

Python Reference

From first script to production-grade data engineering patterns.

Last reviewed 29 Sep 2026 ·

Prerequisites: [DE Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/)

Related: [PySpark](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) · [Airflow](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/) · [LLM APIs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

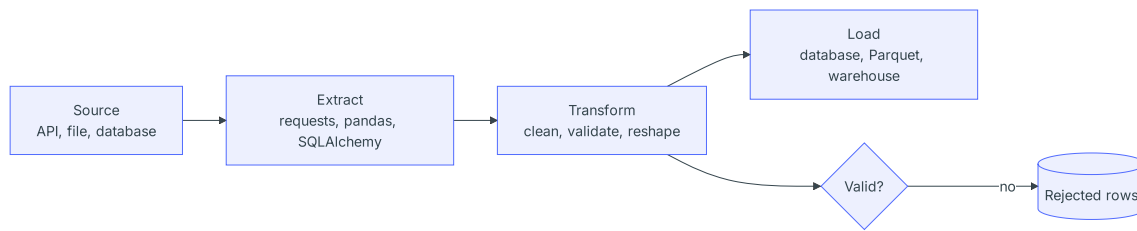
Challenge: SQL is well suited to transforming data inside a database, but pipelines also need to call REST APIs, read files from object storage, retry on transient failures, and react to schema changes. That integration work requires a general-purpose language.

Solution: Python is the common language of the data engineering ecosystem — orchestration code, Spark jobs, streaming consumers, data quality frameworks, and cloud SDKs all use it. SQL typically *transforms* data; Python *moves and orchestrates* it.

Typical DE Python script:

```
requests.get(api) → validate schema → pandas /  
pyarrow → write Parquet to S3 → log + alert  
          extract          guard          transform  
load          observe
```

Priorities for data engineering: Production code must be *idempotent* (safe to rerun), *observable* (logs what it did), *memory-aware* (streams large files instead of loading them whole), and *configurable* (no hardcoded credentials or dates).



On this page

Basics - [Variables & Data Types](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#variables--data-types) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#variables--data-types) - [Operators](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#operators) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#operators) - [Control Flow](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#control-flow) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#control-flow) - [Functions — Basics](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#functions--basics) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#functions--basics) - [Data Structures](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#data-structures) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#data-structures)

Intermediate - [Comprehensions](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#comprehensions) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#comprehensions) - [Error Handling](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#error-handling) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#error-handling) - [File I/O](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#file-io) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#file-io) - [Modules & Imports](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#modules--imports) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#modules--imports) - [OOP Essentials](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#oop-essentials) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#oop-essentials)

Advanced - [Generators & Iterators](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#generators--iterators) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#generators--iterators) - [Decorators](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#decorators) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#decorators) - [Context Managers](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#context-managers) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#context-managers) - [Type Hints](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#type-hints) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#type-hints) - [Useful Standard Library](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#useful-standard-library) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#useful-standard-library) - [Pandas Essentials](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#pandas-essentials) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#pandas-essentials) - [Working with APIs](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#working-with-apis) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#working-with-apis) - [DE Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#de-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#de-patterns)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/#further-reading)

Variables & Data Types

```
# Variables – no declaration needed, dynamically typed
name      = "Alice"          # str
age        = 30               # int
salary     = 95_000.50        # float (underscores for
readability)
active     = True             # bool
missing    = None             # NoneType

# Check type
type(name)          # <class 'str'>
isinstance(age, int) # True

# Multiple assignment
x = y = z = 0
a, b, c = 1, 2, 3          # unpacking

# Constants – Python has no const keyword; UPPER_CASE is
the convention
MAX_RETRIES = 3
DEFAULT_TIMEOUT = 30
```

Strings

```
s = "  Hello, World!  "

# Common methods
s.strip()          # "Hello, World!"
s.lower() / s.upper()
s.replace("World", "Data")
s.split(", ")      # ["  Hello", "World!  "]
s.startswith("Hello") # True (after strip)
", ".join(["a", "b", "c"]) # "a,b,c"
"Alice" in s        # False (case-sensitive)

# Formatting – always prefer f-strings
name, dept = "Alice", "Engineering"
f"Hello, {name}. Dept: {dept}"
f"Salary: {salary:,.2f}"          # → "Salary: 95,000.50"
f"{'left':<10} {'right':>10}"    # padding / alignment
```

```
f"{value!r}"                # repr() of value

# Multi-line string
query = """
    SELECT *
    FROM   employees
    WHERE  active = true
    """
```

