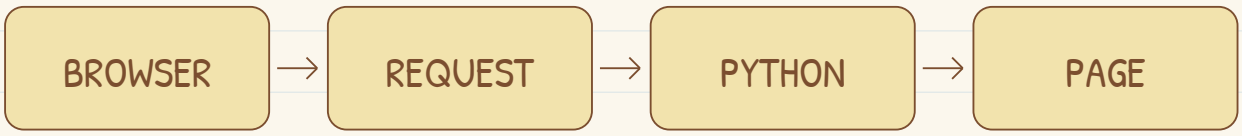
THE PRACTICAL NOTEBOOK

Python for Web

Build clear pages and useful browser experiences.

64 pages / 20 guided lessons / 40 code files



Understand it. Try it. Explain it.

Build server-rendered websites with Python and Flask while learning what the browser does. Follow a request through routing, HTML, templates, forms, validation and a response. The examples use Flask test clients so you can verify behaviour without a public server. Your final project is a small searchable learning library.

PRIVATUS COSMAS / PRIVER

depriver.tech · First edition / October 2026

Your 20-lesson route

Work in order, or use this clickable map to revisit a topic. Each lesson has a learn page, a practice page and an explained solution.

01	A request becomes a response	4
02	HTML with meaning	7
03	Routes and URL parameters	10
04	Templates and escaping	13
05	Reusable page layouts	16
06	CSS and responsive layout	19
07	Forms and GET requests	22
08	POST and validation	25
09	CSRF and form intent	28
10	Redirect after submission	31
11	Cookies and sessions	34
12	Database-backed pages	37
13	Pagination and navigation	40
14	Accessible interaction	43
15	Static files and caching	46
16	Small JavaScript enhancements	49
17	Search-friendly pages	52
18	Error pages that help	55
19	Testing and release checks	58
20	Project: a learning library	61

Setup and study method · 3 References and next steps · 64

Make this book yours

BEFORE YOU START

You should understand Python functions, lists and dictionaries. This book introduces HTML and HTTP gently. It focuses on websites people browse; the separate backend book goes deeper into JSON APIs, authentication boundaries and service operations.

YOUR WORKSPACE

Use Python 3.11 or newer. Create and activate a virtual environment, then install Flask. Save each example in its own file. Run it normally to execute the included checks. To view an example with an app variable in a browser, run python -m flask --app lesson_01_example run, then open the local URL. Keep the development server local.

TERMINAL / FOLLOW THE COMMANDS FOR THIS BOOK

```
python -m pip install Flask
python lesson_01_example.py
python -m flask --app lesson_01_example run
```

A SIMPLE STUDY RHYTHM

1. Read: predict what the example will do before running it.
2. Practise: write your answer using the acceptance cases.
3. Compare: read the solution, explain the difference, then extend it.

The companion ZIP contains lesson_01_example.py through lesson_20_example.py and matching solution files. Run them individually from the extracted folder. Most successful checks are silent; a traceback means a check failed.

READABLE NOTES, PRECISE CODE

Handwritten headings keep the notebook feel; body text and code use clear fonts. Copy runnable code from the ZIP to preserve indentation. The PDF is searchable and its contents links are clickable.

A request becomes a response

THE GOAL

See the complete browser-to-Python cycle.

A browser sends an HTTP request to a URL. The server chooses a handler, performs work and returns a status, headers and body. HTML describes the page; CSS controls presentation; JavaScript can add browser behaviour. Flask connects requests to Python functions. The development server is a learning tool, not a public production server.

lesson_01_example.py

```
from flask import Flask
app = Flask(__name__)

@app.get("/")
def home():
    return "<h1>My learning notebook</h1>"

client = app.test_client()
response = client.get("/")
assert response.status_code == 200
assert b"learning notebook" in response.data
```

The decorator associates GET / with home. The test client sends a request inside the process and inspects the response without opening a network port.

WATCH FOR THIS

Opening a .py file directly in a browser does not run the server. Start Flask and visit its local URL.

Your turn: A request becomes a response

THE CHALLENGE

Add an About page and verify that an unknown path returns 404. Give each route a meaningful function name.

ACCEPTANCE CHECKS

[ ] GET / -> 200

[ ] GET /about -> 200 and your introduction

[ ] GET /missing -> 404

A NUDGE, IF YOU NEED ONE

Add a second @app.get decorator above a separate function.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

A REQUEST BECOMES A RESPONSE

lesson_01_solution.py

```
from flask import Flask
app = Flask(__name__)

@app.get("/")
def home():
    return "<h1>Learning library</h1>"

@app.get("/about")
def about():
    return "<h1>About</h1><p>I build useful software.</p>"

client = app.test_client()
assert client.get("/").status_code == 200
assert b"About" in client.get("/about").data
assert client.get("/missing").status_code == 404
```

WHY IT WORKS

A route that does not exist naturally gives 404. Do not turn every unknown path into a success page, because users and crawlers need accurate status codes.

TAKE IT FURTHER

Open browser developer tools, select Network, and identify the document request and its status.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

HTML with meaning

THE GOAL

Give the page a structure that works without styling.

Semantic HTML explains purpose: headings introduce sections, nav groups navigation and main contains primary content. A page title identifies the browser tab; the h1 identifies the visible page. Use links for navigation and buttons for actions. Start with readable structure before adding decoration.

lesson_02_example.py

```
from flask import Flask
app = Flask(__name__)
PAGE = """<!doctype html><html lang="en">
<head><title>Python notes</title></head>
<body><nav><a href="/">Home</a></nav><main>
<h1>Python notes</h1><p>One idea at a time.</p>
</main></body></html>"""
@app.get("/")
def home():
    return PAGE
assert b"<main>" in app.test_client().get("/").data
```

The response has a language, title, navigation and main content. These elements help readers and assistive technology understand the document.

