សាកលវិទ្យាល័យនរតុន
NORTON UNIVERSITY

Knowledge Representation with JSON

Expert System

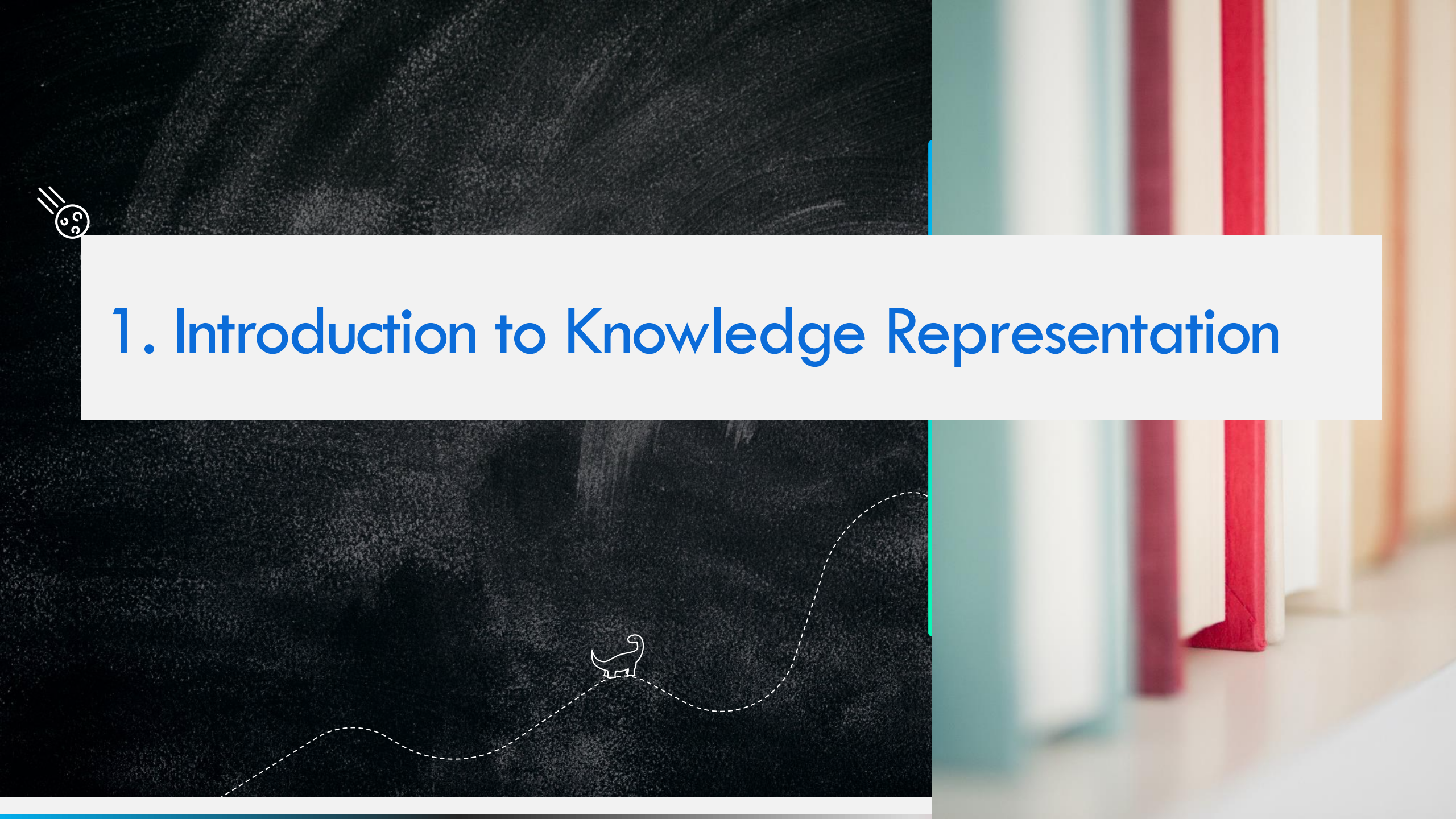
2025 – 2026
Y3 – DCS – NU

By: SEK SOCHEAT
Advisor to DCS and Lecturer
Mobile: 017 879 967
Email: socheatsek@norton-u.com
socheat.sek@gmail.com

Table of Contents:

Objectives:

1. Represent facts & rules in JSON
2. Validate with JSON Schema
3. Query & infer via Flask API



1. Introduction to Knowledge Representation

1. Introduction to Knowledge Representation

What is the Knowledge Representation?

- Knowledge representation (KR) encodes domain knowledge in a machine-readable structure.
- We'll model **facts** (observations) and **rules** (IF–THEN) in JSON, validate the structure with schemas, then serve and reason via Flask.
- The demo domain is **Symptom Diagnosis**.

1. Introduction to Knowledge Representation

Why Knowledge Representation?

- **KR** bridges experts ↔ machines
- Enables reasoning & explanation
- Reusable, auditable, testable

1. Introduction to Knowledge Representation

Facts vs. Rules

- **Facts:** known truths (e.g., “fever”)
- **Rules:** IF conditions THEN conclusion (+ certainty)

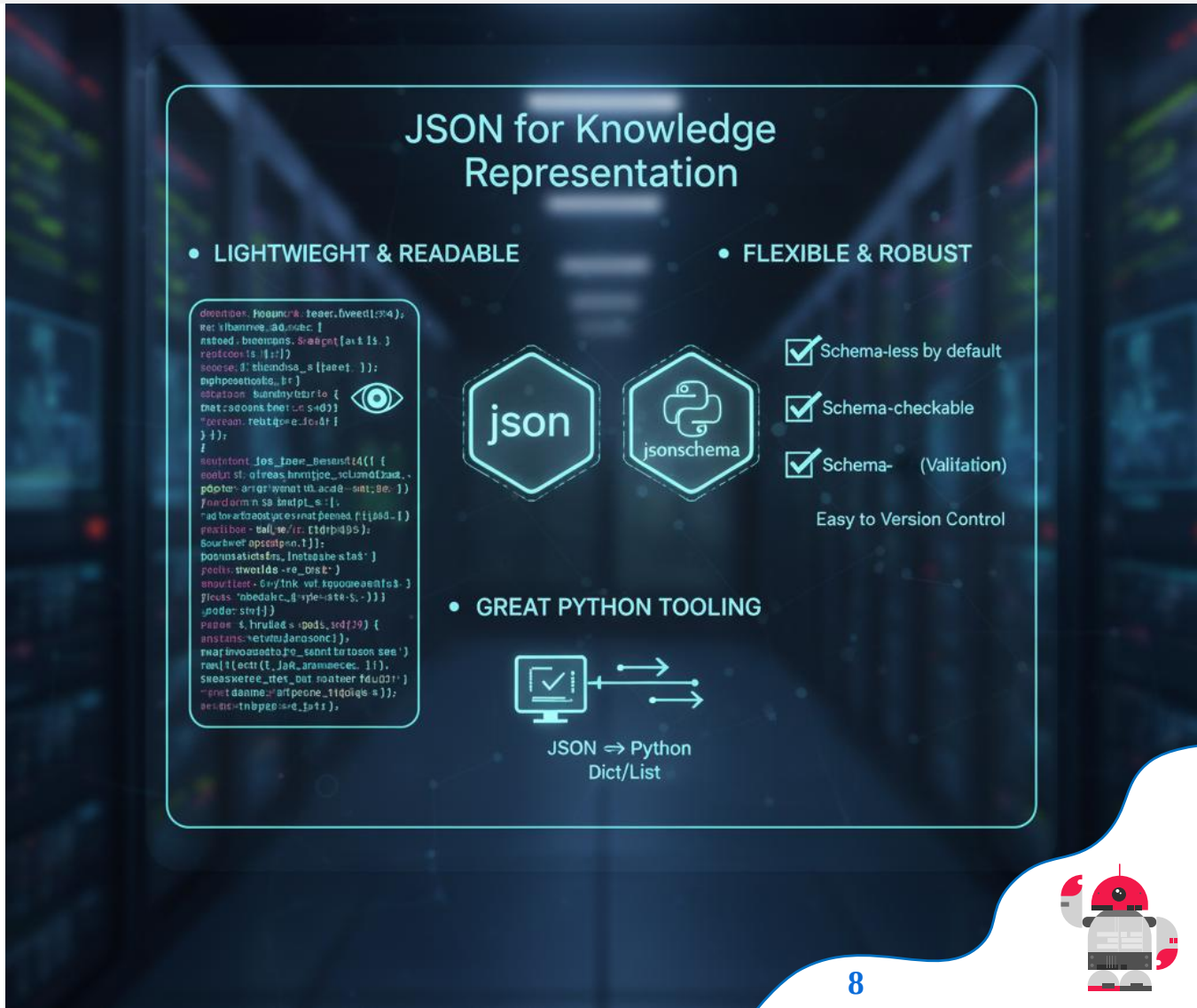
Facts reference **observable** or **asserted** states. **Rules** link facts to derived conclusions. We’ll use **IDs** to connect rules to facts.