Type conversion

```
int("42")          # 42
float("3.14")      # 3.14
str(100)           # "100"
bool(0)            # False

# Falsy values: 0, 0.0, "", [], {}, set(), None
# Everything else is truthy
```

Operators

```
# Arithmetic
10 + 3    # 13      addition
10 - 3    # 7       subtraction
10 * 3    # 30      multiplication
10 / 3    # 3.333   true division (always float)
10 // 3   # 3       floor division
10 % 3    # 1       modulo (remainder)
10 ** 2   # 100     exponentiation

# Comparison
==  !=  <  <=  >  >=

# Logical
and  or  not

# Identity and membership
x is None          # identity check (use for None, not ==)
x is not None
```

```
"key" in my_dict    # membership
3 in [1, 2, 3]      # True

# Walrus operator (Python 3.8+) – assign and test in one
expression
if n := len(data):
    print(f"Processing {n} rows")
```

Control Flow

```
# if / elif / else
if salary > 100_000:
    tier = "senior"
elif salary > 70_000:
    tier = "mid"
else:
    tier = "junior"

# Ternary (one-liner if/else)
tier = "senior" if salary > 100_000 else "other"

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

for i in range(0, 10, 2):   # 0, 2, 4, 6, 8 (start, stop,
    print(i)               step)

for name in ["Alice", "Bob", "Carol"]:
    print(name)

# enumerate – get index + value
for i, name in enumerate(["Alice", "Bob"], start=1):
    print(i, name)          # 1 Alice \n 2 Bob

# zip – iterate two sequences in parallel
for name, score in zip(names, scores):
    print(name, score)
```

```

# dict iteration
for key, value in record.items():
    print(f"{key}: {value}")

# while
i = 0
while i < 10:
    i += 1

# break / continue / else on loop
for n in range(20):
    if n % 2 == 0:
        continue    # skip even numbers
    if n > 9:
        break        # stop loop
else:
    print("loop completed without break") # runs if no
break

```

Functions — Basics

```

# Define a function
def add(a, b):
    return a + b

# Default arguments
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

greet("Alice")          # "Hello, Alice!"
greet("Bob", "Hi")      # "Hi, Bob!"

# Keyword arguments — can pass in any order
greet(greeting="Hey", name="Carol")

# Multiple return values (returns a tuple)
def min_max(numbers):
    return min(numbers), max(numbers)

lo, hi = min_max([3, 1, 4, 1, 5])

```

```
# Docstring
def load_data(path: str) -> list:
    """Load records from a CSV file at the given path."""
    ...
```

args and *kwargs

```
# *args – variable number of positional arguments
def total(*numbers):
    return sum(numbers)

total(1, 2, 3, 4)    # 10

# **kwargs – variable keyword arguments
def log_event(event_type, **metadata):
    print(event_type, metadata)

log_event("click", user_id=42, page="home")
# click {'user_id': 42, 'page': 'home'}

# Unpacking into a function call
args = [1, 2]
kwargs = {"greeting": "Hi"}
add(*args)
greet(**kwargs, name="Alice")
```

Data Structures

List — ordered, mutable

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

# Indexing and slicing
nums[0]        # 1      first element
nums[-1]       # 5      last element
nums[1:3]      # [2, 3] slice (end exclusive)
nums[::-1]     # [5,4,3,2,1] reversed copy
```

```

# Mutating
nums.append(6)           # add to end
nums.extend([7, 8])      # add multiple
nums.insert(0, 0)        # insert at index
nums.remove(3)           # remove first occurrence of value
nums.pop()               # remove and return last
nums.pop(0)              # remove and return at index

# Useful operations
len(nums)                # length
3 in nums                 # membership – O(n)
sorted(nums)             # returns new sorted list
nums.sort(reverse=True)  # sorts in place
nums.count(2)            # count occurrences
nums.index(4)            # find index of value

```

Dictionary — key-value pairs, ordered (Python 3.7+)

```

record = {"id": 1, "name": "Alice", "dept": "Engineering"}

# Access
record["name"]           # "Alice" – raises KeyError if
missing                  # missing
record.get("phone", "N/A") # safe – returns default if
missing                  # missing

# Mutate
record["email"] = "alice@example.com" # add or update
del record["dept"]              # remove key

# Iterate
record.keys()    # dict_keys(["id", "name", "email"])
record.values()  # dict_values([1, "Alice",
"alice@example.com"])
record.items()   # dict_items([("id",1), ("name","Alice"),
...])