WATCH FOR THIS

A visually large paragraph is not a heading. Use HTML meaning first and styling second.

Your turn: HTML with meaning

THE CHALLENGE

Create a reading-list page with one h1, a list of two resources and a meaningful link for each. Do not use click here as both link labels.

ACCEPTANCE CHECKS

[ ] One h1

[ ] A main landmark

[ ] Two descriptive links

[ ] Language declared as en

A NUDGE, IF YOU NEED ONE

Use ul and li for a group of related resources.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

HTML WITH MEANING

lesson_02_solution.py

```
from flask import Flask
app = Flask(__name__)
@app.get("/")
def home():
    return """<!doctype html><html lang="en">
    <head><title>Reading list</title></head><body>
    <main><h1>Reading list</h1><ul>
    <li><a href="/git">Read the Git notebook</a></li>
    <li><a href="/python">Read the Python notebook</a></li>
    </ul></main></body></html>"""
html = app.test_client().get("/").text
assert html.count("<h1>") == 1
assert "<ul>" in html and 'lang="en"' in html
```

WHY IT WORKS

The link destinations are placeholders for later routes; this lesson checks structure, not complete site navigation. Add those routes before publishing.

TAKE IT FURTHER

Navigate the page using only the keyboard and confirm that each link has an understandable label.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Routes and URL parameters

THE GOAL

Represent a resource with a predictable address.

Routes can contain variables such as /lessons/3. Converters validate basic shapes before your handler runs. A missing resource is different from a malformed route. Keep permanent identifiers separate from display titles so changing a title does not break every link.

lesson_03_example.py

```
from flask import Flask, abort
app = Flask(__name__)
lessons = {1: "Variables", 2: "Functions"}
@app.get("/lessons/<int:ident>")
def detail(ident):
    if ident not in lessons:
        abort(404)
    return lessons[ident]
c = app.test_client()
assert c.get("/lessons/1").text == "Variables"
assert c.get("/lessons/99").status_code == 404
```

The integer converter produces a Python integer. It does not prove a lesson exists. The handler performs that domain check.

WATCH FOR THIS

Never turn a URL parameter directly into a filesystem path without a separate path policy.

Your turn: Routes and URL parameters

THE CHALLENGE

Build a tiny city directory using slugs such as mbeya. Return 404 for unknown slugs rather than showing an unrelated city.

ACCEPTANCE CHECKS

[ ] /cities/mbeya -> Mbeya

[ ] /cities/unknown -> 404

[ ] Case-sensitive slug contract is documented

A NUDGE, IF YOU NEED ONE

Use a dictionary keyed by slug and check membership.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

ROUTES AND URL PARAMETERS

lesson_03_solution.py

```
from flask import Flask, abort
app = Flask(__name__)
cities = {"mbeya": "Mbeya", "arusha": "Arusha"}
@app.get("/cities/<slug>")
def city(slug):
    if slug not in cities:
        abort(404)
    return cities[slug]
c = app.test_client()
assert c.get("/cities/mbeya").text == "Mbeya"
assert c.get("/cities/unknown").status_code == 404
```

WHY IT WORKS

Slugs make URLs readable, but the supported spelling is still a contract. A redirect can later support old slugs when names change.

TAKE IT FURTHER

Add a directory index listing every city and link to each detail route using url_for.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Templates and escaping

THE GOAL

Insert values into HTML without treating them as markup.

Templates separate content from page structure. Jinja expressions display values and loops repeat markup. Flask enables HTML escaping in render_template_string; keep the template itself fixed and pass user input as values. Never treat untrusted user text as a new template.

lesson_04_example.py

```
from flask import Flask, render_template_string
app = Flask(__name__)
@app.get("/")
def home():
    return render_template_string(
        "<h1>Hello, {{ name }}</h1>", name="<Asha>")
html = app.test_client().get("/").text
assert "&lt;Asha&gt;" in html
assert "<Asha>" not in html
```

Escaping preserves the text while preventing it from becoming an HTML element. This is different from deleting angle brackets or trusting every saved database value.

WATCH FOR THIS

The safe filter disables escaping. Use it only for content whose safety is established by a deliberate sanitisation policy.

Your turn: Templates and escaping

THE CHALLENGE

Render a list of lesson names using a template loop. Include a name containing an ampersand and verify it is escaped.

ACCEPTANCE CHECKS

[ ] Two inputs -> two li elements

[ ] "AI & ML" appears as AI & ML

[ ] Empty list -> an explicit empty message

A NUDGE, IF YOU NEED ONE

Jinja loops can use an else block when the sequence is empty.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

TEMPLATES AND ESCAPING

lesson_04_solution.py

```
from flask import Flask, render_template_string
app = Flask(__name__)
T = """<ul>{% for name in names %}<li>{{ name }}</li>
{% else %}<li>No lessons yet</li>{% endfor %}</ul>"""
with app.app_context():
    html = render_template_string(T, names=["Git", "AI & ML"])
    assert html.count("<li>") == 2
    assert "AI &amp;amp; ML" in html
    assert "No lessons yet" in render_template_string(T,
                                                      names=[])
```

WHY IT WORKS

The same template handles both populated and empty states. In a larger project, put templates into files under templates/ rather than long Python strings.

