REINVENTING IBM i

The Journey from RPG to Python

BY THE END OF THIS SESSION, YOU WILL KNOW:

- How to install and work with Python natively on IBM I
- How to query DB2 for i from Python and replace batch RPG reports in 40 lines
- Five proven patterns for calling RPG programs from Python (and vice versa)
- A concrete roadmap to start your organisation's Python journey

Koen Decorte • Senior IBM i Architect • CD-Invest nv • Melsele, Belgium

YOUR GUIDE TODAY

Koen Decorte

Founder & Innovator • CD-Invest nv • Antwerp, Belgium

- Belgian IBM i consultancy — 30+ years
- Anthropic AI certified
- PEPPOL Access Point PBE000833
- ERP / Accounting / CRM on IBM i
- Stratum Code Intelligence Platform
- Clients: JORI Furniture, Cras, Deknudt Frames, Formafoam, …
- **We build what others promise**

SECTION 01

Why Python?

Making the case for the world's most loved language on the world's most reliable platform

Python: The Numbers Don't Lie

Why the world is converging on Python — and why IBM i can't ignore it

#1

Most popular language
globally (TIOBE may 2026)
6 times tiobe language of the year

8.2M+

Python developers
worldwide

530K+

packages available
on PyPI

Python is not a trend. It's the default language of AI, data, automation, and the web.
And it runs natively on your IBM i.

IBM i: Stable, Trusted, and Evolving

The platform isn't going anywhere — it just needs new tools

- 40+ years of continuous evolution — no platform has a longer enterprise track record
- 100000+ active installations worldwide across manufacturing, finance, and distribution
- 99.999% uptime target — mission-critical by design, not by accident
- DB2 for i: one of the most reliable and capable databases on the planet
- IBM i is not legacy — the code running on it might be
- Python runs on IBM i via IBM i Open Source

The Generational Cliff Is Real

IBM i shops face an existential workforce problem

Avg RPG dev age: 55

1

Baby boomer RPG developers are approaching retirement. When they leave, they take decades of institutional knowledge with them.

Hard to hire RPG talent

2

University graduates don't learn RPG. Fixed-format scares off mid-career developers. The talent pool is shrinking every year.

Easy to find Python devs

3

Python is taught in every university program. Adding Python to IBM i opens your hiring pool to millions of developers worldwide.

This Is Not RPG vs. Python

It's RPG AND Python — each doing what it does best

RPG Stays In Charge Of

- High-performance DB2 I/O at machine level
- Transaction processing with built-in journaling
- 40 years of battle-tested business logic
- Native IBM i objects and activation groups
- CL integration and job control

Python Takes Over

- REST APIs, JSON, web services in minutes
- AI/ML: scikit-learn, TensorFlow, LangChain
- Data analysis: pandas, NumPy, Polars
- Rich ecosystem: 530K+ packages on PyPI
- Modern developer ergonomics and tooling

IBM i Is Already a Modern Platform

You don't need to re-platform — the modern stack is already here

- ILEastic: open-source HTTP framework for RPG — REST APIs from your existing code
- noxDB: JSON processing library for RPG — parse/create JSON natively in ILE
- SourceOrbit: open-source build system for IBM i — Git + CI/CD integration
- Python, Node.js, Git, nginx — all available via IBM i Open Source (yum)
- VS Code + Code for IBM i: full modern IDE experience for IBM i development
- IBM i Open Source community: ibm.biz/ibmi-oss — active GitHub presence

Three Forces Converging Right Now

The stars align for Python on IBM i — miss this window at your peril

AI is Python-native

1

Every AI library — LangChain, OpenAI SDK, scikit-learn, HuggingFace — is Python first. IBM i shops that don't add Python can't participate in the AI revolution.

Workforce cliff

2

Senior RPG developers retire in the next 5–10 years. Python extends the runway. You can hire Python developers today and run them alongside your RPG core.

API economy

3

Your IBM i data has never been more valuable. Python makes it trivial to expose DB2 as REST APIs consumed by mobile apps, web frontends, and partner systems.

530.000 Packages. Waiting for IBM i.

A taste of what Python brings to your platform

Web / API	Data / BI	AI / ML
FastAPI • Flask • Django • requests	pandas • NumPy • Polars • openpyxl	scikit-learn • LangChain • OpenAI SDK

Automation	Database	PEPPOL / EDI
paramiko • schedule • celery	ibm_db • SQLAlchemy • pyodbc	lxml • zeep • cryptography

What Python Is NOT on IBM i

Setting realistic expectations before we dive in

⚠ **Not a silver bullet:**	Python won't magically replace 30 years of RPG business logic overnight — nor should it.
⚠ **Not as fast as native RPG I/O:**	For high-throughput transaction processing, RPG's native I/O still wins. Use the right tool.
⚠ **Not zero learning curve:**	IBM i Python has quirks: PASE environment, ASCII vs EBCDIC, library paths. Plan for it.
⚠ **Not a reason to abandon IBM i:**	The goal is to EXTEND IBM i, not escape it. Python is an addition, not a replacement.
⚠ **Not yet all releases:**	IBM i still lacks a Rust compiler meaning not all packages will run natively

The Business Case for Python on IBM i

Six arguments your CTO will understand

- Speed to market: Python REST APIs deploy in hours, not weeks of RPG development
- Talent pool: Python developers are 100× easier to hire than RPG developers today
- AI readiness: Python is the only language for serious AI/ML — IBM i needs it now
- Cost: open-source Python stack vs. proprietary tooling — zero license fees
- Risk reduction: a two-language shop is more resilient than a single-language dependency
- IBM endorsement: IBM officially ships Python for IBM i — this is a fully supported path

How IBM i Shops Actually Adopt Python

Three proven entry points — pick the one that fits your organisation

- Pattern A — Automation first: replace CL scripts and shell scripts with Python — low risk, immediate value
- Pattern B — API layer: expose RPG programs and DB2 tables as REST endpoints — fastest modernisation path
- Pattern C — New development: all net-new projects are Python; RPG handles the existing core
- Most shops start with Pattern A — it builds confidence and delivers real savings quickly
- All three patterns are compatible — many organisations run them simultaneously
- The goal is not to choose a pattern and stick to it — it's to start. This week.

What Your Competitors Are Doing

Ignoring Python on IBM i is a strategic risk

- ERP vendors rewriting in Python microservices — faster feature delivery to your shared clients
- Fintech competitors building real-time analytics on IBM i + Python
- Manufacturing clients demanding REST APIs from their IBM i suppliers
- Your best developers are asking for Python — or leaving for shops that offer it
- IBM i shops adding Python report 40–60% faster delivery on new projects
- The question is not 'Should we add Python?' — it's 'How soon before we fall behind?'

CODE COMPARISON

RPG vs Python: Code Side by Side

Seeing is believing — what changes, what stays the same, and what gets dramatically better

What Actually Changes When You Move to Python

Less than you fear. More than you expect.

✗ What Changes

- Syntax — new keywords, indentation
- Toolchain — pip, venv, pytest
- Library model — PyPI vs *LIBL
- Encoding — UTF-8 in PASE (handled)
- Deployment — files, not *PGM objects

✓ What Stays the Same

- Business logic — loops, conditions, SQL
- IBM i security — profiles, authority
- DB2 knowledge — same tables, same SQL
- Your domain expertise (decades of value)
- The platform — IBM i is still underneath

↑ What Gets Better

- Readability — 5 lines vs 50 for reports
- Testing — pytest makes it trivial
- Libraries — 530 K packages on PyPI
- AI/ML — impossible in RPG, native here
- Hiring — Python devs are everywhere

→ **Your IBM i knowledge is your unfair advantage. Python multiplies it — it doesn't replace it.**

Hello World: Your First Comparison

Same result — dramatically different ceremony

RPG — Free-format program

```
**free
ctl-opt dftactgrp(*no) actgrp(*caller);

dcl-s greeting  varchar(50);
dcl-s recipient varchar(30);

recipient = 'IBM i';
greeting  = 'Hello, ' + %trimr(recipient) + '!';
dsply greeting;

*inlr = *on;
return;
```

VS

Python — 3 meaningful lines

```
#!/usr/bin/env python3

recipient = "IBM i"
greeting  = f"Hello, {recipient}!"
print(greeting)
```

→ **RPG: 10 lines, 3 declarations, activation group wiring. Python: 3 meaningful lines. Same output.**

Reading from DB2: Cursor Comparison

Both use the same SQL engine — Python hides the ceremony

RPG — DECLARE / OPEN / FETCH / CLOSE

```
dcl-s wCity    varchar(30) inz('HASSELT');
dcl-s wCustno  char(10);
dcl-s wCustnam varchar(50);
dcl-s wVat     packed(13:2);

exec sql declare c1 cursor for
  select custno, custnam, curvat
  from   cdacclib.custmast
  where  custcty = :wCity
  order  by custnam;

exec sql open c1;
exec sql fetch c1
  into :wCustno, :wCustnam, :wVat;
dow sqlcode = 0;
  processRow(wCustno: wCustnam: wVat);
  exec sql fetch c1
    into :wCustno, :wCustnam, :wVat;
enddo;
exec sql close c1;
```

VS

Python — fetchall() — that's it

```
import ibm_db_dbi

conn   = ibm_db_dbi.connect(
           "*LOCAL", "KOEN", "pwd")
cursor = conn.cursor()

cursor.execute("""
  SELECT custno, custnam, curvat
  FROM   cdacclib.custmast
  WHERE  custcty = ?
  ORDER  BY custnam
""", ("HASSELT",))

for row in cursor.fetchall():
    process_row(row[0], row[1], row[2])

conn.close()
```

→ **Python eliminates DECLARE / OPEN / FETCH / CLOSE. Same DB2 engine, same execution plan, 60 % less code.**

JSON Handling: noxDB vs Native Python

JSON is Python's native data structure — no library, no handle cleanup

RPG — noxDB (external library)

```
// BNDDIR('NOXDB') required
dcl-s jIn   varchar(2048);
dcl-s jOut  varchar(2048);
dcl-s hIn   int(10);
dcl-s hOut  int(10);
dcl-s custno  varchar(10);
dcl-s custnam varchar(50);

hIn    = json_parseString(jIn);
custno  = json_getStr(hIn:'custno');
custnam = json_getStr(hIn:'custnam');

hOut = json_newObject();
json_setStr(hOut:'status':'ok');
json_setStr(hOut:'custno':custno);
json_setStr(hOut:'custnam':custnam);
json_setNum(hOut:'processed':1);
jOut = json_asText(hOut);

json_close(hIn);
json_close(hOut);
```

VS

Python — dict built-in — always available

```
import json
from datetime import datetime

# Parse — one line
data    = json.loads(in_json)
custno  = data["custno"]
custnam = data["custnam"]

# Build response — it's just a dict
response = {
    "status":    "ok",
    "custno":    custno,
    "custnam":   custnam,
    "processed": 1,
    "timestamp": datetime.now().isoformat()
}

out_json = json.dumps(response, indent=2)
# Add a field: response["extra"] = "value"
# Done. No handle. No cleanup.
```

→ **Python dicts ARE JSON. No external library, no handle management, no memory leak risk. Add a field in one line.**