"name" in record # True – key membership, O(1)

# Merge (Python 3.9+)
merged = dict_a | dict_b

```

```
# defaultdict – no KeyError for missing keys
from collections import defaultdict
counts = defaultdict(int)
for word in words:
    counts[word] += 1
```

Tuple — ordered, immutable

```
point = (10, 20)
x, y = point          # unpacking

# Named tuple – readable, self-documenting
from collections import namedtuple
Row = namedtuple("Row", ["id", "name", "salary"])
r = Row(1, "Alice", 90_000)
r.name                # "Alice"
r[0]                  # 1
```

Set — unordered, unique values

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}

a | b    # union:          {1, 2, 3, 4, 5, 6}
a & b    # intersection:  {3, 4}
a - b    # difference:    {1, 2}
a ^ b    # symmetric diff:{1, 2, 5, 6}

3 in a    # True – membership is O(1)

# Deduplication
unique = list(set([1, 2, 2, 3, 3, 3]))  # [1, 2, 3]
```

When to use which

Structure	Use when
<code>list</code>	Ordered sequence, may have duplicates, needs indexing

Structure	Use when
<code>dict</code>	Key-value lookup, named fields
<code>tuple</code>	Immutable record, multiple return values
<code>set</code>	Deduplication, fast membership testing, set math

Comprehensions

```
# List comprehension – [expression for item in iterable if condition]
squares = [x**2 for x in range(10)]
evens    = [x for x in range(20) if x % 2 == 0]

# Dict comprehension
name_to_salary = {emp["name"]: emp["salary"] for emp in employees}

# Set comprehension
unique_depts = {emp["dept"] for emp in employees}

# Generator expression – lazy, doesn't build the full list in memory
total = sum(x**2 for x in range(1_000_000))

# Conditional value
labels = ["high" if s > 100_000 else "low" for s in salaries]

# Nested comprehension – flatten a list of lists
flat = [item for sublist in nested for item in sublist]

# With tuple unpacking
totals = {dept: sum(s for _, s in rows) for dept, rows in grouped.items()}
```

Prefer comprehensions over `map()` / `filter()` for readability. Use generator expressions `( )` instead of list comprehensions `[ ]` when you only need to iterate once — saves memory.

Error Handling

```
# try / except / else / finally
try:
    result = int(user_input)
except ValueError as e:
    print(f"Invalid input: {e}")
except (TypeError, KeyError) as e:
    print(f"Data error: {e}")
else:
    print("Succeeded:", result)    # runs only if no
    exception raised
finally:
    cleanup()                      # always runs – open
    files, connections

# Raise an exception
raise ValueError("salary must be positive")

# Re-raise the current exception
try:
    process(record)
except Exception as e:
    logger.error("Failed: %s", e)
    raise    # re-raise without losing original traceback

# Custom exception hierarchy
class PipelineError(Exception):
    pass

class SchemaValidationError(PipelineError):
    def __init__(self, column, message):
        self.column = column
        super().__init__(f"Column '{column}': {message}")

raise SchemaValidationError("salary", "expected numeric,
got string")
```

File I/O

```
import csv
import json
from pathlib import Path

# — Text files —————
with open("data.txt", "r", encoding="utf-8") as f:
    content = f.read()          # entire file as string
    lines   = f.readlines()     # list of lines

with open("output.txt", "w", encoding="utf-8") as f:
    f.write("Hello\n")

# — CSV —————
with open("orders.csv", newline="", encoding="utf-8") as f:
    reader = csv.DictReader(f)
    for row in reader:          # row is a dict
        print(row["order_id"], row["amount"])

with open("output.csv", "w", newline="", encoding="utf-8")
as f:
    writer = csv.DictWriter(f, fieldnames=["id", "name",
"salary"])
    writer.writeheader()
    writer.writerows(records)

# — JSON —————
with open("config.json") as f:
    config = json.load(f)      # file
                                # dict

with open("output.json", "w") as f:
    json.dump(data, f, indent=2, default=str)  #
default=str handles dates

json.loads('{"key": "value"}')  # string → dict
json.dumps({"key": "value"})    # dict → string

# — pathlib — modern path handling —————
base = Path("/data/warehouse")
file = base / "2024" / "03" / "orders.parquet"  # joins
```

```
with /

file.exists()          # True/False
file.suffix            # ".parquet"
file.stem              # "orders"
file.parent            # Path("/data/warehouse/2024/03")
file.name              # "orders.parquet"

list(base.glob("**/*.parquet"))          # recursive glob
(base / "output").mkdir(parents=True, exist_ok=True)
```