TAKE IT FURTHER

Move the template into templates/list.html and replace render_template_string with render_template.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Reusable page layouts

THE GOAL

Share navigation and document structure across pages.

Template inheritance lets a base layout own the document shell while child templates fill named blocks. This prevents pages from drifting apart when navigation changes. Keep the page title and main content as separate blocks so each page remains identifiable.

lesson_05_example.py

```
from flask import Flask, render_template
from jinja2 import DictLoader
app = Flask(__name__)
app.jinja_loader = DictLoader({
 "base.html": "<title>{% block title %}{% endblock %}</title>"
              "<main>{% block body %}{% endblock %}</main>",
 "home.html": "{% extends 'base.html' %}"
              "{% block title %}Library{% endblock %}"
              "{% block body %}<h1>Library</h1>{% endblock %}"
})
@app.get("/")
def home():
    return render_template("home.html")
assert "<title>Library</title>" in app.test_client().get("/").text
```

DictLoader keeps this example in one runnable file. In an ordinary website the same template names refer to files in the templates directory.

WATCH FOR THIS

Nested or repeated main elements can confuse document structure. Let the base own the main landmark.

Your turn: Reusable page layouts

THE CHALLENGE

Create two child pages that share a footer. Verify that changing the base footer affects both pages.

ACCEPTANCE CHECKS

[ ] Both pages contain the shared footer

[ ] Each page has its own title

[ ] Only the base defines the footer

A NUDGE, IF YOU NEED ONE

Use two child templates and render each inside an application context.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

REUSABLE PAGE LAYOUTS

lesson_05_solution.py

```
from flask import Flask, render_template
from jinja2 import DictLoader
app = Flask(__name__)
app.jinja_loader = DictLoader({
 "base": "{% block body %}{% endblock %}<footer>Priver</footer>",
 "a": "{% extends 'base' %}{% block body %}About{% endblock %}",
 "b": "{% extends 'base' %}{% block body %}Books{% endblock %}"
})
with app.app_context():
    for name in ["a", "b"]:
        assert "<footer>Priver</footer>" in render_template(name)
```

WHY IT WORKS

A shared layout reduces duplication without forcing every page to look identical. Add blocks where a real variation exists, not for every line of HTML.

TAKE IT FURTHER

Add an active navigation state based on the current endpoint and test both child pages.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

CSS and responsive layout

THE GOAL

Make the page comfortable on narrow and wide screens.

CSS changes presentation without changing the server data. A flexible grid can show several cards on a wide screen and one on a narrow screen. Avoid fixed widths that force horizontal scrolling. Set readable line heights, visible focus indicators and sufficient contrast.

lesson_06_example.py

```
from flask import Flask
app = Flask(__name__)
@app.get("/")
def home():
    return """<meta name="viewport" content="width=device-width">
    <style>main{max-width:60rem;margin:auto;padding:1rem}
    .grid{display:grid;gap:1rem;
    grid-template-columns:repeat(auto-fit,minmax(15rem,1fr))}
    a:focus-visible{outline:3px solid navy}</style>
    <main><h1>Books</h1><div class="grid">
    <article>Git</article><article>Python</article></div></main>"""
assert b"viewport" in app.test_client().get("/").data
```

The viewport setting lets mobile browsers use the device width. The example is a layout demonstration; real cards also need headings and meaningful actions.

WATCH FOR THIS

A layout passing an HTML test can still be visually broken. Browser inspection is a separate check.

Your turn: CSS and responsive layout

THE CHALLENGE

Create a CSS rule set for a readable article. Limit text width, keep images inside the column and avoid code blocks stretching the whole page.

ACCEPTANCE CHECKS

[ ] Article width is capped

[ ] Images never exceed their container

[ ] Long code can scroll within its block

A NUDGE, IF YOU NEED ONE

Use max-width for the article and overflow-x:auto for pre.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

CSS AND RESPONSIVE LAYOUT

lesson_06_solution.py

```
css = """
.article { max-width: 68ch; margin-inline: auto;
           padding: 1rem; line-height: 1.8; }
.article img { max-width: 100%; height: auto; }
.article pre { overflow-x: auto; padding: 1rem; }
.article a:focus-visible { outline: 3px solid #244e9a; }
"""
assert "68ch" in css
assert "overflow-x: auto" in css
print(css)
```

WHY IT WORKS

These assertions check the declared rules, not visual quality. Inspect the result at phone and desktop widths and at browser zoom before calling it finished.

TAKE IT FURTHER

Test at 200% zoom and ensure no text or controls become unreachable.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Forms and GET requests

THE GOAL

Use query parameters for shareable search state.

A GET form places its fields in the URL query string. It suits search and filtering because the URL can be bookmarked. A GET request should not change server data. Always validate query values; users can edit the URL directly without using your form.

lesson_07_example.py

```
from flask import Flask, request, render_template_string
app = Flask(__name__)
@app.get("/")
def search():
    query = request.args.get("q", "").strip()
    return render_template_string("""
    <form><label>Search <input name="q" value="{{ q }}"></label>
    <button>Search</button></form><p>Query: {{ q }}</p>""", q=query)
c = app.test_client()
assert "Python" in c.get("/?q=Python").text
```

The current query is displayed back into the field, so the user can refine it. Template escaping also applies inside this quoted attribute.

WATCH FOR THIS

Never use GET to delete a resource: crawlers, previews or prefetching can follow a URL.

Your turn: Forms and GET requests

THE CHALLENGE

Filter a fixed list of titles using a case-insensitive query. Treat an empty query as show all.

