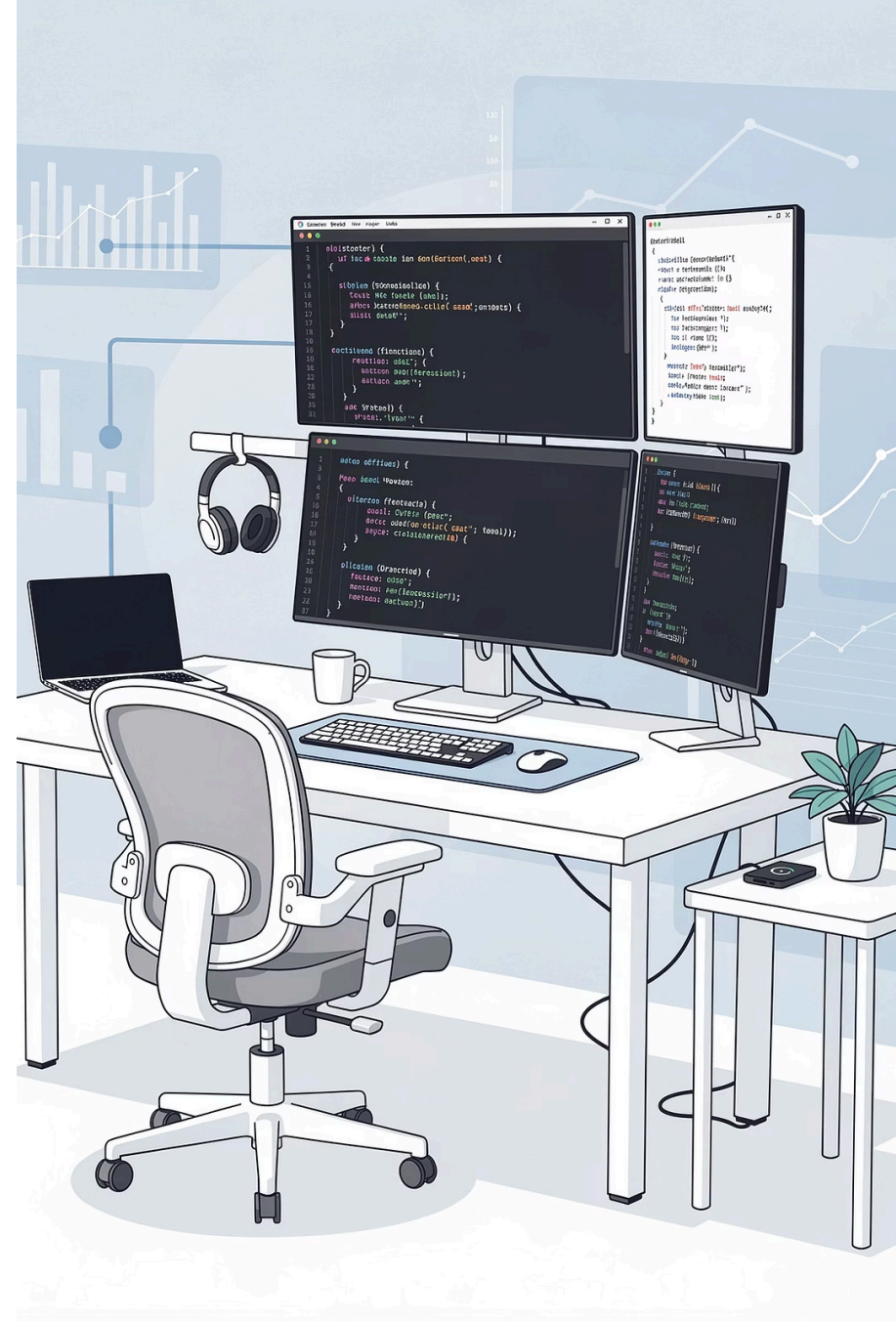


Python Environment & Basics for Data Engineering

Master the foundational tools and concepts that every Data Engineer uses daily. This module covers Python environments, core data structures, and control flow—the building blocks of production data pipelines.



Lesson 1: Python Environments & Development Tools

LESSON 1

Learning Goal: By the end of this lesson, participants will understand how to set up isolated Python environments, choose the right development tool for each scenario, and explain why environment management prevents production failures.

Core Concepts: Environments & Tools

Virtual Environments

Isolated Python setups with specific library versions—prevents conflicts between projects