1. Introduction to Knowledge Representation

JSON as KR Medium

- Lightweight text
- Human & machine readable
- Great Python tooling (json, jsonschema)

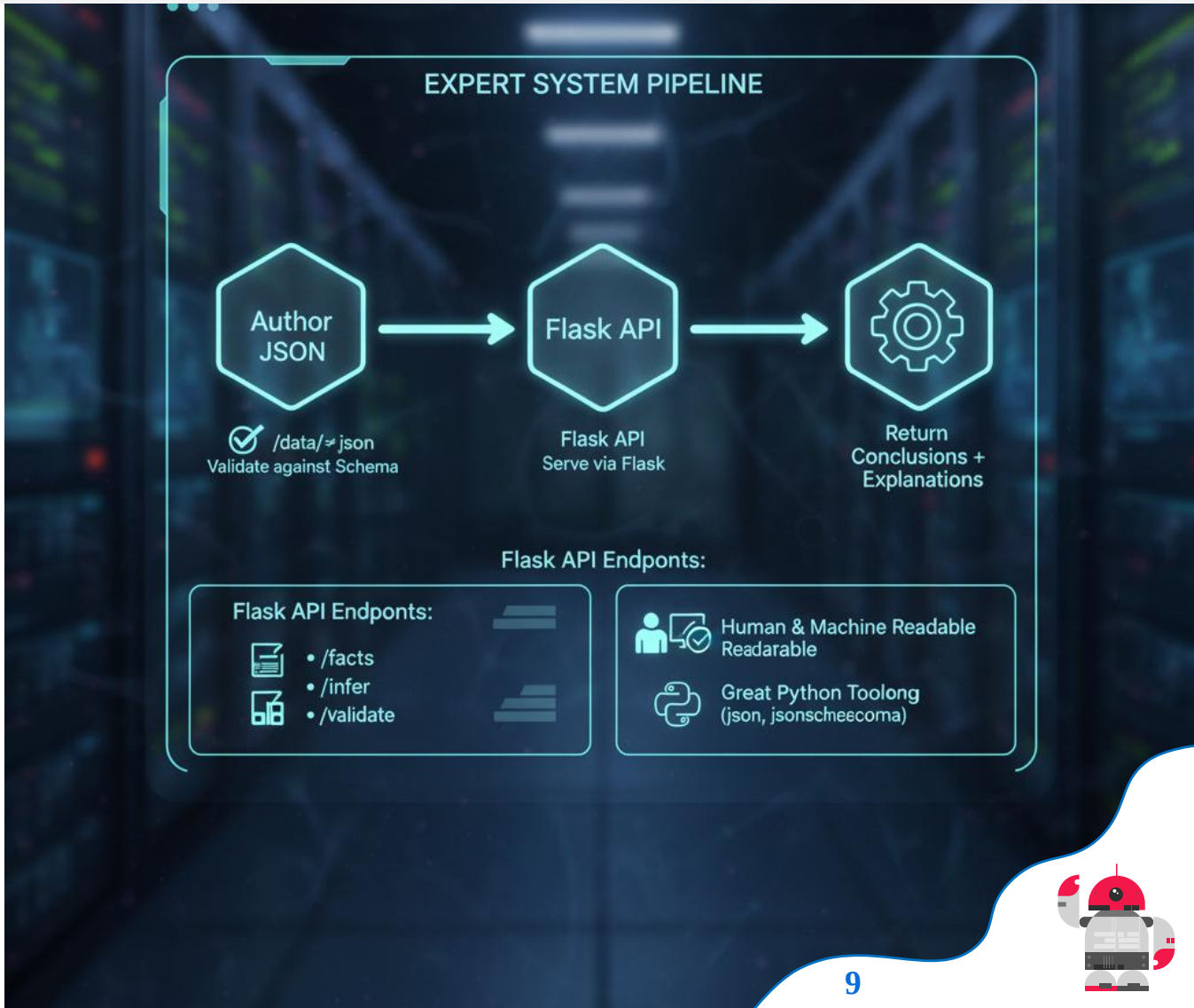
JSON fits KR because it's schema-less by default but schema-checkable; easy to version control; mapable to Python dict/list.

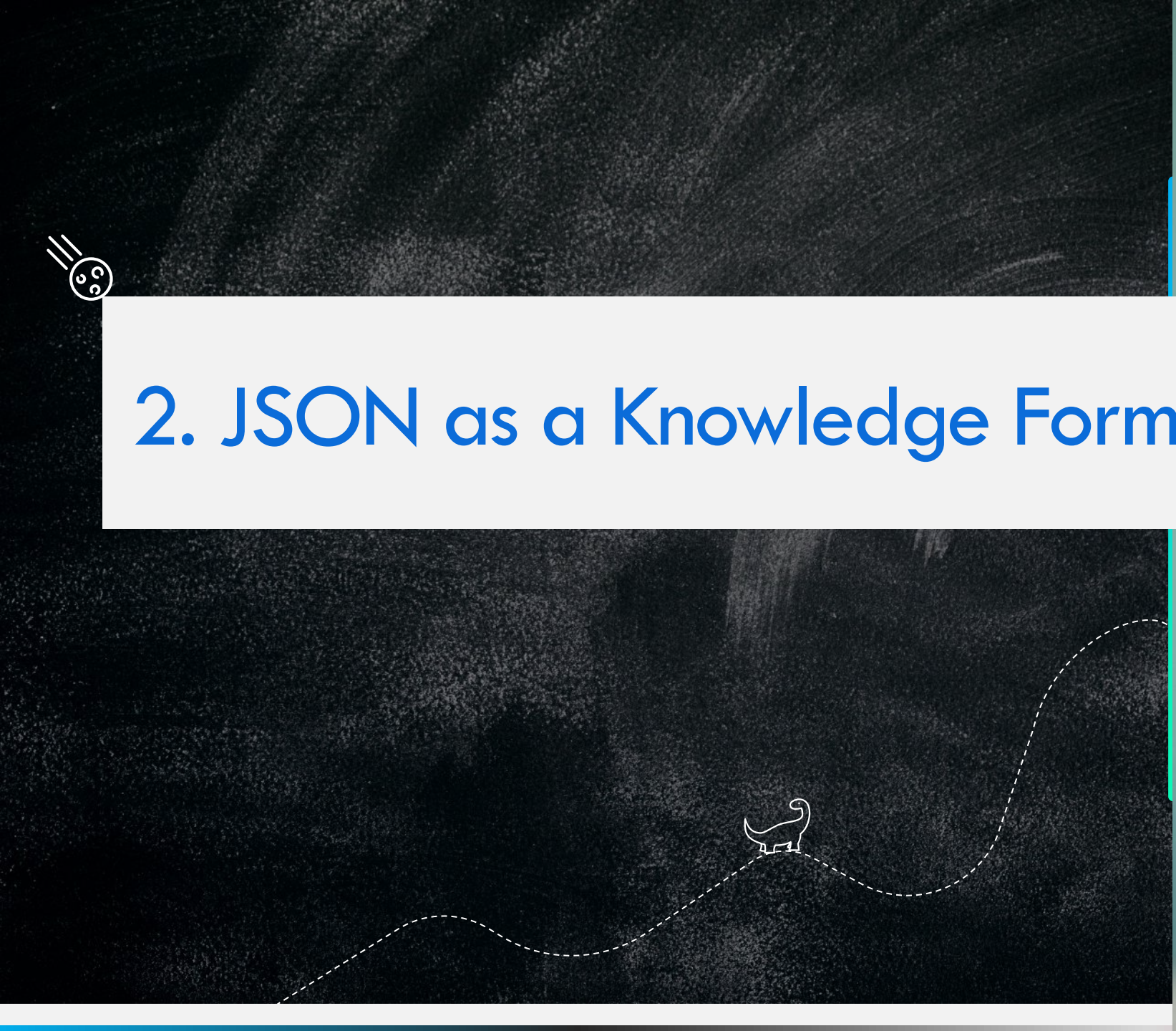


1. Introduction to Knowledge Representation

Architecture Overview

- */data/*.*json → Validation → Flask API → Inference
- ***Endpoints:*** /facts, /rules, /infer, /validate