ACCEPTANCE CHECKS

[ ] q=python -> both Python titles

[ ] q=missing -> empty list

[ ] No q -> all titles

A NUDGE, IF YOU NEED ONE

Normalise both query and title with casefold before membership testing.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

FORMS AND GET REQUESTS

lesson_07_solution.py

```
def search_titles(titles, query):
    query = query.strip().casefold()
    return [title for title in titles
            if query in title.casefold()]

titles = ["Python web", "Git", "Python AI"]
assert search_titles(titles, "PYTHON") == [
    "Python web", "Python AI"]
assert search_titles(titles, "missing") == []
assert search_titles(titles, "") == titles
```

WHY IT WORKS

Keeping filtering in a pure function makes it easy to test independently of Flask. The route becomes input extraction plus rendering.

TAKE IT FURTHER

Add a topic filter and preserve both topic and search query when building navigation links.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

POST and validation

THE GOAL

Check submitted values before treating them as valid data.

POST sends form data in the request body. Server-side validation remains necessary even when HTML has required fields. Distinguish missing values, whitespace-only values and values outside accepted ranges. For this first example, validate and echo a field without storing or changing state.

lesson_08_example.py

```
from flask import Flask, request, render_template_string
app = Flask(__name__)
@app.post("/check")
def check():
    title = request.form.get("title", "").strip()
    if not 2 <= len(title) <= 80:
        return "Title must contain 2-80 characters", 400
    return render_template_string("<p>{{ title }}</p>", title=title)
c = app.test_client()
assert c.post("/check", data={"title": "  "}).status_code == 400
assert c.post("/check", data={"title": "Git"}).status_code == 200
```

The same validation applies whether the submission came from the visible form or a manually constructed request.

WATCH FOR THIS

Browser validation can be bypassed. Do not rely on maxlength or required alone.

Your turn: POST and validation

THE CHALLENGE

Write a pure validator for a contact form with name and message. Return field-specific errors and preserve valid trimmed values.

ACCEPTANCE CHECKS

[ ] Blank name -> name error

[ ] Message under 10 characters -> message error

[ ] Valid input -> no errors

A NUDGE, IF YOU NEED ONE

Return a cleaned dictionary and a separate error dictionary.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

POST AND VALIDATION

lesson_08_solution.py

```
def validate(data):
    clean = {key: data.get(key, "").strip()
             for key in ["name", "message"]}
    errors = {}
    if not 2 <= len(clean["name"]) <= 80:
        errors["name"] = "Use 2-80 characters"
    if not 10 <= len(clean["message"]) <= 1000:
        errors["message"] = "Use 10-1000 characters"
    return clean, errors

assert "name" in validate({})[1]
assert not validate({"name": "Asha",
                     "message": "Please send the guide."})[1]
```

WHY IT WORKS

Field-level errors tell the user what to correct. This validates shape, not whether the person is genuine or whether a message is spam.

TAKE IT FURTHER

Render each error next to its input and associate the text using aria-describedby.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

CSRF and form intent

THE GOAL

Protect state-changing browser requests.

When a browser automatically sends cookies, another site may try to trigger an unwanted action. CSRF protection checks that a state-changing form came from your intended flow. A session token is one mechanism. Production applications should use a maintained CSRF extension and a full policy, not copy a demonstration blindly.

lesson_09_example.py

```
import secrets
from hmac import compare_digest

def valid_csrf(expected, supplied):
    return bool(expected and supplied) and compare_digest(
        expected, supplied)

token = secrets.token_urlsafe(32)
assert valid_csrf(token, token)
assert not valid_csrf(token, "wrong")
assert not valid_csrf("", "")
```

The function rejects missing tokens and compares non-empty values. A token must be bound to the user session and included in the form, not stored as one public constant.

WATCH FOR THIS

Passing a CSRF check does not prove the user owns the resource they want to modify.

Your turn: CSRF and form intent

THE CHALLENGE

Model a state-changing action that verifies a session token before changing a list. Make failed validation leave the list untouched.

ACCEPTANCE CHECKS

[ ] Correct token -> one item added

[ ] Wrong token -> PermissionError

[ ] Failed attempt preserves previous contents

A NUDGE, IF YOU NEED ONE

Check the token before calling append. Use a copy of state to verify failure behaviour.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

CSRF AND FORM INTENT

lesson_09_solution.py

```
from hmac import compare_digest

def add_note(notes, text, expected, supplied):
    if not expected or not supplied:
        raise PermissionError("Missing CSRF token")
    if not compare_digest(expected, supplied):
        raise PermissionError("Invalid CSRF token")
    notes.append(text)

notes = []
add_note(notes, "Learn", "demo-token", "demo-token")
try:
    add_note(notes, "Bad", "demo-token", "wrong")
except PermissionError:
    pass
assert notes == ["Learn"]
```

WHY IT WORKS

The example demonstrates ordering of checks. It is not a complete login or session implementation. Authentication, authorisation and CSRF solve different problems.

TAKE IT FURTHER

Install and configure a maintained Flask CSRF extension, then test missing, expired and valid form tokens.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Redirect after submission

THE GOAL

Avoid accidental repeat actions when a page is refreshed.

