pymaster lectures

This document is generated from the source `.py` files.

Python

01_typed_validation.py

```python
# Technique: Typed Validation Boundary (Parse Early, Trust Internal Data)
# Use When:
# - External input (APIs, files, user JSON) is untrusted
# - Validate once at the boundary
# - Convert to a typed domain model, then keep internal code strict and simple

from dataclasses import dataclass
from typing import Any

@dataclass(frozen=True)
class LineItem:
    name: str
    price: float
    qty: int

def parse_line_item(raw: dict[str, Any]) -> LineItem:
    required = {"name", "price", "qty"}
    missing = required - set(raw)
    if missing:
        raise ValueError(f"missing keys: {sorted(missing)}")

    name = raw["name"]
    price = raw["price"]
    qty = raw["qty"]

    if not isinstance(name, str) or not name.strip():
        raise ValueError("name must be a non-empty string")
    if not isinstance(price, (int, float)):
        raise ValueError(f"price must be numeric for '{name}'")
    if not isinstance(qty, int):
        raise ValueError(f"qty must be an integer for '{name}'")
    if price < 0:
        raise ValueError(f"price must be >= 0 for '{name}'")
    if qty <= 0:
        raise ValueError(f"qty must be > 0 for '{name}'")

    return LineItem(name=name, price=float(price), qty=qty)

def calculate_totals(items: list[LineItem], tax: float) -> dict[str, float]:
    if tax < 0:
        raise ValueError("tax must be >= 0")

    result: dict[str, float] = {}
    for item in items:
        if item.name in result:
            raise ValueError(f"duplicate item name: '{item.name}'")
        result[item.name] = round(item.price * item.qty * (1 + tax), 2)
    return result

if __name__ == "__main__":
    raw_items = [
        {"name": "Book", "price": 12.5, "qty": 2},
        {"name": "Pen", "price": 1.2, "qty": 3},
    ]

    typed_items = [parse_line_item(raw) for raw in raw_items]
    print(calculate_totals(typed_items, tax=0.1))
```

02_custom_exceptions.py

```python
# Technique: Custom Exceptions for Clear Error Contracts
# Use When:
# - Callers can handle failures by category instead of string matching
# - Domain errors are separated from low-level runtime errors
# - Error handling becomes predictable in services and APIs

from dataclasses import dataclass

class PricingError(Exception):
    """Base exception for pricing domain errors."""

class InvalidTaxError(PricingError):
    pass

class InvalidItemError(PricingError):
    pass

class DuplicateItemError(PricingError):
    pass

@dataclass(frozen=True)
class LineItem:
    name: str
    price: float
    qty: int

def validate_item(item: LineItem) -> None:
    if not item.name.strip():
        raise InvalidItemError("name must be non-empty")
    if item.price < 0:
        raise InvalidItemError(f"price must be >= 0 for '{item.name}'")
    if item.qty <= 0:
        raise InvalidItemError(f"qty must be > 0 for '{item.name}'")

def calculate_totals(items: list[LineItem], tax: float) -> dict[str, float]:
    if tax < 0:
        raise InvalidTaxError("tax must be >= 0")

    totals: dict[str, float] = {}
    for item in items:
        validate_item(item)
        if item.name in totals:
            raise DuplicateItemError(f"duplicate item name: '{item.name}'")
        totals[item.name] = round(item.price * item.qty * (1 + tax), 2)
    return totals

def run_pricing(items: list[LineItem], tax: float) -> None:
    """Boundary function (like CLI/API handler)."""
    try:
        totals = calculate_totals(items, tax)
        print("OK:", totals)
    except PricingError as exc:
        print("Pricing failed:", exc)

if __name__ == "__main__":
    good_items = [LineItem("Book", 10.0, 2), LineItem("Pen", 1.5, 3)]
    bad_items = [LineItem("", 10.0, 1)]

    run_pricing(good_items, tax=0.1)
    run_pricing(bad_items, tax=0.1)
```

03_structured_logging.py

```python
# Technique: Structured Logging (Context-Rich, Queryable Logs)
# Use When:
# - Logs are machine-parseable and easier to search in production
# - Every log line carries context (request_id, user_id, item_count, etc.)
# - Failures become diagnosable without reproducing locally

import logging
from dataclasses import dataclass

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s %(levelname)s %(name)s %(message)s",
)
logger = logging.getLogger("pricing")

@dataclass(frozen=True)
class LineItem:
    name: str
    price: float
    qty: int

def calculate_totals(items: list[LineItem], tax: float, request_id: str) -> dict[str, float
    logger.info(
        "event=pricing_start request_id=%s item_count=%d tax=%.4f",
        request_id,
        len(items),
        tax,
    )

    if tax < 0:
        logger.warning("event=pricing_invalid_tax request_id=%s tax=%.4f", request_id, tax)
        raise ValueError("tax must be >= 0")

    totals: dict[str, float] = {}
    for item in items:
        if item.price < 0 or item.qty <= 0:
            logger.warning(
                "event=pricing_invalid_item request_id=%s item_name=%s price=%.2f qty=%d",
                request_id,
                item.name,
                item.price,
                item.qty,
            )
            raise ValueError(f"invalid item: {item.name}")
        totals[item.name] = round(item.price * item.qty * (1 + tax), 2)

    logger.info(
        "event=pricing_success request_id=%s unique_items=%d",
        request_id,
        len(totals),
    )
    return totals

if __name__ == "__main__":
    request_id = "req-2026-02-26-001"
    items = [LineItem("Book", 12.5, 2), LineItem("Pen", 1.2, 3)]

    try:
        totals = calculate_totals(items=items, tax=0.1, request_id=request_id)
        print("Totals:", totals)
    except ValueError as exc:
        logger.exception("event=pricing_failed request_id=%s error=%s", request_id, exc)
```