2. JSON as a Knowledge Format

2. JSON as a Knowledge Format

JSON Basics

- Objects {}, Arrays [], Primitives
- Strings, numbers, booleans, null

We'll avoid denormalized structures; use IDs for references; keep fields predictable.

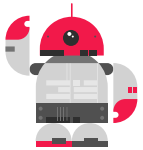
☑ **Valid JSON**

- No trailing commas (every list or object must end cleanly).
- No comments allowed (// or /* ... */ not permitted).
- Keys and string values must be in double quotes.
- Only these value types: string, number, boolean, null, object, array.

✗ **Invalid JSON examples**

{ "a": 1, } ← ✗ trailing comma

{ "a": 1 // comment } ← ✗ comment not allowed



2. JSON as a Knowledge Format

Fact Record Design

- ***Fields:*** id, description, value, tags?
- ***Minimal viable fact:*** id, description, value (boolean). Optionally tags for taxonomy linking.

☑ **Valid JSON:**

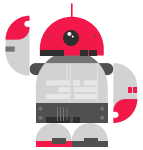
{ "stressStableIDs": true }

❌ **Invalid JSON examples:**

{ stressStableIDs: true } // ❌ keys must be in double quotes

{ "stressStableIDs": true, } // ❌ trailing comma

{ "stressStableIDs": True } // ❌ `true` must be lowercase



2. JSON as a Knowledge Format

Example facts.json

```json
[
  {
    "id": "f1",
    "description": "Patient has fever",
    "value": true,
    "tags": [
      "symptom"
    ]
  },
  {
    "id": "f2",
    "description": "Patient has cough",
    "value": true,
    "tags": [
      "symptom"
    ]
  },
  {
    "id": "f3",
    "description": "Patient has headache",
    "value": false,
    "tags": [
      "symptom"
    ]
  },
  {
    "id": "f4",
    "description": "Patient reports fatigue",
    "value": true,
    "tags": [
      "symptom"
    ]
  },
  {
    "id": "f5",
    "description": "Patient has sore throat",
    "value": false,
    "tags": [
      "symptom"
    ]
  }
]
```

2. JSON as a Knowledge Format

Rule Record Design

• ***Fields:*** id, conditions, conclusion, certainty, explain

• ***conditions:*** list of fact IDs that must all be true.

certainty in [0..1]. explain helps UI/trace.

That describes a rule inference confidence calculation often used in expert systems or fuzzy rule engines:

Rule confidence = (minimum of matched condition confidences) × (rule certainty factor)

Explanation:

- Each condition (premise) in the rule has a match confidence — how strongly the data satisfies it.
- The minimum (the weakest link) represents the overall truth of the if-part (using fuzzy logic's AND = min operator).
- Multiply that minimum by the rule's intrinsic certainty factor (CF) to scale the conclusion strength.

2. JSON as a Knowledge Format

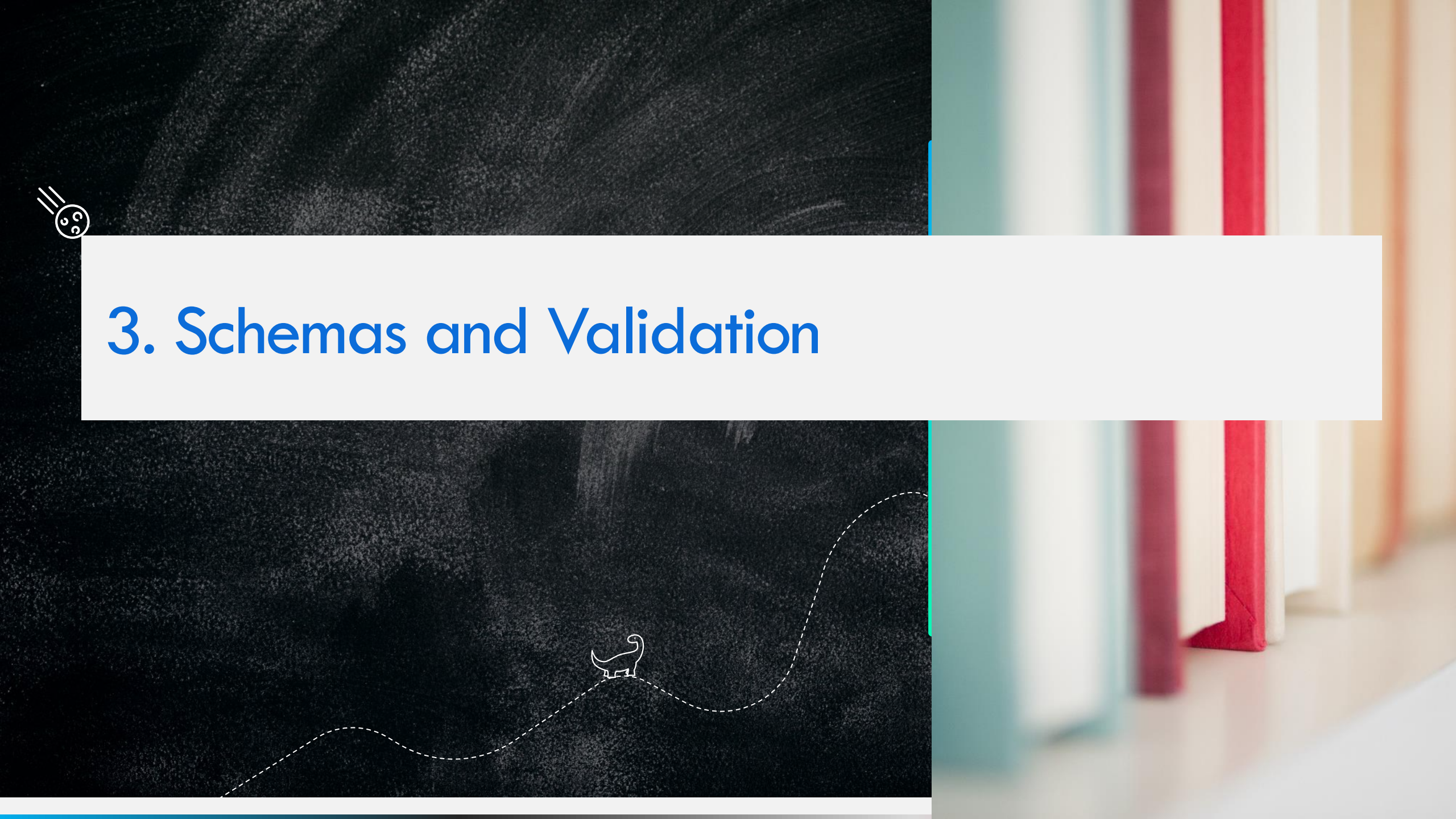
Example rules.json

```json
[
    {
        "id": "r1",
        "conditions": [
            "f1",
            "f2"
        ],
        "conclusion": "flu",
        "certainty": 0.8,
        "explain": "Fever and cough suggest influenza."
    },
    {
        "id": "r2",
        "conditions": [
            "f2",
            "f4"
        ],
        "conclusion": "common_cold",
        "certainty": 0.6,
        "explain": "Cough with fatigue often indicates a cold."
    },
    {
        "id": "r3",
        "conditions": [
            "f3",
            "f5"
        ],
        "conclusion": "migraine_possible",
        "certainty": 0.5,
        "explain": "Headache with sore throat may indicate migraine or related conditions."
    }
]
```

2. JSON as a Knowledge Format

Taxonomy (Optional)

```
{
    "parent": {
        "f1": "high_temperature",
        "f2": "respiratory_symptom",
        "f4": "general_malaise",
        "high_temperature": "vital_abnormality",
        "respiratory_symptom": "respiratory_issue"
    }
}
```



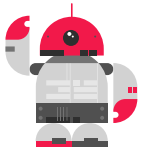
3. Schemas and Validation

3. Schemas and Validation

Why JSON Schema?