Building a REST Endpoint: ILEastic vs FastAPI

FastAPI generates OpenAPI docs, validation, and error handling automatically

RPG — ILEastic (manual wiring)

```
/copy QCOPYSRC,IREST
/copy QCOPYSRC,IRESTUTIL

dcl-proc getCustomer export;
  dcl-pi *n;
    req  likeds(httpRequest);
    resp likeds(httpResponse);
  end-pi;
  dcl-s custno   varchar(10);
  dcl-s custnam  varchar(50);
  dcl-s result   int(10);
  custno = web_getQueryParam(req:'id');
  exec sql select custnam into :custnam
    from custmast where custno = :custno;
  if sqlcode = 100;
    web_setResponseStatus(resp:404);
    return;
  endif;
  result = json_newObject();
  json_setStr(result:'custno':custno);
  json_setStr(result:'name':custnam);
  web_setResponseJson(resp:result);
  json_close(result);
end-proc;
```

VS

Python — FastAPI (10 lines + auto docs)

```
from fastapi import FastAPI, HTTPException
import ibm_db_dbi

app = FastAPI()

@app.get("/customer/{custno}")
async def get_customer(custno: str):
    conn = ibm_db_dbi.connect(
               "*LOCAL", "USR", "PWD")
    cur  = conn.cursor()
    cur.execute(
        "SELECT custnam FROM custmast"
        " WHERE custno = ?", (custno,))
    row = conn.close() or cur.fetchone()
    if not row:
        raise HTTPException(status_code=404)
    return {"custno": custno,
            "name":   row[0].strip()}
# /docs → Swagger UI auto-generated
```

→ **FastAPI: type validation, 404 handling, Swagger docs, async I/O — all automatic. ILEastic requires every line by hand.**

Error Handling: MONITOR vs try / except

Python exception hierarchy lets you handle DB errors, logic errors, and surprises separately

RPG — MONITOR / ON-ERROR (all-or-nothing)

```
dcl-s wVat     packed(13:2);
dcl-s wErrMsg  varchar(200);

monitor;
  exec sql
    select curvat into :wVat
    from   cdacclib.custmast
    where  custno = :wCustno;

  if sqlcode = 100;
    wErrMsg = 'Customer '
            + %trimr(wCustno)
            + ' not found';
    return *off;
  elseif sqlcode <> 0;
    wErrMsg = 'SQL err: '
            + %char(sqlcode);
    return *off;
  endif;

on-error;
  // catches ALL other RPG errors
  wErrMsg = 'Unexpected RPG error';
  return *off;
```

VS

Python — try / except / finally (surgical)

```
import ibm_db_dbi, logging
from decimal import Decimal

try:
    cursor.execute(
        "SELECT curvat FROM custmast"
        " WHERE custno = ?", (custno,))
    row = cursor.fetchone()
    if row is None:
        raise ValueError(
            f"Customer {custno} not found")
    vat = Decimal(str(row[0]))

except ibm_db_dbi.Error as e:
    logging.error(f"DB error: {e}")
    raise          # re-raise for caller

except ValueError as e:
    logging.warning(str(e))
    vat = Decimal("0.00")

finally:
    conn.close()   # always runs
```

→ **Python distinguishes DB errors / business logic / unexpected failures. MONITOR catches all or nothing.**

Calling External APIs: HTTPAPI vs requests

Python makes HTTP calls trivially simple — one line for the common case

RPG — HTTPAPI library (requires *LIBL setup)

```
H BNDDIR('HTTPAPI')

dcl-s url     varchar(256);
dcl-s resData varchar(8192);
dcl-s rc      int(10);
dcl-s hJson   int(10);

url = 'https://api.vatcheck.eu/v2/'
    + %trimr(vatNo);

rc = http_req(
       url
      :'GET'
      :*omit
      :resData
      :%len(resData)
      :0);

if rc <> 0;
  // parse error details from resData
  return *off;
endif;
// parse resData with noxDB...
```

VS

Python — requests (pip install requests)

```
import requests

# GET request — one line
url  = f"https://api.vatcheck.eu/v2/{vat}"
resp = requests.get(url, timeout=10)
resp.raise_for_status()   # 4xx/5xx → exception
data = resp.json()

# POST with JSON body + auth header
result = requests.post(
    "https://api.peppol.eu/invoice",
    json={
        "sender":   "BE0123456789",
        "receiver": "BE0987654321",
        "amount":   1250.00
    },
    headers={
        "Authorization": f"Bearer {token}"
    },
    timeout=30).json()
```

→ **requests.get(url).json() — that's the common case. Timeout, auth headers, JSON body: one argument each.**

Batch Processing: CL + RPG vs Python

Python collapses the CL + RPG two-layer architecture into a single file

RPG — CL submit + RPG program (2 objects)

```
/* RUNJOB.CLP — scheduled via ADDJOBSCDE */
PGM PARM(&DATE)
  DCL VAR(&DATE) TYPE(*CHAR) LEN(10)
  SBMJOB CMD(CALL PGM(CDACCLIB/RPTGEN) +
             PARM(&DATE)) +
         JOBD(CDACCLIB/RPRTJD) +
         JOBQ(QBATCH)
ENDPGM

/* RPTGEN.RPGLE — 300+ lines:
   file declarations, OPEN/READ loop,
   internal array totals, OVRPRTF,
   STRPRTF → spool → manual Excel  */
```

VS

Python — Python — one file, one deploy

```
import ibm_db_dbi, pandas as pd
import smtplib, schedule, time, logging

REPORT_SQL = """
  SELECT salesrep, SUM(amount) tot
  FROM   cdacclib.salesdet
  GROUP  BY salesrep
  ORDER  BY tot DESC
"""

def generate_report():
    conn = ibm_db_dbi.connect(
               "*LOCAL","RPT","pwd")
    df = pd.read_sql(REPORT_SQL, conn)
    df.to_excel(f"/reports/{today}.xlsx")
    send_email(f"/reports/{today}.xlsx",
               to="sales@jori.be")
    logging.info(f"Report sent: {today}")

schedule.every().day.at("06:00").do(
    generate_report)
while True:
    schedule.run_pending(); time.sleep(60)
```

→ **CL + RPG: 2 objects, 2 compile steps, OVRPRTF, manual Excel export. Python: 1 file, schedules itself, sends Excel.**

Testing Your Code: RPG vs pytest

The hardest part of RPG to modernise — Python makes it a non-event

RPG — The RPG testing reality

```
// RPG has no built-in test framework.

// Option A: manual CALL + inspect
CALL PGM(CALCVAT) PARM('INV001'
  x'00000000' x'00000000')
// Check DSPLY output or WRKJOBLOG

// Option B: RPGUNIT (open source)
// Requires separate install, limited
// mock / stub support, and still needs
// a live IBM i connection to run.

// Either way:
//   • No coverage measurement
//   • Cannot run in CI/CD pipeline
//   • Cannot run without real IBM i
//   • Slow: compile → call → inspect
//   • No parameterised test cases
//   • Debugging: DSPLY + WRKJOBLOG
```

VS

Python — pytest — industry standard

```
# test_vatcalc.py — runs anywhere
import pytest
from vatcalc import calculate_vat

def test_standard_rate_belgium():
    assert calculate_vat(100, "BE") == 21.0

def test_zero_rate():
    assert calculate_vat(100, "EXEMPT") == 0.0

@pytest.mark.parametrize("amount,ctry", [
    (100, "BE"), (500, "NL"), (1000, "DE")
])
def test_multiple_countries(amount, ctry):
    result = calculate_vat(amount, ctry)
    assert result >= 0

def test_negative_raises():
    with pytest.raises(ValueError):
        calculate_vat(-50, "BE")

# pytest -v --cov=vatcalc
# Coverage: 97%   Tests: 5 passed  0.12 s
```

→ **pytest runs on your laptop, in GitHub Actions, in Docker — no IBM i needed. Parameterised, mocking, coverage built in.**

When to Use RPG, When to Use Python

A practical decision guide for your IBM i shop — not an ideology, a toolbox

Use Case / Task	RPG	Python	Recommendation
High-volume DB2 transaction I/O	✓ RPG	○ Possible	RPG wins on raw native I/O throughput
Complex commitment control	✓ RPG	○ ibm_db	Journal boundaries are safest in RPG
Battle-tested business logic	✓ Keep	⚠ Risk	If it isn't broken, don't touch it
New REST API endpoints	○ ILEastic	✓ Python	FastAPI: 10 lines. ILEastic: 40+.
Data analysis & reporting	✗ Manual	✓ Python	pandas replaces hundreds of lines
AI / ML integration	✗ No	✓ Only option	LangChain, OpenAI are Python-only
Calling external HTTP APIs	○ HTTPAPI	✓ Python	requests: 1 line vs 40 in RPG
Automation & scheduling	○ CL	✓ Python	schedule + subprocess beats CL/SBMJOB
Unit / integration testing	○ RPGUNIT	✓ Python	pytest runs without IBM i, in CI/CD
All new features going forward	○ Works	✓ Recommended	Python ecosystem grows 100× faster

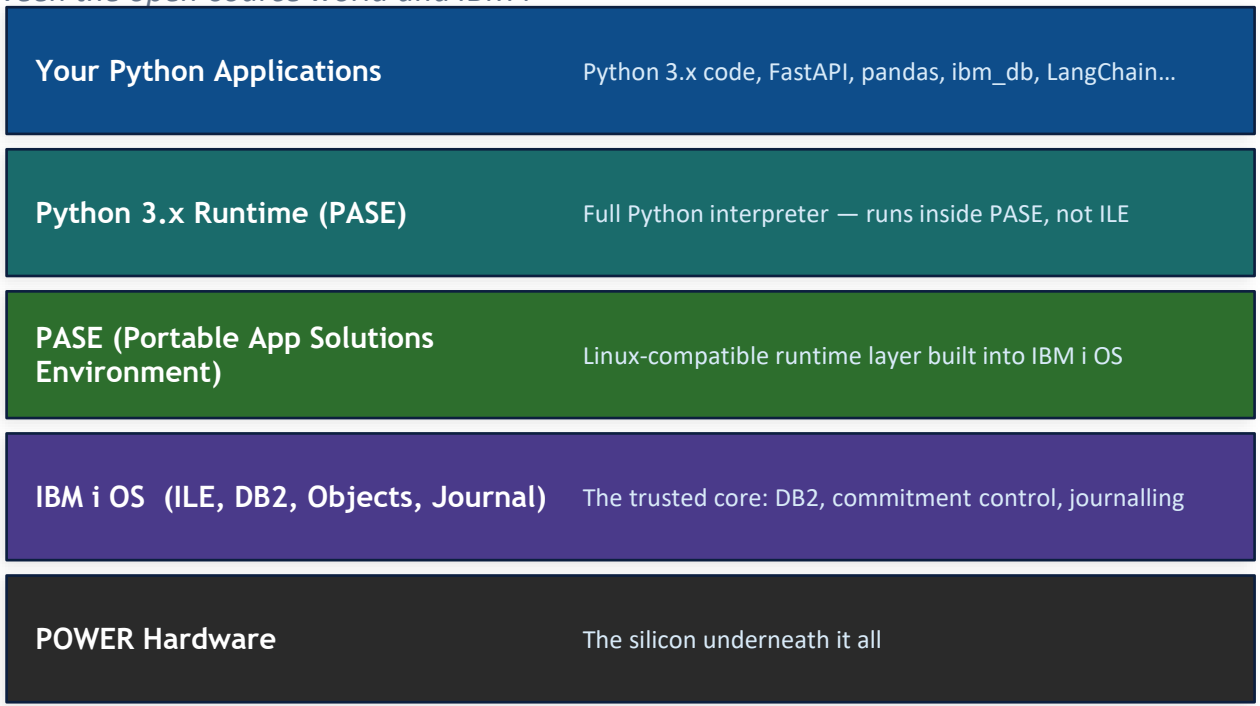
SECTION 02

Python on IBM i

Installation, environment setup, and running your first Python program on the green machine

How Python Runs on IBM i

PASE: the bridge between the open-source world and IBM i



PASE provides a full Linux-compatible runtime on IBM i. Python lives in PASE — not ILE. It can still call ILE programs via itoolkit.