Jupyter Notebooks

Interactive cell-by-cell execution for exploration and prototyping

Python Scripts

Sequential top-to-bottom execution for automation and production pipelines

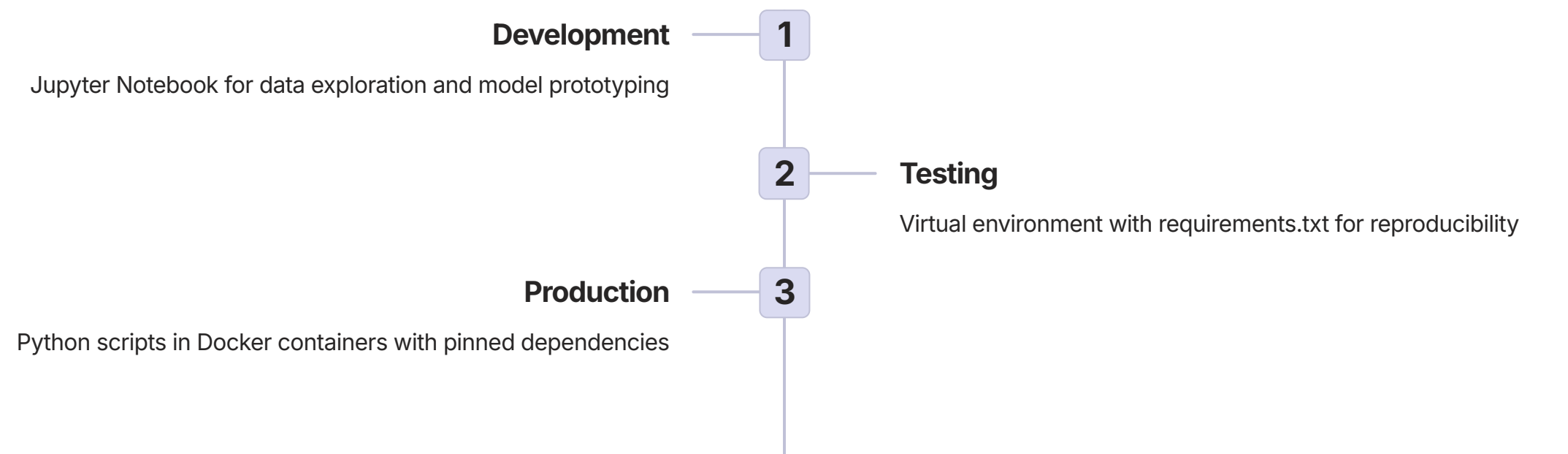
Why This Matters

In production data engineering:

- Code runs on laptops, servers, and scheduled jobs
- Library version mismatches cause silent failures
- Environment consistency = predictable pipelines

- ❏ Trainer cue: Ask about past experiences with "it works on my machine" bugs

Real-World Context: Production Scenarios



Common Tools in Data Pipelines



Airflow

Schedules Python scripts as DAG tasks



Docker

Packages environments for consistent deployment

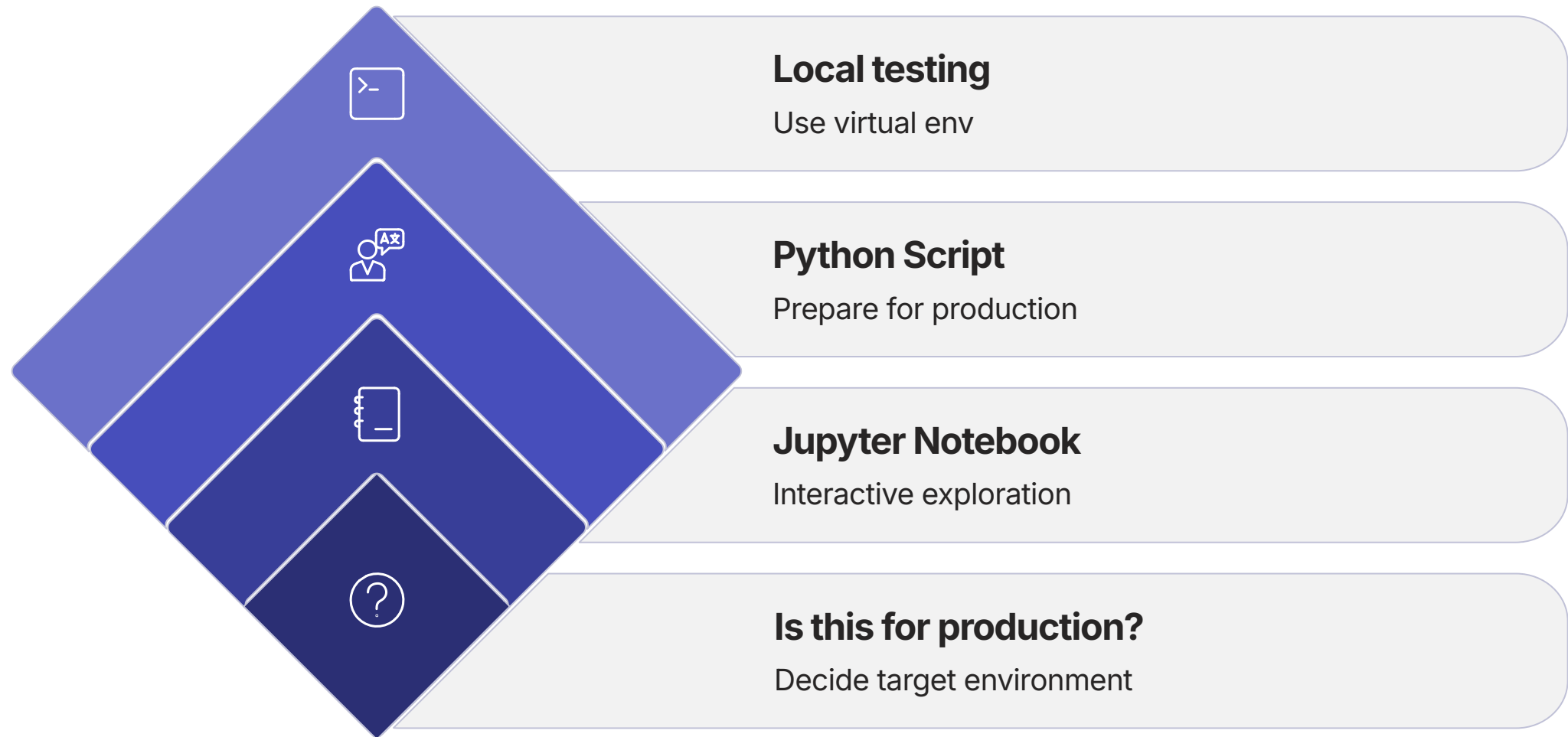


Git

Version control works best with .py scripts (not notebooks)

📌 Trainer cue: Emphasize that notebooks are NOT for production—they're for exploration only

Visual Reference: When to Use Each Tool



This decision tree guides tool selection based on the development phase. Interactive exploration demands Jupyter's immediate feedback, while production systems require the reliability and version control compatibility of Python scripts.