- Enforce structure
- Catch errors early
- Better debugging

Schemas declare required fields, types, and bounds; validation ensures engine safety.



3. Schemas and Validation

Fact Schema

```python
# schemas.py
# Centralized JSON Schema definitions for the expert system

facts_array_schema = {
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "type": "array",
    "items": {
        "type": "object",
        "additionalProperties": False,
        "required": ["id", "description", "value"],
        "properties": {
            "id": {"type": "string", "minLength": 1},
            "description": {"type": "string"},
            "value": {"type": "boolean"},
            "tags": {"type": "array", "items": {"type": "string"}}
        }
    }
}
```

3. Schemas and Validation

Rule Schema

```
19
20  rules_array_schema = {
21      "$schema": "https://json-schema.org/draft/2020-12/schema",
22      "type": "array",
23      "items": {
24          "type": "object",
25          "additionalProperties": False,
26          "required": ["id", "conditions", "conclusion"],
27          "properties": {
28              "id": {"type": "string", "minLength": 1},
29              "conditions": {
30                  "type": "array",
31                  "minItems": 1,
32                  "items": {"type": "string", "minLength": 1}
33              },
34              "conclusion": {"type": "string", "minLength": 1},
35              "certainty": {"type": "number", "minimum": 0.0, "maximum": 1.0},
36              "explain": {"type": "string"}
37          }
38      }
39  }
```

3. Schemas and Validation

Array Schemas and Validation Demo

- ***Wrap for lists:*** facts.json, rules.json

facts_array_schema = {"type": "array", "items": fact_schema}

rules_array_schema = {"type": "array", "items": rule_

- ***Demo:*** *Validate valid vs invalid JSON*

from jsonschema import validate, ValidationError

validate(facts, facts_array_schema)

validate(rules, rules_array_schema)

schema}

4. Designing Facts & Rules

4. Designing Facts & Rules

schemas/schemas.py

schemas > schemas.py > ...

```python
# schemas.py
# Centralized JSON Schema definitions for the expert system

facts_array_schema = {
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "type": "array",
    "items": {
        "type": "object",
        "additionalProperties": False,
        "required": ["id", "description", "value"],
        "properties": {
            "id": {"type": "string", "minLength": 1},
            "description": {"type": "string"},
            "value": {"type": "boolean"},
            "tags": {"type": "array", "items": {"type": "string"}}
        }
    }
}

rules_array_schema = {
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "type": "array",
    "items": {
        "type": "object",
        "additionalProperties": False,
        "required": ["id", "conditions", "conclusion"],
        "properties": {
            "id": {"type": "string", "minLength": 1},
            "conditions": {
                "type": "array",
                "minItems": 1,
                "items": {"type": "string", "minLength": 1}
            },
            "conclusion": {"type": "string", "minLength": 1},
            "certainty": {"type": "number", "minimum": 0.0, "maximum": 1.0},
            "explain": {"type": "string"}
        }
    }
}
```

4. Designing Facts & Rules

data/facts.json

data > {} facts.json > ...

```
[
    {
        "id": "f1",
        "description": "Patient has fever",
        "value": true,
        "tags": [
            "symptom"
        ]
    },
    {
        "id": "f2",
        "description": "Patient has cough",
        "value": true,
        "tags": [
            "symptom"
        ]
    },
    {
        "id": "f3",
        "description": "Patient has headache",
        "value": false,
        "tags": [
            "symptom"
        ]
    },
    {
        "id": "f4",
        "description": "Patient reports fatigue",
        "value": true,
        "tags": [
            "symptom"
        ]
    },
    {
        "id": "f5",
        "description": "Patient has sore throat",
        "value": false,
        "tags": [
            "symptom"
        ]
    }
]
```

4. Designing Facts & Rules

data/rules.json

data > {} rules.json > ...

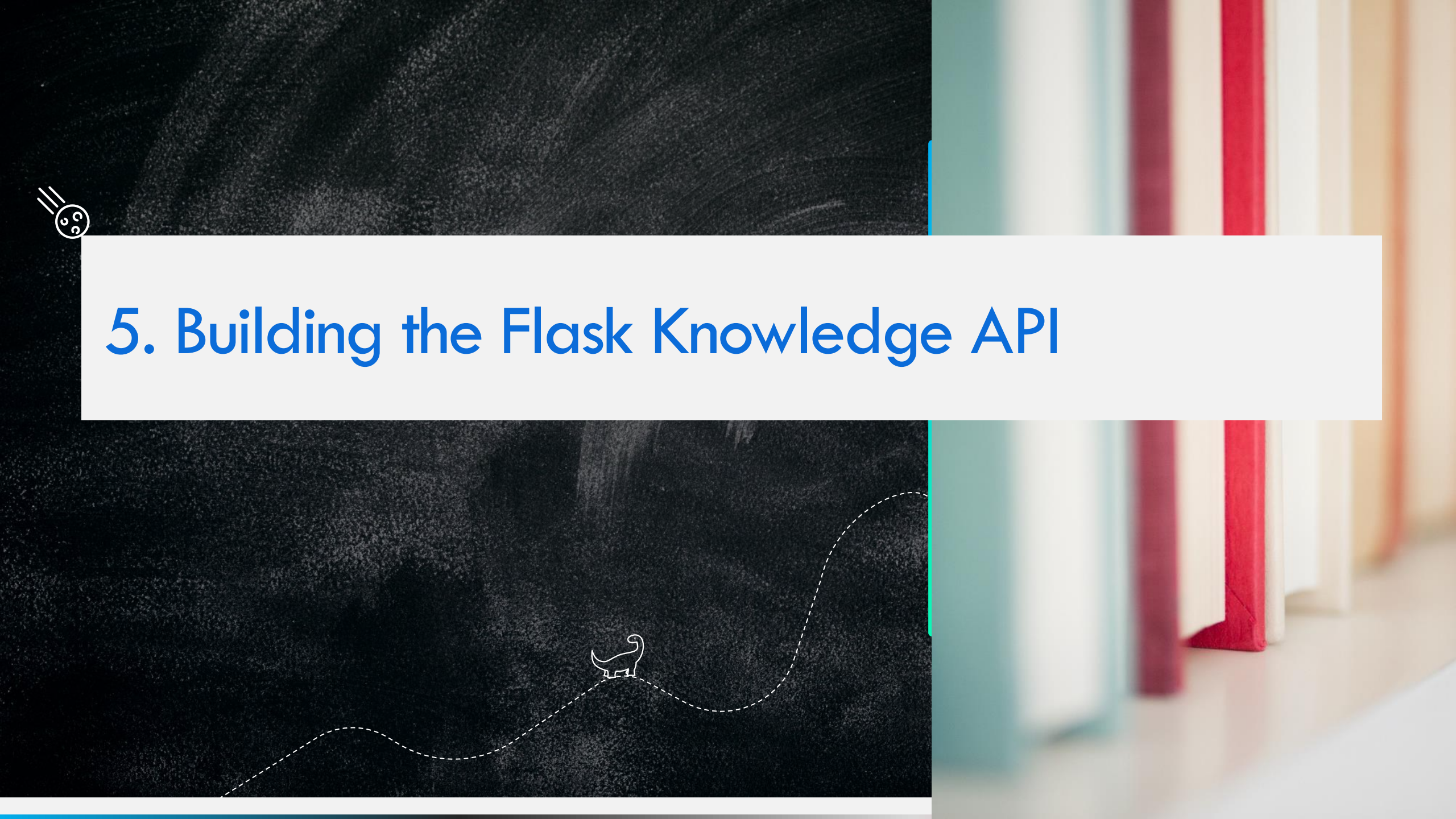
```json
[
    {
        "id": "r1",
        "conditions": [
            "f1",
            "f2"
        ],
        "conclusion": "flu",
        "certainty": 0.8,
        "explain": "Fever and cough suggest influenza."
    },
    {
        "id": "r2",
        "conditions": [
            "f2",
            "f4"
        ],
        "conclusion": "common_cold",
        "certainty": 0.6,
        "explain": "Cough with fatigue often indicates a cold."
    },
    {
        "id": "r3",
        "conditions": [
            "f3",
            "f5"
        ],
        "conclusion": "migraine_possible",
        "certainty": 0.5,
        "explain": "Headache with sore throat may indicate migraine or related conditions."
    }
]
```

