Python Concepts as per FHIR related use cases

# Objective

This project is a Python-based hands-on training kit for learning healthcare IT workflows using HL7® FHIR® standards. It walks through:

* Core Python concepts (dict, JSON, file handling)
* CSV → FHIR resource conversion
* Working with NDJSON and Bundles
* Posting to and parsing from FHIR servers
* Practical use cases like member attribution, coverage, and claim handling

# Prerequisite

FHIR REST API & Awareness or FHIR Resource Structure

Python core Programming concepts

# 📂 Modules Covered:

| **Module** | **Description** |
| --- | --- |
| module\_1\_json\_dict/ | Basic Python dict and JSON handling |
| module\_2\_file\_handling/ | Read, write, and convert between CSV/JSON/NDJSON |
| module\_3\_fhir\_resource/ | Using fhir.resources to build and validate FHIR objects |
| module\_4\_use\_cases/ | Real-world integration: from raw data to FHIR bundle |

# 1. Understanding JSON and Python Dictionaries

In FHIR workflows, almost all resource data is exchanged in JSON format. In Python, this JSON data is represented as a `dict` object.

Key Concepts:  
- `json.loads(json\_string)` → Convert JSON string to Python dict  
- `json.dumps(dict)` → Convert Python dict to JSON string  
- JSON is ideal for serialization and data exchange (e.g., between client ↔ FHIR server)

# 2. File Handling: Reading and Writing JSON/CSV/NDJSON

FHIR data often originates from CSV files (e.g., member lists, claims) or JSON files (e.g., saved resources).

Key Concepts:  
- Use `open()` and `json.load()` to read JSON files into dict  
- Use `csv.DictReader` to read CSV rows as dictionaries  
- Use `json.dump()` or `csv.DictWriter` to write back to file  
- Structure file paths into `input/` and `output/` folders for traceability

# 3. Nested Dictionaries and Lists

FHIR resources often have deeply nested structures — such as `Patient.name[0].family`.

Key Concepts:  
- Access nested data with `dict["key"]["subkey"]`  
- Use `.get()` with defaults to avoid KeyError  
- Loop through arrays of data (e.g., multiple `name`, `address`, `coding`)

# 4. Data Transformation: Raw to FHIR

Once data is extracted from files, it's converted into FHIR resources using:  
- The `fhir.resources` Python package  
- Structured models like `Patient()`, `Coverage()`, `Bundle()`

Before this, raw dictionaries are shaped into FHIR-compatible structures.

# 5. NDJSON: Newline-Delimited JSON

FHIR bulk data (like claims, coverage, or patient history) is often exported/imported in NDJSON format, where each line is an independent JSON object.

Key Concepts:  
- Loop through list of dicts and write each as a line using `json.dumps()`  
- Read NDJSON line-by-line with `json.loads(line)`

# 6. Validating FHIR Resource Structures

FHIR resources can be validated using the `fhir.resources` package:  
- `Patient(\*\*dict)` → validates the dict against FHIR structure  
- `model\_validate()` (Pydantic v2+) is used for parsing/validation  
- Errors can be caught with `try/except` using `pydantic.ValidationError`

# 7. FHIR Server Interaction

FHIR servers (e.g., HAPI, Azure Health Data Service) accept `GET`, `POST`, and `PUT` requests.

Key Concepts:  
- Use `requests.get()` and `requests.post()` for interacting with FHIR servers  
- Use `Bundle` of type `transaction` to send multiple resources together  
- Handle `\_count`, `\_total=accurate`, pagination, and query filters like `name=`, `birthdate=`

# 8. Bundling Resources

When sending data to FHIR servers:  
- Use `Bundle(type='transaction')`  
- Include multiple `entry` items with `resource` and `request` details  
- Resources like `Coverage` and `EOB` are linked using `Reference(reference='Patient/123')`

# 9. Handling Errors and Logging Invalid Data

Not all data passes validation. You must:  
- Catch and log `ValidationError`s  
- Write failed rows to a separate `error\_output/` folder  
- Use `uuid4()` to assign IDs only to validated resources

# 10. Folder Organization for Projects

Please go through the sample code which is simple POC level code for real use cases which are implemented at Onyx.

Organize your files to ensure maintainability:

Python-Healthcare/  
├── input/ # Raw CSV and JSON files  
├── output/ # Processed FHIR bundles and resources  
├── error\_output/ # Invalid records for review  
├── module\_\*\_<module\_name>/  
│\_\_\_\_docs/   
└── README.md

# Final Tip for Learners

The journey from CSV → JSON → FHIR-compliant resource → FHIR server is made easy by mastering:  
- Core Python data structures (`dict`, `list`, `str`)  
- JSON file manipulation  
- FHIR model classes via `fhir.resources`  
- Good validation and logging practices

# Next

Handling large amount of data using bigdata technology – Azure DataBrick (PySpark)

FHIR – REST API in details (must have)