📌 Trainer cue: Whiteboard exercise—ask participants to classify their current projects

Python Basics: Variables & Data Types

Core Data Types

int

Whole numbers: `count = 42`

float

Decimals: `price = 99.5`

str

Text: `name = "Alex"`

bool

True/False: `is_active = True`


Why Types Matter

Data types drive every transformation:

- Revenue calculations require numeric types
- String operations enable text cleaning
- Boolean logic powers filtering and validation

Python infers types automatically—no explicit declaration needed, but understanding types prevents errors.

Activity 1 – Create and Run Your First Notebook

 HANDS-ON ACTIVITY

Objective

Set up a Jupyter Notebook environment and execute Python statements to build confidence with interactive code execution.

What You'll Do

- Launch Jupyter and create a new notebook
- Execute print statements and expressions
- Create variables and observe outputs
- Modify values and re-run cells

Expected Outcome

A saved notebook demonstrating:

- Cell-by-cell execution workflow
- Variable creation and manipulation
- Difference between cell output and print()

 Filename: `week1_activity_first_notebook.ipynb`

Lesson 2: Core Python Data Structures

LESSON 2

Learning Goal: By the end of this lesson, participants will select the appropriate data structure for specific data engineering tasks and explain how Python structures map to JSON and database records.

The Four Essential Data Structures

List

Ordered, mutable sequences

```
["Alex", "Maria"]
```

Tuple

Ordered, immutable records

```
(10.5, 20.3)
```

Set

Unordered, unique values

```
{101, 102, 103}
```

Dictionary

Key-value pairs

```
{"id": 101}
```

Why Each Exists

Lists: Store rows/records during processing

Tuples: Return multiple values from functions

Sets: Deduplicate and check membership fast

Dictionaries: Map directly to JSON and database records

Real-World Context: Data Structure Usage

Production Pipeline Patterns



API Response

JSON → dict/list



Deduplication

list → set → list



Lookup Table

list → dict ($O(1)$ access)



Database Insert

dict → SQL params



Trainer cue: Show live example of JSON from a public API and discuss structure choice

JSON ↔ Python Mapping

JSON Format

```
{
  "students": [
    {
      "id": 1,
      "name": "Alex",
      "completed": true
    },
    {
      "id": 2,
      "name": "Maria",
      "completed": false
    }
  ]
}
```

Python Equivalent

```
{
  "students": [
    {
      "id": 1,
      "name": "Alex",
      "completed": True
    },
    {
      "id": 2,
      "name": "Maria",
      "completed": False
    }
  ]
}
```

Nearly identical! This direct mapping makes Python ideal for API work and data lake processing.

When to Use Which Structure



List

- Processing CSV rows
- Collecting transformation results
- Maintaining order matters



Tuple

- Database query results
- Function returns (x, y coordinates)
- Immutable config values




Set

- Removing duplicate IDs
- Fast membership checks
- Comparing datasets (union, intersection)



Dictionary

- Parsing API responses
- Building lookup tables
- Structured record representation

 Trainer cue: Ask participants to classify a scenario you describe

Common Pitfalls



Using Lists for Lookups

$O(n)$ linear search—use dictionaries for $O(1)$ access



Modifying Tuples


Immutable after creation—raises `TypeError`



Expecting Set Order

Sets are unordered—use lists if sequence matters

Activity 2 – Iterating Over Lists and Dictionaries

 HANDS-ON ACTIVITY

Objective

Master iteration patterns used in every data transformation—processing records, filtering values, and aggregating results.

What You'll Do

- Iterate over lists with for loops
- Access dictionary keys, values, and items
- Process list-of-dictionaries (JSON-like data)
- Implement filtering and counting logic

Expected Outcome

A notebook demonstrating:

- keys(), values(), items() usage
- Nested structure navigation
- Conditional processing inside loops
- Data validation patterns

 Filename: `week1_activity_iterating_structures.ipynb`

Lesson 3: Control Flow and Loops

LESSON 3

Learning Goal: By the end of this lesson, participants will implement conditional logic and iteration patterns to handle real-world data variability—validation, filtering, and error handling.