After a successful state-changing POST, a redirect can send the browser to a GET confirmation page. This is the Post/Redirect/Get pattern. It improves normal refresh behaviour but does not make an action idempotent; duplicate network submissions still need a separate policy.

lesson_10_example.py

```
from flask import Flask, redirect, url_for
app = Flask(__name__)
@app.post("/submit")
def submit():
    return redirect(url_for("thanks"), code=303)
@app.get("/thanks")
def thanks():
    return "Thank you"
c = app.test_client()
r = c.post("/submit")
assert r.status_code == 303
assert r.headers["Location"] == "/thanks"
```

This route changes no data; it isolates redirect behaviour. A real form validates and safely stores its data before redirecting.

WATCH FOR THIS

Do not redirect to arbitrary user-supplied external URLs without a destination policy.

Your turn: Redirect after submission

THE CHALLENGE

Write a test that follows the redirect and checks both the final page and the intermediate status.

ACCEPTANCE CHECKS

[ ] Initial POST -> 303

[ ] Final page -> Thank you

[ ] Exactly one redirect in history

A NUDGE, IF YOU NEED ONE

Use follow_redirects=True on the test client.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

REDIRECT AFTER SUBMISSION

lesson_10_solution.py

```
from flask import Flask, redirect
app = Flask(__name__)
@app.post("/submit")
def submit():
    return redirect("/thanks", code=303)
@app.get("/thanks")
def thanks():
    return "Thank you"
r = app.test_client().post("/submit", follow_redirects=True)
assert r.text == "Thank you"
assert len(r.history) == 1
assert r.history[0].status_code == 303
```

WHY IT WORKS

The test distinguishes navigation behaviour from the final content. A 200 final response alone would not tell you how the browser arrived there.

TAKE IT FURTHER

Add a validation failure that returns the form directly with errors and does not redirect.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Cookies and sessions

THE GOAL

Remember small pieces of state across requests.

HTTP requests are independent. Cookies let the browser send a value back later. Flask's default session stores signed data in a cookie: signing detects tampering but does not encrypt its contents. Keep secrets and large records out of it. Use a strong server-side secret key supplied through deployment configuration.

lesson_11_example.py

```
from flask import Flask, session
app = Flask(__name__)
app.config["SECRET_KEY"] = "local-demo-not-for-production"
@app.get("/")
def home():
    return session.get("theme", "light")
with app.test_client() as client:
    with client.session_transaction() as state:
        state["theme"] = "dark"
    assert client.get("/").text == "dark"
```

The test client manages the cookie exchange. The hardcoded key is only for this local example and must not become a deployed application secret.

WATCH FOR THIS

Never put passwords, API tokens or private documents into a signed-but-readable session cookie.

Your turn: Cookies and sessions

THE CHALLENGE

Write a settings helper that accepts only light or dark. Unknown values should fall back to light rather than entering the session unchanged.

ACCEPTANCE CHECKS

[ ] dark -> dark

[ ] light -> light

[ ] unexpected -> light

A NUDGE, IF YOU NEED ONE

Use a small allowlist for supported values.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

COOKIES AND SESSIONS

lesson_11_solution.py

```
def normalise_theme(value):
    return value if value in {"light", "dark"} else "light"

assert normalise_theme("dark") == "dark"
assert normalise_theme("light") == "light"
assert normalise_theme("<script>") == "light"
settings = {"theme": normalise_theme("dark")}
assert settings == {"theme": "dark"}
```

WHY IT WORKS

An allowlist gives configuration a defined shape. It does not replace template escaping when values are later displayed.

TAKE IT FURTHER

Configure Secure, HttpOnly and SameSite cookies for an HTTPS deployment and test the resulting headers.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Database-backed pages

THE GOAL

Keep persistent content separate from presentation.

A website should query stored data and render a view rather than mixing HTML into database records. SQLite is useful for small local applications. Use placeholders for values, close connections and design an upgrade path when multiple users or deployments increase concurrency.

lesson_12_example.py

```
import sqlite3
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lessons (id INTEGER, title TEXT)")
con.executemany("INSERT INTO lessons VALUES (?, ?)",
                [(1, "Git"), (2, "Python")])
rows = con.execute("SELECT id, title FROM lessons ORDER BY id")
assert list(rows) == [(1, "Git"), (2, "Python")]
con.close()
```

The database provides rows; a template decides how to present them. This in-memory database disappears when the connection closes.

WATCH FOR THIS

Formatting user input into an SQL string creates an injection risk even when the input comes from your own form.

Your turn: Database-backed pages

THE CHALLENGE

Write a search query that treats a supplied title as a value. Test a string containing an apostrophe without changing SQL syntax.

ACCEPTANCE CHECKS

[ ] Exact title lookup returns one row

[ ] Unknown title returns no row

[ ] Apostrophe in title is handled as data

A NUDGE, IF YOU NEED ONE

Use WHERE title = ? and pass a one-item tuple.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

DATABASE-BACKED PAGES

lesson_12_solution.py

```
import sqlite3
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE books (title TEXT)")
con.execute("INSERT INTO books VALUES (?)", ("Priver's notes",))
def find(title):
    return con.execute("SELECT title FROM books WHERE title = ?",
                       (title,)).fetchone()
assert find("Priver's notes") == ("Priver's notes",)
assert find("missing") is None
con.close()
```

WHY IT WORKS

Parameter binding separates SQL instructions from values. Dynamic column names need an allowlist because placeholders represent values, not identifiers.

TAKE IT FURTHER

Connect the query to a Flask detail route and render a 404 when fetchone returns None.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Pagination and navigation

THE GOAL

Show a manageable slice while preserving context.