Modules & Imports

```
# Import a module
import os
import datetime

# Import specific names
from pathlib import Path
from datetime import date, timedelta

# Import with alias
import pandas as pd
import numpy as np

# Import everything (avoid – pollutes namespace)
from os.path import *

# Your own module (file: utils.py)
from utils import load_config, validate_schema

# Conditional import – handle optional dependencies
try:
    import ujson as json    # faster JSON library if
    available
except ImportError:
    import json
```

Project structure

```
my_pipeline/  
  __init__.py  
  config.py      # Config dataclass, env var loading  
  extract.py     # Source connectors  
  transform.py   # Business logic  
  load.py        # Destination writers  
  utils.py       # Shared helpers  
  main.py        # Entrypoint
```

OOP Essentials

```
class Pipeline:  
    default_retries = 3      # class variable – shared  
    across all instances  
  
    def __init__(self, name: str, source: str):  
        self.name = name    # instance variable  
        self.source = source  
        self._runs = 0      # _ prefix =  
        internal/private by convention  
  
    # Regular method  
    def run(self) -> None:  
        self._runs += 1  
        print(f"Running {self.name} (run #{self._runs})")  
  
    # Property – computed attribute, accessed like a  
    variable  
    @property  
    def run_count(self) -> int:  
        return self._runs  
  
    # Class method – receives class, not instance; used as  
    factory  
    @classmethod  
    def from_config(cls, config: dict) -> "Pipeline":  
        return cls(config["name"], config["source"])
```

```

    # Static method — no class or instance; utility
function in namespace
    @staticmethod
    def validate_source(source: str) -> bool:
        return source.startswith(("s3://", "gs://",
"abfss://"))

    def __repr__(self) -> str:
        return f"Pipeline(name={self.name!r})"

# Inheritance
class IncrementalPipeline(Pipeline):
    def __init__(self, name, source,
watermark_col="updated_at"):
        super().__init__(name, source)    # call parent
__init__
        self.watermark_col = watermark_col

    def run(self) -> None:
        print(f"Incremental from {self.watermark_col}")
        super().run()                    # call parent
method

```

Dataclass — less boilerplate for data-holding classes

```

from dataclasses import dataclass, field

@dataclass
class Config:
    db_host: str
    db_port: int = 5432
    db_name: str = "analytics"
    tags: list = field(default_factory=list)

cfg = Config(db_host="localhost")
cfg.db_host    # "localhost"
cfg.db_port    # 5432

```

Generators & Iterators

```
# Generator function – uses yield; returns one item at a time
# The caller controls when the next item is produced
def read_in_chunks(filepath, chunk_size=1_000):
    with open(filepath) as f:
        chunk = []
        for line in f:
            chunk.append(line)
            if len(chunk) == chunk_size:
                yield chunk
                chunk = []          # reset after yielding
        if chunk:
            yield chunk            # final partial chunk

# Memory efficient – only one chunk in memory at a time
for chunk in read_in_chunks("large_file.csv"):
    process(chunk)

# Generator expression – lazy list comprehension
squares = (x**2 for x in range(1_000_000))  # nothing
computed yet
next(squares)    # 0
next(squares)    # 1
list(squares)    # forces evaluation of remainder

# yield from – delegate to a sub-generator
def chain_files(paths):
    for path in paths:
        yield from read_in_chunks(path)

# Useful pattern: infinite sequence with a sentinel
def counter(start=0):
    n = start
    while True:
        yield n
        n += 1
```

Decorators

Decorators wrap a function to add behaviour (timing, logging, retrying) without modifying its body.

```
import time, functools

# Simple decorator
def timer(func):
    @functools.wraps(func)      # preserves func's name
    and docstring
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        print(f"{func.__name__} took {time.perf_counter() -
start:.3f}s")
        return result
    return wrapper

@timer
def load_data(path):
    ...

# Decorator with arguments
def retry(max_attempts=3, delay=1.0, exceptions=
(Exception,)):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(1, max_attempts + 1):
                try:
                    return func(*args, **kwargs)
                except exceptions as e:
                    if attempt == max_attempts:
                        raise
                    print(f"Attempt {attempt} failed: {e}.
Retrying in {delay}s...")
                    time.sleep(delay)
            return wrapper
        return decorator

@retry(max_attempts=3, delay=2, exceptions=
```

```
(requests.Timeout,))
def fetch_data(url):
    ...

# Logging decorator
def log_calls(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        logger.info("Calling %s", func.__name__)
        result = func(*args, **kwargs)
        logger.info("%s completed", func.__name__)
        return result
    return wrapper
```