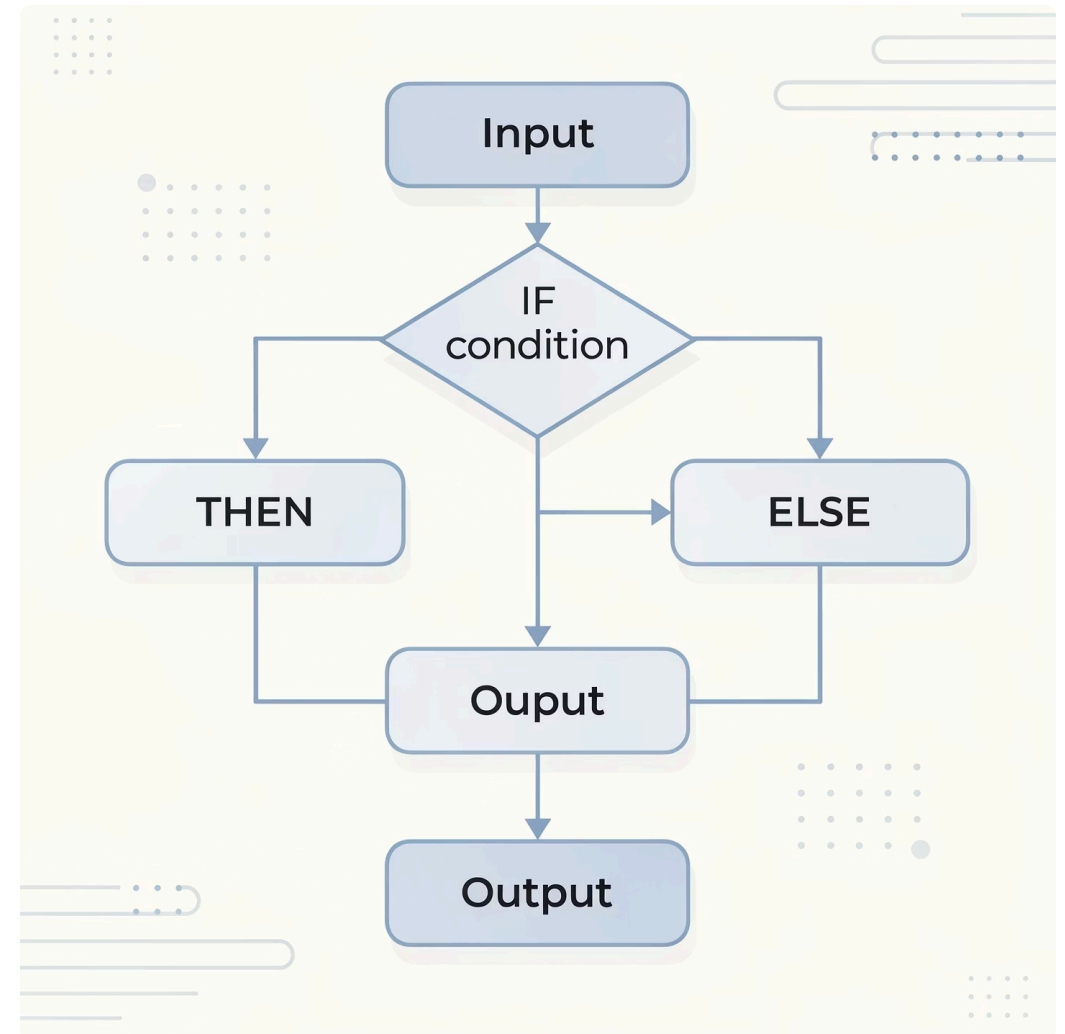
Control Flow: Decision Making

if / elif / else

```
if condition:  
    # Execute if True  
elif another_condition:  
    # Execute if first False  
else:  
    # Execute if all False
```

Used For:

- Data validation
- Business rule application
- Error handling



📌 Trainer cue: Relate to previous lesson—filtering dictionary values based on conditions

Loops: Repetition

for Loop

When: Iterating over known collections

Use: Processing lists, dicts, files

```
for item in items:  
    process(item)
```

while Loop

When: Condition-based repetition

Use: Retries, polling, unknown length

```
while condition:  
    do_something()  
    update_condition()
```

❏ **Warning:** while loops risk infinite execution if condition never becomes False

Loop Control: break and continue

break

Exits loop immediately when error or target found

```
for item in data:  
    if item < 0:  
        break # Stop  
    process(item)
```

continue

Skips current iteration, moves to next

```
for item in data:  
    if item == 0:  
        continue # Skip  
    process(item)
```

Real-World Context: Pipeline Logic

Common Data Engineering Patterns

01

Validation

Check if age > 0, break on invalid

03

Transformation

Apply business rules with if/elif

02

Filtering

Skip records with null values (continue)

04

Aggregation

Count valid records in for loop



Trainer cue: Show Apache Airflow task example where failure triggers break equivalent

When Control Flow Prevents Failures

In production systems, unhandled data issues cascade into downstream failures. Early detection with break statements and validation logic prevents:

- Corrupted database inserts
- Incomplete aggregations
- Silent data quality degradation

Defensive programming through control flow = reliable pipelines

Practical Example: Data Validation

```
ages = [18, 21, -1, 30]


for age in ages:
    if age < 0:
        print("Invalid age found")
        break # Stop processing
    elif age < 18:
        print("Minor")
    else:
        print("Adult")
```

Output:

```
Adult
Adult
Invalid age found
```

This pattern appears in every ETL pipeline—validate early, fail fast.

Activity 3 – Control Flow and Data Processing

 HANDS-ON ACTIVITY

Objective

Apply conditional logic and loops to implement data validation, filtering, and aggregation—core skills for pipeline development.


