

Introduction to Python Functions and Data Structures



- Training roadmap: from core syntax to building real data pipelines



- Why they matter: automation, data cleaning, and API integration in corporate work



- Functions transform data; choosing the right structure drives efficient, readable code



- Prerequisites: basic Python syntax, plus a hands-on, exercise-driven learning approach



Defining and Calling Functions

`def`

- The `def` keyword creates reusable blocks of logic



- Parameters define the expected inputs; arguments are actual values

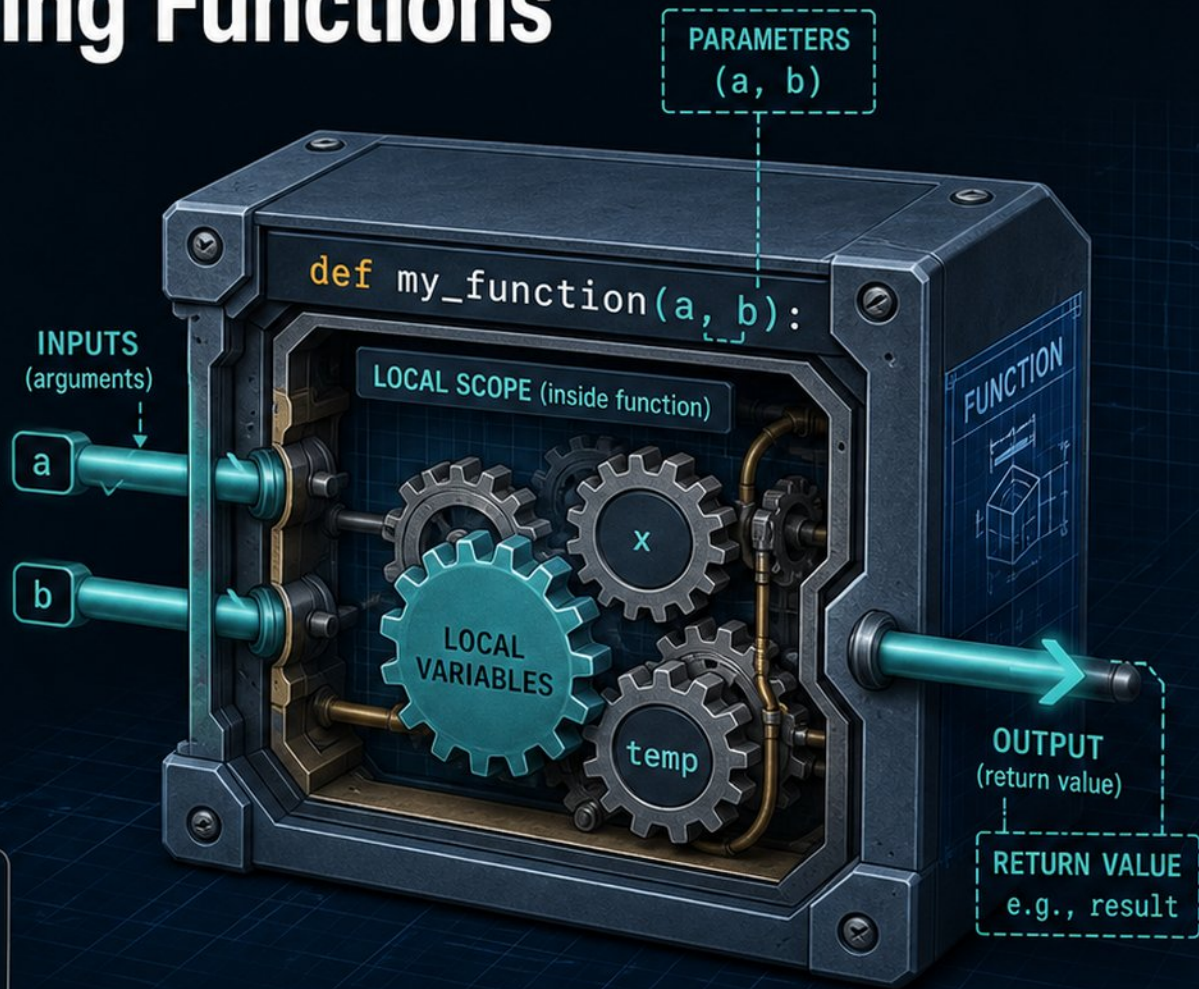


- Every function returns a value—`None` when no return is specified



- Scope determines variable visibility: local, enclosing, global, built-in

```
1 def add(a, b):  
2     total = a + b    # local variable  
3     return total  
4  
5 result = add(3, 4)   # arguments: 3, 4  
6 print(result)        # returns 7  
7
```