Installing Python on IBM i

Three commands and you're running Python 3 on your IBM i

```
# From a 5250 terminal or SSH session
call qp2term                          # Open PASE terminal
yum install python3                   # Install Python 3.x
yum install python3-pip               # Install pip (package manager)
yum install python3-ibm_db            # Install IBM i DB2 driver
python3 --version                     # Verify: Python 3.x
python3 -c "print('Hello, IBM i!')"  # First program

# Alternative: ACS GUI
# Tools → Open Source Package Management → Available Packages → python3 → Install

# Requires: IBM i V7R4M0+, 5770-SS1 Option 33 (PASE), internet access
```

Requires: IBM i V7R4M0+ • 5770-SS1 Option 33 (PASE) • Internet access for yum • Zero licensing cost

Virtual Environments: Python's Clean Rooms

Essential practice — isolate dependencies per project

```
# Create a virtual environment for your project
cd /home/koen/myproject
python3 -m venv venv                  # Create isolated environment
source venv/bin/activate              # Activate it
# Install packages (isolated — won't affect system Python)
pip install fastapi uvicorn ibm_db pandas
# Your app runs with pinned dependencies
python3 main.py
# Deactivate when done
deactivate
# Share dependencies via requirements.txt
pip freeze > requirements.txt         # Export pinned versions
pip install -r requirements.txt       # Reproduce anywhere
```

- One venv per project — never install to system Python
- Commit requirements.txt to Git — reproducible builds across environments
- Use .env files for IBM i library lists and credentials (python-dotenv)

Your First Python Program on IBM i

From hello world to a useful IBM i utility in 20 lines

```
#!/usr/bin/env python3
"""List active jobs on IBM i — using itoolkit"""
import itoolkit
import itoolkit.transport as tp

# Connect to IBM i via native PASE transport
conn = tp.DatabaseTransport('*LOCAL')
itool = itoolkit.iToolKit()

# Call QUSLJOB API
itool.add(itoolkit.iSrvPgm('QUSLJOB', 'QSYS', 'QUSLJOB'))
itool.call(conn)
jobs = itool.dict_out('QUSLJOB')

for job in jobs:
    print(f"Job: {job['JOB_NAME']:<10}  User: {job['USER_NAME']:<10}  Status: {job['JOB_STATUS']}")
```

itoolkit is IBM's Python library for calling IBM i APIs and ILE programs directly from PASE Python.

Setting Up Your Python Dev Environment

SSH + VS Code + Git: a modern workflow for IBM i Python

1

Enable SSH on IBM i

```
STRTCPSVR SERVER(*SSHD)
CHGTCPSVR SVRSPCVAL(*SSHD) AUTOSTART(*YES)
```

2

Connect VS Code

```
Extensions: Remote-SSH + Code for IBM i + Python
ssh koen@my-ibmi-host  ← in VS Code terminal
```

3

Clone your project

```
git clone https://github.com/myorg/ibmi-python.git
python3 -m venv venv && source venv/bin/activate
pip install -r requirements.txt
```

The EBCDIC Gotcha — Know It Before It Bites

IBM i is EBCDIC. PASE is ASCII. Python lives in PASE.

- IBM i native objects (files, programs, data) use EBCDIC encoding — a completely different byte mapping
- PASE (where Python runs) uses ASCII/UTF-8 — these are two different worlds on the same box
- ibm_db handles encoding conversion transparently — but you must understand why it exists
- Streaming data from IFS? Specify encoding='utf-8' explicitly or you'll get garbage
- Practical rule: use ibm_db for DB2 access — never raw file I/O for IBM i native files
- When in doubt: test with a known-value query first and print repr(value) to see what you have

PASE Environment Variables You Must Know

Set these before running any IBM i Python script

```
# Add to /home/user/.profile or use python-dotenv in your project

JAVA_HOME=/QOpenSys/pkgs/lib/jvm/openjdk-11   # for JDBC fallbacks
PATH=$PATH:/QOpenSys/pkgs/bin                  # yum-installed tools findable
PYTHONPATH=/home/koen/myproject                # add your project root
LIBPATH=/QOpenSys/pkgs/lib                     # shared libs for ibm_db
DB2_HOME=/QIBM/ProdData/OS400/DataBase         # for direct DB2 connections

# Or use python-dotenv in your app:
# pip install python-dotenv
# from dotenv import load_dotenv; load_dotenv()
```

- Store in /home/user/.profile for persistent settings across sessions
- Use .env + python-dotenv for project-specific configuration — never hardcode credentials

Scheduling Python Jobs on IBM i

IBM i's job scheduler can run Python — no cron daemon needed

- CL wrapper: SBMJOB CMD(QSH CMD('python3 /home/koen/job.py')) JOBD(PYJOBD)
- JOBD: create a job description that sets PASE environment variables automatically
- IBM i job scheduler (ADDJOBSCDE) triggers the CL wrapper on a recurring schedule
- Alternative: use Python's schedule library inside a persistent job for fine-grained timing
- For event-driven jobs: use data queue triggers → Python consumer picks up the event
- Monitor via WRKACTJOB — Python jobs appear as QP2SHELL or QZSHSH entries

Logging and Monitoring Python on IBM i

Your Python jobs need to be observable — build this in from day one

- Python logging module: write structured log files to IFS at /logs/myapp/ — rotated daily
- Send critical errors to a DB2 table — queryable from any 5250 screen by ops staff
- Integrate with IBM i message queues: use itoolkit QMHSNDPM API for native alerts
- Structured logging (JSON) enables log aggregation tools like Splunk or ELK
- Alert on errors: Python → SMTP → your existing IBM i alert notification system
- Build observability from day one — don't add it after the first production outage

SECTION 03

DB2 + Python

Querying DB2 for i from Python — the most important skill you'll learn today

ibm_db: Python's DB2 Gateway

The official IBM DB2 driver for Python — fast, full-featured, free

```
import ibm_db_dbi  # DB-API 2.0 compatible wrapper

# Connect to local IBM i DB2
conn = ibm_db_dbi.connect(database="*LOCAL", user="KOEN", password="mypassword")

# Simple query — always use ? placeholders, never string-format SQL
cursor = conn.cursor()
cursor.execute("""
    SELECT CUSTNO, CUSTNAM, CUSTCTY, CURVAT
    FROM   CDACCLIB.CUSTMAST
    WHERE  CUSTCTY = ?
    ORDER  BY CUSTNAM
""", ("HASSELT",))

for row in cursor.fetchall():
    print(f"{row[0]:<10}  {row[1]:<30}  {row[3]:.2f}")

cursor.close()
conn.close()
```

**LOCAL = no network needed • DB-API 2.0 = same API as any Python DB driver • Library = SQL schema*

pandas Meets DB2 for i

Turn your IBM i data into a pandas DataFrame in three lines

```
import ibm_db_dbi, pandas as pd

conn = ibm_db_dbi.connect(database="*LOCAL", user="KOEN", password="secret")

# Load entire result set into a DataFrame
df = pd.read_sql("""
    SELECT   ARTNO, ARTOMSCHR, ARTFAM, PRIJS
    FROM     JORI.JRARTIK
    WHERE    ARTFAM IN ('SOFA', 'CHAIR')
    ORDER BY ARTFAM, PRIJS DESC
""", conn)

# Instant analysis — no SQL gymnastics
print(df.groupby("ARTFAM")["PRIJS"].agg(["mean", "min", "max", "count"]))
df.to_excel("jori_products.xlsx", index=False)   # Export to Excel
df.to_json("products.json", orient="records")     # Export to JSON
```

- Analytics: group, pivot, aggregate DB2 data with pandas in seconds
- Export: Excel, CSV, JSON, Parquet — send your IBM i data anywhere instantly
- ML-ready: pandas DataFrames plug directly into scikit-learn — DB2 → AI pipeline

Calling DB2 Stored Procedures from Python

Leverage your existing SQL routines — no rewrite needed

```
import ibm_db
conn = ibm_db.connect("DATABASE=*LOCAL;UID=KOEN;PWD=secret;", "", "")

# Assume: CREATE PROCEDURE CDACCLIB.CALC_INVOICE_TOTAL
#           (IN  p_invno CHAR(10),
#            OUT p_total DECIMAL(13,2),
#            OUT p_vat   DECIMAL(13,2))

stmt = ibm_db.callproc(conn, "CDACCLIB.CALC_INVOICE_TOTAL",
    ("INV-0042198", 0.00, 0.00))   # IN param, two OUT params

# ibm_db returns updated tuple with OUT param values filled in
invno, total, vat = stmt
print(f"Invoice {invno}: Total = €{total:.2f}  (VAT: €{vat:.2f})")

# Any RPG procedure registered as a SQL stored procedure is callable this way
ibm_db.close(conn)
```

Your RPG stored procedures become instantly callable from Python, Flask, FastAPI — or any HTTP client. No wrapper code needed.

Using Mapepire

Using IBM's new DB2 client

```
from mapepire_python import connect

creds = {
  "host": "SERVER",
  "port": 8076,
  "user": "USER",
  "password": "PASSWORD",
}

with connect(creds) as conn:
    with conn.execute("select * from sample.employee") as cursor:
        result = cursor.fetchone()
        print(result)
```

Using ODBC from Python (my preferred option)

Using standard ODBC is now the fastest way to connect to IBM i

```
import os, pyodbc
print("Start")
conn = pyodbc.connect(
    "Driver={IBM i Access ODBC Driver};"
    f"System=localhost;"
    f"UID=KOEN;"
    f"PWD=XXXX;"
)
print("Conn")
cur = conn.cursor()
cur.execute("select current timestamp from sysibm.sysdummy1")
print(cur.fetchone()[0])
cur.close(); conn.close()
```

Fast standard access to IBM i

ODBC Driver

Fixes on the driver

- Make sure that in the home directory subfolder .iSeriesAccess of the user connecting to the system the following file is set to read only : cwb_userprefs.ini
- There is a bug in older versions of the driver causing it to grow and each time degrading the performance of ODBC
- We found the bug and helped IBM solve it.
- Install version 1.1.0.29-0 of the ibm-iaccess package – it has the fix or do what is above

CONNECTION DEEP DIVE

Mapepire vs ODBC

Two very different roads to the same DB2 table

Mapepire

WebSocket → Server Component → DB2

VS

ODBC

Native Driver → DB2 directly

How Each Approach Reaches Your DB2 Data

Every extra hop costs time — Mapepire adds two; ODBC adds zero

Mapepire (mapepire-python)

1. **Python Script**
   Your code calls mapepire-python
2. **WebSocket (TLS)**
   Network I/O to IBM i port 8076
3. **Mapepire Server (IBM i)**
   Node.js daemon on IBM i PASE
4. **Db2 for i**
   Final query execution

⚠ 4 hops • serialisation/deserialisation overhead • TLS each request

ODBC (ibm_db_dbi / ibm_db)