4. Designing Facts & Rules

(Optional) data/taxonomy.json

data > {} taxonomy.json > ...

```
{
    "parent": {
        "f1": "high_temperature",
        "f2": "respiratory_symptom",
        "f4": "general_malaise",
        "high_temperature": "vital_abnormality",
        "respiratory_symptom": "respiratory_issue"
    }
}
```



5. Building the Flask Knowledge API

5. Building the Flask Knowledge API

Directory Structure and installing VENV

A. Installing VENV (Windows OS)

1. Create VENV via PowerShell in VS Code Terminal

   python -m venv .venv

2. Activate .venv

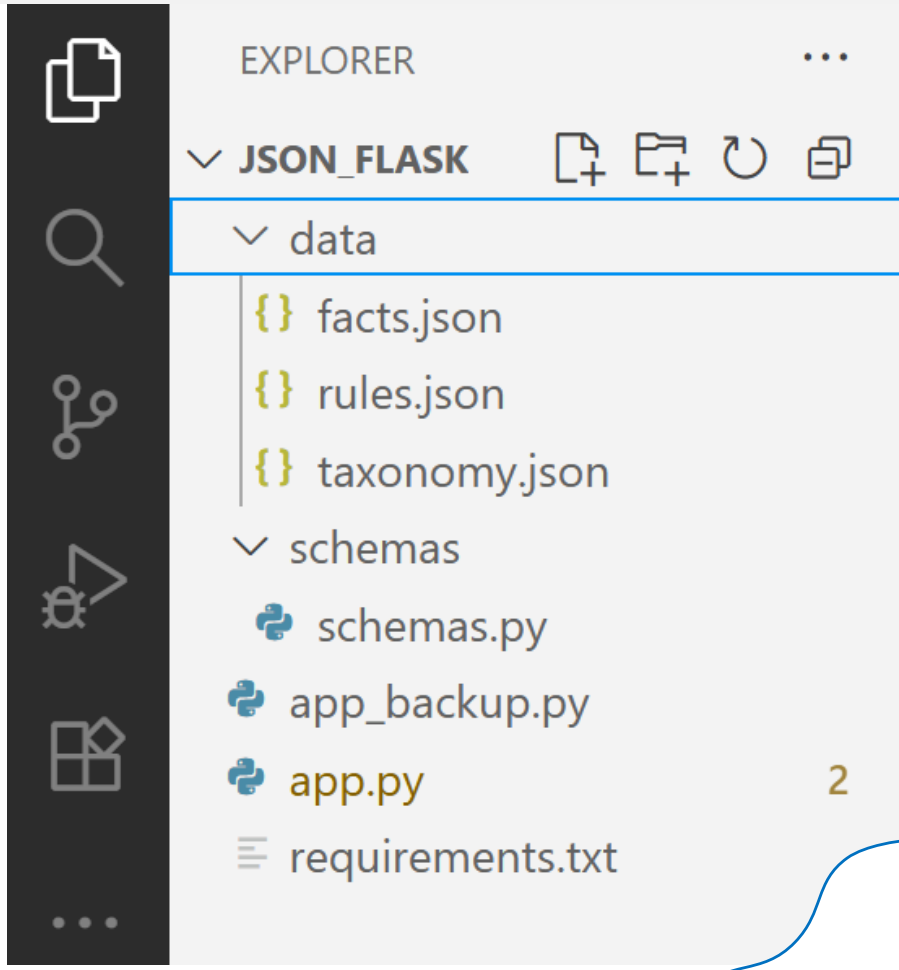
   .venv\scripts\activate.ps1

3. Upgrade pip to the latest version

   python -m pip install --upgrade pip

4. Install Flask Module

   pip install Flask



5. Building the Flask Knowledge API