SCOPE LAYERS (Outermost → Innermost)



BUILT-IN SCOPE
e.g., print, len



GLOBAL SCOPE
module level



ENCLOSING SCOPE
(if any)



LOCAL SCOPE
function variables

Function Design Patterns



- Default and keyword args for flexible, self-documenting calls



- `*args` and `**kwargs` handle variable numbers of inputs



- Pure functions avoid side effects, simplifying testing and debugging



- Type hints (`int | None`, `list[str]`) improve readability and tooling

```
def func(a: int, b: str = "default",  
        *args: Any, **kwargs: Any) -> None:
```

***args**
Variable positional arguments
Collects extra positional arguments into a tuple

****kwargs**
Variable keyword arguments
Collects extra keyword arguments into a dict

Defaults

Default and keyword argument values

-> None
Returns no value (explicitly)

b: str = "default"
Uses "default" when no argument is provided

LEGEND

- Collects
- Provides default
- Return type

Type Hints in Practice

- ▶ Type hints serve as live documentation, checked by tools like mypy
- ▶ Modern syntax: ``int | None``, generics like ``list[str]``, ``TypedDict``
- ▶ Add annotations gradually, starting with public function signatures
- ▶ Demonstration: annotate a data function and run mypy in real time

BEFORE

```
def load_user(user_id):  
  
    users = [  
        {'id': 1, 'name': 'Ada', 'email': 'ada@example.com'},  
        {'id': 2, 'name': 'Grace', 'email': 'grace@example.com'},  
    ]  
  
    for u in users:  
        if u['id'] == user_id:  
            return u  
  
    return None
```

T

TypedDict
defines the
data shape

f(x)

Parameter
annotated
with type

→

Return type
declared

[]

Generic
collection
with type

AFTER

```
from typing import TypedDict  
  
class User(TypedDict):  
    id: int  
    name: str  
    email: str  
  
def load_user(user_id: int) -> User | None:  
    users: list[User] = [  
        {'id': 1, 'name': 'Ada', 'email': 'ada@example.com'},  
        {'id': 2, 'name': 'Grace', 'email': 'grace@example.com'},  
    ]  
    for u in users:  
        if u['id'] == user_id:  
            return u  
  
    return None
```

mypy (real-time)

```
$ mypy example.py
```

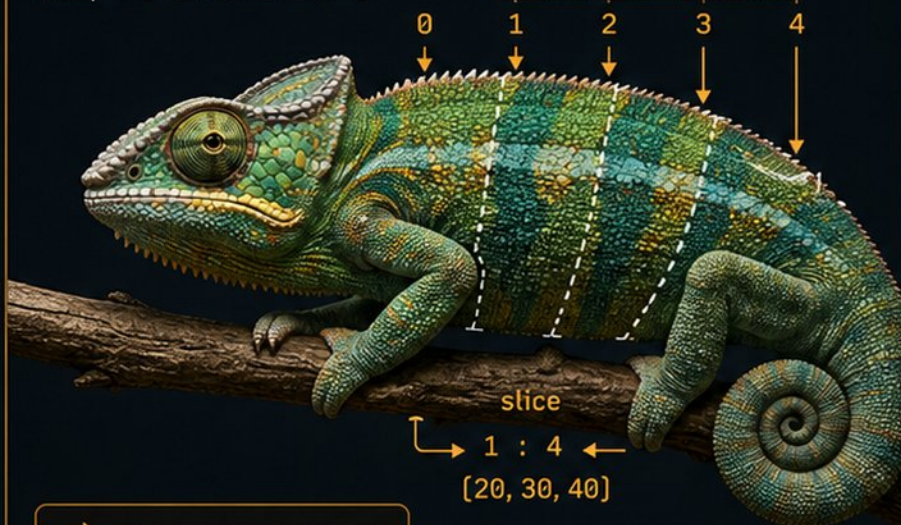
```
Success: no issues found in 1 source file
```

Lists and Tuples

PYTHON CORE CONCEPTS

LIST (mutable)