1. **Python Script**
   Your code calls ibm_db_dbi
2. **IBM iAccess ODBC Driver**
   Native C driver, zero serialisation
3. **Db2 for i**
   Direct query execution

✓ **3 hops • native binary protocol • persistent connection**

Identical SQL — Radically Different Wiring

Both are PEP 249 compatible — but install path, server requirements, and results shape differ

Mapepire — mapepire-python

```
# pip install mapepire-python
# Requires: Mapepire Server daemon
#           running on IBM i (port 8076)

from mapepire_python import connect

creds = {
    "host":     "ibmi.cdinvest.eu",
    "port":      8076,
    "user":     "KOEN",
    "password": "secret",
}

with connect(creds) as conn:
    with conn.execute("""
        SELECT custno, custnam, curvat
        FROM   cdacclib.custmast
        WHERE  custcty = 'HASSELT'
        ORDER  BY custnam
    """) as cursor:
        rows = cursor.fetchall()

# Result is a list of dicts — JSON shape:
# [{"CUSTNO":"000123","CUSTNAM":"JORI","CURVAT":21.0}, ...]
# Metadata included: useful for APIs
```

VS

ODBC — ibm_db_dbi (10-15× faster)

```
# pip install ibm-db  (+ IBM iAccess driver)
# Requires: IBM iAccess ODBC driver installed
#           No server daemon needed

import ibm_db_dbi as db

conn = db.connect(
    "DATABASE=*LOCAL;"
    "UID=KOEN;PWD=secret",
    "", "")

cursor = conn.cursor()
cursor.execute("""
    SELECT custno, custnam, curvat
    FROM   cdacclib.custmast
    WHERE  custcty = 'HASSELT'
    ORDER  BY custnam
""")
rows = cursor.fetchall()
conn.close()

# Result is a list of tuples — native types:
# [("000123","JORI",21.0), ...]
# Zero serialisation overhead
```

→ **API surface is nearly identical (PEP 249) — the difference is entirely in transport and data serialisation**

ODBC Is 10-15× Faster Than Mapepire

Measured on CD-Invest IBM i V7R5M0 production equivalent — same query, same data, same Python version

10-15×

faster with ODBC

on real CD-Invest IBM i workloads

Why is Mapepire slower?

1. **Network round-trip**
   Every query crosses TLS WebSocket to port 8076 and back — even on localhost
2. **JSON serialisation**
   All data is serialised to JSON in the server daemon and deserialised in Python
3. **Node.js intermediary**
   Mapepire Server is a Node.js process that mediates every statement
4. **Connection setup overhead**
   New WebSocket handshake + TLS negotiation per connection
5. **Result shape inflation**
   JSON envelope (id, metadata, is_done, success) adds bytes per result set

ODBC performance advantage

Native binary protocol • No serialisation • Persistent pooled connections • Direct DB2 engine access without intermediary process • Tuples (not dicts) = smaller memory footprint

Performance showdown

ODBC is still fastest connection to your IBM i DB2

```
[2026-05-16 14:52:48] Connection closed.

============================================================
PROFILING RESULTS
============================================================
         270 function calls in 0.043 seconds

   Ordered by: cumulative time
   List reduced from 125 to 30 due to restriction <30>
```

```
============================================================
PROFILING RESULTS
============================================================
         5194 function calls (5181 primitive calls) in 0.755 seconds
```

The Crossover: When Mapepire's async Pool Wins

10–15× slower was measured on a single sequential query. The picture flips under concurrent load.

ODBC + ThreadPoolExecutor ("pooling")

```
from concurrent.futures import ThreadPoolExecutor
import ibm_db_dbi as db

def run_query(sql):
    conn = db.connect("*LOCAL","U","P","","")
    return conn.cursor().execute(sql).fetchall()

# 10 concurrent queries → 10 OS threads
with ThreadPoolExecutor(max_workers=10) as ex:
    results = list(ex.map(run_query, queries))
```

✓ **Works — queries run in parallel on IBM i**

⚠ Each thread needs its own OS thread stack (~1 MB each)

⚠ Thread scheduling overhead grows with concurrency

✗ At 50+ concurrent queries: thread contention kills throughput

✗ No built-in backpressure — pool exhaustion is your problem

Mapepire Pool (true async)

```
import asyncio
from mapepire_python import Pool, PoolOptions

async def main():
  async with Pool(options=PoolOptions(
      creds="./mapepire.ini",
      max_size=10, starting_size=5)) as pool:
    # 10 queries fan out across 10 IBM i jobs
    results = await asyncio.gather(
        *[pool.execute(sql) for sql in queries])
```

✓ **asyncio — one thread, one event loop, zero thread overhead**

✓ **Each PoolJob = separate IBM i server job → true DB2 parallelism**

✓ **Coroutines are ~4 KB vs ~1 MB for OS threads — 250× lighter**

✓ **asyncio.gather() fans N queries to N jobs simultaneously**

⚠ Each PoolJob still pays WebSocket overhead per query

1 query — Sequential	5-20 queries — Moderate concurrency	50+ queries — High concurrency

Mapepire vs ODBC: Full Comparison

Every dimension that matters for IBM i Python development

Feature	Mapepire (mapepire-python)	ODBC (ibm_db_dbi)
Raw query throughput	○ Slow (WebSocket + JSON)	✓ 10–15× faster (native binary)
Works from anywhere (no VPN)	✓ Yes — WebSocket over internet	✗ ODBC driver must be installed locally
Server setup required	✗ Yes — Mapepire daemon on IBM i	✓ None — driver on client only
Async / concurrent queries	✓ Native async, Pool object	○ Thread-based only, no async
Result format	○ JSON dicts + metadata envelope	✓ Native tuples, no overhead
PEP 249 compatible	✓ Yes (connect / cursor)	✓ Yes (ibm_db_dbi)
Kerberos / SSO auth	✓ Yes (built-in)	✓ Yes (ODBC DSN)
TLS / certificate support	✓ Yes (v0.3.0+)	✓ Yes (ODBC driver)
pandas read_sql() integration	○ Manual via fetchall()	✓ Direct: pd.read_sql(sql, conn)
CL / program calls	✗ SQL only	✗ SQL only (use itoolkit for CL)
Works on non-IBM i clients	✓ Any OS, no driver install	○ Needs IBM iAccess ODBC driver
Streaming / pagination	✓ rows_to_fetch control	✓ fetchmany() standard
Recommended for new projects	○ Remote/cloud access scenarios	✓ On-prem, performance-critical

Pick the Right Tool for the Right Job

They solve different problems — don't let the WebSocket URL distract you from the performance data

Use Mapepire when…

- ✓ Your Python code runs outside the IBM i network (cloud, SaaS, CI/CD)
- ✓ You need true async concurrency — Pool object manages multiple DB2 jobs
- ✓ You cannot install IBM iAccess ODBC driver on the client machine
- ✓ You are building a remote data API and the round-trip is network-bound anyway
- ✓ Your team works on Windows/Mac and driver management is a support burden

Use ODBC (ibm_db_dbi) when…

- ✓ Performance is critical — batch ETL, real-time analytics, high-volume reports
- ✓ Your script runs on IBM i PASE or in the same network (*LOCAL DSN = zero hops)
- ✓ You use pandas — pd.read_sql() works natively, no manual fetchall() needed
- ✓ You want the simplest possible setup: pip install ibm-db, connect, done
- ✓ You measured 10–15× slower Mapepire and can reproduce it — trust the data

Bottom line: ODBC for everything on-prem and performance-critical. Mapepire for remote/cloud access without driver management.

SQLAlchemy ORM with DB2 for i

Use Python's most popular ORM — connected to your IBM i DB2

- SQLAlchemy abstracts DB2 behind Python objects — no raw SQL required for standard queries
- Install: pip install sqlalchemy ibm_db_sa (IBM's official SQLAlchemy dialect)
- Connection: create_engine('ibm_db_sa://KOEN:pwd@localhost/MYLIB')
- Define models: class Customer(Base): custno = Column(String(10), primary_key=True)
- Query: session.query(Customer).filter(Customer.city == 'Brussels').all()
- Works with FastAPI, Django, Flask — the full Python web ecosystem is now yours

Python DB2 Best Practices

Patterns that make your code production-ready

❌ Common Mistakes

```
conn = ibm_db_dbi.connect(...)
cursor.execute("SELECT * FROM " + table)  # SQL INJECTION!
data = cursor.fetchall()
# Forgot to close — resource leak!
```

✅ Production Pattern

```
@contextmanager
def db2_connection():
    conn = ibm_db_dbi.connect(*LOCAL)
    try: yield conn
    finally: conn.close()  # Always closes
with db2_connection() as conn:
    cursor.execute("SELECT... WHERE CUSTNO=?",(no,))
```

Python DB2 Performance Tips

Keep your Python ↔ DB2 queries fast

- Use executemany() for bulk inserts — 10–100× faster than looping execute() one row at a time
- Fetch in batches: cursor.fetchmany(1000) — never fetchall() on millions of rows
- Use server-side cursors for large result sets — avoids memory exhaustion in PASE
- Minimise columns: SELECT only what you need — no SELECT * in production code
- Indexes still matter: Python SQL goes through the same DB2 query optimizer as RPG
- Connection pooling: ibm_db_dbi supports it — reuse connections across API requests

Transaction Management in Python

ACID compliance from Python — DB2 journalling works for you

- ibm_db_dbi uses autocommit=False by default — explicit conn.commit() required
- conn.rollback() to undo on error — always wrap in try/except/finally
- Python context managers (with statements) make commit/rollback clean and reliable
- DB2 for i journal-based commitment control applies — same guarantees as RPG
- Savepoints supported: ibm_db.exec_immediate(conn, 'SAVEPOINT sp1 ON ROLLBACK RETAIN CURSORS')
- Multi-table transactions work exactly as in RPG — journal applies to all Python SQL

IBM i Data Type Mapping

Packed decimal, zoned, dates — Python handles them all

DECIMAL / NUMERIC	→	**Python Decimal**	from decimal import Decimal — never float for money
DATE / TIME / TIMESTAMP	→	**Python datetime**	date, datetime — fully timezone-aware
CHAR / VARCHAR	→	**Python str**	Auto-converted, trailing spaces stripped by ibm_db
BINARY / VARBINARY	→	**Python bytes**	Use for raw binary data and BLOBs
Packed decimal	→	**Returned as str**	Convert with Decimal(val) — ibm_db behaviour

Always use Decimal not float for monetary values — float loses precision, guaranteed.