04_dependency_injection.py

```
# Technique: Dependency Injection (Design for Testability)
# Use When:
# - Business logic does not hardcode external systems (DB, API, clock, email)
# - You can swap real implementations with fakes in tests
# - Code becomes modular and easier to evolve

from dataclasses import dataclass
from typing import Protocol

@dataclass(frozen=True)
class LineItem:
    name: str
    price: float
    qty: int

class TaxProvider(Protocol):
    def get_tax_rate(self, region: str) -> float:
        ...

class StaticTaxProvider:
    def __init__(self, rates: dict[str, float]) -> None:
        self._rates = rates

    def get_tax_rate(self, region: str) -> float:
        if region not in self._rates:
            raise ValueError(f"unknown region: {region}")
        return self._rates[region]

class PricingService:
    def __init__(self, tax_provider: TaxProvider) -> None:
        self._tax_provider = tax_provider

    def calculate_totals(self, items: list[LineItem], region: str) -> dict[str, float]:
        tax = self._tax_provider.get_tax_rate(region)
        if tax < 0:
            raise ValueError("tax must be >= 0")

        totals: dict[str, float] = {}
        for item in items:
            if item.qty <= 0 or item.price < 0:
                raise ValueError(f"invalid item: {item.name}")
            totals[item.name] = round(item.price * item.qty * (1 + tax), 2)

        return totals
```

```
if __name__ == "__main__":
    provider = StaticTaxProvider({"US-CA": 0.0825, "US-NY": 0.088})
    service = PricingService(provider)

    items = [LineItem("Book", 12.5, 2), LineItem("Pen", 1.2, 3)]
    print(service.calculate_totals(items, region="US-CA"))
```

05_pytest_unit_tests.py

```
# Technique: Unit Testing with pytest (Fast, Focused, Deterministic)
# Use When:
# - Tests verify behavior and protect against regressions
# - Unit tests run fast because they isolate one unit of logic
# - Parametrization reduces repetitive test code

from dataclasses import dataclass

@dataclass(frozen=True)
class LineItem:
    name: str
    price: float
    qty: int

def calculate_total(item: LineItem, tax: float) -> float:
    if tax < 0:
        raise ValueError("tax must be >= 0")
    if item.price < 0:
        raise ValueError("price must be >= 0")
    if item.qty <= 0:
        raise ValueError("qty must be > 0")

    return round(item.price * item.qty * (1 + tax), 2)

# Example pytest tests (put in a test file, e.g. tests/test_pricing.py)
# ---------------------------------------------------------------
# import pytest
# from 05_pytest_unit_tests import LineItem, calculate_total
#
# def test_calculate_total_happy_path():
#     item = LineItem(name="Book", price=10.0, qty=2)
#     assert calculate_total(item, tax=0.1) == 22.0
#
# @pytest.mark.parametrize(
#     "item,tax,expected_error",
#     [
#         (LineItem("Book", -1.0, 1), 0.1, "price must be >= 0"),
#         (LineItem("Book", 10.0, 0), 0.1, "qty must be > 0"),
#         (LineItem("Book", 10.0, 1), -0.01, "tax must be >= 0"),
#     ],
# )
# def test_calculate_total_validation_errors(item, tax, expected_error):
#     with pytest.raises(ValueError, match=expected_error):
#         calculate_total(item, tax)
```

```python
if __name__ == "__main__":
    # Demo run only; real verification should be in pytest tests.
    sample = LineItem(name="Book", price=10.0, qty=2)
    print(calculate_total(sample, tax=0.1))
```

06_context_managers.py

```
# Technique: Context Managers (Reliable Resource Cleanup)
# Use When:
# - Resources (files, DB sessions, locks, timers) are always cleaned up
# - Cleanup runs even when exceptions happen
# - Code is safer and easier to reason about

from contextlib import contextmanager
from time import perf_counter

@contextmanager
def timed_block(label: str):
    start = perf_counter()
    try:
        yield
    finally:
        duration_ms = (perf_counter() - start) * 1000
        print(f"{label} took {duration_ms:.2f} ms")

def load_numbers(path: str) -> list[int]:
    numbers: list[int] = []
    # Built-in context manager closes file automatically.
    with open(path, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line:
                continue
            numbers.append(int(line))
    return numbers

def compute_sum(path: str) -> int:
    with timed_block("compute_sum"):
        values = load_numbers(path)
        return sum(values)

if __name__ == "__main__":
    sample_path = "/tmp/pymaster_numbers.txt"

    with open(sample_path, "w", encoding="utf-8") as f:
        f.write("10\n20\n30\n")

    total = compute_sum(sample_path)
    print("Total:", total)
```

07_dataclasses_vs_classes.py

```python
# Technique: Dataclasses vs Regular Classes (Choose the Right Model)
# Use When:
# - `@dataclass` removes boilerplate for data containers
# - Regular classes are better when behavior/lifecycle dominates
# - Clear model choice improves readability and maintainability

from dataclasses import dataclass

@dataclass(frozen=True)
class Money:
    amount: float
    currency: str

    def add(self, other: "Money") -> "Money":
        if self.currency != other.currency:
            raise ValueError("cannot add different currencies")
        return Money(amount=round(self.amount + other.amount, 2), currency=self.currency)

class InvoiceService:
    def __init__(self, tax_rate: float) -> None:
        if tax_rate < 0:
            raise ValueError("tax_rate must be >= 0")
        self._tax_rate = tax_rate

    def total_with_tax(self, subtotal: Money) -> Money:
        taxed = subtotal.amount * (1 + self._tax_rate)
        return Money(amount=round(taxed, 2), currency=subtotal.currency)

if __name__ == "__main__":
    line1 = Money(10.0, "USD")
    line2 = Money(15.5, "USD")
    subtotal = line1.add(line2)

    service = InvoiceService(tax_rate=0.1)
    total = service.total_with_tax(subtotal)

    print("Subtotal:", subtotal)
    print("Total:", total)
```