Directory Structure and installing VENV

B. Installing VENV (MacOS or Linux)

1. Create VENV in VS Code Terminal

   python3 -m venv .venv

2. Activate .venv

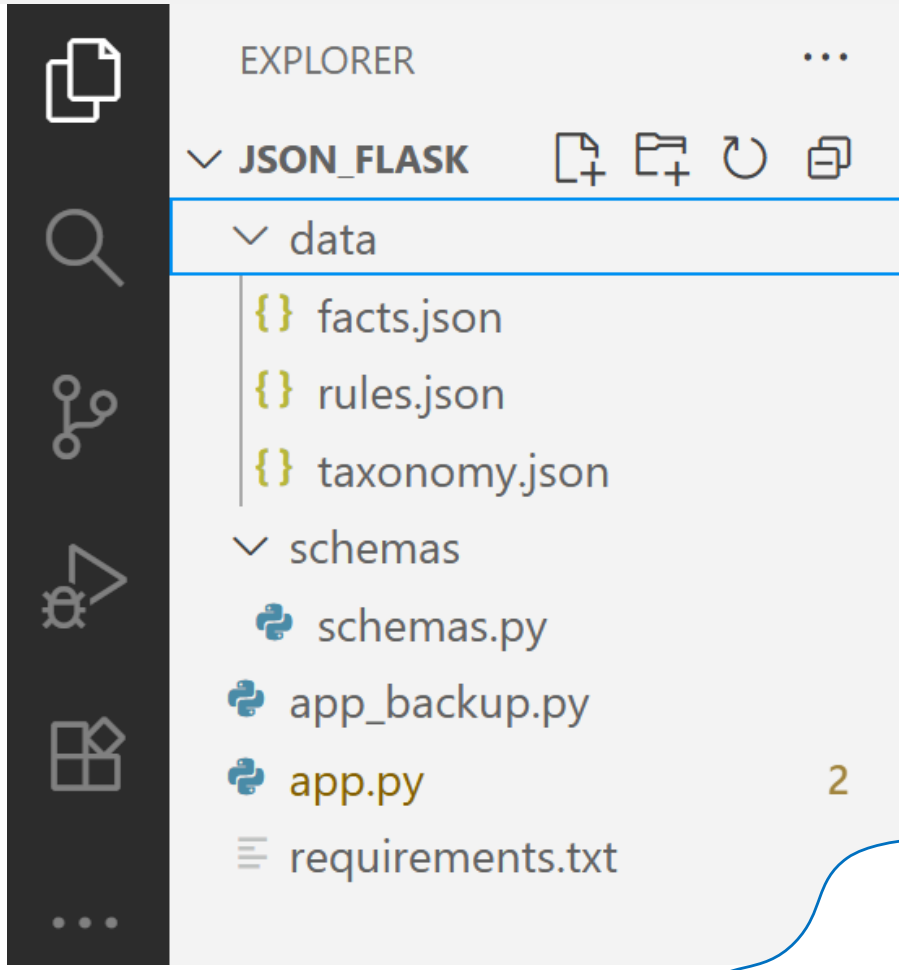
   source .venv/bin/activate

3. Upgrade pip to the latest version

   python3 -m pip install --upgrade pip

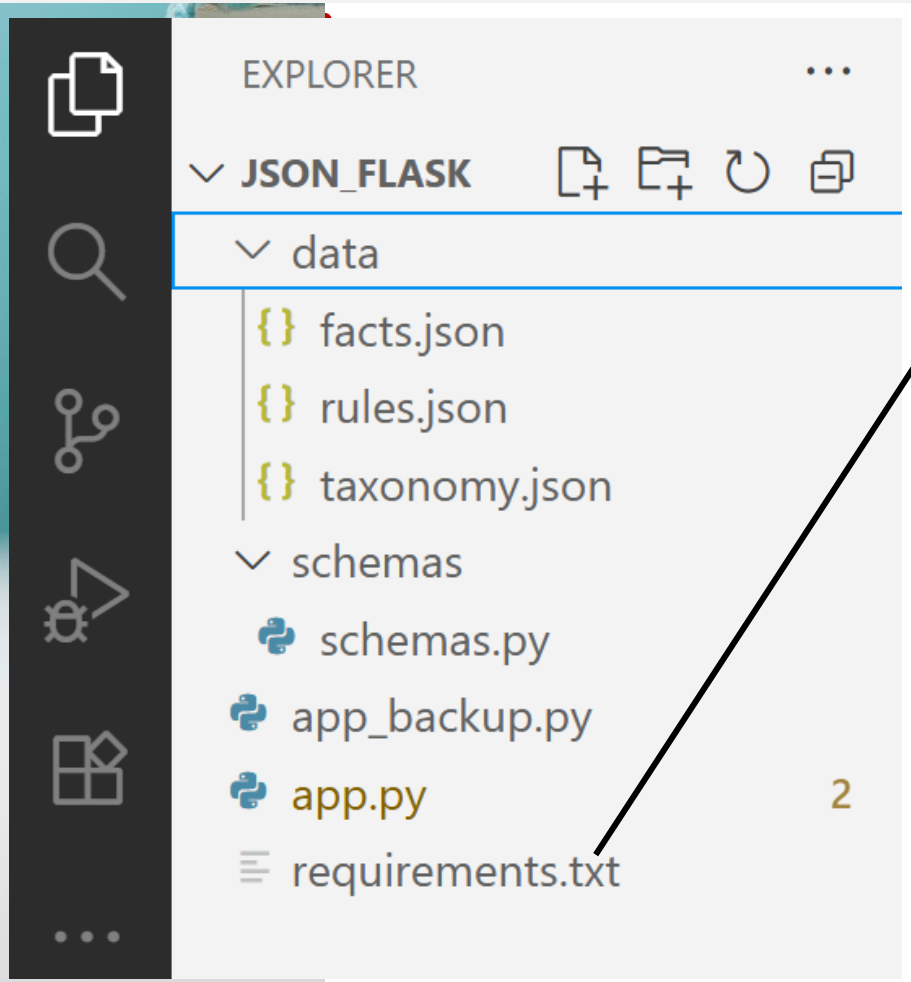
4. Install Flask Module

   pip install Flask



5. Building the Flask Knowledge API

Directory Structure



requirements.txt

```
Flask==2.3.3
jsonschema==4.19.0
Werkzeug==3.0.3
```

Install Instruction from Terminal in VS Code:

1. MacOS and Linux:

python -m venv .venv && source .venv/bin/activate && pip install -r requirements.txt

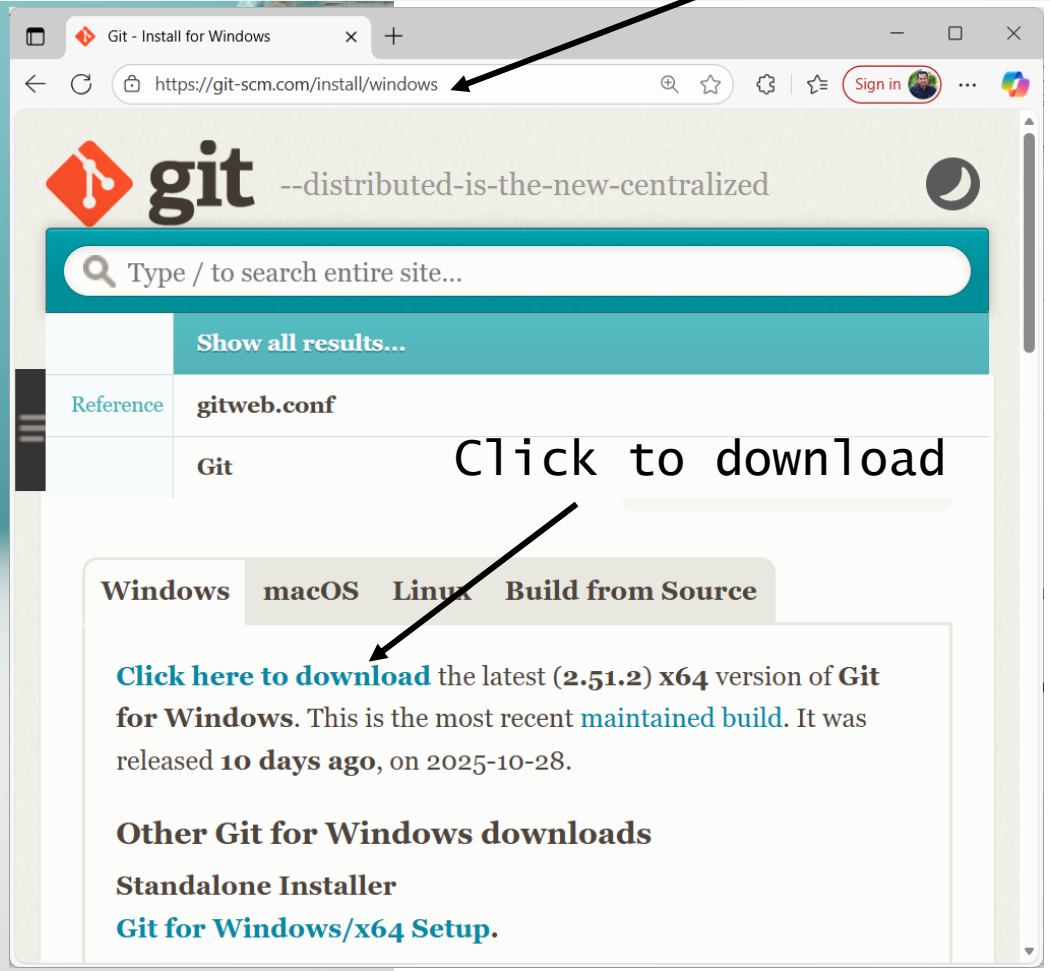
2. Window via Git Bash:

python -m venv .venv && source .venv/scripts/activate && pip install -r requirements.txt