Pagination divides a larger list into pages. Validate the requested page and keep the sort order stable. When search filters change, reset to a valid page. Offset pagination is easy to learn; cursor pagination can behave better for large or frequently changing datasets.

lesson_13_example.py

```
def page_items(values, page, size=3):
    if page < 1 or size < 1:
        raise ValueError("Positive page and size required")
    start = (page - 1) * size
    return values[start:start + size]

assert page_items(list(range(8)), 2) == [3, 4, 5]
assert page_items([], 1) == []
```

The visible page number starts at one; list indexes start at zero. The translation happens once in the start calculation.

WATCH FOR THIS

Without a stable sort and unique tie-breaker, records can move unpredictably between pages.

Your turn: Pagination and navigation

THE CHALLENGE

Return items plus has_previous and has_next flags for a fixed list. Keep page size positive.

ACCEPTANCE CHECKS

[ ] 8 items, page 1, size 3 -> next only

[ ] 8 items, page 3, size 3 -> previous only

[ ] Empty first page -> neither

A NUDGE, IF YOU NEED ONE

has_next asks whether page times size is less than total length.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

PAGINATION AND NAVIGATION

lesson_13_solution.py

```
def paginate(values, page, size=3):
    if page < 1 or size < 1:
        raise ValueError("Invalid pagination")
    start = (page - 1) * size
    return {"items": values[start:start+size],
            "previous": page > 1,
            "next": page * size < len(values)}

assert paginate(list(range(8)), 1)["next"]
assert not paginate(list(range(8)), 3)["next"]
assert paginate([], 1) == {
    "items": [], "previous": False, "next": False}
```

WHY IT WORKS

This pure helper does not decide whether an out-of-range page should be empty or 404; the route must choose and document that behaviour.

TAKE IT FURTHER

Build next and previous URLs with the current search query preserved.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Accessible interaction

THE GOAL

Make controls understandable without a mouse.

Accessibility is part of the structure, not a final decoration. Inputs need labels, buttons need meaningful names and focus must remain visible. Error messages should explain how to recover. Colour alone should not carry the only signal. Test keyboard order and browser zoom in addition to automated checks.

lesson_14_example.py

```
html = """<form>
<label for="query">Search lessons</label>
<input id="query" name="q" aria-describedby="hint">
<p id="hint">Search by title or topic.</p>
<button type="submit">Find lessons</button>
</form>"""
assert 'for="query"' in html
assert 'id="query"' in html
assert 'aria-describedby="hint"' in html
```

The label and input are connected by the identifier. The hint provides extra guidance without replacing the label.

WATCH FOR THIS

A placeholder disappears as the user types and is not a replacement for a persistent label.

Your turn: Accessible interaction

THE CHALLENGE

Design an error message for a missing title. Include a visible explanation and mark the input invalid without removing its label.

ACCEPTANCE CHECKS

[ ] Label remains visible

[ ] aria-invalid=true when invalid

[ ] Error text is connected to the input

A NUDGE, IF YOU NEED ONE

Use aria-describedby to reference an error element with a stable id.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

ACCESSIBLE INTERACTION

lesson_14_solution.py

```
html = """<label for="title">Lesson title</label>
<input id="title" name="title" aria-invalid="true"
       aria-describedby="title-error">
<p id="title-error">Enter a title with 2-80 characters.</p>"""
assert 'aria-invalid="true"' in html
assert 'aria-describedby="title-error"' in html
assert 'id="title-error"' in html
```

WHY IT WORKS

These checks confirm attributes, not the entire assistive-technology experience. Read the form with a screen reader when possible and test focus after submission.

TAKE IT FURTHER

Add an error summary with links that move focus to the relevant invalid inputs.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Static files and caching

THE GOAL

Serve assets with predictable names and sensible cache rules.

CSS, images and JavaScript are static assets. Flask can serve a static directory during development; production may place assets behind a dedicated server or CDN. Long-lived caching works best with versioned or content-hashed filenames. Personalised pages need more careful cache decisions.

lesson_15_example.py

```
from flask import Flask, url_for
app = Flask(__name__)
with app.test_request_context():
    assert url_for("static", filename="site.css") == (
        "/static/site.css")
    assert url_for("static", filename="covers/python.webp") == (
        "/static/covers/python.webp")
```

url_for builds paths from Flask configuration rather than scattering hardcoded prefixes through templates. The example does not create the asset files.

WATCH FOR THIS

Do not publicly cache an authenticated user's personal page under a shared cache key.

Your turn: Static files and caching

THE CHALLENGE

Create a route returning a public text resource with a short cache lifetime. Verify the response header.

ACCEPTANCE CHECKS

[ ] GET /notice.txt -> 200

[ ] Content-Type is text/plain

[ ] Cache-Control contains max-age=60

A NUDGE, IF YOU NEED ONE

Use make_response and set headers explicitly.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

STATIC FILES AND CACHING

lesson_15_solution.py

```
from flask import Flask, make_response
app = Flask(__name__)
@app.get("/notice.txt")
def notice():
    response = make_response("New guide available")
    response.headers["Content-Type"] = "text/plain; charset=utf-8"
    response.headers["Cache-Control"] = "public, max-age=60"
    return response
r = app.test_client().get("/notice.txt")
assert r.status_code == 200
assert "max-age=60" in r.headers["Cache-Control"]
```

WHY IT WORKS

The header allows public caching for sixty seconds. It is appropriate only because every reader receives the same non-private content.

TAKE IT FURTHER

Add a versioned CSS filename and explain how a new release changes the URL to bypass old cached content.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Small JavaScript enhancements

THE GOAL

Keep a useful HTML experience when scripts fail.