08_abc_vs_protocol.py

```python
# Technique: ABC vs Protocol (Nominal vs Structural Interfaces)
# Use When:
# - You choose interface style based on control and flexibility needs
# - ABC enforces explicit inheritance (nominal typing)
# - Protocol enables duck-typed compatibility with static checks (structural typing)

from abc import ABC, abstractmethod
from typing import Protocol

# ---------- ABC example (explicit inheritance required) ----------
class PaymentGatewayABC(ABC):
    @abstractmethod
    def charge(self, amount: float) -> str:
        pass

class StripeGateway(PaymentGatewayABC):
    def charge(self, amount: float) -> str:
        if amount <= 0:
            raise ValueError("amount must be > 0")
        return f"stripe_charge_id_{int(amount * 100)}"

def checkout_with_abc(gateway: PaymentGatewayABC, amount: float) -> str:
    return gateway.charge(amount)

# ---------- Protocol example (inheritance optional) ----------
class PaymentGatewayProtocol(Protocol):
    def charge(self, amount: float) -> str:
        ...

class MockGateway:
    # No inheritance from Protocol; still compatible by signature.
    def charge(self, amount: float) -> str:
        if amount <= 0:
            raise ValueError("amount must be > 0")
        return f"mock_charge_id_{int(amount * 100)}"

def checkout_with_protocol(gateway: PaymentGatewayProtocol, amount: float) -> str:
    return gateway.charge(amount)

if __name__ == "__main__":
    stripe = StripeGateway()
    print("ABC checkout:", checkout_with_abc(stripe, 19.99))

    mock = MockGateway()
```

```python
print("Protocol checkout:", checkout_with_protocol(mock, 19.99))
```

09_asyncio_basics.py

```python
# Technique: asyncio Basics (Concurrent I/O without Threads)
# Use When:
# - Efficient for many I/O-bound tasks (HTTP calls, DB/network waits)
# - Clear control over concurrency with `await` and task groups
# - Better throughput when tasks spend time waiting

import asyncio
from dataclasses import dataclass

@dataclass(frozen=True)
class Product:
    sku: str
    base_price: float

async def fetch_discount(sku: str) -> float:
    # Simulated I/O latency (e.g., external API call)
    await asyncio.sleep(0.2)
    return 0.1 if sku.startswith("A") else 0.05

async def price_with_discount(product: Product) -> tuple[str, float]:
    discount = await fetch_discount(product.sku)
    final_price = round(product.base_price * (1 - discount), 2)
    return product.sku, final_price

async def price_all(products: list[Product]) -> dict[str, float]:
    tasks = [price_with_discount(product) for product in products]
    pairs = await asyncio.gather(*tasks)
    return dict(pairs)

async def main() -> None:
    products = [
        Product("A-BOOK", 20.0),
        Product("B-PEN", 5.0),
        Product("A-NOTE", 10.0),
    ]
    priced = await price_all(products)
    print(priced)

if __name__ == "__main__":
    asyncio.run(main())
```

10_retries_and_backoff.py

```python
# Technique: Retries with Exponential Backoff (Resilient External Calls)
# Use When:
# - External services fail transiently (timeouts, rate limits, network blips)
# - Controlled retries improve reliability without spamming dependencies
# - Backoff + jitter reduces synchronized retry storms

import random
import time
from typing import Callable

class TransientError(Exception):
    pass

class PermanentError(Exception):
    pass

def compute_backoff_delay(attempt: int, base_delay: float, max_delay: float) -> float:
    delay = min(base_delay * (2 ** (attempt - 1)), max_delay)
    jitter = random.uniform(0, delay * 0.25)
    return delay + jitter

def with_retries(
    func: Callable[[], str],
    *,
    attempts: int = 4,
    base_delay: float = 0.2,
    max_delay: float = 2.0,
) -> str:
    if attempts < 1:
        raise ValueError("attempts must be >= 1")

    for attempt in range(1, attempts + 1):
        try:
            return func()
        except PermanentError:
            # Do not retry non-transient failures.
            raise
        except TransientError as exc:
            if attempt == attempts:
                raise RuntimeError(f"failed after {attempts} attempts") from exc

            sleep_for = compute_backoff_delay(attempt, base_delay, max_delay)
            print(f"attempt={attempt} transient_error='{exc}' retry_in={sleep_for:.2f}s")
            time.sleep(sleep_for)
    raise RuntimeError("unexpected retry flow")

def flaky_call_factory() -> Callable[[], str]:
    state = {"count": 0}

    def flaky_call() -> str:
        state["count"] += 1
        if state["count"] < 3:
            raise TransientError("temporary timeout")
        return "success"

    return flaky_call

if __name__ == "__main__":
    call = flaky_call_factory()
    result = with_retries(call, attempts=5, base_delay=0.1, max_delay=0.5)
    print("result:", result)
```