What You'll Do

- Write if/elif/else validation logic
- Implement for loops with break/continue
- Filter records based on conditions
- Count and aggregate valid data

Expected Outcome

A notebook demonstrating:

- Data quality checks
- Early failure detection
- Conditional transformation logic
- Record counting and metrics

 Filename: `week1_activity_control_flow.ipynb`

Module Summary

WEEK 1 RECAP

Lesson 1

Environments & Tools

Virtual environments prevent conflicts; notebooks for exploration, scripts for production

Lesson 2

Data Structures

Lists, tuples, sets, dicts—choose based on mutability, order, and uniqueness needs

Lesson 3

Control Flow

if/elif/else for decisions; for/while for iteration; break/continue for control

Key Takeaways



Environment Isolation

Virtual environments are not optional—they're essential for reproducible pipelines



Structure Selection

Dictionaries map to JSON—the most important structure for API and data lake work



Defensive Processing

Validate early, fail fast—control flow prevents cascading failures

Tools in Your Data Engineering Toolkit



Python

Core language for transformation logic



Jupyter

Interactive exploration and prototyping



Docker

Environment packaging for deployment



Git

Version control for .py scripts



Airflow

Orchestration of Python DAGs



pip/Poetry

Dependency management

What Makes Python Ideal for Data Engineering

```
1 import pandas as pd
2 import sys
3 import pandas as pd
4
5 df = pd.read_csv("sales.csv")
6
7 sales_df = {
8     (n
9      df.dropna()
10      .rename(columns={"date": "order_date",
11                      "region": "sales_region"}))
12      .astype({"sales": "salesd_x", "float"})
13      .assign(revenue=lambda x: x["sales"] * 1.2):
14      .groupby(["sales_region", "order_date"]);
15      .aggtotal_sales=("sales", sum),
16      .total_revenue=("revenue", sum))
17 }
18 sales_df.to_csv("sales_summary.csv",
19               index=False)
20 sales_df = {n
```

Technical Advantages

- **Readable syntax** reduces cognitive load
- **Rich ecosystem** (pandas, requests, sqlalchemy)
- **JSON compatibility** via native dict/list
- **Cross-platform** consistency
- **Strong community** and documentation

These factors make Python the default choice for data transformation layers in modern architectures.

Environment Management Best Practices

One Environment Per Project

Never mix dependencies across projects—create isolated venvs

Pin Versions

Use requirements.txt with exact versions (pandas==2.0.0, not pandas>=2.0.0)

Document Setup

Include setup instructions in README.md for new team members

Use .gitignore

Exclude venv/ directories from version control—environments are not portable

Jupyter vs Scripts: Decision Matrix

Scenario	Jupyter Notebook	Python Script
Exploring new dataset	✔ Ideal	✘ Too rigid
Scheduled ETL job	✘ Not suitable	✔ Required
Team code review	✘ Difficult (JSON)	✔ Easy (plain text)
Visualizing results	✔ Inline charts	✘ Requires save
CI/CD integration	✘ Complex	✔ Standard

📌 Rule of thumb: Prototype in notebooks, productionize in scripts

Understanding Mutability

Mutable (Can Change)

List

```
x = [1, 2]
x.append(3)
# [1, 2, 3]
```

Dictionary

```
d = {"a": 1}
d["b"] = 2
# {"a": 1, "b": 2}
```

Set

```
s = {1, 2}
s.add(3)
# {1, 2, 3}
```

Immutable (Fixed)

Tuple

```
t = (1, 2)
t[0] = 3
# TypeError!
```

String

```
s = "hello"
s[0] = "H"
# TypeError!
```

int/float/bool

```
x = 5
# Cannot modify
# Must reassign
```

Immutability = safety—use tuples for fixed configs to prevent accidental modification

List Operations Cheat Sheet

Common Methods

<code>.append(x)</code>	Add to end
<code>.insert(i, x)</code>	Add at index
<code>.remove(x)</code>	Remove first match
<code>.pop(i)</code>	Remove and return
<code>.sort()</code>	Sort in place
<code>.reverse()</code>	Reverse order

Slicing

```
data = [0, 1, 2, 3, 4]
```

```
data[1:3] # [1, 2]
```

```
data[:2] # [0, 1]
```

```
data[2:] # [2, 3, 4]
```

```
data[-1] # 4
```

```
data[::2] # [0, 2, 4]
```

Dictionary Operations Cheat Sheet

Access & Modification

```
d = {"name": "Alex", "age": 25}
```

```
# Access
```

```
d["name"]      # "Alex"
```

```
d.get("city", "N/A") # Default
```

```
# Modify
```

```
d["age"] = 26
```

```
d["city"] = "Madrid"
```

```
# Delete
```

```
del d["age"]
```

```
d.pop("city")
```

Iteration Methods

```
d = {"a": 1, "b": 2}
```

```
# Keys only
```


```
for key in d.keys():  
    print(key)
```

```
# Values only
```

```
for val in d.values():  
    print(val)
```

```
# Both
```

```
for k, v in d.items():  
    print(k, v)
```

 Pro tip: Use `.get()` with defaults to avoid `KeyError` in production

Set Operations for Data Deduplication

Union (|)

All unique elements from both sets

$\{1, 2\} \mid \{2, 3\} \# \{1, 2, 3\}$

Intersection (&)

Only common elements

$\{1, 2\} \& \{2, 3\} \# \{2\}$

Difference (-)

Elements in first but not second

$\{1, 2\} - \{2, 3\} \# \{1\}$

Real-World Set Usage

Example: Finding New Users

```
# Previous month users
previous = {101, 102, 103, 104}

# Current month users
current = {103, 104, 105, 106}

# New users this month
new_users = current - previous
print(new_users) # {105, 106}

# Retained users
retained = current & previous
print(retained) # {103, 104}
```