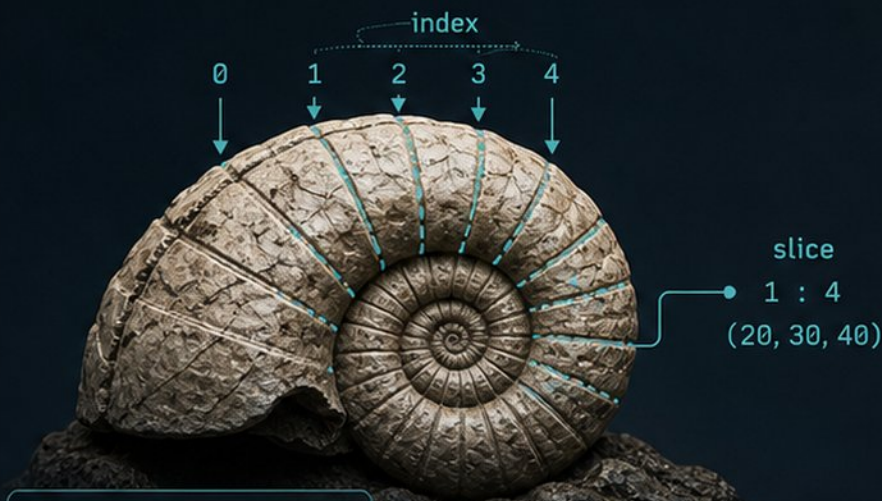
Example: [10, 20, 30, 40, 50]



Mutable: can change size and modify elements.

TUPLE (immutable)

Example: (10, 20, 30, 40, 50)



Immutable: fixed size and cannot be changed.



Create, index, and slice sequences with zero-indexing and slice syntax



Lists are dynamic and mutable; tuples are immutable records



Use `append`, `extend`, `pop`, `sort`, and sequence unpacking



List comprehensions: transform and filter data in one concise line

```
1 # Create
2 nums = [10, 20, 30, 40, 50]
3 point = (3, 4, 5)
4
5 # Indexing (zero-indexed)
6 nums[0]      # 10
7 nums[2]      # 30
8 point[1]     # 4
9
10 # Slicing
11 nums[1:4]    # [20, 30, 40]
12 nums[:3]     # [10, 20, 30]
13 point[1:1]   # (4, 5)
14
```

```
# List operations
nums.append(60)      # [10, 20, 30, 40, 50, 60]
nums.extend([70, 80]) # [10, 20, 30, 40, 50, 60, 70, 80]
last = nums.pop()    # 80
nums.sort()          # [10, 20, 30, 40, 50, 60, 70]

# Sequence unpacking
a, b, c = nums[0], nums[1], nums[2] # a=10, b=20, c=30
x, y, z = point                # x=3, y=4, z=5

# List comprehension
squares = [n*n for n in nums if n % 20 == 0]
# [20, 40, 60]
```

Dictionaries and Sets



Create, access, and update key-value pairs with ``get``, ``keys``, ``values``, ``items``



Sets enforce uniqueness and support union, intersection, and difference operations



Build mappings and collections concisely using dict and set comprehensions



Count occurrences with ``dict``, deduplicate with ``set``, and enable fast lookups

dict



set

Shared Concepts

Unordered (conceptually)

#

Hashable Elements

Fast Lookups



UNION
 $A \cup B$



INTERSECTION
 $A \cap B$



DIFFERENCE
 $A - B$



DIFFERENCE
 $B - A$



Iterating and Transforming Data

- ▶ - for loops use `iter()` and `next()` protocols under the hood
- ▶ - Built-in powerhouses: `enumerate`, `zip`, `map`, and `filter`
- ▶ - Generator expressions use lazy evaluation for memory efficiency
- ▶ - Custom generator functions with `yield` for large datasets
- ▶ - Choose the right iteration tool for readability and performance

```
# Lazy iteration with a generator
def packet_stream(source):
    for item in source:
        yield transform(item)
for packet in packet_stream(data):
    process(packet)
```



`iterate()`
Get an iterator



`next()`
Get the next item



`transform()`
Process / filter / map



`repeat...`
As needed, lazily

Combining Functions with Data Structures



- Pass functions as arguments to `'sorted'`, `'max'`, and `'min'` using the `'key'` parameter.



- Build reusable components by writing functions that return collections like `lists` or `dicts`.



- Structure data pipelines with `'map'`, `'filter'`, and `'reduce'` for clean, readable workflows.



- Hands-on: clean and summarize a small `CSV-like` dataset using functions and `list/dict` operations.

```
def by_amount(row):  
    return row['amount']
```

Examples

```
sorted(rows, key=by_amount)  
max(rows, key=lambda r: r['amount'])  
min(rows, key=lambda r: r['amount'])
```



RAW CSV-LIKE DATA (sample)

date	product	category	amount
2024-05-01	Widget	A	120
2024-05-01	Gadget	B	80
2024-05-02	Widget	A	150
...	...	...	...