11_configuration_management.py

```python
# Technique: Configuration Management (Single Source of Runtime Settings)
# Use When:
# - Keeps environment-specific values out of business logic
# - Makes local/dev/prod behavior explicit and auditable
# - Fails fast when required config is missing

import os
from dataclasses import dataclass

@dataclass(frozen=True)
class AppConfig:
    app_env: str
    db_url: str
    request_timeout_s: float

def load_config() -> AppConfig:
    app_env = os.getenv("APP_ENV", "dev")
    db_url = os.getenv("DB_URL")
    timeout_raw = os.getenv("REQUEST_TIMEOUT_S", "2.0")

    if not db_url:
        raise RuntimeError("DB_URL is required")

    try:
        timeout = float(timeout_raw)
    except ValueError as exc:
        raise RuntimeError("REQUEST_TIMEOUT_S must be numeric") from exc

    if timeout <= 0:
        raise RuntimeError("REQUEST_TIMEOUT_S must be > 0")

    return AppConfig(app_env=app_env, db_url=db_url, request_timeout_s=timeout)

if __name__ == "__main__":
    os.environ.setdefault("DB_URL", "postgresql://localhost:5432/pymaster")
    config = load_config()
    print(config)
```

12_pathlib_and_file_io.py

```
# Technique: pathlib + Explicit File I/O
# Use When:
# - `Path` objects are clearer and less error-prone than string paths
# - File operations become cross-platform and composable
# - Encoding/newline behavior is explicit

from pathlib import Path

def write_report(path: Path, lines: list[str]) -> None:
    path.parent.mkdir(parents=True, exist_ok=True)
    payload = "\n".join(lines) + "\n"
    path.write_text(payload, encoding="utf-8")

def read_report(path: Path) -> list[str]:
    text = path.read_text(encoding="utf-8")
    return [line for line in text.splitlines() if line.strip()]

if __name__ == "__main__":
    report_path = Path("/tmp/pymaster") / "daily_report.txt"
    write_report(report_path, ["status=ok", "items=3"])
    print(read_report(report_path))
```

13_iterators_and_generators.py

```python
# Technique: Generators for Streaming Data
# Use When:
# - Processes large inputs lazily instead of loading everything in memory
# - Keeps pipelines composable and efficient
# - Simplifies line-by-line or record-by-record processing

from collections.abc import Iterable, Iterator

def parse_ints(rows: Iterable[str]) -> Iterator[int]:
    for row in rows:
        row = row.strip()
        if not row:
            continue
        yield int(row)

def non_negative(values: Iterable[int]) -> Iterator[int]:
    for value in values:
        if value >= 0:
            yield value

def squared(values: Iterable[int]) -> Iterator[int]:
    for value in values:
        yield value * value

if __name__ == "__main__":
    rows = ["10", "-3", "5", "", "2"]
    pipeline = squared(non_negative(parse_ints(rows)))
    print(list(pipeline))
```

14_lru_cache.py

```python
# Technique: Caching with functools.lru_cache
# Use When:
# - Avoids repeating expensive pure computations
# - Improves latency with minimal code
# - Makes performance tuning explicit and bounded

import time
from functools import lru_cache

@lru_cache(maxsize=128)
def currency_rate(base: str, quote: str) -> float:
    # Simulate expensive call.
    time.sleep(0.2)
    rates = {("USD", "EUR"): 0.92, ("USD", "JPY"): 148.0}
    if (base, quote) not in rates:
        raise ValueError(f"unsupported pair: {base}/{quote}")
    return rates[(base, quote)]

if __name__ == "__main__":
    print(currency_rate("USD", "EUR"))
    print(currency_rate("USD", "EUR"))
    print(currency_rate.cache_info())
```

15_sqlite_repository_pattern.py

```
# Technique: Repository Pattern with sqlite3
# Use When:
# - Keeps SQL access isolated from business logic
# - Enables easier testing and future DB migration
# - Defines clear persistence boundaries

import sqlite3
from dataclasses import dataclass

@dataclass(frozen=True)
class User:
    id: int
    email: str

class UserRepository:
    def __init__(self, conn: sqlite3.Connection) -> None:
        self._conn = conn

    def init_schema(self) -> None:
        self._conn.execute(
            "CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, email TEXT NOT NULL U
        )
        self._conn.commit()

    def create(self, email: str) -> User:
        cur = self._conn.execute("INSERT INTO users (email) VALUES (?)", (email,))
        self._conn.commit()
        return User(id=int(cur.lastrowid), email=email)

    def get(self, user_id: int) -> User | None:
        row = self._conn.execute("SELECT id, email FROM users WHERE id = ?", (user_id,)).fet
        if row is None:
            return None
        return User(id=row[0], email=row[1])

if __name__ == "__main__":
    conn = sqlite3.connect(":memory:")
    repo = UserRepository(conn)
    repo.init_schema()
    created = repo.create("dev@example.com")
    print(repo.get(created.id))
```

16_argparse_cli.py

```python
# Technique: CLI Design with argparse
# Use When:
# - Provides discoverable commands and help text
# - Validates user input at the command boundary
# - Makes scripts reusable in automation and CI

import argparse

def calculate_total(price: float, qty: int, tax: float) -> float:
    if price < 0 or qty <= 0 or tax < 0:
        raise ValueError("invalid input values")
    return round(price * qty * (1 + tax), 2)

def main() -> int:
    parser = argparse.ArgumentParser(description="Compute line total with tax")
    parser.add_argument("--price", type=float, required=True)
    parser.add_argument("--qty", type=int, required=True)
    parser.add_argument("--tax", type=float, default=0.0)
    args = parser.parse_args()

    try:
        total = calculate_total(args.price, args.qty, args.tax)
    except ValueError as exc:
        print(f"error: {exc}")
        return 2

    print(total)
    return 0

if __name__ == "__main__":
    raise SystemExit(main())
```