Why Sets Win

- **$O(1)$ membership checks** vs $O(n)$ in lists
- **Automatic deduplication**
- **Built-in set algebra**

Common in:

- User cohort analysis
- Data quality checks
- Record reconciliation

Nested Data Structures

Real-world data is rarely flat—JSON APIs return nested structures:

```
{
  "user": {
    "id": 101,
    "profile": {
      "name": "Alex",
      "location": "Madrid"
    },
    "orders": [
      {"id": 1, "total": 99.5},
      {"id": 2, "total": 45.0}
    ]
  }
}
```

Access Pattern:

```
data["user"]["profile"]["name"] # "Alex"
data["user"]["orders"][0]["total"] # 99.5
```

 Trainer cue: Live demo parsing real API response (e.g., GitHub, OpenWeather)

List Comprehensions

Traditional Loop

```
numbers = [1, 2, 3, 4, 5]
squares = []

for n in numbers:
    squares.append(n ** 2)

print(squares)
# [1, 4, 9, 16, 25]
```

List Comprehension

```
numbers = [1, 2, 3, 4, 5]

squares = [n ** 2 for n in numbers]

print(squares)
# [1, 4, 9, 16, 25]
```

Same result, cleaner syntax

With Filtering:

```
even_squares = [n ** 2 for n in numbers if n % 2 == 0] # [4, 16]
```

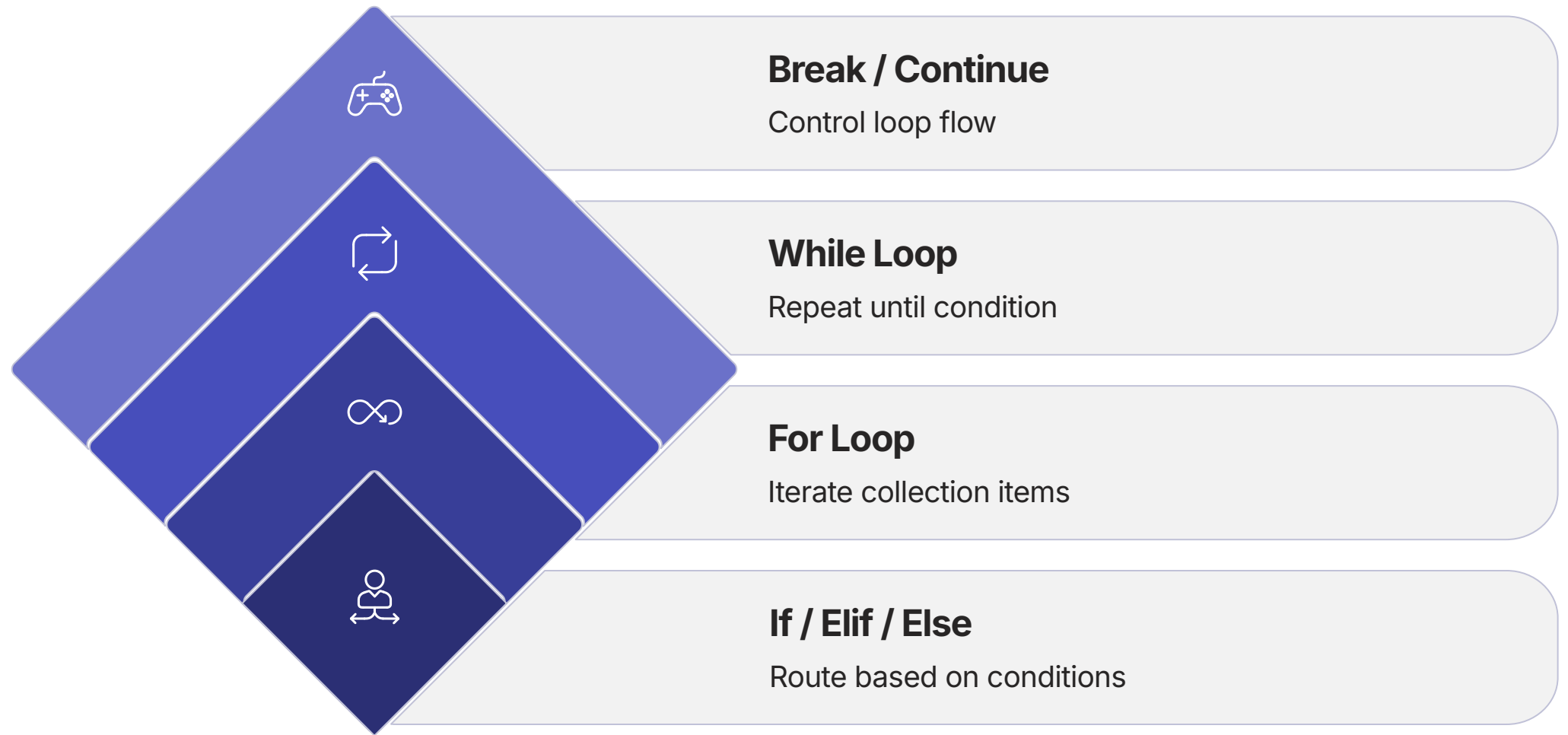
Dictionary Comprehensions

Build lookup tables efficiently:

```
students = [  
    {"id": 1, "name": "Alex"},  
    {"id": 2, "name": "Maria"}  
]  
  
# Traditional approach  
lookup = {}  
for s in students:  
    lookup[s["id"]] = s["name"]  
  
# Dictionary comprehension  
lookup = {s["id"]: s["name"] for s in students}  
# {1: "Alex", 2: "Maria"}
```

This pattern transforms lists into fast $O(1)$ lookup dictionaries—critical for join operations in data pipelines.