Context Managers

Guarantee setup and teardown — even when exceptions occur.

```
# Built-in: open(), database connections
with open("data.csv") as f:
    data = f.read()
# file closed automatically

# Custom context manager via contextlib
from contextlib import contextmanager

@contextmanager
def db_transaction(conn):
    try:
        yield conn          # code in the `with` block
                             runs here
        conn.commit()
    except Exception:
        conn.rollback()
        raise

with db_transaction(conn) as c:
    c.execute("UPDATE ...")
    c.execute("INSERT ...")
# commits if no error, rolls back if exception
```

```
# Context manager via class (__enter__ / __exit__)
class Timer:
    def __enter__(self):
        self.start = time.perf_counter()
        return self

    def __exit__(self, *args):
        self.elapsed = time.perf_counter() - self.start

with Timer() as t:
    process_data()
print(f"Took {t.elapsed:.3f}s")
```

Type Hints

Type hints don't enforce anything at runtime — they document intent and enable static analysis tools (mypy, pyright).

```
# Basic hints
def greet(name: str, times: int = 1) -> str:
    return (name + " ") * times

# Optional – value or None
from typing import Optional
def find_user(user_id: int) -> Optional[dict]:
    ...

# Modern Python 3.10+ – use | instead of Optional/Union
def find_user(user_id: int) -> dict | None:
    ...

# Collections
def process(records: list[dict]) -> list[dict]: ...
def lookup(mapping: dict[str, int]) -> None: ...
def dedupe(items: set[str]) -> list[str]: ...

# Callable
from collections.abc import Callable
def apply(func: Callable[[int], str], values: list[int]) ->
```

```

list[str]:
    return [func(v) for v in values]

# Type aliases
Row      = dict[str, object]
Records = list[Row]

def transform(records: Records) -> Records: ...

# Generics
from typing import TypeVar
T = TypeVar("T")

def first(items: list[T]) -> T:
    return items[0]

```

Useful Standard Library

datetime

```

from datetime import date, datetime, timedelta

today      = date.today()
now        = datetime.now()
yesterday  = today - timedelta(days=1)
diff       = datetime(2024, 3, 15) - datetime(2024, 1, 1)
diff.days  # 74

now.strftime("%Y-%m-%d")          # "2024-03-15"
now.strftime("%Y-%m-%d %H:%M:%S") # "2024-03-15 10:30:00"
datetime.strptime("2024-03-15", "%Y-%m-%d")
datetime.fromisoformat("2024-03-15T10:30:00")

```

os & environment variables

```

import os

os.environ["DB_HOST"]          # raises KeyError if
missing

```

```
os.environ.get("DB_PORT", "5432")    # safe with default

# Never hardcode credentials – always load from environment
DB_HOST = os.environ["DB_HOST"]
DB_PASS = os.environ["DB_PASSWORD"]
```

logging

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s %(levelname)s %(name)s - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
)
logger = logging.getLogger(__name__)

logger.debug("Processing row %d", row_id)
logger.info("Loaded %d records", count)
logger.warning("Missing values in column: %s", col)
logger.error("Connection failed: %s", str(e))
logger.exception("Unexpected error")    # error + full
                                         traceback
```

itertools

```
import itertools

# Chain multiple iterables into one
list(itertools.chain([1, 2], [3, 4]))    # [1, 2, 3, 4]

# Slice a generator
list(itertools.islice(gen, 100))         # first 100 items

# Group consecutive items by key (input must be sorted by key)
for key, group in itertools.groupby(sorted_rows, key=lambda
r: r["dept"]):
    print(key, list(group))
```

```
# Cartesian product
for x, y in itertools.product([1, 2], ["a", "b"]):
    print(x, y)    # 1a, 1b, 2a, 2b

# Batch into chunks (Python 3.12+: itertools.batched)
def batched(iterable, n):
    it = iter(iterable)
    while chunk := list(itertools.islice(it, n)):
        yield chunk
```

Pandas Essentials

```
import pandas as pd
import numpy as np

# — Reading data —————
df = pd.read_csv("orders.csv", parse_dates=["created_at"])
df = pd.read_parquet("orders.parquet")
df = pd.read_json("orders.json")
df = pd.DataFrame({"id": [1, 2], "name": ["Alice", "Bob"]})

# — Inspection —————
df.shape          # (rows, cols)
df.dtypes
df.head(5)
df.describe()     # stats for numeric columns
df.info()         # column types + null counts
df.isnull().sum() # nulls per column

# — Selecting data —————
df["name"]                # single
column → Series
df[["name", "salary"]]    # multiple
columns → DataFrame
df.loc[0]                 # row by
label
df.iloc[0]                # row by
position
df[df["salary"] > 90_000] # boolean
```