17_type_narrowing.py

```python
# Technique: Type Narrowing and Type Guards
# Use When:
# - Reduces runtime type errors
# - Lets static checkers prove safety after validation
# - Makes heterogeneous input handling explicit

from typing import Any, TypeGuard

def is_price_payload(value: Any) -> TypeGuard[dict[str, float]]:
    if not isinstance(value, dict):
        return False
    return isinstance(value.get("price"), (int, float)) and isinstance(value.get("tax"), (i

def compute_from_unknown(payload: Any) -> float:
    if not is_price_payload(payload):
        raise ValueError("payload must contain numeric price and tax")

    price = float(payload["price"])
    tax = float(payload["tax"])
    if price < 0 or tax < 0:
        raise ValueError("price and tax must be >= 0")
    return round(price * (1 + tax), 2)

if __name__ == "__main__":
    print(compute_from_unknown({"price": 10.0, "tax": 0.1}))
```

18_threadpool_concurrency.py

```python
# Technique: ThreadPoolExecutor for Blocking I/O
# Use When:
# - Speeds up independent blocking operations
# - Keeps code simple when APIs are synchronous
# - Controls concurrency level explicitly

import time
from concurrent.futures import ThreadPoolExecutor, as_completed

def fetch_profile(user_id: int) -> str:
    # Simulated blocking I/O call.
    time.sleep(0.2)
    return f"user-{user_id}"

def fetch_many(user_ids: list[int]) -> list[str]:
    results: list[str] = []
    with ThreadPoolExecutor(max_workers=4) as pool:
        futures = [pool.submit(fetch_profile, user_id) for user_id in user_ids]
        for future in as_completed(futures):
            results.append(future.result())
    return results

if __name__ == "__main__":
    print(fetch_many([1, 2, 3, 4, 5]))
```

19_http_client_pattern.py

```python
# Technique: HTTP Client Wrapper Pattern
# Use When:
# - Centralizes timeout, headers, retries, and error mapping
# - Keeps endpoint callers thin and consistent
# - Makes integration points easy to mock in tests

from dataclasses import dataclass

class ApiClientError(Exception):
    pass

@dataclass(frozen=True)
class UserDTO:
    id: int
    email: str

class ApiClient:
    def __init__(self, base_url: str, timeout_s: float = 2.0) -> None:
        self._base_url = base_url.rstrip("/")
        self._timeout_s = timeout_s

    def get_user(self, user_id: int) -> UserDTO:
        # Placeholder for real HTTP request (requests/httpx).
        # In production, this method would call an API endpoint.
        if user_id <= 0:
            raise ApiClientError("user_id must be positive")
        return UserDTO(id=user_id, email=f"user{user_id}@example.com")

if __name__ == "__main__":
    client = ApiClient("https://api.example.com", timeout_s=2.0)
    print(client.get_user(1))
```

20_project_structure_example.py

```python
# Technique: Professional Project Structure
# Use When:
# - Separates concerns (domain, infra, interfaces)
# - Improves onboarding and discoverability
# - Scales better as codebase size grows

PROJECT_LAYOUT = {
    "src/pymaster/domain/": ["models.py", "services.py"],
    "src/pymaster/infra/": ["db.py", "http_client.py"],
    "src/pymaster/interfaces/": ["cli.py", "api.py"],
    "tests/": ["test_services.py", "test_api.py"],
}

def print_layout() -> None:
    for folder, files in PROJECT_LAYOUT.items():
        print(folder)
        for file in files:
            print(f"  - {file}")

if __name__ == "__main__":
    print_layout()
```

21_invariants_and_properties.py

```
# Technique: Encapsulation with Invariants + @property
# Use When:
# - Objects protect their own correctness (invariants)
# - Callers use a simple API; representation can change safely
# - Properties expose computed values without leaking internal state

from dataclasses import dataclass

@dataclass
class BankAccount:
    _balance_cents: int

    def __post_init__(self) -> None:
        if self._balance_cents < 0:
            raise ValueError("balance must be >= 0")

    @property
    def balance(self) -> float:
        return self._balance_cents / 100

    def deposit(self, amount: float) -> None:
        if amount <= 0:
            raise ValueError("deposit must be > 0")
        self._balance_cents += int(round(amount * 100))

    def withdraw(self, amount: float) -> None:
        if amount <= 0:
            raise ValueError("withdrawal must be > 0")
        cents = int(round(amount * 100))
        if cents > self._balance_cents:
            raise ValueError("insufficient funds")
        self._balance_cents -= cents

if __name__ == "__main__":
    acct = BankAccount(0)
    acct.deposit(10.25)
    acct.withdraw(3.00)
    print("balance:", acct.balance)
```

22_classmethods_and_factories.py

```python
# Technique: @classmethod Factories (Multiple Constructors)
# Use When:
# - Object creation rules stay centralized
# - Intent is explicit (from_env, from_json, from_dsn)
# - Constructor changes can remain backwards-compatible

import os
from dataclasses import dataclass

@dataclass(frozen=True)
class DbConfig:
    host: str
    port: int

    @classmethod
    def from_env(cls) -> "DbConfig":
        host = os.getenv("DB_HOST", "localhost")
        port_raw = os.getenv("DB_PORT", "5432")
        try:
            port = int(port_raw)
        except ValueError as exc:
            raise ValueError("DB_PORT must be an int") from exc
        if not (1 <= port <= 65535):
            raise ValueError("DB_PORT out of range")
        return cls(host=host, port=port)

    @classmethod
    def from_dsn(cls, dsn: str) -> "DbConfig":
        # Minimal parser: "host:port".
        if ":" not in dsn:
            raise ValueError("dsn must be 'host:port'")
        host, port_raw = dsn.split(":", 1)
        return cls(host=host, port=int(port_raw))

if __name__ == "__main__":
    os.environ.setdefault("DB_HOST", "db")
    os.environ.setdefault("DB_PORT", "5432")
    print(DbConfig.from_env())
    print(DbConfig.from_dsn("db:5432"))
```