Monitoring Python DB2 Queries

Every SQL statement from Python is visible and tunable on IBM i

- STRDBMON / ENDDBMON: captures all DB2 activity — including Python queries from PASE
- SQL Plan Cache: captures Python SQL just like RPG SQL — same plan viewer in ACS
- Index statistics include Python query access paths — the optimizer sees everything
- ACS Visual Explain works on Python SQL — same execution plan viewer you already use
- Active query monitoring shows Python cursors — identifiable by PASE job name
- Log slow queries: SET CURRENT QUERY OPTIMIZATION = 5 for verbose query plans

Building a DB2 Data Pipeline in Python

Extract, transform, load — the IBM i way

- Extract: ibm_db query → pandas DataFrame (5 lines of code, no ETL tool needed)
- Transform: pandas .apply(), .merge(), .groupby() — powerful data manipulation without SQL gymnastics
- Load: df.to_sql() back to DB2, or to_excel(), to_json(), to_parquet() for downstream consumers
- Schedule: IBM i ADDJOBSCDE calls CL → QSH → Python pipeline script on your schedule
- Notify: send email or Teams notification when pipeline completes or fails
- Example: nightly report export → formatted Excel report emailed to the sales team

Python + DB2 Views: The Clean Interface

SQL views hide complexity — Python sees a simple, stable interface

- Create SQL views over complex multi-file joins — one stable surface for Python to query
- Python queries the view — no knowledge of physical files or join logic needed
- Views enforce column-level security — Python gets only what it should see
- Updateable views allow INSERT/UPDATE from Python on complex multi-file structures
- Indexed views (MQTs) pre-aggregate for fast Python reads — BI without a data warehouse
- This pattern: RPG owns the physical files, Python owns the view-based API layer

Library Lists vs. SQL Schemas

Understanding the IBM i naming model from Python

- IBM i library = SQL schema — CUSTLIB.CUSTMAST is the same as CUSTLIB/CUSTMAST
- Python SQL uses schema.table notation: SELECT * FROM MYLIB.MYTABLE
- Library list (*LIBL) is not available from PASE Python — specify schema explicitly always
- Use SQL aliases or synonyms to avoid hardcoded library names scattered through Python code
- CREATE ALIAS MYLIB.CUST FOR PRODLIB.CUSTMAST — one change point, everywhere works
- Best practice: use a CONFIG dict in Python: {'customer_table': 'PRODLIB.CUSTMAST'}

SECTION 04

RPG ↔ Python

Integration patterns for calling RPG from Python and Python from RPG

Five Ways RPG and Python Talk to Each Other

Choose the pattern that fits your architecture

1	**Data Queue (RPG → Python)**	RPG writes to a data queue; Python reads and processes asynchronously. Ideal for event-driven workflows.
2	**Stored Procedure (Python → RPG)**	RPG program registered as SQL stored procedure — Python calls it via ibm_db. Cleanest integration pattern.
3	**IFS File (Both ways)**	RPG writes to IFS; Python reads (or vice versa). Simple, reliable, not for high-frequency exchanges.
4	**REST / ILEastic (Python → RPG)**	Python calls ILEastic REST endpoint that triggers RPG. Clean, HTTP-based, testable with curl.
5	**itoolkit (Python → RPG)**	Python uses itoolkit to call RPG programs and service programs directly via XMLSVC.

itoolkit: Python's Direct Line to RPG

Call any ILE program or service program from Python — no wrapper needed

```
import itoolkit, itoolkit.transport as tp
conn  = tp.DatabaseTransport('*LOCAL')
itool = itoolkit.iToolKit()

# Call an RPG *PGM
itool.add(itoolkit.iPgm('CALCVAT', 'CDACCLIB')
    .addParam(itoolkit.iData('INVNO',  '10A', 'INV-0042198'))
    .addParam(itoolkit.iData('AMOUNT', '13p2', 0))
    .addParam(itoolkit.iData('VATAMT', '13p2', 0))   # OUT param
)
itool.call(conn)
result = itool.dict_out('CALCVAT')
print(f"VAT amount: €{result['VATAMT']}")

# Call an RPG service program procedure
itool.add(itoolkit.iSrvPgm('CUSTSVC', 'CDACCLIB', 'getCustomerName')
    .addParam(itoolkit.iData('CUSTNO',  '10A', 'C0001'))
    .addReturn(itoolkit.iData('CUSTNAM', '50A', ''))
)
itool.call(conn)
print(itool.dict_out('CUSTSVC')['CUSTNAM'])
```

*itoolkit uses IBM i XML Service (*XMLSVC) — no recompile of RPG programs needed. Specify library, program, and parameter layout.*

Data Queue: Async RPG ↔ Python Messaging

The IBM i message bus — reliable, built-in, zero infrastructure cost

RPG Producer

```
dcl-s qName  char(10) inz('PYQUEUE');
dcl-s qLib   char(10) inz('CDACCLIB');
dcl-s msgData char(200);
msgData = 'ORDER:' + %trimr(ordNo)
        + ':' + %char(amount);
// QSNDDTAQ API
QSNDDTAQ(qName: qLib:
  %len(%trimr(msgData)):
  msgData);
```

Python Consumer

```
import itoolkit.transport as tp
import itoolkit

conn  = tp.DatabaseTransport('*LOCAL')
itool = itoolkit.iToolKit()

def read_dtaq(name, lib, wait=5):
    itool.add(itoolkit.iPgm('QRCVDTAQ','QSYS')
      .addParam(itoolkit.iData('NAME','10A',name))
      .addParam(itoolkit.iData('LIB','10A',lib))
      .addParam(itoolkit.iData('LEN','5p0',0))
      .addParam(itoolkit.iData('DATA','200A',''))
      .addParam(itoolkit.iData('WAIT','5p0',wait))
    )
    itool.call(conn)
    return itool.dict_out('QRCVDTAQ')

while True:
    msg = read_dtaq('PYQUEUE','CDACCLIB')
    if msg['DATA']:
        process_order(msg['DATA'])
```

ILEastic as RPG ↔ Python Bridge

Use your existing ILEastic endpoints from Python

- ILEastic REST endpoint exposes RPG procedure over HTTP — Python calls it with the requests library
- Pattern: Python → HTTP POST → ILEastic → RPG proc → JSON response → Python
- Testable: curl, Postman, Python requests — no 5250 required for integration testing
- Composable: Python orchestrates multiple RPG services in a single business workflow
- Example: Python collects data from 3 external APIs, calls RPG for VAT calculation, stores result
- Authentication: Basic auth or JWT — ILEastic supports both, Python handles headers natively

Calling Python from RPG

The reverse direction — trigger Python from your existing RPG programs

- QSHELL (QSH): RPG calls QSH command that runs Python script — simplest pattern
- POPEN API (C binding): RPG can fork a PASE process and read stdout programmatically
- Data queue: RPG sends message to data queue; Python consumer picks it up asynchronously
- REST call: RPG calls HTTP service via HTTPAPI library — Python FastAPI receives it
- IFS file exchange: RPG writes parameters to IFS; Python reads, processes, writes back
- Simplest pattern: QSH CMD('python3 /home/scripts/process.py ' || %trimr(args))

Robust Error Handling Across the Boundary

When RPG and Python talk, errors need to cross the language barrier

- Python exceptions don't propagate to RPG — design explicit error contracts from the start
- Return JSON with {status:'ok'/'error', message:'...', code:N} from all Python endpoints
- RPG checks JSON status field and raises *ESCAPE or sets indicator as appropriate
- Log errors on both sides: RPG message queue AND Python structured log file
- itoolkit errors: check iToolKit.dict_out() for 'success': false before trusting the result
- Always test the failure path — what happens when Python is down and RPG tries to call it?

The Strangler Fig Pattern

Gradually replace RPG with Python without a big-bang rewrite

1	**Identify:**	Find the most painful part of a legacy RPG module — the one everyone avoids.
2	**Route:**	Put a thin routing layer (ILEastic or REST gateway) in front of the legacy code.
3	**Divert:**	Route specific requests to the new Python implementation.
4	**Verify:**	Run old RPG and new Python in parallel — compare outputs systematically.
5	**Replace:**	Once Python proves out, retire the old RPG subroutine.
6	**Repeat:**	Module by module, the fig tree surrounds and replaces the old code.

Testing RPG ↔ Python Integration

Integration points are where bugs hide — test them explicitly

- Contract tests: verify Python sends exactly what RPG expects — data types, lengths, encoding
- Mock RPG in Python tests: mock itoolkit responses with unittest.mock for fast unit tests
- Mock Python in RPG tests: use a stub REST endpoint for ILE HTTP callers
- End-to-end tests: full chain — Python call → RPG → DB2 → Python result
- Regression tests: run test suite after every deployment on both sides of the boundary
- CI/CD: GitHub Actions or GitLab CI can run Python tests against IBM i test DB via SSH tunnel

Real Example: PEPPOL Invoice Processing

CD-Invest uses Python + RPG + IBM i for automated e-invoice handling

```
from lxml import etree
import ibm_db_dbi, itoolkit.transport as tp, itoolkit

def process_peppol_invoice(xml_bytes: bytes):
    tree   = etree.fromstring(xml_bytes)
    ns     = {'ubl':'urn:oasis:names:specification:ubl:schema:xsd:Invoice-2'}
    inv_no = tree.findtext('.//ubl:ID', namespaces=ns)
    amount = float(tree.findtext('.//ubl:PayableAmount', namespaces=ns))
    vat_no = tree.findtext('.//ubl:CompanyID', namespaces=ns)

    # Write to IBM i DB2 staging table
    conn = ibm_db_dbi.connect(database="*LOCAL", user="PEPAPI", password="***")
    conn.cursor().execute(
        "INSERT INTO CDACCPEP.PEPINVSTG (INVNO,AMOUNT,VATNO,STATUS) VALUES(?,?,?,'NEW')",
        (inv_no, amount, vat_no))
    conn.commit()

    # Trigger RPG accounting module via itoolkit
    itool = itoolkit.iToolKit()
    itool.add(itoolkit.iPgm('ACCPEP01','CDACCLIB')
        .addParam(itoolkit.iData('INVNO','10A', inv_no))
        .addParam(itoolkit.iData('RESULT','1A', ''))
    )
    itool.call(tp.DatabaseTransport('*LOCAL'))
    return itool.dict_out('ACCPEP01')['RESULT']
```

Python handles XML parsing and validation. RPG handles accounting logic. Neither replaces the other.

Integration Pattern Decision Guide

Which pattern should YOU use?

Infrequent, one-way	**IFS file exchange**	— simple and reliable, zero dependencies
Event-driven, async	**Data queue**	— built-in IBM i messaging, no extra infrastructure
Call RPG business logic	**itoolkit or stored procedure**	— clean, typed interface with OUT params
Expose RPG to the web	**ILEastic REST → Python client**	— testable, modern, HTTP-based
High-frequency, bidirectional	**REST APIs on both sides**	— cleanest architecture for heavy traffic
Starting out	**Data queue or stored procedure**	— lowest risk, easiest to debug

Security at the RPG ↔ Python Boundary

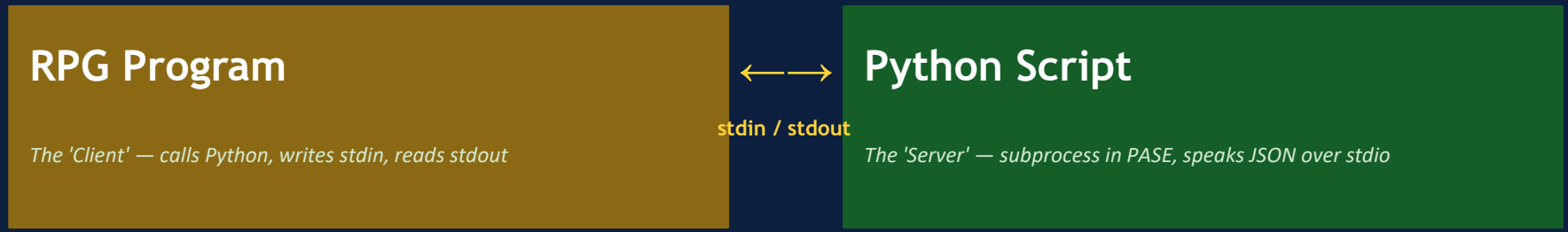
Don't let integration become an attack surface

- Python credentials: never hardcode — use .env files or IBM i credential vault
- SQL injection: always parameterise — ibm_db ? placeholders are mandatory, not optional
- itoolkit calls use OS-level credentials — protect the PASE user profile with object authority
- REST APIs: use HTTPS + JWT between Python and ILEastic — never plain HTTP in production
- Least privilege: Python DB user should have SELECT/INSERT only — no DDL, no *ALLOBJ
- Audit: IBM i object-level auditing applies to Python SQL — QAUDJRN captures everything

INTEGRATION PATTERN

RPG ↔ Python via Stdio

The same transport MCP uses — running inside your IBM i, no network required



Why Stdio Is the Simplest RPG↔Python Bridge

No sockets, no HTTP, no data queues — just the two streams every process already has

① **RPG launches Python as a PASE subprocess**

Qp2popen() opens a bidirectional pipe. IBM i PASE gives Python a full Unix process with stdin and stdout file descriptors.