```

filter
df.query("salary > 90000 and dept == 'Eng'") # query
string

# — Adding / transforming columns —————
df["annual"] = df["salary"] * 12
df["name"]   = df["name"].str.upper()
df["dept"]   = df["dept"].str.strip().str.lower()
df["tier"]   = pd.cut(df["salary"],
                      bins=[0, 70_000, 100_000, np.inf],
                      labels=["junior", "mid", "senior"])

df["score"] = df["salary"].apply(lambda x: round(x / 1000,
1))

# — Handling nulls —————
df.dropna(subset=["salary"])
df.fillna({"salary": 0, "phone": "N/A"})

# — GroupBy —————
df.groupby("dept")["salary"].mean()
df.groupby("dept").agg(
    headcount = ("id", "count"),
    avg_salary = ("salary", "mean"),
    max_salary = ("salary", "max"),
)

# — Merge / Join —————
merged = pd.merge(employees, departments, on="dept_id",
how="left")

# — Reshape —————
# Pivot – rows to columns
pivot = df.pivot_table(index="dept", columns="year",
                      values="revenue", aggfunc="sum")

# Melt – wide to long
long = pd.melt(df, id_vars=["id"], value_vars=["q1", "q2",
"q3"],
              var_name="quarter", value_name="revenue")

# — Sort & deduplicate —————

```

```

df.sort_values("salary", ascending=False)
df.drop_duplicates(subset=["email"])
df.drop_duplicates(subset=["id"], keep="last")

# — Writing —————
df.to_csv("output.csv", index=False)
df.to_parquet("output.parquet", index=False,
compression="snappy")
df.to_json("output.json", orient="records", lines=True)  #
NDJSON

```

Working with APIs

```

import requests
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry

# Session with built-in retry
def create_session(retries=3, backoff=1):
    session = requests.Session()
    retry = Retry(
        total=retries,
        backoff_factor=backoff,
        status_forcelist=[429, 500, 502, 503, 504],
    )
    session.mount("https://",
HTTPAdapter(max_retries=retry))
    return session

session = create_session()

# GET
resp = session.get(
    "https://api.example.com/orders",
    params={"status": "shipped", "limit": 100},
    headers={"Authorization": f"Bearer {token}"},
    timeout=30,
)
resp.raise_for_status()  # raises HTTPError on 4xx/5xx
data = resp.json()

```

```

# POST
resp = session.post(
    "https://api.example.com/ingest",
    json={"records": records},    # auto-sets Content-Type:
    application/json
)

# Pagination — fetch all pages
def fetch_all(base_url, params, page_size=100):
    results, page = [], 1
    while True:
        resp = session.get(base_url, params={**params,
"page": page, "limit": page_size})
        resp.raise_for_status()
        batch = resp.json().get("data", [])
        if not batch:
            break
        results.extend(batch)
        page += 1
    return results

```

DE Patterns

Config management — never hardcode secrets

```

import os
from dataclasses import dataclass

@dataclass(frozen=True)
class Config:
    db_host: str
    db_port: int
    db_name: str
    db_pass: str
    s3_bucket: str

    @classmethod
    def from_env(cls) -> "Config":
        # Read env vars when called, not at import time —

```

```

so importing the
    # module never crashes and tests can build a Config
directly.
    return cls(
        db_host    = os.environ["DB_HOST"],
        db_port    = int(os.environ.get("DB_PORT",
"5432")),
        db_name     = os.environ["DB_NAME"],
        db_pass     = os.environ["DB_PASSWORD"],
        s3_bucket   = os.environ["S3_BUCKET"],
    )

config = Config.from_env()

```

Idempotent pipeline task

```

def run_daily_load(logical_date: date) -> None:
    """Load orders for a specific date. Safe to rerun."""
    date_str = logical_date.strftime("%Y-%m-%d")

    conn.execute("DELETE FROM orders WHERE order_date =
%s", (date_str,))

    raw      = extract_orders(date_str)
    cleaned = [transform(r) for r in raw if is_valid(r)]

    conn.executemany("INSERT INTO orders VALUES (%s, %s,
%s)", cleaned)
    conn.commit()
    logger.info("Loaded %d orders for %s", len(cleaned),
date_str)

```

Chunked processing — avoid loading everything into memory

```

def process_large_file(path: str, chunk_size: int = 10_000) -> None:
    total = 0
    for chunk_df in pd.read_csv(path,
chunksize=chunk_size):
        processed = transform(chunk_df)

```

```

write_to_db(processed)
total += len(chunk_df)
logger.info("Processed %d rows so far", total)

```

Schema validation