23_abc_and_template_method.py

```python
# Technique: ABC + Template Method
# Use When:
# - One stable workflow is implemented once
# - Subclasses customize only specific steps
# - Keeps rules consistent and reduces duplication

from abc import ABC, abstractmethod

class ReportGenerator(ABC):
    def generate(self) -> str:
        header = self.header()
        body = self.body()
        footer = self.footer()
        return "\n".join([header, body, footer])

    @abstractmethod
    def header(self) -> str:
        raise NotImplementedError

    @abstractmethod
    def body(self) -> str:
        raise NotImplementedError

    def footer(self) -> str:
        return "-- end --"

class SalesReport(ReportGenerator):
    def header(self) -> str:
        return "Sales Report"

    def body(self) -> str:
        return "total=1234"

if __name__ == "__main__":
    print(SalesReport().generate())
```

24_composition_over_inheritance.py

```python
# Technique: Composition Over Inheritance
# Use When:
# - Inheritance is strong coupling; composition stays flexible
# - Behaviors can be swapped at runtime (policies/strategies)
# - Avoids fragile base-class problems

from dataclasses import dataclass
from typing import Protocol

@dataclass(frozen=True)
class LineItem:
    name: str
    price: float
    qty: int

class DiscountPolicy(Protocol):
    def discount_rate(self, item: LineItem) -> float:
        ...

class NoDiscount:
    def discount_rate(self, item: LineItem) -> float:
        return 0.0

class NamePrefixDiscount:
    def __init__(self, prefix: str, rate: float) -> None:
        self._prefix = prefix
        self._rate = rate

    def discount_rate(self, item: LineItem) -> float:
        return self._rate if item.name.startswith(self._prefix) else 0.0

class Pricing:
    def __init__(self, discount_policy: DiscountPolicy) -> None:
        self._discount_policy = discount_policy

    def total(self, item: LineItem, tax: float) -> float:
        if tax < 0:
            raise ValueError("tax must be >= 0")
        if item.price < 0 or item.qty <= 0:
            raise ValueError(f"invalid item: {item.name}")

        rate = self._discount_policy.discount_rate(item)
        if not (0 <= rate < 1):
            raise ValueError("discount rate out of range")
        subtotal = item.price * item.qty
        discounted = subtotal * (1 - rate)
        return round(discounted * (1 + tax), 2)

if __name__ == "__main__":
    item = LineItem("A-BOOK", 20.0, 1)
    print(Pricing(NoDiscount()).total(item, tax=0.1))
    print(Pricing(NamePrefixDiscount("A-", 0.2)).total(item, tax=0.1))
```

25_mixins.py

```python
# Technique: Mixins (Optional Shared Behavior)
# Use When:
# - Reuses small behavior without deep inheritance
# - Keeps feature sets composable
# - Avoids monolithic base classes

from dataclasses import dataclass

class JsonSerializableMixin:
    def to_json(self) -> dict[str, object]:
        return self.__dict__.copy()

class AuditMixin:
    def audit_line(self) -> str:
        return f"audit type={type(self).__name__}"

@dataclass
class User(JsonSerializableMixin, AuditMixin):
    id: int
    email: str

if __name__ == "__main__":
    user = User(id=1, email="dev@example.com")
    print(user.to_json())
    print(user.audit_line())
```

26_dunder_methods.py

```python
# Technique: Dunder Methods (__repr__, __str__, ordering)
# Use When:
# - Good __repr__ helps debugging and logging
# - Rich comparisons enable sorting and consistent behavior
# - Value objects behave like values

from dataclasses import dataclass

@dataclass(frozen=True, order=True)
class Version:
    major: int
    minor: int
    patch: int

    def __str__(self) -> str:
        return f"{self.major}.{self.minor}.{self.patch}"

if __name__ == "__main__":
    versions = [Version(1, 2, 0), Version(1, 1, 9), Version(2, 0, 0)]
    print("sorted:", [str(v) for v in sorted(versions)])
    print("repr:", versions[0])
```

27_dataclass_slots.py

```
# Technique: Dataclasses with slots (Memory + Attribute Safety)
# Use When:
# - `slots=True` reduces per-instance overhead for many objects
# - Prevents accidental attribute creation (typos become errors)
# - Works well for large collections of objects

from dataclasses import dataclass

@dataclass(frozen=True, slots=True)
class Point:
    x: float
    y: float

if __name__ == "__main__":
    p = Point(1.0, 2.0)
    print(p)
    # This would fail (good): p.z = 3.0
```

28_descriptors.py

```python
# Technique: Descriptors (Reusable Attribute Validation)
# Use When:
# - Centralizes repeated validation logic
# - Enforces invariants across multiple classes
# - Useful for lightweight domain models

from __future__ import annotations

class PositiveFloat:
    def __set_name__(self, owner: type, name: str) -> None:
        self._private_name = f"_{name}"

    def __get__(self, obj: object, objtype: type | None = None) -> float:
        if obj is None:
            return self  # type: ignore[return-value]
        return float(getattr(obj, self._private_name))

    def __set__(self, obj: object, value: float) -> None:
        if not isinstance(value, (int, float)):
            raise TypeError("value must be numeric")
        if value <= 0:
            raise ValueError("value must be > 0")
        setattr(obj, self._private_name, float(value))

class Product:
    price = PositiveFloat()

    def __init__(self, name: str, price: float) -> None:
        self.name = name
        self.price = price

if __name__ == "__main__":
    p = Product("Book", 10.0)
    print(p.name, p.price)
```