DATA PIPELINE (Python)

```
clean = map(parse_row, rows)  
valid = filter(is_valid, clean)  
summary = reduce(aggregate, valid, init_summary())  
  
# summary is a dict with totals and counts
```

SUMMARY (dict)

```
{  
    'total_amount': 3500,  
    'top_category': 'A',  
    'transactions': 128,  
    'by_category': {'A': 2100, 'B': 900, 'C': 500}  
}
```

Advanced Data Structures from the `collections` Module



- `namedtuple`: lightweight, immutable data objects with named fields for clarity.



- `defaultdict` and `Counter`: specialized dicts for counting, grouping, and avoiding `KeyError`.



- `deque`: efficient double-ended queue for stack and queue operations.



- Choosing the right structure: trade-offs between lists, dicts, sets, and `collections` types.



Error Handling and Defensive Programming

KEY PRACTICES

01

- Use ``try`/`except`/`else`/`finally`` to catch specific errors and clean up resources

02

- Validate inputs and data structure integrity early to fail fast with clear messages

03

- Raise and create custom exceptions for domain-specific error conditions

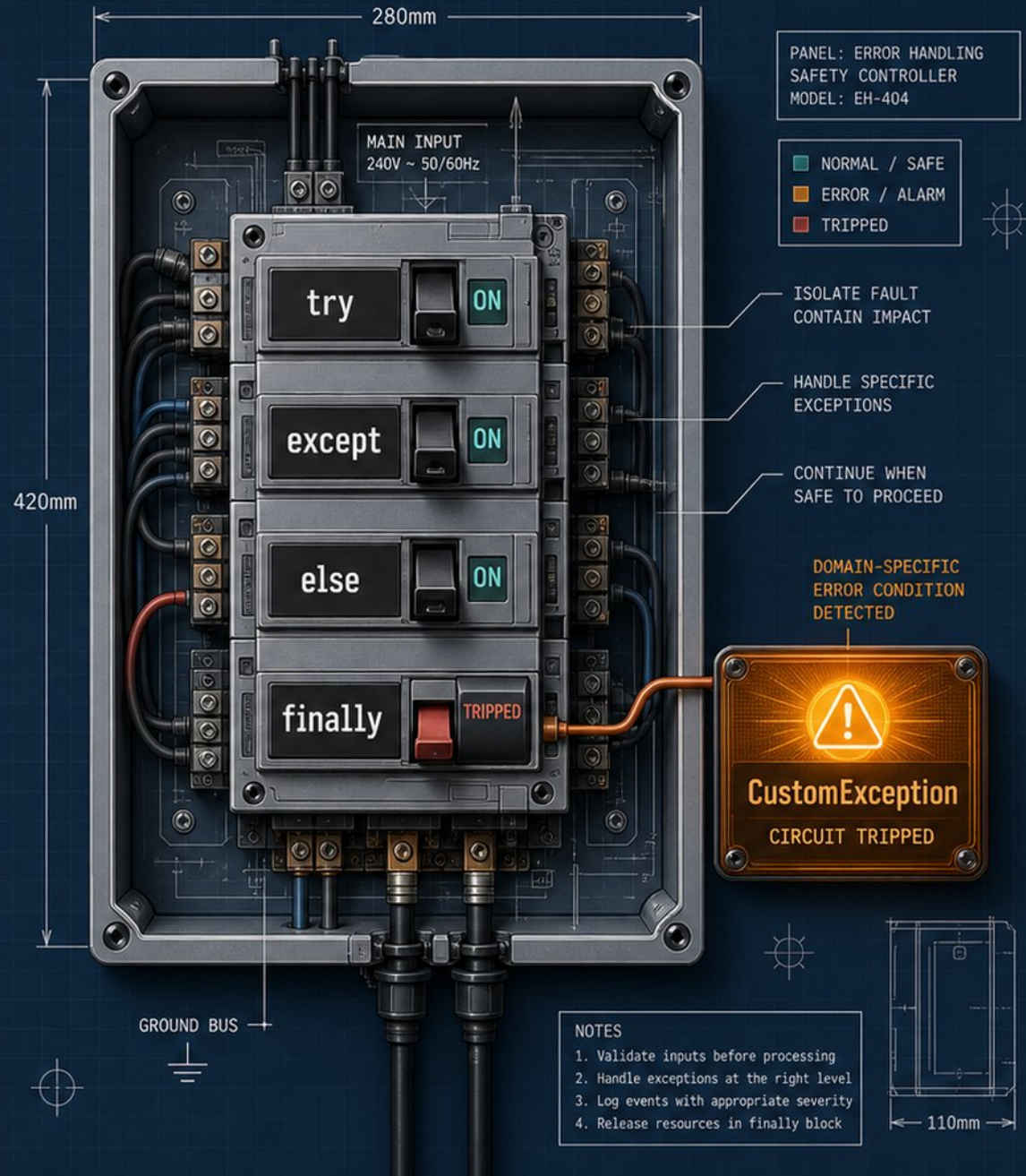
04

- Replace ``print`` with ``logging`` for production-grade debugging and monitoring



SAFETY FIRST

Anticipate errors. Validate early.
Fail safely. Recover cleanly.



