brackets to reach into nested dictionaries and lists, and use `.get()` for keys that may be missing. Save a response to a file to work offline.
- The network fails as a matter of course. Always set a **timeout**, and catch `Timeout`, `ConnectionError`, `HTTPError`, and their parent `RequestException`, most specific first. Keep **API keys** in **environment variables**, read with `os.environ.get`.

Practice

Do these in your `day22` folder, with `requests` installed in your virtual environment (22.2). Type each program yourself, run it, and only then open the answer. Your output will differ from the sample output wherever it comes from a live call.

Question 13 Short answer

Three fields from a public API. Call a free public JSON API and print three fields from the response. (Open-Meteo's place search, <https://geocoding-api.open-meteo.com/v1/search>, is a good one: give it `name` and `count`.)

Question 14 Short answer

A safe function. Wrap the call in a function that takes a city name, uses a timeout, handles network and HTTP errors, and returns the place's dictionary, or `None` if anything went wrong. Test it with a real city, a made-up name, and your Wi-Fi switched off.

Question 15 Short answer

Save it, then load it without the network. Save a response to a JSON file. Then write a second program that loads it back and prints the same three fields, without calling the API.

Review questions

Question 16 Debugging

This program should print the temperature in Paris right now. It stops with `KeyError: 'current'`. Find every problem, and fix the program.

```
1 import requests
2
3 lat, lon = 48.85, 2.35
4 response = requests.get(
5     f"https://api.open-meteo.com/v1/forecast?latitude={lat}&longitude={lon}"
6 )
7 data = response.json()
8 print(data["current"]["temperature_2m"])
```

Question 17 Tracing

This program parses a body like one from the place search, with one place's population missing. What does it print?

review_trace.py

```
1 import json
2
3 body = """
4 {"results": [
5     {"name": "Springfield", "admin1": "Illinois", "population": 114394},
6     {"name": "Springfield", "admin1": "Missouri"}
7 ]}
8 """
9 data = json.loads(body)
10
11 for place in data["results"]:
12     people = place.get("population", "unknown")
13     print(f"{place['name']}, {place['admin1']}: {people}")
14 known = sum(1 for place in data["results"] if "population" in place)
15 print(len(data["results"]), "places,", known, "with a population")
```

Question 18 Multiple choice

A program's `except` clauses are in this order: `RequestException`, then `Timeout`, then `ConnectionError`. The server takes 30 seconds to answer a request sent with `timeout=5`. Which clause runs?

- A. `Timeout`, because it names the problem exactly
- B. `RequestException`, because it is first and `Timeout` is its child
- C. Both `RequestException` and `Timeout`, one after the other
- D. None of them: the request waits the full 30 seconds and succeeds

Question 19 Short answer

A classmate's weather program has its requests, parsing, and printing all in one long function. Explain how splitting it like `weather.py` and `report.py` lets them test the program with `pytest` (Day 20), and why tests should not call the real API.

Question 20 Multiple choice

A program works on your laptop, but your friend runs the same file and gets `HTTPError: 401 Client Error`. What is the most likely cause?

- A. Your friend's internet connection is down
- B. Your friend has not set the API key's environment variable, or set it in a different terminal
- C. The server is broken
- D. The URL has a typo

Question 21 Short answer

Ben pasted his program, including `API_KEY = "..."` with his real key, into a public forum post. He has now deleted the post. Is the key safe again? What should he do, and how should he change the program?

Go further

The requests documentation's Quickstart covers today's tools and a few more, such as sending data with POST. Open-Meteo's API documentation lists every measurement and query parameter its forecast service accepts.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can name the parts of a request and a response, and the parts of a URL. Reread: 22.1 Requests and Responses
- ☐ I can install requests, fetch a URL, and use `status_code`, `text`, and `json()`. Reread: 22.2 Fetching Data with requests
- ☐ I can say what a status code means from its first digit. Reread: 22.3 Status Codes
- ☐ I can explain why `raise_for_status()` belongs right after `requests.get`. Reread: 22.4 Stopping on Errors with `raise_for_status()`
- ☐ I can reach into nested dictionaries and lists in a JSON response, and use `.get()` for optional keys. Reread: 22.5 Reading a JSON Response
- ☐ I can send query parameters with `params=` instead of building the URL myself. Reread: 22.6 Query Parameters
- ☐ I can set a timeout and catch network errors in the right order. Reread: 22.7 Timeouts and Network Errors

☐ I can keep an API key in an environment variable and read it with `os.environ.get`. Reread: 22.8
Keeping API Keys Out of Your Code

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

C. The server decides how the request went and puts the status code at the top of its response.

- **A** is wrong: the client chooses the method and sends it in the request.
- **B** is wrong: the host is part of the URL, which the client sends to say which server to ask.
- **D** is wrong: the query string is part of the URL too. It carries the client's extra details to the server.