29_context_manager_class.py

```python
# Technique: Context Manager as a Class (__enter__/__exit__)
# Use When:
# - Stateful context managers are sometimes clearer than generator-based
# - Useful for resource objects (connections, locks, tracing spans)
# - __exit__ controls whether exceptions are suppressed

from __future__ import annotations

from time import perf_counter

class Timer:
    def __init__(self, label: str) -> None:
        self._label = label
        self._start: float | None = None

    def __enter__(self) -> "Timer":
        self._start = perf_counter()
        return self

    def __exit__(self, exc_type, exc, tb) -> bool:
        if self._start is None:
            return False
        duration_ms = (perf_counter() - self._start) * 1000
        print(f"{self._label} took {duration_ms:.2f} ms")
        return False

if __name__ == "__main__":
    with Timer("work"):
        total = sum(range(100_000))
    print("total:", total)
```

30_strategy_pattern.py

```python
# Technique: Strategy Pattern (Swap Algorithms Cleanly)
# Use When:
# - Avoids large if/else trees for behavior selection
# - Makes algorithms testable in isolation
# - Enables runtime selection (config, feature flags)

from dataclasses import dataclass
from typing import Protocol

@dataclass(frozen=True)
class Order:
    subtotal: float

class ShippingStrategy(Protocol):
    def shipping_cost(self, order: Order) -> float:
        ...

class FlatRateShipping:
    def __init__(self, fee: float) -> None:
        self._fee = fee

    def shipping_cost(self, order: Order) -> float:
        return self._fee

class FreeOverThreshold:
    def __init__(self, threshold: float, fee: float) -> None:
        self._threshold = threshold
        self._fee = fee

    def shipping_cost(self, order: Order) -> float:
        return 0.0 if order.subtotal >= self._threshold else self._fee

class CheckoutService:
    def __init__(self, shipping: ShippingStrategy) -> None:
        self._shipping = shipping

    def total(self, order: Order) -> float:
        if order.subtotal < 0:
            raise ValueError("subtotal must be >= 0")
        return round(order.subtotal + self._shipping.shipping_cost(order), 2)

if __name__ == "__main__":
    order = Order(subtotal=49.99)
    print(CheckoutService(FlatRateShipping(5.0)).total(order))
print(CheckoutService(FreeOverThreshold(50.0, 5.0)).total(order))
```

31_pure_functions.py

```python
# Technique: Pure Functions (Predictable, Testable Code)
# Use When:
# - Pure functions are deterministic: same inputs, same output
# - No hidden side effects makes testing and reasoning easy
# - Encourages clean boundaries between logic and I/O

from dataclasses import dataclass

@dataclass(frozen=True)
class LineItem:
    name: str
    price: float
    qty: int

def line_total(item: LineItem, tax: float) -> float:
    if tax < 0 or item.price < 0 or item.qty <= 0:
        raise ValueError("invalid inputs")
    return round(item.price * item.qty * (1 + tax), 2)

if __name__ == "__main__":
    print(line_total(LineItem("Book", 10.0, 2), tax=0.1))
```

32_higher_order_functions.py

```python
# Technique: Higher-Order Functions (Functions that Take/Return Functions)
# Use When:
# - Lets you inject behavior without inheritance
# - Enables reusable transformations and policies
# - Keeps code declarative and composable

from collections.abc import Callable

def make_multiplier(factor: float) -> Callable[[float], float]:
    def multiply(x: float) -> float:
        return x * factor

    return multiply

def apply_all(values: list[float], fn: Callable[[float], float]) -> list[float]:
    return [fn(v) for v in values]

if __name__ == "__main__":
    times_two = make_multiplier(2)
    print(apply_all([1.0, 2.5, 3.0], times_two))
```

33_functor_map_filter.py

```python
# Technique: map/filter + Comprehensions (Declarative Data Transforms)
# Use When:
# - Expresses transformation intent clearly
# - Reduces mutable state and loop noise
# - Easy to test by comparing input/output

def normalize_emails(raw: list[str]) -> list[str]:
    return [e.strip().lower() for e in raw if e.strip()]

if __name__ == "__main__":
    data = [" Dev@Example.com ", "", "ADMIN@EXAMPLE.COM"]
    print(normalize_emails(data))
```

34_reduce_and_folds.py

```python
# Technique: reduce (Folds) for Aggregation
# Use When:
# - Captures aggregation logic as a single expression
# - Useful when building up a value with a clear accumulator
# - Works well with immutable accumulator patterns

from functools import reduce

def total_length(words: list[str]) -> int:
    return reduce(lambda acc, w: acc + len(w), words, 0)

if __name__ == "__main__":
    print(total_length(["a", "bb", "ccc"]))
```

35_partial_and_currying.py

```python
# Technique: functools.partial (Practical Currying)
# Use When:
# - Pre-binds arguments to create specialized functions
# - Reduces repetitive parameter passing
# - Creates clearer APIs for pipelines

from functools import partial

def compute_total(price: float, qty: int, tax: float) -> float:
    if price < 0 or qty <= 0 or tax < 0:
        raise ValueError("invalid inputs")
    return round(price * qty * (1 + tax), 2)

if __name__ == "__main__":
    compute_with_tax = partial(compute_total, tax=0.1)
    print(compute_with_tax(price=10.0, qty=2))
```