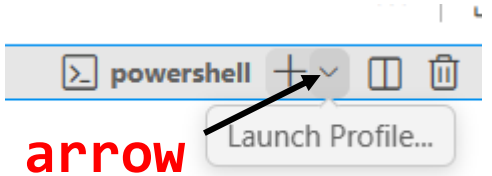
5. Building the Flask Knowledge API

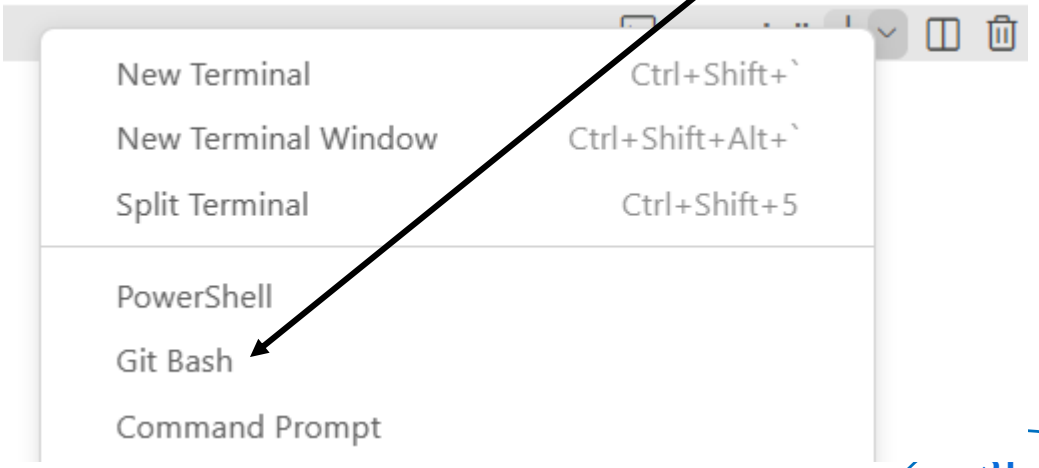
Install Git Bash

https://git-scm.com/install/windows



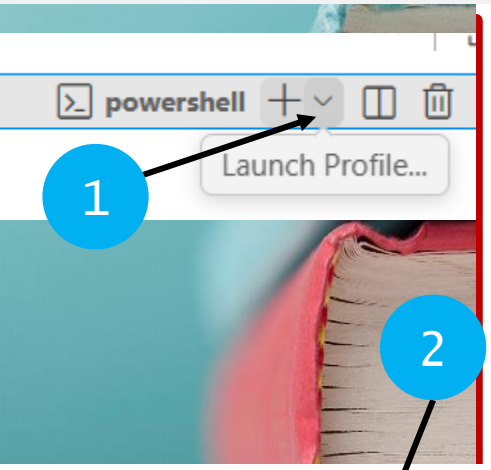
1. Installing Git-Bash on your PC.
2. Close VS Code and reopen it.
3. In the terminal
   >> Choose a **drop-down arrow**
   >> Choose the **Git Bash**

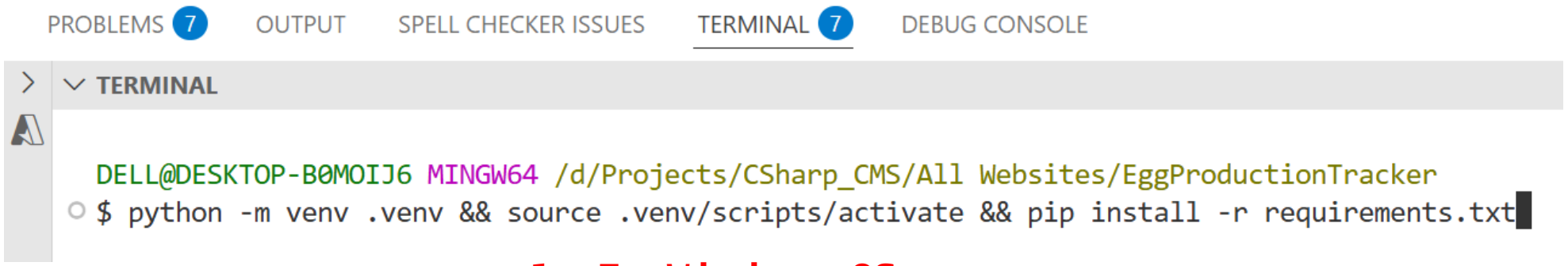


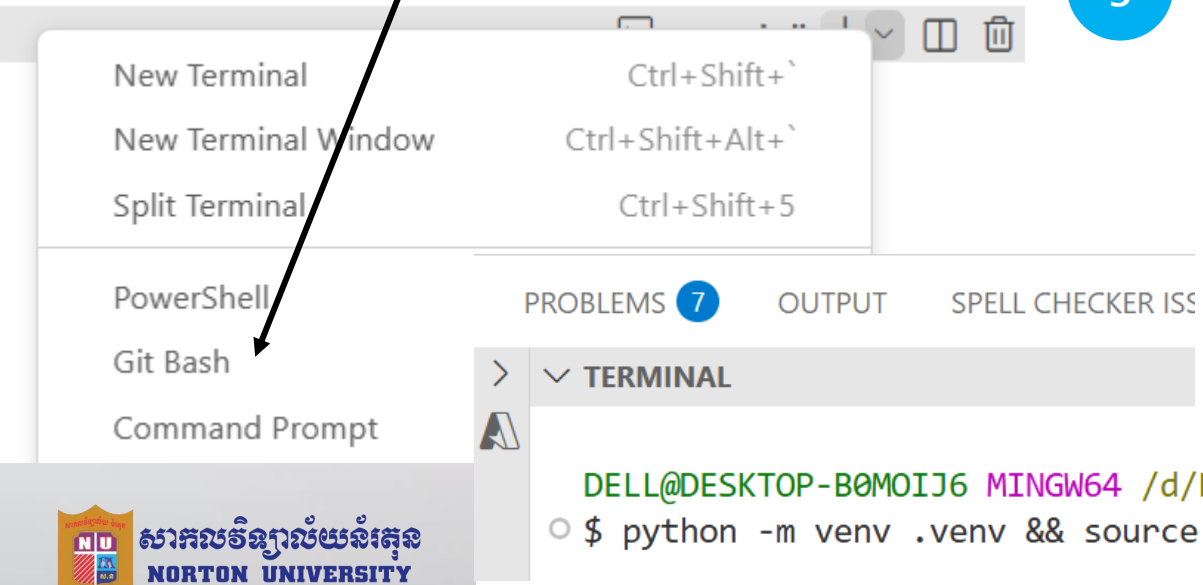


5. Building the Flask Knowledge API

Install “venv” via Git Bash in Terminal







1. In Windows OS:

Copy and paste commands into the Git Bash terminal, then enter

2. In MacOS or Linux:

Copy and paste commands into the Git Bash terminal, then enter

5. Building the Flask Knowledge API

app.py > ...

```python
# app.py
from flask import Flask, jsonify, request
import json
import os
from jsonschema import validate, ValidationError

APP_ROOT = os.path.dirname(os.path.abspath(__file__))
DATA_DIR = os.path.join(APP_ROOT, "data")
os.makedirs(DATA_DIR, exist_ok=True)

# ---------------------------
# JSON Schemas
# ---------------------------

facts_array_schema = {
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "type": "array",
    "items": {
        "type": "object",
        "additionalProperties": False,
        "required": ["id", "description", "value"],
        "properties": {
            "id": {"type": "string", "minLength": 1},
            "description": {"type": "string"},
            "value": {"type": "boolean"},
            "tags": {"type": "array", "items": {"type": "string"}}
        }
    }
}

```

Full app.py (Part 1

5. Building the Flask Knowledge API

Full app.py (Part 2)

```python
rules_array_schema = {
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "type": "array",
    "items": {
        "type": "object",
        "additionalProperties": False,
        "required": ["id", "conditions", "conclusion"],
        "properties": {
            "id": {"type": "string", "minLength": 1},
            "conditions": {
                "type": "array",
                "minItems": 1,
                "items": {"type": "string", "minLength": 1}
            },
            "conclusion": {"type": "string", "minLength": 1},
            "certainty": {"type": "number", "minimum": 0.0, "maximum": 1.0},
            "explain": {"type": "string"}
        }
    }
}

```

5. Building the Flask Knowledge API

Full app.py (Part 3)

```python
# ----------------------------
# Helpers
# ----------------------------

def load_json(filename: str):
    """Load JSON from ./data/<filename>."""
    path = os.path.join(DATA_DIR, filename)
    if os.path.exists(path):
        with open(path, "r", encoding="utf-8") as f:
            return json.load(f)
    else:
        raise FileNotFoundError(f"Required file not found: {path}")

def error_payload(e: ValidationError, filelabel: str):
    """Produce rich error payload for schema validation."""
    return {
        "file": filelabel,
        "error": e.message,
        "path": list(e.path),
        "schema_path": list(e.schema_path)
    }

def clamp01(x: float) -> float:
    """Clamp numeric confidence to [0,1]."""
    try:
        return max(0.0, min(1.0, float(x)))
    except Exception:
        return 0.0

```

5. Building the Flask Knowledge API

Full app.py (Part 4)

```python
# ---------------------------
# Load facts / rules / taxonomy
# ---------------------------

try:
    facts = load_json("facts.json")
    rules = load_json("rules.json")
    taxonomy = load_json("taxonomy.json")
except FileNotFoundError as e:
    print(f"ERROR: {e}")
    print("Please ensure data/facts.json, data/rules.json, and data/taxonomy.json exist.")
    exit(1)

# Validate on startup
try:
    validate(instance=facts, schema=facts_array_schema)
    validate(instance=rules, schema=rules_array_schema)
except ValidationError as e:
    print(f"ERROR: Data validation failed: {e.message}")
    exit(1)

PARENT = taxonomy.get("parent", {})
FACT_VALUE = {f["id"]: bool(f["value"]) for f in facts}

def ancestors(concept: str):
    """Yield all ancestors of a concept via PARENT mapping."""
    seen = set()
    cur = concept
    while cur in PARENT and PARENT[cur] not in seen:
        parent = PARENT[cur]
        seen.add(parent)
        yield parent
        cur = parent
```