Building a Data Cleaning Pipeline –

Part 1: Load and Profile

- Load messy e-commerce transactions CSV with pandas and inspect raw shape
- Profile data: check dtypes, null counts, and value ranges for every column
- Standardize column names to lowercase snake_case using `.rename()`
- Validate raw input against a Pandera schema model before cleaning

transactions_dirty.csv

⚠️ MESSY DATA DETECTED

	Order ID	Customer ID	Product	Quantity	Price	Order Date	Discount	Notes
0	1001	CUST-001	Widget A	2	19.99	2024-01-05	5%	-
1	1002	cust-002	Widget B	NA	29.99	01/06/2024	10%	gift
2	1003		Widget A	1	FREE	2024-01-07	-	-
3	1004	CUST-004	Widget C	three	39.50	2024/01/08	-	priority
4	1005	CUST-005	Widget B	5	29,99	Jan 9, 2024	15%	-
5	1006	CUST-006	Widget A	0	0	2024-01-10	-	test
6	1007	CUST-007	Widget C	NA	39.50	2024-01-11	5%	-
7	1008	CUST-008	Widget B	2.0	29.99	2024-13-01	-	-
...	...	...	...	...	...	...	...	...

PROFILING SUMMARY

Column	Dtype (inferred)	Null Count	Null %	Value Range / Notes	Issues
Order ID	int64	0	0%	1001 - 1050	✓
Customer ID	object (mixed)	2	20%	Mixed case, inconsistent IDs	⚠️
Product	object	0	0%	3 unique values	✓
Quantity	object (mixed)	2	20%	Values: int, float, text, NA	⚠️
Price	object (mixed)	1	10%	Currency as text, commas	⚠️
Order Date	object (mixed)	0	0%	Multiple date formats	⚠️
Discount	object	3	30%	Percent as text, blanks	⚠️
Notes	object	5	50%	Many blanks	⚠️



Building a Data Cleaning Pipeline – Part 2:

Clean and Transform



1. Handle missing values: numeric median, categorical mode/constant imputation



2. Coerce data types: parse mixed-format dates, strings to numeric



3. Remove duplicates and normalize text: strip, lower, standardize categories



4. Clip outliers using the IQR method to reasonable bounds



RAW DATA



CLEAN &
TRANSFORM



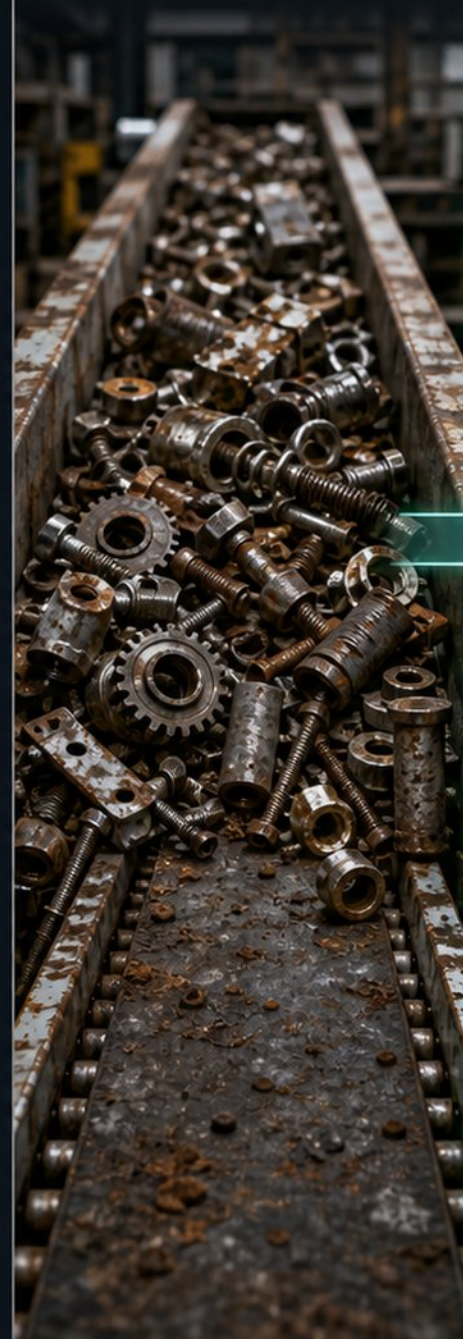
STRUCTURED
DATA



ANALYSIS
& INSIGHTS

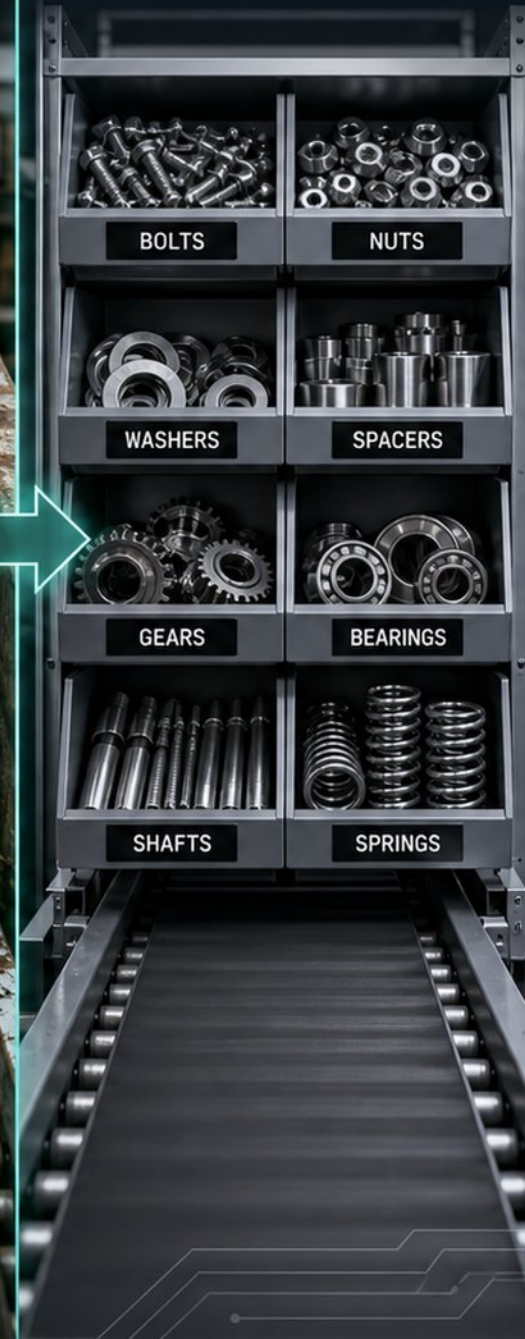
BEFORE

CHAOTIC & DIRTY



AFTER

CLEAN & ORGANIZED



Building a Data Cleaning Pipeline – Part 3: Validate, Output, and Audit



- Validate cleaned output with schema enforcement and data quality checks



- Write clean data to Parquet and SQLite for downstream consumption



- Generate a structured audit log: row counts, quarantine reasons, stage durations



- Organize pipeline code using .pipe() for method chaining and ColumnTransformer



id	name	age	email	country	signup_date
1	Alice	28	alice@example.com	US	2024-01-15
2	Bob	34	bob@example.com	CA	2024-01-16
3	Charlie	45	charlie@example.com	UK	2024-01-17
4	Diana	31	diana@example.com	US	2024-01-18
5	Evan	29	evan@example.com	AU	2024-01-19
...	...	...	...	...	...

AUDIT LOG

Run ID: 2024-05-15_1015
Pipeline: customer_cleaning
Started: 2024-05-15 10:15:22

ROW COUNTS

Raw Input Rows	1,250,000
Valid Rows	1,187,345
Quarantined Rows	62,655
Output Rows	1,187,345

QUARANTINE REASONS

Invalid Email Format	22,134
Missing Required Field	18,921
Out of Range Value	12,876
Duplicate Record	8,724

STAGE DURATIONS

Extract	
Validate	00:01:23
Transform	00:02:47
Output	00:03:56
Audit	00:01:18
	00:00:42