Control Flow: The Complete Picture



Control flow is the skeleton of every data pipeline. Conditions route data through validation, transformation, and error handling. Loops process collections. Break/continue manage exceptions.

if/elif/else: Syntax Patterns

Single Condition

```
if value > 0:  
    print("Positive")
```

If-Else

```
if value > 0:  
    print("Positive")  
else:  
    print("Non-positive")
```

Multiple Conditions

```
if value > 0:  
    print("Positive")  
elif value == 0:  
    print("Zero")  
else:  
    print("Negative")
```

Nested

```
if value > 0:  
    if value < 10:  
        print("Small positive")
```

Comparison Operators

Operator	Meaning	Example
<code>==</code>	Equal to	<code>x == 5</code>
<code>!=</code>	Not equal to	<code>x != 0</code>
<code>></code>	Greater than	<code>x > 10</code>
<code><</code>	Less than	<code>x < 100</code>
<code>>=</code>	Greater or equal	<code>x >= 18</code>
<code><=</code>	Less or equal	<code>x <= 65</code>
<code>in</code>	Membership	<code>"a" in ["a", "b"]</code>

Logical Operators

and

Both conditions must be True

```
if age >= 18 and has_license:  
    print("Can drive")
```

or

At least one condition must be True

```
if weekend or holiday:  
    print("Day off")
```

not

Negates the condition

```
if not is_valid:  
    print("Invalid")
```

📄 Combine for complex logic: `if (x > 0 and x < 10) or y == 5:`

for Loop: Syntax Variations

Basic Iteration

```
for item in [1, 2, 3]:  
    print(item)
```

With range()

```
for i in range(5):  
    print(i) # 0, 1, 2, 3, 4
```

With enumerate()

```
for i, val in enumerate(["a", "b"]):  
    print(i, val) # 0 a, 1 b
```

Dictionary Iteration

```
d = {"x": 1, "y": 2}  
for key, val in d.items():  
    print(key, val)
```

Nested Loops

```
for i in range(2):  
    for j in range(3):  
        print(i, j)
```

range() Function

Generates sequences of numbers—essential for numeric iteration:

range(stop)

```
range(5)  
# 0, 1, 2, 3, 4
```

range(start, stop)

```
range(2, 5)  
# 2, 3, 4
```

range(start, stop, step)

```
range(0, 10, 2)  
# 0, 2, 4, 6, 8
```

Common use: processing fixed-size batches, indexing operations, generating sequences.

while Loop: Use Cases

Retry Logic

```
retries = 0
max_retries = 3

while retries < max_retries:
    if api_call_succeeds():
        break
    retries += 1
    print(f"Retry {retries}")
```

User Input

```
valid = False

while not valid:
    value = input("Enter value: ")
    if validate(value):
        valid = True
    else:
        print("Invalid, try again")
```



Critical: Always update the condition variable inside the loop

break: Early Exit

Finding First Match

```
target = "Maria"
students = ["Alex", "Maria", "John"]

for student in students:
    if student == target:
        print("Found!")
        break
# Stops after "Maria"
```

Error Detection

```
data = [10, 20, -5, 30]

for value in data:
    if value < 0:
        print("Invalid data")
        break
    process(value)
# Stops at -5
```

break prevents unnecessary processing after finding what you need

continue: Skip Iteration

Filtering Nulls

```
records = [  
    {"name": "Alex"},  
    {"name": None},  
    {"name": "Maria"}  
]  
  
for record in records:  
    if record["name"] is None:  
        continue # Skip  
    print(record["name"])
```

Skipping Invalid Values

```
values = [10, 0, 20, 0, 30]  
  
for v in values:  
    if v == 0:  
        continue # Skip zeros  
    result = 100 / v  
    print(result)
```

continue allows selective processing without nested conditions

Comparison: break vs continue

Aspect	break	continue
Action	Exit loop completely	Skip to next iteration
Use When	Found target or error	Skip invalid records
Remaining Code	Not executed	Executes for valid items

Data Validation Pattern

Production-grade validation combines all control flow concepts:

```
def validate_records(records):
    valid_count = 0
    invalid_count = 0

    for record in records:
        # Skip missing IDs
        if "id" not in record:
            invalid_count += 1
            continue

        # Stop on critical error
        if record["id"] < 0:
            print("Critical error: negative ID")
            break

        # Validate age
        age = record.get("age", 0)
        if age < 0 or age > 150:
            invalid_count += 1
            continue

        # Record is valid
        valid_count += 1

    return valid_count, invalid_count
```