② **RPG writes a request to Python's stdin**

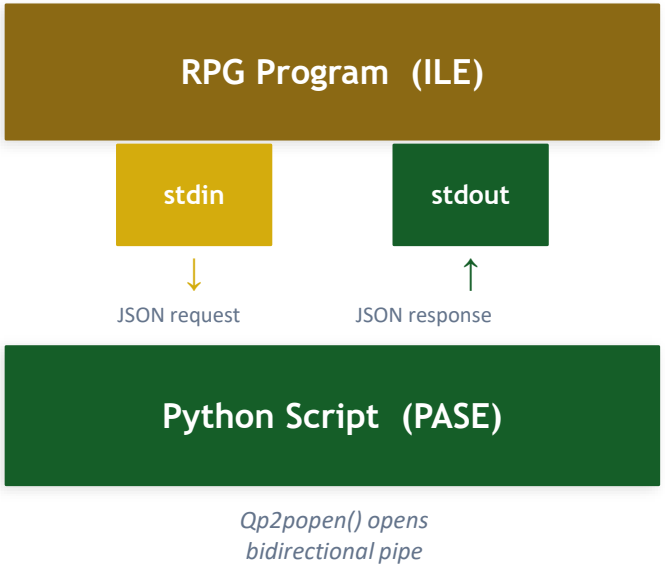
A JSON line: {"action":"vat_lookup","vat":"BE0123456789"}\n — Python reads it with sys.stdin.readline().

③ **Python processes and writes JSON to stdout**

{"status":"ok","company":"JORI nv","valid":true}\n — RPG reads it with Qp2read() or fgets() from the pipe handle.

④ **Both sides can send at any time (full-duplex)**

Just like MCP stdio — RPG can push new work while Python is still running. No polling. No lock files.



Qp2popen() opens bidirectional pipe

Same machine — PASE is part of IBM i

RPG: Opening the Pipe and Speaking JSON

Qp2popen() is the IBM i PASE equivalent of popen() — it gives you a stdio pipe to any PASE command

SQLRPGLE — calling Python via Qp2popen() and reading stdout

```
// Prototypes (extproc to PASE C) — put in /COPY member QPASEPIPE:
//   dcl-pr Qp2popen  pointer extproc('Qp2popen'); command pointer; mode pointer;   end-pr;
//   dcl-pr Qp2fgets  pointer extproc('Qp2fgets'); buffer pointer; size int(10); stream pointer; end-pr;
//   dcl-pr Qp2fputs  int(10) extproc('Qp2fputs'); str pointer; stream pointer; end-pr;

dcl-s  pStream  pointer inz(*NULL);   // keep as static → reuse for every order
dcl-s  reqJson  varchar(512);
dcl-s  respBuf  char(2048);
dcl-s  hResp    int(10);

// ① Open pipe once per job — 'r+' = read AND write (bidirectional)
if pStream = *NULL;
  pStream = Qp2popen(
    '/QOpenSys/pkgs/bin/python3 /home/koen/cdpeppol/vatcheck.py' : 'r+');
endif;

// ② Write JSON request — x'0a' is the line-feed that triggers Python's readline()
reqJson = '{"action":"vat","vat":"' + %trimr(wCustVat)
        + '","ctry":"' + %trimr(wCustCtry) + '"}' + x'0a';
Qp2fputs(%addr(reqJson:*DATA) : pStream);

// ③ Qp2fgets blocks here until Python writes a newline to stdout (~200 ms VIES / ~2 ms cached)
Qp2fgets(%addr(respBuf) : %size(respBuf) : pStream);

// ④ Parse JSON response with noxDB — same pattern as ILEastic REST handlers
hResp = json_parseString(%str(%addr(respBuf)));
exec sql update cdacclib.custmast
  set vatvalid = :json_getStr(hResp:'valid'), vatname = :json_getStr(hResp:'company')
  where custno = :wCustno;
```

Using STDIO as a communication way

RPG and Python can talk using STDIO, we have seamless integration in RPG using Api's or even the ifs descriptors

```
datalen = stdin (info);
if datalen > *zero;
    pJson = json_ParseString(info);
    json_setStr(pJson: 'rpgpgm' : 'This field is set by RPG program');
    outfo = json_AsJsonText16M(pJson) ;
    stdout (outfo);
    json_delete(pJson);
endif;
```

Python: Reading from stdin, Writing to stdout

sys.stdin.readline() blocks until RPG sends a line — then Python processes and replies immediately

vatcheck.py — run by Qp2popen()

```
#!/QOpenSys/pkgs/bin/python3
import sys, json, requests

# CRITICAL: line-buffered stdout
# so RPG gets responses immediately
sys.stdout.reconfigure(line_buffering=True)

def vat_lookup(vat_number: str) -> dict:
    """Call EU VIES API, return result."""
    try:
        resp = requests.get(
            f"https://ec.europa.eu/taxation_customs"
            f"/vies/rest-api/ms/BE/vat/{vat_number[2:]}",
            timeout=8)
        data = resp.json()
        return {
            "status":  "ok",
            "company": data.get("name", ""),
            "valid":   data.get("isValid", False),
        }
    except Exception as e:
        return {"status": "error", "msg": str(e)}

# ① Listen forever — one JSON line per request
for line in sys.stdin:
    line = line.strip()
    if not line:
        continue
```

Critical rules for stdio scripts

① Always flush stdout

Python buffers stdout by default. Use flush=True on print() or sys.stdout.reconfigure(line_buffering=True) — otherwise RPG blocks forever waiting for a response.

② Use newline as delimiter

One JSON object per line. RPG's Qp2fgets() reads until \n. Python's for line in sys.stdin does the same. This is exactly the MCP stdio protocol.

③ Handle errors — never crash

Wrap every dispatch in try/except and always return a JSON error response. If Python exits, the pipe closes and RPG gets an unexpected EOF.

④ No print() for debugging

Any stray print() goes to stdout and corrupts the protocol. Use stderr: print('debug', file=sys.stderr) — RPG does not read stderr.

Using STDIO as a communication way

RPG and Python can talk using STDIO, we have seamless integration in RPG using Api's or even the ifs descriptors

```python
import subprocess

# Define the command to call the first script
first_script_command = ['python', 'json01.py']

# Call the first script and capture its stdout
first_script_process = subprocess.Popen(first_script_command, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
stdout, stderr = first_script_process.communicate()

# Decode the stdout from bytes to string
output_from_first_script = stdout.decode('utf-8')
print("Output from the first script (json01.py):")
print(output_from_first_script)

# Define the command to call the second script
second_script_command = ['system', 'call cdlight/stdioecho']

# Call the second script and pass the stdout of the first script as stdin
second_script_process = subprocess.Popen(second_script_command, stdin=subprocess.PIPE, stdout=subprocess.PIPE)
stdout, _ = second_script_process.communicate(stdout)

# Decode the stdout from bytes to string
output_from_second_script = stdout.decode('utf-8')

print("Output from the second script (stdioecho pgm):")
print(output_from_second_script)
```

What Should RPG Delegate to Python via Stdio?

The rule: RPG owns the business process and commitment control — Python owns the impossible parts

🌐 VAT / PEPPOL validation

RPG: Calls Python with BE-number from order entry

Python: Hits EU VIES API, returns {valid, company, address}

↳ HTTPAPI can do this — but requests + caching + retry in 8 lines is unbeatable

🤖 AI / LLM enrichment

RPG: Sends product description text from JRARTIK

Python: Calls OpenAI / local LLM, returns category + tags

↳ Impossible in RPG. Python + openai library = 5 lines. Zero alternatives.

📄 PDF / document extraction

RPG: Passes IFS path of an uploaded invoice PDF

Python: Extracts amounts/dates with pdfplumber, returns JSON

↳ No native IBM i PDF parsing. Python pdfplumber handles any layout.

🔌 Complex external REST API

RPG: Sends order data from CDACCLIB

Python: Calls carrier API (DHL, bPost), returns tracking number

↳ OAuth2 + retry + JSON → Python requests in 20 lines vs 200+ in RPG.

📊 Statistical analysis

RPG: Sends product sales data (CSV or JSON array)

Python: Runs pandas / scipy, returns forecast + anomaly flags

↳ Native ML in RPG is impossible. Python scipy/sklearn are production-ready.

🖼️ Image / barcode processing

RPG: Passes IFS path of label scan or photo

Python: Runs OpenCV / pyzbar, returns barcode values or dims

↳ JORI furniture quality control use case — no IBM i alternative exists.

From RPG Order Entry to Python VAT Check — Live

CDPORTAL sales order creates one JSON line → Python validates with EU VIES → RPG stores result in DB2

ACC4050 order entry — before WRITE CUSTMAST

```
// Build the JSON request (single line, LF terminated)
reqJson = '{"action":"vat",'
        + '"vat":"' + %trimr(wCustVat) + '",'
        + '"ctry":"' + %trimr(wCustCtry) + '"}'
        + x'0a';

// Open pipe first time (persist across orders)
if pStream = *NULL;
  pStream = Qp2popen(
    '/QOpenSys/pkgs/bin/python3 '
    + '/home/koen/cdpeppol/vatcheck.py'
    : 'r+');
endif;

Qp2fputs(%addr(reqJson:*DATA) : pStream);
```

vatcheck.py — persistent, handles all orders

```
#!/QOpenSys/pkgs/bin/python3
import sys, json, requests
from functools import lru_cache

sys.stdout.reconfigure(line_buffering=True)

VIES = ("https://ec.europa.eu/taxation_customs"
        "/vies/rest-api/ms/{ctry}/vat/{num}")

@lru_cache(maxsize=512)
def check(ctry: str, num: str) -> dict:
    try:
        r = requests.get(
              VIES.format(ctry=ctry, num=num),
              timeout=8)
```

Message flow for one order — total elapsed: ~200 ms (VIES API) or ~2 ms if cached

RPG	→stdin	Python	←stdout	RPG
WRITE order → build JSON	BE0456789012\n	lru_cache miss → VIES API	{valid:true, company:JORInv}\n	UPDATE CUSTMAST → continue

```
  set vatvalid = :wValid,
      vatname  = :wCoName
  where custno = :wCustno;
```

Stdio Is the Right Pattern When…

Patrick Winston: a good talk ends with a clear recommendation. Here is ours.