Completed: 2024-05-15 10:24:08

STATUS: SUCCESS

INGEST



VALIDATE



Schema &
Data Quality

TRANSFORM



Clean &
Standardize

OUTPUT



Parquet | SQLite

AUDIT



Log & Metrics

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

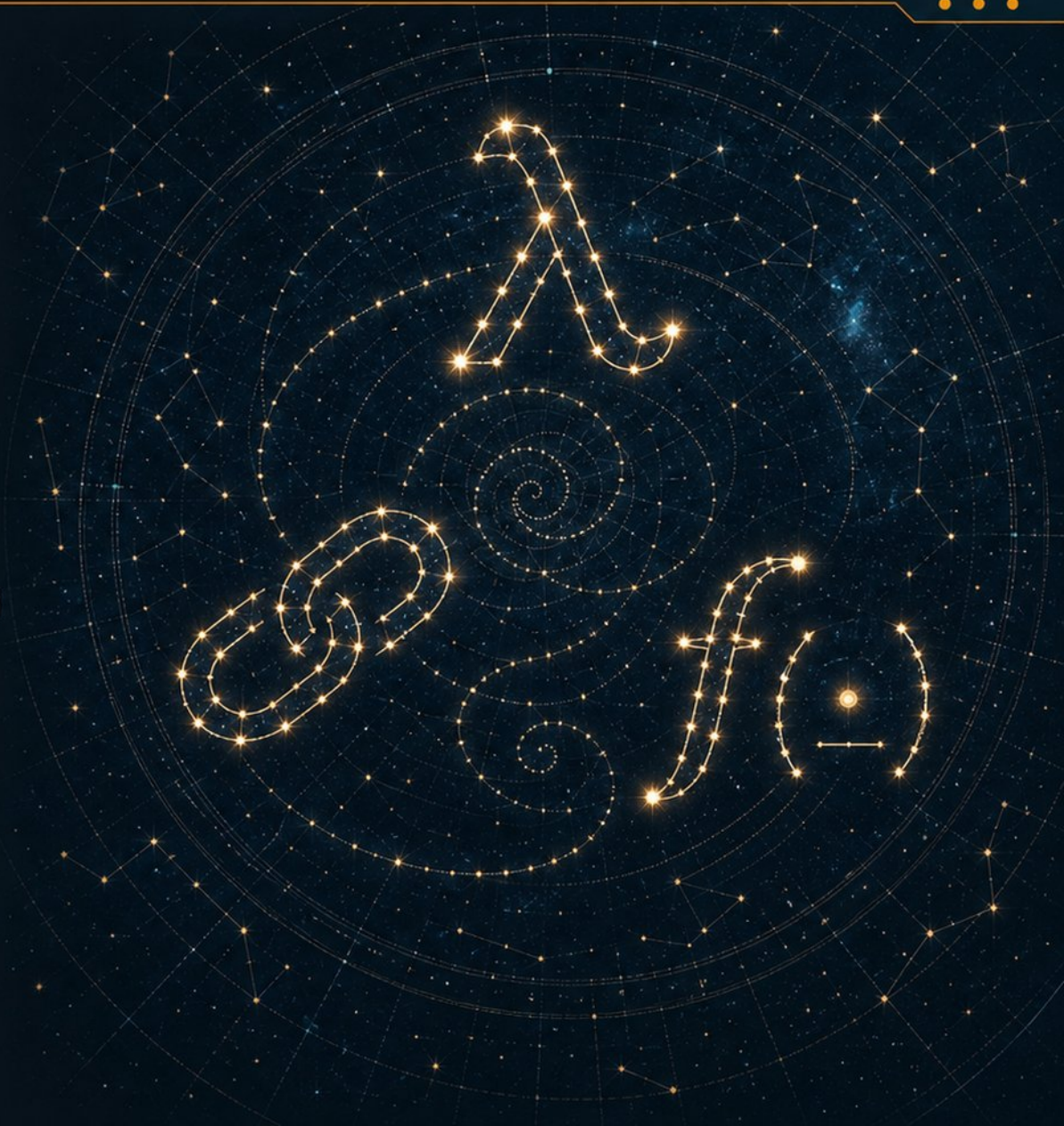
numeric_features = ['age', 'income', 'balance']
categorical_features = ['country', 'segment']

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features),
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features)
    ], remainder='drop'
)
```