Progressive enhancement starts with working links and forms, then adds convenience in JavaScript. A client-side filter can reduce visible items, but the underlying list should still be readable without it. Keep user text out of innerHTML unless sanitisation is deliberately handled.

lesson_16_example.py

```
script = """const input = document.querySelector('#filter');
const cards = document.querySelectorAll('[data-title]');
input.addEventListener('input', () => {
  const query = input.value.trim().toLowerCase();
  for (const card of cards) {
    card.hidden = !card.dataset.title.includes(query);
  }
});"""
assert "card.hidden" in script
print(script)
```

This Python snippet stores JavaScript for discussion; it is not a browser execution test. Cards need normalised data-title values and a labelled filter input.

WATCH FOR THIS

A JavaScript failure should not hide all content before the script has successfully initialised.

Your turn: Small JavaScript enhancements

THE CHALLENGE

Write a Python helper that prepares a safe searchable label from a title and tags. Test mixed case and extra spaces.

ACCEPTANCE CHECKS

[ ] Title and tags are both searchable

[ ] Output is lowercase and trimmed

[ ] Empty tags still work

A NUDGE, IF YOU NEED ONE

Join text values, then casefold and collapse whitespace.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

SMALL JAVASCRIPT ENHANCEMENTS

lesson_16_solution.py

```
def search_label(title, tags):
    return " ".join((title + " " + " ".join(tags)).split()).casefold()

assert search_label(" Python  Web ", ["Flask"]) == (
    "python web flask")
assert search_label("Git", []) == "git"
```

WHY IT WORKS

Pass the label through normal template escaping when rendering it into an attribute. Normalisation is not HTML escaping.

TAKE IT FURTHER

Add a visible no-results message and announce the result count with an aria-live region.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Search-friendly pages

THE GOAL

Give each real page a clear identity.

Search engines and link previews benefit from descriptive titles, useful summaries, canonical URLs and real visible content. These are communication tools, not a guarantee of ranking. Use an absolute canonical URL for the preferred address. Keep private management pages out of public navigation and mark them appropriately.

lesson_17_example.py

```
from html import escape

def metadata(title, description, canonical):
    return (f"<title>{escape(title)}</title>"
            f'<meta name="description" content="{escape(description)}">'
            f'<link rel="canonical" href="{escape(canonical)}">')
html = metadata("Python guide", "Learn by building.",
                "https://example.org/guides/python")
assert "canonical" in html
```

This helper illustrates metadata shape. In a Flask application, a fixed template is usually clearer than building HTML fragments manually.

WATCH FOR THIS

Duplicated keyword-heavy titles can make pages less useful to readers. Write for the actual page purpose.

Your turn: Search-friendly pages

THE CHALLENGE

Create metadata for two distinct pages and verify each has a unique title and canonical URL. Use a controlled site base URL.

ACCEPTANCE CHECKS

[ ] Home and guide titles differ

[ ] Canonical URLs are absolute

[ ] Description contains no fabricated claims

A NUDGE, IF YOU NEED ONE

Keep canonical construction based on known route paths, not an arbitrary Host header.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

SEARCH-FRIENDLY PAGES

lesson_17_solution.py

```
SITE = "https://example.org"
pages = [
    {"path": "/", "title": "Learning library"},
    {"path": "/python", "title": "Python practical guide"},
]
for page in pages:
    page["canonical"] = SITE + page["path"]
assert len({p["title"] for p in pages}) == 2
assert all(p["canonical"].startswith(SITE) for p in pages)
```

WHY IT WORKS

The controlled base prevents a request-specific host from being reflected into canonical metadata. Page content still needs to deserve the description.

TAKE IT FURTHER

Add a sitemap containing only public canonical pages and verify every URL resolves.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Error pages that help

THE GOAL

Explain failures while preserving correct status codes.

A useful error page tells the reader what happened and how to continue. A missing page should still return 404, even if the HTML looks polished. Unexpected failures should be logged for maintainers without exposing secrets or stack traces to visitors.

lesson_18_example.py

```
from flask import Flask
app = Flask(__name__)
@app.errorhandler(404)
def missing(error):
    return '<h1>Page not found</h1><a href="/">Go home</a>', 404
r = app.test_client().get("/absent")
assert r.status_code == 404
assert b"Go home" in r.data
```

The handler changes presentation but preserves the error status. The error argument is supplied by Flask and does not need to be shown to the user.

WATCH FOR THIS

Debug mode can expose sensitive internals. Never expose the development debugger publicly.

Your turn: Error pages that help

THE CHALLENGE

Create a 400 validation response and a 404 missing-resource response. Keep their messages distinct and test their status codes.

ACCEPTANCE CHECKS

[ ] Invalid request -> 400 and correction guidance

[ ] Unknown resource -> 404 and navigation guidance

[ ] Neither response includes a traceback

A NUDGE, IF YOU NEED ONE

Test the response status and body together.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

ERROR PAGES THAT HELP

lesson_18_solution.py

```
from flask import Flask, abort
app = Flask(__name__)
@app.get("/invalid")
def invalid():
    return "Enter a valid page number", 400
@app.get("/missing")
def missing():
    abort(404)
c = app.test_client()
assert c.get("/invalid").status_code == 400
assert c.get("/missing").status_code == 404
assert "Traceback" not in c.get("/missing").text
```

WHY IT WORKS

Different failures need different recovery actions. A user should not be told to retry indefinitely when the requested resource does not exist.

TAKE IT FURTHER

Design a maintenance page with an appropriate temporary-unavailability status and a useful retry message.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Testing and release checks

THE GOAL

Verify behaviour before exposing the site to readers.