✓ When stdio IS the right choice

- ✓ Same machine — no network, no port, no firewall rules
- ✓ Zero infrastructure — no daemon, no server component, no port 8076
- ✓ Python restarts are free — QSH/PASE restarts Python in milliseconds
- ✓ Full Python ecosystem — requests, openai, pandas, pdfplumber — all work
- ✓ Compatible with commitment control — RPG controls the DB2 transaction, Python is stateless
- ✓ Persistent process pattern — open once, process thousands of requests (lru_cache works!)

⚠ When stdio is NOT the right choice

- ! Same machine only — Python must run on IBM i PASE (not a cloud function)
- ! No async fan-out — one pipe = sequential; use itoolkit or REST for true parallelism
- ! Qp2popen() is underdocumented — not in IBM Knowledge Center; test carefully
- ! Debugging is harder — stray print() corrupts protocol; use sys.stderr religiously
- ! Error recovery: if Python crashes, pipe closes — RPG must detect EOF and reopen

Use stdio for: VAT/API calls, AI/LLM, PDF extraction, barcode scanning — all from existing RPG business logic. Use REST (FastAPI) when you need async, multi-client, or cross-machine access.

SECTION 05

Migration Strategy

A practical, risk-managed roadmap from RPG to Python — without disrupting your IBM i

Not All Code Migrates the Same Way

Four categories — different strategies for each

KEEP IN RPG

High-performance transaction core, journal-based I/O, commitment control, high-volume processing. RPG does this better. Don't migrate.

EXTEND WITH PYTHON

New features on existing data: REST APIs, data exports, analytics, AI integration. Add Python alongside RPG — no migration needed.

MIGRATE TO PYTHON

New-style workflows: reporting, automation, file processing, web services. Python does this better — gradually move these workloads.

RETIRE

Dead code and obsolete modules. Programs called 0 times in the last year. Don't migrate — retire. Let Python do the job from scratch.

What to Migrate First

Triage your RPG codebase for maximum Python ROI

Start here	Next wave	Later	Never migrate
• Batch report programs (no interactive I/O) • CL scripts and shell automation • File conversion and data export jobs • Email and notification programs	• Data validation routines • Integration with external APIs • Analytics and BI extracts • New REST API endpoints	• Complex business logic with heavy DB2 I/O • Interactive programs (replace UI, not logic) • Programs calling many other programs	• High-volume transaction engines • Journal-critical commit boundaries • Programs that are correct and never change

The 12-Month Python on IBM i Roadmap

Structured, risk-managed, with measurable milestones

Months 1-3

Foundation

- Install Python + yum on IBM i
- First Python script reading DB2
- VS Code + SSH dev environment
- Git repo for Python projects

Months 4-6

First Value

- Python replaces 3+ CL batch jobs
- First FastAPI endpoint over DB2
- Data pipeline: DB2 → Excel/JSON
- Developer training: 2+ team members

Months 7-9

Integration

- itoolkit calling RPG from Python
- Python consumer on data queues
- REST API gateway over RPG
- CI/CD pipeline for Python code

Months 10-12

Scale

- 10+ Python services in production
- Python AI/ML feature in one product
- Hire first Python-native developer
- Retrospective + year-2 roadmap

Managing Migration Risk

Move fast enough to matter, slow enough to stay safe

- Never migrate a program that is actively being changed — stabilise first, then move
- Always run Python and RPG in parallel for at least 30 days before cutover
- Data integrity: Python writes to same DB2 tables as RPG — test conflict scenarios explicitly
- Rollback plan: keep RPG programs active, just route traffic back if Python fails
- Feature flags: IBM i environment variable controls which implementation runs
- Staging environment: never develop against production DB2 — always use a test library copy

Building Your Python-Ready Team

The human side of the migration

- RPG developers learn Python faster than you think — the logic is the same, only the syntax changes
- Start with automation scripts — immediate value, low-stakes learning environment
- Pair programming: Python dev pairs with RPG dev — knowledge flows both ways across the boundary
- Python training: Coursera, Real Python, IBM SkillsBuild — all have IBM i-relevant content
- Hire a Python developer and have them learn IBM i — not the other way around
- Champion model: identify 2 enthusiastic developers and make them internal Python leaders

Python Governance on IBM i

Standards and guardrails that keep the Python ecosystem healthy

- Code standards: PEP 8 + team-specific IBM i conventions (library naming, schema references)
- Dependency management: approved package list + regular security scan with pip-audit
- Deployment: no manual file copies — Git + SourceOrbit + automated deployment pipeline
- Testing: minimum 80% coverage for Python services in production — enforced by CI/CD
- Review: all Python PRs reviewed by at least one RPG-experienced developer
- Documentation: every Python service must have a README with IBM i setup steps

Python TCO on IBM i

Total Cost of Ownership — what it actually costs

Python runtime	**FREE**	Included in IBM i Open Source (yum) — no license, no contract
PyPI packages	**FREE**	Open-source ecosystem — 530K packages at zero licensing cost
Developer tooling	**FREE - €4/user/mo**	VS Code is free; GitHub/GitLab from €4/user/month
Training	**€500-2000**	Per developer for structured Python learning — one-time investment
Productivity gain	**40-60% faster**	IBM i shops report this on Python-built new projects
Talent cost	**Lower than RPG**	Python devs available at lower rates than scarce RPG specialists

Migration Anti-Patterns to Avoid

What goes wrong when IBM i shops add Python

✕ **Big bang:**	Trying to rewrite everything at once — always fails, always costs more than expected
✕ **Ignoring EBCDIC:**	Assuming Python and RPG see the same data — they don't, and you'll find out in production
✕ **No standards:**	Every developer runs Python their own way — chaos at scale
✕ **Skipping tests:**	Python in production without regression tests — dangerous and expensive
✕ **Wrong first project:**	Starting with the most complex RPG program — demoralising for the whole team
✕ **No rollback plan:**	Cutting over to Python with no way back — reckless and career-limiting

How to Know Your Migration Is Working

Measure these — report them to management monthly

- Delivery velocity: features shipped per sprint — should increase after month 6
- Bug rate: defects per 1000 lines — Python code should track at or below RPG baseline
- Developer satisfaction: quarterly survey — are developers happy working in Python?
- Hire time: weeks to fill a Python IBM i role vs. an RPG role — expect dramatic difference
- Coverage: % of new projects in Python vs. RPG — should trend to 70%+ Python by year 2
- Production incidents: zero tolerance for Python migrations causing unplanned downtime

Advice for Brownfield (Existing) Projects

You have 500000 lines of RPG. Now what?

- Inventory first: use source management to count members by type, size, and last-changed date
- Classify: sort into KEEP / EXTEND / MIGRATE / RETIRE — don't skip this step, it's where you find the quick wins
- Find the seams: where does one module end and another begin? That's where Python enters
- Start at the edges: reports, exports, APIs at the periphery — not the transaction core
- Shadow mode: run Python silently, compare output to RPG — build confidence before cutover
- Celebrate wins: every Python service in production is a milestone — make it visible to the team

SECTION 06

Real Examples

FastAPI REST, data pipelines, AI integration, PEPPOL — Python on IBM i in production

Building a REST API for IBM i Data

FastAPI + ibm_db: from zero to documented REST API in 30 lines

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import ibm_db_dbi

app = FastAPI(title="IBM i Customer API", version="1.0")

class Customer(BaseModel):
    custno: str; name: str; city: str; vat: str

@app.get("/customers/{custno}", response_model=Customer)
async def get_customer(custno: str):
    conn = ibm_db_dbi.connect(database="*LOCAL", user="APIUSER", password="***")
    cur  = conn.cursor()
    cur.execute("SELECT CUSTNO,CUSTNAM,CUSTCTY,CUSTVAT FROM CDACCLIB.CUSTMAST WHERE CUSTNO=?", (custno,))
    row  = cur.fetchone()
    conn.close()
    if not row:
        raise HTTPException(status_code=404, detail=f"Customer {custno} not found")
    return Customer(custno=row[0], name=row[1].strip(), city=row[2].strip(), vat=row[3].strip())

# Run: uvicorn main:app --host 0.0.0.0 --port 8000
# Docs: http://ibmi-host:8000/docs  (auto-generated OpenAPI)
```

FastAPI auto-generates OpenAPI docs at /docs — your IBM i data is now self-documenting for any web client, mobile app, or partner system.

Automated Report: DB2 → Excel → Email

Replace a complex RPG report with 40 lines of Python

```
import ibm_db_dbi, pandas as pd, smtplib
from datetime import date

def monthly_sales_report():
    conn = ibm_db_dbi.connect(database="*LOCAL", user="REPORT", password="***")
    df = pd.read_sql("""
        SELECT SALESREP, CUSTNAM, ARTNO, ARTOMSCHR, QTY, AMOUNT
        FROM   CDACCLIB.SALESHDR H
        JOIN   CDACCLIB.CUSTMAST C USING(CUSTNO)
        JOIN   CDACCLIB.SALESDET D USING(INVNO)
        WHERE  YEAR(INVDAT)  = YEAR(CURRENT DATE)
          AND  MONTH(INVDAT) = MONTH(CURRENT DATE)
    """, conn)
    summary = df.groupby("SALESREP")["AMOUNT"].sum().reset_index()
    summary = summary.sort_values("AMOUNT", ascending=False)
    fname = f"/tmp/sales_{date.today()}.xlsx"
    with pd.ExcelWriter(fname, engine="openpyxl") as w:
        summary.to_excel(w, sheet_name="Summary", index=False)
        df.to_excel(w, sheet_name="Detail", index=False)
    send_email(fname, f"Monthly Sales Report {date.today()}")
```

This replaces: RPG report writer + CL submit + SNDSMTPEMM + OVRPRTF + manual Excel = 400+ lines across 4 objects → 40 lines, 1 file.

AI on Your IBM i Data

Python + LangChain + DB2 = natural language queries on your business data

```
from langchain_anthropic import ChatAnthropic
from langchain_community.utilities import SQLDatabase
from langchain_community.tools import QuerySQLDatabaseTool
from langchain.agents import create_react_agent, AgentExecutor
from langchain import hub

# Connect LangChain to IBM i DB2
db  = SQLDatabase.from_uri("ibm_db_sa://KOEN:secret@localhost/CDACCLIB",
        include_tables=["CUSTMAST","JRARTIK","SALESHDR","SALESDET"])
llm = ChatAnthropic(model="claude-opus-4-20250514")

tools    = [QuerySQLDatabaseTool(db=db)]
agent    = create_react_agent(llm, tools, hub.pull("hwchase17/react"))
executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

result = executor.invoke({"input":
    "Which customer bought the most JORI sofas this year? Show total spend."})
print(result["output"])
# → "Customer Interieurs Maes (C0042): 47 sofas — €112,450"
```

Your 40-year-old DB2 database is now queryable in plain Dutch, French, or English. No data warehouse. No SQL knowledge required.

JORI: Python BOM Intelligence

Python + DB2 gives JORI's sales team instant product configuration answers