```
(df_raw
 .pipe(validate_schema)
 .pipe(run_data_quality_checks)
 .pipe(preprocessor.fit_transform)
 .pipe(write_parquet, path='data/clean/customers.parquet')
 .pipe(write_sqlite, path='data/clean/customers.db')
 .pipe(generate_audit_log)
)
```

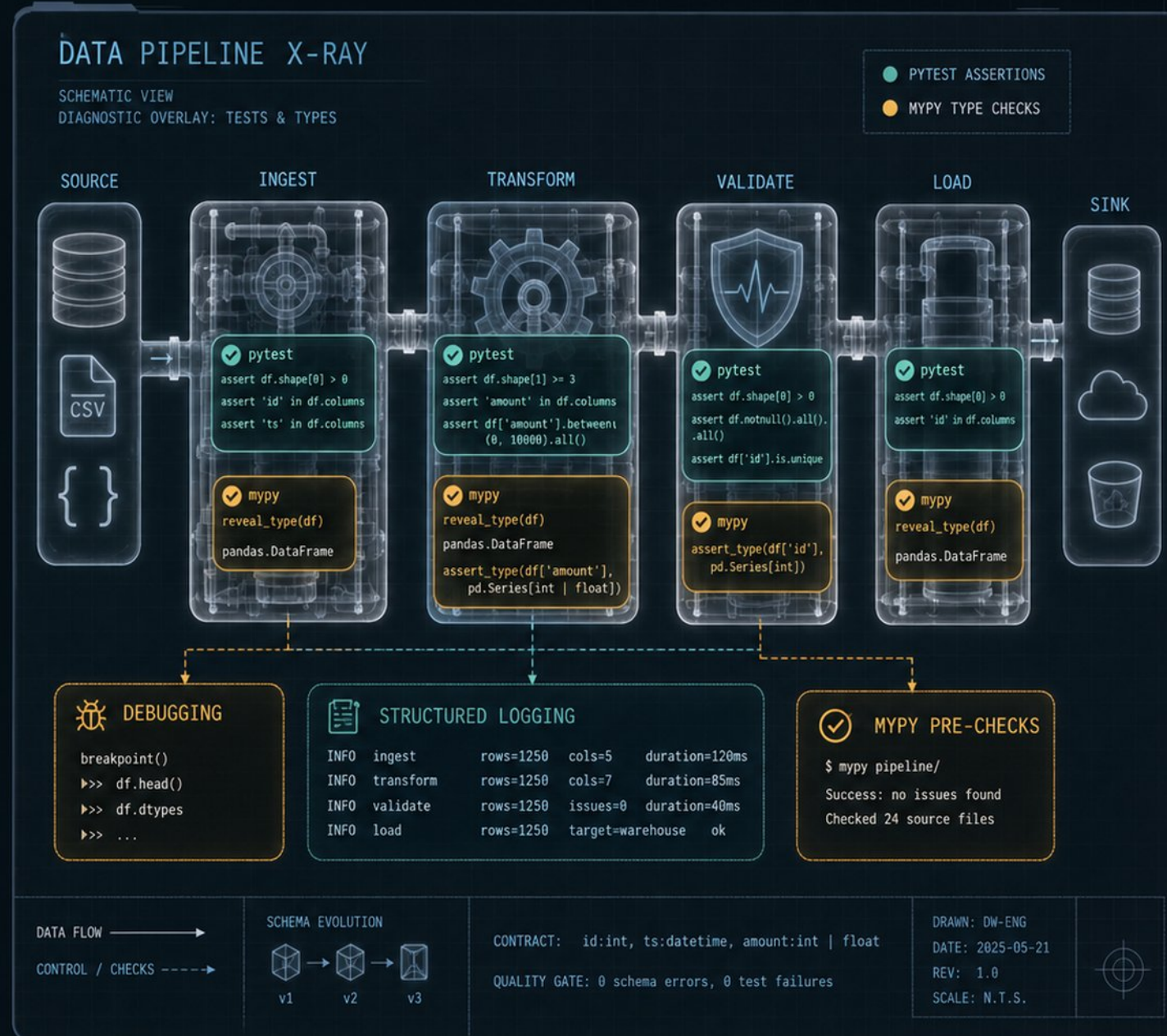
Functional Programming Patterns and `itertools`

- ▶ - Lambda expressions create small, anonymous functions inline
- ▶ - map, filter, and reduce transform data without explicit loops
- ▶ - itertools provides chain, groupby, product, and combinations
- ▶ - functools.partial locks arguments to build specialized functions
- ▶ - Chain generators to process large files with minimal memory



Testing and Debugging Your Code

- Write unit tests with `pytest` for functions and transformations
- Assert on row counts, column presence, and value ranges
- Verify type hints during development with `assert_type` and `reveal_type`
- Debug using `breakpoint()`, structured logging, and `mypy` pre-checks



Real-World Application and Next Steps

- ▶ - End-to-end mini-project review: patterns and structure
- ▶ - Avoid type hint drift, unlogged errors, and untested edge cases
- ▶ - Choose optimal data structures; use generators for large data
- ▶ - Learning paths: data engineering, API development, AI/ML
- ▶ - Performance: avoid quadratic loops in production code



PersonWise.

PersonWise • AI digital-human course platform



Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/aeb2c183/public