```

REQUIRED_COLUMNS = {"order_id", "customer_id", "amount",
"created_at"}
NON_NULLABLE      = ["order_id", "amount"]

def validate(df: pd.DataFrame) -> None:
    missing = REQUIRED_COLUMNS - set(df.columns)
    if missing:
        raise SchemaValidationError(f"Missing columns:
{missing}")

    for col in NON_NULLABLE:
        n = df[col].isnull().sum()
        if n > 0:
            raise SchemaValidationError(f"'{col}' has {n}
nulls")

    if (df["amount"] < 0).any():
        raise SchemaValidationError("Negative amounts
found")

```

Watermark-based incremental load

```

def get_last_watermark(conn, table: str) -> datetime:
    row = conn.execute(
        "SELECT MAX(updated_at) FROM watermarks WHERE
table_name = %s", (table,)
    ).fetchone()
    return row[0] or datetime(2000, 1, 1)

def save_watermark(conn, table: str, ts: datetime) -> None:
    conn.execute("""
        INSERT INTO watermarks (table_name, updated_at)
        VALUES (%s, %s)
        ON CONFLICT (table_name) DO UPDATE SET updated_at =

```

```

EXCLUDED.updated_at
    """, (table, ts))

def incremental_load(conn, source_conn, table: str) ->
None:
    watermark = get_last_watermark(conn, table)
    rows = source_conn.execute(
        "SELECT * FROM orders WHERE updated_at > %s",
        (watermark,)
    ).fetchall()
    if rows:
        load(conn, rows)
        save_watermark(conn, table, max(r["updated_at"] for
r in rows))
        logger.info("Loaded %d new rows into %s",
len(rows), table)

```

Common Pitfalls

Pitfall	Symptom	Fix
Mutable default argument <code>def f(rows=[])</code>	Rows leak between calls; list keeps growing	Default to <code>None</code> , then <code>rows = rows or []</code> inside
Reading env vars at import time (module-level or dataclass defaults)	Importing the module crashes when a var is missing; tests can't override config	Read config inside a function / <code>from_env()</code> classmethod
Bare <code>except:</code> or <code>except Exception: pass</code>	Failures swallowed; pipeline "succeeds" with no data	Catch specific exceptions; log and re-raise anything you can't handle
<code>pd.read_csv</code> on a file bigger than RAM	<code>MemoryError</code> or the container gets OOM-killed	<code>chunksize=</code> , read Parquet with column selection, or use DuckDB/Polars/Spark
Letting pandas infer dtypes	IDs with leading zeros lose them; mixed columns become <code>object</code> ; ints become floats when NULLs appear	Pass <code>dtype=</code> explicitly; use nullable types (<code>Int64</code> , <code>string</code>)

Pitfall	Symptom	Fix
<code>df.apply(lambda ...)</code> row by row	100× slower than it needs to be	Vectorized ops (<code>df["a"] * df["b"]</code> , <code>.str</code> , <code>.dt</code> , <code>np.where</code>)
Naive datetimes	Off-by-hours bugs around DST and across regions	Store and compute in UTC with tz-aware datetimes; convert only for display
<code>requests.get(url)</code> with no timeout	Job hangs forever on a stuck connection	Always pass <code>timeout=</code> ; use a <code>Session</code> with <code>Retry</code>
<code>print()</code> instead of logging	No timestamps or levels; nothing reaches the log aggregator	<code>logging</code> with a module logger; structured (JSON) logs in production
Unpinned dependencies	A new library release breaks prod overnight	Pin versions (<code>requirements.txt</code> / lock file); upgrade deliberately

Cheat Sheet

Task	Code
Env var with default	<code>os.environ.get("DB_PORT", "5432")</code>
Required env var	<code>os.environ["DB_HOST"]</code> (raises <code>KeyError</code> if missing)
Today / yesterday (UTC)	<code>datetime.now(timezone.utc).date() · ... - timedelta(days=1)</code>
Parse / format dates	<code>datetime.strptime(s, "%Y-%m-%d") · d.strftime("%Y-%m-%d") · d.isoformat()</code>
Read JSON / write NDJSON	<code>json.load(f) · f.write(json.dumps(r) + "\n")</code>
Stream a big file	<code>for line in open(path): · pd.read_csv(path, chunksize=100_000)</code>
Batch an iterable	<code>itertools.batched(rows, 1000)</code> (3.12+)
Read Parquet columns only	<code>pd.read_parquet(p, columns=["id", "amount"])</code>

Task	Code
Dedupe keep latest	<code>df.sort_values("updated_at").drop_duplicates("id", keep="last")</code>
Group + multiple aggs	<code>df.groupby("k").agg(n=("id", "count"), total=("amt", "sum"))</code>
HTTP with retries	<code>Session() + HTTPAdapter(max_retries=Retry(...)), always timeout=</code>
Logger	<code>logger = logging.getLogger(__name__)</code>
Timer	<code>start = time.perf_counter(); ...; time.perf_counter() - start</code>