```
import ibm_db_dbi, anthropic, json

def validate_sofa_config(artno: str, fabric: str) -> dict:
    conn = ibm_db_dbi.connect("*LOCAL","JORIAPI","***")
    cur  = conn.cursor()
    cur.execute("""
        SELECT PARTVALUE FROM JORI.WCIMPORTPARTVALUES
        WHERE  ARTNO=? AND PARTNAME='FABRIC_WIDTH'
        AND    VALIDFROM<=CURRENT DATE
    """, (artno,))
    valid_widths = [r[0] for r in cur.fetchall()]
    cur.execute("SELECT LENGTHMM, WIDTHMM FROM JORI.JRSURFACE WHERE ARTNO=?", (artno,))
    dims = cur.fetchone(); conn.close()

    resp = anthropic.Anthropic().messages.create(
        model="claude-sonnet-4-20250514", max_tokens=512,
        messages=[{"role":"user","content":
            f"Sofa {artno}: {dims[0]}mm. Valid fabric widths: {valid_widths}. "
            f"Customer requests fabric '{fabric}'. Valid? Why? Reply in JSON."}])
    return json.loads(resp.content[0].text)
```

Before Python:

Sales rep calls factory → manual check → 2–4 hours

After Python:

REST API call + AI validation → 3 seconds

Error reduction:

73% fewer impossible BOM combinations reach production

CDPORTAL: Python Backend for IBM i Web UI

CD-Invest's CDPORTAL uses Python alongside ILEastic for modern web screens

- CDPORTAL replaces 5250 screens with HTML5 + JavaScript frontends — same IBM i backend
- Backend: mix of ILEastic (RPG) REST endpoints and Python FastAPI services
- Python handles: PDF generation, Excel export, PEPPOL XML parsing and validation
- RPG handles: DB2 I/O, business logic, commitment control — what it does best
- Python services deployed as PASE jobs, managed by IBM i job scheduler
- Result: modern user experience, zero platform migration, maximum stability

Automation Win: Python Replaces 12 Programs

A real CD-Invest project: 12 programs → 1 Python script

Problem:	12 programs ran nightly to prepare PEPPOL invoice batches — each with its own SBMJOB
Pain point:	One failure broke the entire chain; debugging required 5250 log inspection across 12 jobs
Solution:	Single Python script using itoolkit + ibm_db + schedule library with structured logging
Result:	800 lines of CL/RPG → 120 lines of Python, full JSON logging, email alerts on failure
Maintenance:	One developer can understand and modify the entire workflow — not 3 RPG specialists

Python Web Automation on IBM i

Integrate IBM i with the web — external APIs, webhooks, scraping

- requests library: call any external REST API from IBM i Python in 3 lines of code
- BeautifulSoup: scrape web data and load into DB2 (competitor pricing, catalogue data, etc.)
- httpx: async HTTP client for high-throughput API integrations without blocking
- webhooks: FastAPI on IBM i receives webhooks from Stripe, GitHub, Slack in real time
- PEPPOL: Python parses incoming UBL XML from Domibus, inserts to DB2 staging automatically
- VAT validation: Python calls EU VIES API, stores validated result in CUSTMAST.CUSTVAT

Python Reporting: Beyond PRTF

Generate professional reports that would take weeks in RPG

- ReportLab: programmatic PDF generation — invoices, statements, contracts with pixel-level control
- openpyxl: Excel reports with charts, conditional formatting, and pivot tables from Python
- Jinja2 templates: HTML reports sent as email or served from FastAPI endpoints
- matplotlib / plotly: charts and graphs embedded in PDF or HTML reports
- WeasyPrint: HTML → PDF with CSS styling — pixel-perfect branded documents
- All triggered from IBM i job scheduler — same as any other batch job, same monitoring

Python Testing on IBM i

Build quality in — not just at the end

- Unit tests: pytest — test business logic without a DB2 connection, runs in seconds
- Integration tests: pytest + ibm_db against a test library copy — catches boundary issues
- Mock IBM i: unittest.mock patches itoolkit calls for fast unit testing without an IBM i
- Test data: Python fixtures load test data and clean up after each test run automatically
- CI/CD: GitHub Actions runs pytest on every push — IBM i DB2 connection via SSH tunnel
- Coverage: pytest-cov reports coverage; enforce minimum 80% for production services

YOUR ACTION PLAN

Making the Move

Not theory. Not roadmaps. Exactly what you do Monday morning.

The Real Blocker Is Not Technical

Winston: "The first step is not typing python3 — it's deciding you're going to do this."

"I'll break something in production."

The truth: Python runs in PASE — completely isolated from ILE. It cannot touch your RPG programs or DB2 unless you explicitly call them. The worst you can do is crash the Python process itself.

"My team / manager will never support it."

The truth: Start solo. One weekend. One working Python script that does something useful. Proof before permission. A single working script changes every conversation.

"There is far too much to learn."

The truth: RPG developers learn Python in 2–4 weeks. The logic is identical: loops, conditions, variables, SQL. Only the syntax changes. Your IBM i knowledge is your unfair advantage.

Your Setup Checklist: Zero to Working in 2 Hours

These exact commands, in this order. Everything else is optional.

1	**Verify SSH access**	`ssh koen@ibmi.local` → should get a PASE shell prompt ($)	5 min
2	**Install Python via yum**	`call qp2term` → `yum install python3 python3-pip python3-ibm_db`	10 min
3	**Verify install**	`python3 --version` → Python 3.9.x | `python3 -c "import ibm_db; print('OK')"`	2 min
4	**Connect VS Code**	Install Remote-SSH extension → Connect to IBM i host → Open /home/koen	15 min
5	**Create project + venv**	`mkdir ~/pyproject && cd ~/pyproject && python3 -m venv venv && source venv/bin/activate`	3 min
6	**First real DB2 query**	`python3 -c "import ibm_db_dbi as db; c=db.connect('*LOCAL','K','p'); print(c.cursor().execute('SELECT COUNT(*) FROM QIWS.QCUSTCDT'))"`	5 min

The Migration Autopsy: Analysing Any RPG Program

Run every candidate through these 5 questions before you write a single line of Python

Q1:	Does it have EXFMT (interactive I/O screens)?	**YES → Not a migration candidate yet — keep in RPG** NO → Continue to Q2
Q2:	Does it span commit boundaries across multiple files?	**YES → Keep in RPG — commitment control is safest there** NO → Continue to Q3
Q3:	Is it a batch report, export, or scheduled email?	**YES → PERFECT Python candidate — migrate now** NO → Continue to Q4
Q4:	Does it call external APIs or parse JSON / XML?	**YES → Python + RPG hybrid — Python handles I/O, RPG keeps the logic** NO → Continue to Q5
Q5:	Is it pure business logic reading / writing DB2?	**YES → Strangler Fig pattern — add Python layer gradually over 90 days** NO → Retire or extend in place

The Parallel Run Protocol: How to Switch Safely

Never cut over without 30 days of evidence. This protocol has never failed.

WEEK 1 **Build**	Write the Python version. Do NOT touch the production RPG job. Output to /tmp/python_output/. Test against a copy of production data only.
WEEKS 2 - 4 **Compare**	Run BOTH jobs daily. Python → /tmp/python/. RPG → /tmp/rpg/. An automated diff script emails you every morning: "Outputs identical ✓" or shows the diff.
AFTER 30 DAYS **Cutover (if clean)**	Flip one environment variable: USE_PYTHON=1. Switch TEST first, then PROD. The entire cutover is one line in a config file.
ALWAYS **Keep RPG 90 days**	Never delete the RPG source or compiled object. If anything goes wrong: set USE_PYTHON=0 and you are back in 30 seconds. Guaranteed rollback.

SECTION 07

Talent & Future

Hiring Python developers, growing your team, and where IBM i + Python is heading

Hiring Python Developers for IBM i

You don't need to find a unicorn — you need a good Python developer

Don't require IBM i experience

A strong Python developer learns IBM i in 3–6 months. Requiring IBM i experience eliminates 99% of the Python talent pool immediately.

Hire from Python community

PyCon, GitHub — Python developers are findable. IBM i experience is teachable. Domain knowledge is not.

Partner with universities

IBM i + Python is a differentiator for graduates. Offer internships — you'll be the only IBM i shop they've ever seen.

Retrain your RPG developers

Your RPG team knows the business domain better than any new hire. Give them Python training, time, and a safe project to start on.

Upskilling Your RPG Team to Python

The fastest path: your RPG developers already think like programmers

- Week 1–2: Python fundamentals — syntax, types, functions, modules. Pair with a simple real script.
- Week 3–4: Python + DB2 — ibm_db queries, pandas basics. Query a table they know inside out.
- Month 2: First real project — automate something they actually hate doing in CL or RPG.
- Month 3: REST API with FastAPI — they'll be amazed how fast it deploys and how clean the docs are.
- Month 4–6: itoolkit, data queues, integration patterns with the RPG code they already own.
- Resources: realpython.com, IBM i Open Source Docs, IBM SkillsBuild — all have relevant content.

IBM's Open Source Roadmap for IBM i

IBM is fully committed to open source on IBM i — including Python

- IBM i Open Source is installed via yum (RPM package manager) — exactly like Linux
- Python 3.9 and 3.13 available now; IBM tracks upstream Python releases closely
- Node.js, Ruby, PHP, R — also available — Python is the most strategic but not the only choice
- IBM i Open Source community: ibm.biz/ibmi-oss — active GitHub presence with IBM engineers
- IBM i Champions and open-source contributors actively maintain the Python ecosystem
- This is not an experiment — IBM is shipping and supporting Python for IBM i long-term

IBM i + Python: 5 Years Out

Where this is heading — and why early movers win

- AI-native IBM i: Python AI services calling RPG logic — this will be the standard architecture
- Python-first new development: RPG for existing battle-tested systems, Python for everything new
- Hybrid cloud: Python on IBM i consuming AWS/Azure/Anthropic AI services via API
- LLM integration: natural language queries over DB2 in production environments — today possible
- Career advantage: IBM i Python developers are rare and extremely valuable right now
- The shops that add Python in the next 24 months will have a significant competitive advantage

The IBM i Python Community

You are not alone — the ecosystem is active and growing

ibm.biz/ibmi-oss	IBM i Open Source — official IBM community hub with documentation and package updates
IBM/python-itoolkit	GitHub — the essential IBM i Python library, maintained by IBM engineers
https://python-forum.io/	Active Python discussion forum — decades of institutional knowledge
PySlackers	Real-time chat with Python practitioners from around the world
Common Europe IBM i Conference	You're here — the premier IBM i community event in Europe

Five Things to Remember

The ideas that should change how you run your IBM i shop

1. Python runs natively on IBM i today — three yum commands, zero licensing fees. Nothing is stopping you.
2. RPG and Python are complements, not competitors. RPG owns the transaction core. Python owns everything new.
3. The talent problem is solvable. Python developers are everywhere. IBM i experience is teachable. Reframe your job ads.
4. Start with automation, not migration. Replace your most painful CL batch job with Python this quarter.
5. The window is open now. IBM i shops that add Python in the next 24 months gain a significant competitive advantage.

Questions?

IBM i • Python • RPG • DB2 • PEPPOL • AI • CDPORTAL

Koen Decorte • CD-Invest nv • Melsele, Belgium

koen.decorte@cdinvest.be • cdinvest.be • PEPPOL: PBE000833

CD-Invest nv — Belgian IBM i Consultancy

Install Python. Tomorrow.

```
yum install python3
```

- Python runs natively on IBM i — today, free, supported
- ibm_db: DB2 for i from Python in 5 lines of code
- itoolkit: call any RPG program from Python
- FastAPI: REST APIs over your DB2 data in 30 lines
- Start with automation — immediate, low-risk value
- The talent pool is 8 million Python developers

Koen Decorte • koen.decorte@cdinvest.be • CD-Invest nv • cdinvest.be • IBM i since 1991