Common Pitfalls: Control Flow



Indentation Errors

Python uses indentation for block structure—
inconsistent spacing breaks code



Infinite while Loops

Always ensure condition becomes False—
add safety counter



= VS ==

= assigns, == compares—mixing them
causes logic errors

Defensive Programming Checklist

- **Validate inputs before processing**

Check for None, empty strings, negative numbers

- **Use break for early exit on errors**

Don't process remaining data if critical failure detected

- **Log skipped records**

Track what was filtered out for debugging

- **Add safety counters to while loops**

Prevent infinite loops with max iteration limits

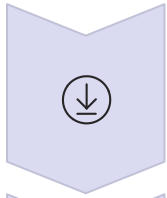
- **Test edge cases**

Empty lists, single items, duplicate values

Code Example: ETL Validation

```
def process_etl_batch(records):  
    """Extract, Transform, Load with validation"""  
    processed = []  
    errors = []  
  
    for record in records:  
        try:  
            # Extract  
            if not isinstance(record, dict):  
                errors.append(f'Invalid type: {type(record)}')  
                continue  
  
            # Validate required fields  
            required = ["id", "timestamp", "value"]  
            if not all(k in record for k in required):  
                errors.append(f'Missing fields in {record}')  
                continue  
  
            # Transform  
            record["value"] = float(record["value"])  
            record["processed_at"] = datetime.now()  
  
            # Load (simulate)  
            processed.append(record)  
  
        except Exception as e:  
            errors.append(f'Error processing {record}: {e}')  
  
    return processed, errors
```

Real-World Pipeline Example



Ingest

Read from API/file



Validate

if/elif/else checks



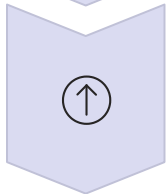
Filter

continue on invalid



Transform

for loop processing



Load

Write to database

Next Steps: Building on Fundamentals



What Comes Next

With Python fundamentals mastered, you're ready for:

- **Functions and modules** for code reusability
- **Error handling** with try/except
- **File I/O** for reading CSVs, JSON, logs
- **pandas** for DataFrame operations
- **SQL integration** via sqlalchemy
- **API interactions** with requests

These fundamentals underpin every advanced topic.

Industry Standards: PEP 8

PEP 8 is Python's official style guide—following it makes code readable and maintainable:

Indentation

4 spaces (not tabs)

Line Length

Max 79 characters

Naming

snake_case for functions/variables

Whitespace

Blank lines between functions

Teams enforce PEP 8 with linters like **flake8** and **black** (auto-formatter).

Version Control for Data Engineers

Git Best Practices

- Commit .py scripts, not .ipynb notebooks
- Use .gitignore for venv/, __pycache__/- Write descriptive commit messages
- Branch for features, merge to main
- Include requirements.txt in every repo

Sample .gitignore

```
# Virtual environments
venv/
env/

# Python cache
__pycache__/
*.pyc

# Jupyter checkpoints
.ipynb_checkpoints/

# Data files (too large)
*.csv
*.parquet

# Secrets
.env
config.ini
```

Testing Python Code

Data Engineers test transformation logic to prevent silent failures:

```
def clean_age(age):  
    """Return age if valid, else None"""  
    if age < 0 or age > 150:  
        return None  
    return age  
  
# Test cases  
assert clean_age(25) == 25    # Valid  
assert clean_age(-5) is None  # Invalid negative  
assert clean_age(200) is None # Invalid high  
assert clean_age(0) == 0     # Edge case
```

Frameworks like **pytest** automate testing—critical for CI/CD pipelines.

Debugging Strategies

01

Print Debugging

Add `print()` at checkpoints to inspect values

03

Use Type Checking

Check types with `type()` and `isinstance()`

02

Read Error Messages

Python errors point to line numbers—read stack traces carefully

04

Interactive Debugger

Use `pdb` or IDE debuggers for step-through execution

Performance Considerations

Optimization Tips

- Use sets for membership tests ($O(1)$ vs $O(n)$)
- Avoid repeated `list.append()` in tight loops
- Prefer list comprehensions over for loops
- Use generators for large datasets (memory efficient)
- Profile with `timeit` before optimizing

Example: Set vs List

```
# Slow:  $O(n)$  for each check  
ids_list = [1, 2, 3, ..., 10000]  
if 9999 in ids_list: # Slow!
```

```
# Fast:  $O(1)$  for each check  
ids_set = {1, 2, 3, ..., 10000}  
if 9999 in ids_set: # Instant!
```

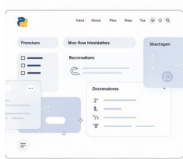
📌 Rule: Optimize for readability first, performance second (unless at scale)

Documentation Practices

Well-documented code is maintainable code—use docstrings and comments strategically:

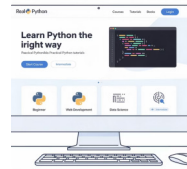
```
def process_records(records, min_age=18):  
    """  
    Process and validate customer records.  
  
    Args:  
    records (list): List of dict records with 'age' field  
    min_age (int): Minimum age threshold (default: 18)  
  
    Returns:  
    tuple: (valid_records, invalid_count)  
  
    Example:  
    >>> records = [{"age": 25}, {"age": 15}]  
    >>> valid, invalid = process_records(records)  
    >>> len(valid)  
    1  
    """  
    valid = []  
    invalid = 0  
  
    for record in records:  
        if record.get("age", 0) >= min_age:  
            valid.append(record)  
        else:  
            invalid += 1  
  
    return valid, invalid
```

Career Resources



Python Docs

Official reference for all built-in functions and libraries



Real Python

In-depth tutorials and video courses for all skill levels



Stack Overflow

Community Q&A—search before asking, read accepted answers



GitHub Projects

Read production code from popular Python libraries

You're Ready to Build

What You've Mastered

3

Core Lessons

Environments, structures, control flow

3

Hands-On Activities

Practical notebook exercises completed

60

Slides

Comprehensive learning journey

Next step: Apply these skills to real datasets. Start with a CSV file, load it into Python, clean the data using control flow, and output the results. You have the tools—now build.

[Start Your First Project](#)

[Review Key Concepts](#)