Reread *22.1 Requests and Responses* if you chose differently.

Question 2

The host is `geocoding-api.open-meteo.com`, the path is `/v1/search`, and the query string is `?name=Paris&count=2` (the part after the `?`). A correct answer stops the host at the first `/` after `https://`, and starts the query string at `?`.

Reread *22.1 Requests and Responses* if your answer differed.

Question 3

B. `response.text` is the body exactly as it arrived, one string. `response.json()` reads that string as JSON, and a JSON object (`{...}`) becomes a Python `dict`.

- **A** is wrong: turning JSON text into Python values is the whole job of `json()`, so it does not give back a string.
- **C** is wrong: it has them the wrong way round.
- **D** is wrong: `text` is never converted. That is why you cannot write `response.text["current"]`.

Reread *22.2 Fetching Data with requests* if you chose differently.

Question 4

```
403 check the API key
200 use the data
404 fix the request
502 try again later
429 wait, then try again
```

The checks run in order and the first match returns. 403 is not 200 or 429, but it is in the tuple `(401, 403)`. 200 matches the first `if`. 404 matches nothing: it is not 200, 429, 401, or 403, and it is not 500 or more, so it reaches the last line. 502 is 500 or more. 429 matches the second `if` before the `>= 500` test is reached.

Reread *22.3 Status Codes* if your output differed.

Question 5

C. Without `raise_for_status()`, a 404 is not an error to requests, so the call returns normally. The error body is valid JSON, so `json()` gives the dictionary `{'reason': 'Not Found', 'error': True}`, and the program prints `Got a response`. That dictionary has no `"current"` key, so the last line raises `KeyError: 'current'`.

- **A** is wrong: `requests.get` never raises for a status code. Only `raise_for_status()` does.
- **B** is wrong: the error body is JSON too.
- **D** is wrong: indexing with `[...]` raises `KeyError` for a missing key; only `.get()` gives `None`.

The fix is `response.raise_for_status()` right after the call, which stops the program with a message naming the 404. Reread *22.4 Stopping on Errors with `raise_for_status()`* if you chose differently.

Question 6

```
US
True
['Europe/Paris', 'America/Chicago']
0
```

`data["results"][1]` is the **second** place (indexes start at 0): Paris, Texas, whose `"country_code"` is `"US"`. Line 8 compares the two populations, 2138551 and 24782, so it prints `True`. Line 9 is a list comprehension (Day 11) that takes the `"timezone"` of each place in order. Line 10 asks for a key the response does not have, so `.get` gives its default, 0.

Reread *22.5 Reading a JSON Response* if your output differed.

Question 7

(a) `data["current_units"]["wind_speed_10m"]`, which is `'km/h'`. (b) `data["daily"]["temperature_2m_min"][1]`, which is `10.0`: the lows are a list, and index 1 is the second day. (c) `len(data["daily"]["time"])`, which is 3 (any of the daily lists would do, since they all have one item per day).

A correct answer goes from the outside in: first the top-level key, then the inner key, then a list index where the value is a list.

Reread *22.5 Reading a JSON Response* if your answer differed.

Question 8

A. `params=` takes a dictionary whose keys are the parameter names and whose values are the values. `requests` encodes the space in `green tea` as `+`, turns 2 into `"2"`, and joins the pairs with `&`.

- **B** is wrong: braces around plain strings with no colons make a **set** (Day 7), not a dictionary, so there are no names and values.
- **C** is wrong: a flat list does not pair names with values, and the URL should not end in `?`, since `requests` adds it.
- **D** is wrong: the keys and values are swapped, so it would send `green+tea=q&2=page`.

Reread *22.6 Query Parameters* if you chose differently.

Question 9

B. With no connection, `requests.get` raises `requests.exceptions.ConnectionError`. It is not a `Timeout`, so the first clause is skipped, and the second clause matches. After it prints, the function carries on to `return None`.

- **A** is wrong: the exception is caught inside the function, so there is no traceback.
- **C** is wrong: that clause would also match, but Python runs only the **first** matching clause, and `ConnectionError` comes earlier.

- **D** is wrong: with no network at all, the connection fails at once. Even when a server is silent, `timeout=10` stops the wait.

Reread 22.7 *Timeouts and Network Errors* if you chose differently.

Question 10

Model answer: without a timeout, requests may wait for the silent server for a very long time, so the program appears frozen: no error, no message, and no further checks, possibly for hours. With `timeout=10`, the wait ends after 10 seconds with `requests.exceptions.Timeout`, which the program can catch, report, and then carry on to the next check a minute later.