Dataclass record	<code>@dataclass(frozen=True) class Order: id: int; amount: float</code>
Virtual env	<code>python -m venv .venv && source .venv/bin/activate</code>

Choose the right container: list (ordered, duplicates) · tuple (fixed record) · set (fast membership, unique) · dict (lookup by key) · generator (one pass, constant memory)

Interview Questions

Q: What is a generator and why is it useful in data pipelines? A: A generator is a function that uses `yield` to produce values one at a time, pausing between them, instead of building a whole list in memory. For pipelines this means you can stream a 50 GB file or a paginated API through transform and load steps using constant memory. The trade-off is that a generator can only be consumed once and doesn't support indexing or `len()`.

Q: What's the difference between a list and a tuple? When would you use a set? A: Lists are mutable and tuples are immutable; tuples are hashable (if their contents are), so they can be dict keys or set members — useful for composite keys like `(customer_id, date)`. Sets store unique values with $O(1)$ average membership tests, so `if id in seen_ids` against a set of a million IDs is fast, whereas the same check against a list is a linear scan.

Q: Explain the mutable default argument problem. A: Default values are evaluated once, when the function is defined — not on each call. So `def add(row, rows=[])` shares one list across every call that doesn't pass `rows`, and data leaks between calls. The fix is `rows=None` and `if rows is None: rows = []` inside the function. The same thing happens with dataclass defaults and module-level code: anything computed at definition time runs once, at import.

Q: How do you process a file that is larger than memory in Python? A: Stream it. For text, iterate line by line; for CSV, `pd.read_csv(..., chunksize=N)` and process each chunk; for Parquet, read only the columns you need or iterate row groups with `pyarrow`. If you need joins or aggregations over the whole dataset, reach for DuckDB or Polars (which spill to disk and are multi-threaded) or Spark if it's truly large.

Q: What makes a Python pipeline task idempotent? A: Rerunning it for the same input produces the same end state. Practically: parameterize on the logical date instead of `datetime.now()`, write with overwrite-partition or `DELETE + INSERT / MERGE` instead of plain appends, write to a temp location and atomically rename or swap, and make side effects (emails, API calls) conditional on not having already happened.

Q: What is the GIL and does it matter for data engineering? A: The Global Interpreter Lock lets only one thread execute Python bytecode at a time in CPython. It doesn't matter much for I/O-bound work (API calls, database queries) — threads release the GIL while waiting, so `ThreadPoolExecutor` speeds those up. It does limit CPU-bound pure-Python work; for that, use `multiprocessing / ProcessPoolExecutor`, or libraries whose heavy lifting runs in native code (NumPy, pandas, Polars, PyArrow). Free-threaded CPython builds (3.13+) are removing this limit, but most production stacks still assume the GIL.

Q: How do you handle secrets in Python pipelines? A: Never in code or Git. Read them from environment variables injected at runtime, or directly from a secrets manager (AWS Secrets Manager, Vault, Airflow connections/Databricks secret scopes). Use `.env` files only locally, and keep them in `.gitignore`. Avoid logging config objects that contain secrets.

Further Reading

- [Python documentation](https://docs.python.org/3/) (<https://docs.python.org/3/>) — especially the tutorial and the standard library reference
 - [pandas user guide](https://pandas.pydata.org/docs/user_guide/index.html) (https://pandas.pydata.org/docs/user_guide/index.html)
 - *Fluent Python* — Luciano Ramalho (O'Reilly). The best "intermediate to advanced" Python book.
 - *Python for Data Analysis* — Wes McKinney (free online at wesmckinney.com/book (<https://wesmckinney.com/book>)), by the creator of pandas
 - [Polars](https://docs.pola.rs/) (<https://docs.pola.rs/>) and [DuckDB Python API](https://duckdb.org/docs/api/python/overview) (<https://duckdb.org/docs/api/python/overview>) — faster options once pandas runs out of memory
 - [Real Python](https://realpython.com/) (<https://realpython.com/>) — practical tutorials on almost every topic above
-

Previous: [SQL](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) · **Next:** [Data Modeling](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/) · **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 29, 2026