Use automated checks for routes, validation and escaping, then manual browser checks for layout and interaction. A production release needs an appropriate WSGI server, HTTPS, configuration secrets and a rollback plan. Debug settings and development data must not leak into the deployed environment.

lesson_19_example.py

```
from flask import Flask

def create_app(testing=False):
    app = Flask(__name__)
    app.config["TESTING"] = testing
    @app.get("/health")
    def health():
        return {"status": "ok"}
    return app

app = create_app(True)
assert app.test_client().get("/health").json == {"status": "ok"}
```

An application factory creates isolated app instances. It makes testing easier and avoids hidden global configuration shared between tests.

WATCH FOR THIS

A health route saying ok does not prove database writes, backups or external integrations work.

Your turn: Testing and release checks

THE CHALLENGE

Create two application instances with different labels and prove their responses do not leak into one another.

ACCEPTANCE CHECKS

[ ] First app returns first label

[ ] Second app returns second label

[ ] Both route statuses are 200

A NUDGE, IF YOU NEED ONE

Capture the label inside the factory or store it in app.config.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

TESTING AND RELEASE CHECKS

lesson_19_solution.py

```
from flask import Flask

def create_app(label):
    app = Flask(__name__)
    @app.get("/")
    def home():
        return label
    return app

first = create_app("First")
second = create_app("Second")
assert first.test_client().get("/").text == "First"
assert second.test_client().get("/").text == "Second"
```

WHY IT WORKS

Isolation prevents tests from passing only because a previous test left useful state behind. Deployment also needs explicit configuration rather than mutable import-time assumptions.

TAKE IT FURTHER

Write a release checklist covering mobile layout, keyboard use, broken links, errors, HTTPS and recovery.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Project: a learning library

THE GOAL

Combine routing, search, templates and a detail page.

The final project is a small read-only website. It deliberately has no login or writes, so you can focus on navigation and presentation. Store resources as structured data, use GET for search and render titles through a template. A missing identifier must produce a real 404.

lesson_20_example.py

```
from flask import Flask, request, render_template_string
app = Flask(__name__)
BOOKS = {1: "Python Web", 2: "Git Notebook"}
@app.get("/")
def home():
    q = request.args.get("q", "").casefold()
    rows = [(i, t) for i, t in BOOKS.items() if q in t.casefold()]
    return render_template_string("""<h1>Library</h1>
    {% for i,t in rows %}<p><a href="/books/{{ i }}">{{ t }}</a></p>
    {% else %}<p>No matching books</p>{% endfor %}""", rows=rows)
c = app.test_client()
assert "Python Web" in c.get("/?q=python").text
assert "Git Notebook" not in c.get("/?q=python").text
```

The example implements the index. The practice adds the missing detail route and closes the navigation loop. Keep the project read-only until you have a complete authentication and write-protection design.

WATCH FOR THIS

A nice card that links to a missing detail page is an incomplete user journey. Test the destination, not only the card markup.

Your turn: Project: a learning library

THE CHALLENGE

Add the detail route, return 404 for unknown IDs and include a link back to the library. Test a valid ID, invalid ID and non-matching search.

ACCEPTANCE CHECKS

[ ] /books/1 -> title and back link

[ ] /books/99 -> 404

[ ] /?q=absent -> empty-state message

A NUDGE, IF YOU NEED ONE

Use abort(404) and a fixed template with a title variable.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

PROJECT: A LEARNING LIBRARY

lesson_20_solution.py

```
from flask import Flask, abort, render_template_string
app = Flask(__name__)
BOOKS = {1: "Python Web", 2: "Git Notebook"}
@app.get("/")
def home():
    return "<h1>Library</h1>"
@app.get("/books/<int:ident>")
def detail(ident):
    if ident not in BOOKS:
        abort(404)
    return render_template_string("""<h1>{{ title }}</h1>
    <a href="/">Back to library</a>""", title=BOOKS[ident])
c = app.test_client()
assert "Python Web" in c.get("/books/1").text
assert "Back to library" in c.get("/books/1").text
assert c.get("/books/99").status_code == 404
```

WHY IT WORKS

This standalone solution isolates the new route. Merge it with the index from the worked example, then run all acceptance checks against the combined app. Add the shared layout and responsive styles from earlier lessons.

TAKE IT FURTHER

Finish the portfolio version with a search form, shared layout, cover images, PDF links, accessible empty state and a README explaining how to run it.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Sources & next steps

Use these official references when you need the full API contract or a deeper explanation. Documentation can evolve; check the version used by your own environment.

Flask quickstart

https://flask.palletsprojects.com/en/stable/quickstart/

Flask tutorial

https://flask.palletsprojects.com/en/stable/tutorial/

Testing websites

https://flask.palletsprojects.com/en/stable/testing/

Template documentation

https://jinja.palletsprojects.com/en/stable/templates/

Security considerations

https://flask.palletsprojects.com/en/stable/web-security/

Production deployment

https://flask.palletsprojects.com/en/stable/deploying/

WHAT THE CHECKS ESTABLISH

All 40 companion scripts were executed independently during preparation. Their assertions check the small fixtures shown here. They do not establish production readiness, security under attack, browser accessibility or performance at scale.

YOUR NEXT PROJECT

Choose a small problem you can explain. Write its inputs, outputs and success checks. Build a minimal version, test one failure path, and ask another learner to follow your README. Record what you would improve next.

Keep your notes. Keep experimenting.