A correct answer says that without a timeout the program hangs instead of failing, and that the timeout turns a hang into an exception the program can handle.

Reread 22.7 *Timeouts and Network Errors* if your reasoning differed.

Question 11

C. The key stays on your computer, outside the file, so sharing the program shares only the variable's name.

- **A** is wrong: anyone who sees the file sees the key.
- **B** is wrong: Python ignores comments, but people reading the file do not, so the key is still shared. And the program could not use it.
- **D** is wrong: some APIs do take the key as a query parameter, but typing it into the URL in your code puts it in the file again. Read it from the environment first, then pass it in `params=`.

Reread 22.8 *Keeping API Keys Out of Your Code* if you chose differently.

Question 12

Model answer: an environment variable set with `export` exists only in the terminal where it was typed, and in programs started from that terminal. The second window never got it, so `os.environ.get("STUDY_API_KEY")` returned `None` and the program stopped. She should run the program in the first window, or set the variable again in the second one.

A correct answer says the variable belongs to one terminal session, like an activated virtual environment.

Reread 22.8 *Keeping API Keys Out of Your Code* if your reasoning differed.

Question 13

One possible solution, `city_fields.py`:

```
1 import requests
2
3 response = requests.get(
4     "https://geocoding-api.open-meteo.com/v1/search",
5     params={"name": "Tokyo", "count": 1},
6     timeout=10,
7 )
8 response.raise_for_status()
9 place = response.json()["results"][0]
10 print("Name:", place["name"])
11 print("Country:", place["country"])
12 print("Population:", place["population"])
```

Sample output:

```
Name: Tokyo
Country: Japan
Population: 9733276
```

How it works: `params=` builds `?name=Tokyo&count=1`, `raise_for_status()` stops at once on an error code, and `json()` gives a dictionary whose `"results"` key holds a list with one place. `[0]` takes that place, and three keys are printed. Before writing the `print` lines, print `response.json()` once to see which keys there are. A search for a name with no matches would raise `KeyError: 'results'`; Practice 2 handles that.

Reread 22.2 *Fetching Data with requests* and 22.5 *Reading a JSON Response* if you got stuck.

Question 14

One possible solution:

```
1 from typing import Any
2
3 import requests
4
5 SEARCH_URL = "https://geocoding-api.open-meteo.com/v1/search"
6
7
8 def find_city(name: str) -> dict[str, Any] | None:
9     """Return the first place called name, or None on any failure."""
10    try:
11        response = requests.get(
12            SEARCH_URL, params={"name": name, "count": 1}, timeout=10
13        )
14        response.raise_for_status()
15        data = response.json()
16    except requests.exceptions.Timeout:
17        print("The search took too long. Try again later.")
18        return None
19    except requests.exceptions.ConnectionError:
20        print("Could not connect. Check your internet connection.")
21        return None
22    except requests.exceptions.RequestException as error:
23        print(f"The search failed: {error}")
24        return None
25    results = data.get("results", [])
26    if not results:
27        print(f"No place called {name}.")
28        return None
29    return results[0]
30
31
32 for name in ["Tokyo", "Qzxqzxq"]:
33     place = find_city(name)
34     if place is not None:
35         print(f"{place['name']}, {place['country']}")
```

Sample output, online:

```
Tokyo, Japan
No place called Qzxqzxq.
```

With the Wi-Fi off, both searches print `Could not connect`. Check your internet connection.

How it works: the `try` block covers the request, the status check, and `json()`, and nothing else, so a bug in the code below it is not hidden. The last `except` is the parent `RequestException`, which catches `HTTPError` and `JSONDecodeError` along with anything else from requests. A made-up name is not a network error: the search succeeds with no `"results"` key, which `.get` turns into an empty list. Every path returns either a dictionary or `None`, as the hint promises, and the loop checks for `None` before using the result.

Reread 22.7 *Timeouts and Network Errors* if your function differed.

Question 15

One possible solution. `save_city.py` calls the API once and saves the parsed body:

```
1 import json
2
3 import requests
4
5 response = requests.get(
6     "https://geocoding-api.open-meteo.com/v1/search",
7     params={"name": "Tokyo", "count": 1},
8     timeout=10,
9 )
10 response.raise_for_status()
11 with open("city.json", "w", encoding="utf-8") as file:
12     json.dump(response.json(), file, indent=2)
13 print("Saved city.json")
```

`load_city.py` never imports requests, so it works with no connection at all, and gives the same answer every run:

`load_city.py`

```
1 import json
2 from pathlib import Path
3
4 path = Path("city.json")
5 if not path.exists():
6     print("No saved copy yet: run save_city.py first")
7 else:
8     with path.open(encoding="utf-8") as file:
9         data = json.load(file)
10     place = data["results"][0]
11     print(f"Loaded {path} without the network")
12     print("Name:", place["name"])
13     print("Country:", place["country"])
14     print("Population:", place["population"])
```

```
Loaded city.json without the network
Name: Tokyo
Country: Japan
Population: 9733276
```

How it works: `json.dump(..., indent=2)` writes the dictionary from `response.json()` back out as readable JSON (Day 14), and `json.load` reads it back as the same dictionary, so the loading program reaches into it exactly as Practice 1 did. `path.exists()` (Day 14) gives a clear message if you run the loader before the saver. Saving responses like this lets you work on the rest of a program without calling the API every run, which is quicker, kinder to free services, and repeatable. (`json.dump` writes a character such as `"` as the code `"` by default; it loads back as `"`.)

Reread 22.5 *Reading a JSON Response* if you got stuck.

Question 16

The request succeeds, but it never asks for current conditions: Open-Meteo includes "current" only when the `current` query parameter names the measurements. So the response has no "current" key. Two other problems would cause trouble on another day: there is no timeout, so a silent server could freeze the program, and there is no `raise_for_status()`, so an error response would also end in a confusing `KeyError`. Fixed:

```
1 import requests
2
3 params = {"latitude": 48.85, "longitude": 2.35, "current": "temperature_2m"}
4 response = requests.get(
5     "https://api.open-meteo.com/v1/forecast", params=params, timeout=10
6 )
7 response.raise_for_status()
8 data = response.json()
9 print(data["current"]["temperature_2m"])
```

Building the query string with `params=` also makes the missing parameter easy to spot. When a key you expected is missing, print the whole response before guessing.

Reread 22.4 *Stopping on Errors with `raise_for_status()`* and 22.6 *Query Parameters* if you missed a problem.

Question 17

```
Springfield, Illinois: 114394
Springfield, Missouri: unknown
2 places, 1 with a population
```

`json.loads` turns the string into a dictionary whose "results" is a list of two dictionaries. For the first, `.get("population", "unknown")` finds 114394. The second has no "population" key, so `.get` gives the default, "unknown", and the loop carries on instead of raising `KeyError`. Line 14 counts the places that have the key: "population" in `place` checks keys (Day 8), and `sum` of a generator of 1s counts them (Day 11).

Reread 22.5 *Reading a JSON Response* if your output differed.

Question 18

B. Python checks the clauses from the top and runs only the first one that matches. A `Timeout` is a kind of `RequestException`, so the first clause matches, and the other two can never run for any requests failure.

- **A** is wrong: the exact match comes too late. Order matters more than exactness.
- **C** is wrong: at most one `except` clause runs per exception.
- **D** is wrong: after 5 seconds without an answer, requests gives up and raises `Timeout`.

Move `RequestException` to the end. Reread *22.7 Timeouts and Network Errors* if you chose differently.

Question 19

Model answer: with the parsing and formatting in `report.py`, a test can load a saved response such as `forecast.json` and check, for example, that `daily_lines(data)[1]` is `"2026-10-16: 10.0 to 17.2, rain 8.7 mm"`. That answer is known and never changes. A test that called the real API would fail whenever the internet is down or the server is busy, even though the code is fine, and the weather changes every hour, so there is no fixed expected answer. That breaks Day 20's rules that a test is repeatable and has a known answer. Only the thin network part in `weather.py` is left untested by these tests, and it is short enough to check by hand.

A correct answer says that saved responses give fixed, known inputs, and that live calls make tests depend on the network and on data that changes.

Reread *Putting it together* if your reasoning differed.

Question 20

B. 401 means the request was missing a valid key. The key lives in an environment variable on **your** computer, not in the file, so your friend needs their own key set in the terminal they run from. That is the price, and the point, of keeping keys out of code.

- **A** is wrong: with no connection there would be no response at all, so no status code; requests raises `ConnectionError`.
- **C** is wrong: a broken server sends a 5xx code, not 401.
- **D** is wrong: a typo in the path usually gives 404, and the same file works for you.

Reread *22.3 Status Codes* and *22.8 Keeping API Keys Out of Your Code* if you chose differently.

Question 21

Model answer: no. Once a key has been public, even for a minute, he must assume someone copied it. He should go to the service's website, create a new key, and delete (revoke) the old one, so the copied key stops working. Then he should remove the key from the program, read it with `os.environ.get("NAME")`, stop with a clear message if it is `None`, and set the variable in his terminal before running. Next time he can share the file safely, because it holds only the variable's name.

A correct answer says that deleting the post is not enough, that the key must be replaced, and that the new key belongs in an environment variable.

Reread *22.8 Keeping API Keys Out of Your Code* if your reasoning differed.