5. Building the Flask Knowledge API

Full app.py (Part 5)

```python

def expand_observations(obs_conf: dict) -> dict:
    """Propagate observation confidence up the taxonomy."""
    if not PARENT:
        return obs_conf
    expanded = dict(obs_conf)
    for fid, c in list(obs_conf.items()):
        for anc in ancestors(fid):
            expanded[anc] = max(expanded.get(anc, 0.0), c)
    return expanded

def evaluate_rule(rule: dict, obs_conf: dict):
    """Return (fires:boolean, score:float)."""
    cond_ids = rule.get("conditions", [])
    if not cond_ids:
        return (False, 0.0)

    cond_scores = []
    for cid in cond_ids:
        c = clamp01(obs_conf.get(cid, 0.0))
        if c <= 0.0:
            return (False, 0.0)
        cond_scores.append(c)

    base = min(cond_scores)
    cf = clamp01(rule.get("certainty", 1.0))
    return (True, base * cf)

```

5. Building the Flask Knowledge API

Full app.py (Part 6)

```python
# ---------------------------
# Flask app & routes
# ---------------------------

app = Flask(__name__)

@app.get("/health")
def health():
    return jsonify({"status": "ok"})

@app.get("/facts")
def get_facts():
    """Return all facts from data/facts.json"""
    return jsonify(facts)

@app.get("/rules")
def get_rules():
    """Return all rules from data/rules.json"""
    return jsonify(rules)

@app.route("/validate", methods=["GET", "POST"])
def validate_payloads():
    """Validate facts and/or rules against schemas"""
    if request.method == "GET":
        return jsonify({
            "usage": "POST JSON with 'facts' and/or 'rules' to validate.",
            "schemas": ["facts_array_schema", "rules_array_schema"]
        })

```

5. Building the Flask Knowledge API

Full app.py (Part 7)

```
171     payload = request.get_json(force=True, silent=True) or {}
172     errors = []
173
174     if "facts" in payload:
175         try:
176             validate(payload["facts"], facts_array_schema)
177         except ValidationError as e:
178             errors.append(error_payload(e, "facts"))
179
180     if "rules" in payload:
181         try:
182             validate(payload["rules"], rules_array_schema)
183         except ValidationError as e:
184             errors.append(error_payload(e, "rules"))
185
186     return jsonify({"valid": len(errors) == 0, "errors": errors})
187
188 @app.route("/infer", methods=["GET", "POST"])
189 def infer():
190     """
191     Inference endpoint supporting both GET and POST.
192
193     GET: http://127.0.0.1:5000/infer?facts=f1,f2&weights=f1:1.0,f2:0.8&useTrueFacts=false
194     POST: JSON body with facts, weights, useTrueFacts
195     """
196
197     # Parse parameters (GET or POST)
198     if request.method == "GET":
199         # Query parameters
200         facts_param = request.args.get("facts", "")
201         weights_param = request.args.get("weights", "")
202         use_true_facts = request.args.get("useTrueFacts", "false").lower() == "true"
203
```

5. Building the Flask Knowledge API

Full app.py (Part 8)

```
204         observed_ids = set(f.strip() for f in facts_param.split(",") if f.strip())
205         weights = {}
206         if weights_param:
207             for pair in weights_param.split(","):
208                 if ":" in pair:
209                     fid, conf = pair.split(":", 1)
210                     try:
211                         weights[fid.strip()] = float(conf.strip())
212                     except ValueError:
213                         pass
214     else:
215         # JSON body
216         payload = request.get_json(force=True, silent=True) or {}
217         observed_ids = set(payload.get("facts", []))
218         weights = payload.get("weights", {})
219         use_true_facts = bool(payload.get("useTrueFacts", False))
220 
221     # Add true facts if requested
222     if use_true_facts:
223         observed_ids.update([fid for fid, val in FACT_VALUE.items() if val is True])
224 
225     # Build observation confidence map
226     obs_conf = {}
227     for fid in observed_ids:
228         obs_conf[fid] = clamp01(weights.get(fid, 1.0))
229 
230     # Expand via taxonomy
231     obs_conf = expand_observations(obs_conf)
232 
```

5. Building the Flask Knowledge API

Full app.py (Part 8)

```python
    # Evaluate rules
    fired = []
    for rule in rules:
        fires, score = evaluate_rule(rule, obs_conf)
        if fires and score > 0.0:
            fired.append({
                "rule": rule["id"],
                "conclusion": rule["conclusion"],
                "score": round(score, 3),
                "explain": rule.get("explain", ""),
                "conditions": rule.get("conditions", [])
            })

    # Aggregate conclusions
    aggregate = {}
    for r in fired:
        k = r["conclusion"]
        aggregate[k] = max(aggregate.get(k, 0.0), r["score"])

    ranked = sorted(
        [{"conclusion": k, "score": round(v, 3)} for k, v in aggregate.items()],
        key=lambda x: x["score"],
        reverse=True
    )

    return jsonify({
        "matched_rules": fired,
        "ranked_conclusions": ranked,
        "observations": [{"id": k, "confidence": v} for k, v in sorted(obs_conf.items())]
    })

if __name__ == "__main__":
    app.run(
        host="0.0.0.0",
        port=int(os.getenv("PORT", "5000")),
        debug=os.getenv("FLASK_DEBUG") == "1"
    )
```

6. Reasoning & Demonstrations

6. Reasoning & Demonstrations

Reasoning Model & Certainty

- Conservative min() over conditions
- Multiply by rule certainty
- Aggregate by max per conclusion

This is a simple, explainable certainty flow. Variants: average, product, Bayesian, Dempster-Shafer.

We pick min*certainty to be intuitive and safe.

6. Reasoning & Demonstrations

Demo 1: Classic Flu

- Observations: f1 fever =1.0, f2 cough =1.0

Request:

```
curl -s -X POST http://127.0.0.1:5000/infer \
  -H "Content-Type: application/json" \
  -d '{"facts":["f1","f2"]}' | jq
```

Expected:

- r1 fires → score = min(1,1)*0.8 = 0.8
- Conclusion flu: 0.8

6. Reasoning & Demonstrations

Demo 2: Cold vs Flu Tie-Breaker

- Observations: f1,f2,f4 (fever, cough, fatigue)

- r1: 0.8 (flu)

- r2: 0.6 (cold)

Ranked conclusions: flu (0.8) > cold (0.6)

6. Reasoning & Demonstrations

Demo 3: Partial Confidence

- Weighted facts

```
curl -s -X POST http://127.0.0.1:5000/infer \
  -H "Content-Type: application/json" \
  -d '{"facts":["f1","f2"], "weights":{"f2":0.7}}' | jq
```

r1 score = min(1.0, 0.7)*0.8 = 0.56 → flu 0.56.

6. Reasoning & Demonstrations

Validation Endpoint Demo

- Send broken rule → see errors

```
curl -s -X POST http://127.0.0.1:5000/validate \
  -H "Content-Type: application/json" \
  -d
'{"rules":[{"id":"rX","conditions":["f1"],"conclusion":"flu","cert
ainty":1.2}]}' | jq
```

Expect invalid with an error explaining the certainty bound.

Thank You

SEK SocheaT

socheatsek@norton-u.com

4. Homework: Additional Exercises on Rule-Based Decision-Making

Expanding the Diagnosis System

Task: Add new symptoms and diagnoses to the simple expert system from class. Try to include:

- **New Symptoms:** like sore throat, runny nose, headache.
- **New Diagnoses:** such as “strep throat” or “common allergies.”

Instructions:

- Update your system to include these symptoms.
- Create at least two new rules to handle these added symptoms.

Goal: Understand how adding more conditions enhances the system’s ability to make accurate diagnoses.