36_closures_and_state.py

```python
# Technique: Closures (State without Classes)
# Use When:
# - Encapsulates state in a small, testable unit
# - Useful for lightweight counters, memoization, configuration
# - Avoids over-engineering with classes

from collections.abc import Callable

def make_counter() -> Callable[[], int]:
    state = {"count": 0}

    def inc() -> int:
        state["count"] += 1
        return state["count"]

    return inc

if __name__ == "__main__":
    c = make_counter()
    print(c(), c(), c())
```

37_pattern_matching.py

```python
# Technique: Structural Pattern Matching (match/case)
# Use When:
# - Clear handling of variant-shaped inputs
# - Replaces long if/elif chains
# - Works well for parsing commands and events

def handle_event(event: dict[str, object]) -> str:
    match event:
        case {"type": "user_created", "id": int(user_id)}:
            return f"create user {user_id}"
        case {"type": "user_deleted", "id": int(user_id)}:
            return f"delete user {user_id}"
        case {"type": str(t)}:
            return f"unknown event type: {t}"
        case _:
            return "invalid event"

if __name__ == "__main__":
    print(handle_event({"type": "user_created", "id": 1}))
    print(handle_event({"type": "x", "id": 2}))
    print(handle_event({"no": "shape"}))
```

38_itertools_pipelines.py

```python
# Technique: itertools Pipelines (Lazy, Composable Data Processing)
# Use When:
# - Efficient for large sequences (lazy evaluation)
# - Powerful building blocks for data pipelines
# - Keeps memory usage low

import itertools

def chunked(values: list[int], size: int) -> list[list[int]]:
    if size <= 0:
        raise ValueError("size must be > 0")
    it = iter(values)
    out: list[list[int]] = []
    while True:
        chunk = list(itertools.islice(it, size))
        if not chunk:
            break
        out.append(chunk)
    return out

if __name__ == "__main__":
    print(chunked([1, 2, 3, 4, 5], 2))
```

39_immutable_data.py

```python
# Technique: Immutable Data (Safer by Default)
# Use When:
# - Reduces bugs from shared mutable state
# - Makes functions easier to reason about
# - Fits naturally with concurrency

from dataclasses import dataclass

@dataclass(frozen=True)
class Cart:
    items: tuple[str, ...]

def add_item(cart: Cart, item: str) -> Cart:
    if not item:
        raise ValueError("item must be non-empty")
    return Cart(items=cart.items + (item,))

if __name__ == "__main__":
    cart = Cart(items=())
    cart2 = add_item(cart, "Book")
    print(cart, cart2)
```

40_functional_error_handling.py

```python
# Technique: Functional Error Handling (Result Pattern)
# Use When:
# - Makes success/failure explicit in return types
# - Avoids using exceptions for expected control flow
# - Works well in pipelines and batch processing

from dataclasses import dataclass

@dataclass(frozen=True)
class Ok:
    value: float

@dataclass(frozen=True)
class Err:
    error: str

Result = Ok | Err

def parse_float(raw: str) -> Result:
    try:
        return Ok(float(raw))
    except ValueError:
        return Err(f"not a float: {raw!r}")

if __name__ == "__main__":
    for raw in ["1.2", "x"]:
        res = parse_float(raw)
        match res:
            case Ok(value=v):
                print("ok", v)
            case Err(error=e):
                print("err", e)
```

41_logging_dictconfig.py

```python
# Technique: Centralized Logging with dictConfig
# Use When:
# - One place to configure log level, format, and handlers
# - Consistent, structured-ish logs across modules
# - Easy to tune behavior per environment (dev/stage/prod)

import logging
import logging.config

def configure_logging(level: str = "INFO") -> None:
    logging.config.dictConfig(
        {
            "version": 1,
            "disable_existing_loggers": False,
            "formatters": {
                "default": {
                    "format": "%(asctime)s %(levelname)s %(name)s %(message)s",
                }
            },
            "handlers": {
                "console": {
                    "class": "logging.StreamHandler",
                    "formatter": "default",
                }
            },
            "root": {
                "level": level,
                "handlers": ["console"],
            },
        }
    )

logger = logging.getLogger("app")

def run_job(job_id: str) -> None:
    logger.info("event=job_start job_id=%s", job_id)
    try:
        # business logic
        result = 2 + 2
        logger.info("event=job_success job_id=%s result=%d", job_id, result)
    except Exception:
        logger.exception("event=job_failed job_id=%s", job_id)
        raise
```

```python
if __name__ == "__main__":
    configure_logging("INFO")
    run_job("job-001")
```

42_typed_settings_env.py

```python
# Technique: Typed Settings Loaded from Environment
# Use When:
# - Clear, validated configuration contracts
# - Fails fast when required settings are missing/invalid
# - Keeps environment details out of business logic

import os
from dataclasses import dataclass

@dataclass(frozen=True)
class Settings:
    app_env: str
    service_name: str
    timeout_s: float

def load_settings() -> Settings:
    app_env = os.getenv("APP_ENV", "dev")
    service_name = os.getenv("SERVICE_NAME", "pymaster")
    timeout_raw = os.getenv("TIMEOUT_S", "2.0")

    try:
        timeout_s = float(timeout_raw)
    except ValueError as exc:
        raise RuntimeError("TIMEOUT_S must be numeric") from exc

    if timeout_s <= 0:
        raise RuntimeError("TIMEOUT_S must be > 0")

    return Settings(app_env=app_env, service_name=service_name, timeout_s=timeout_s)

if __name__ == "__main__":
    os.environ.setdefault("APP_ENV", "dev")
    os.environ.setdefault("SERVICE_NAME", "billing")
    os.environ.setdefault("TIMEOUT_S", "1.5")
    print(load_settings())
```

43_request_scoped_contextvars.py

```python
# Technique: Request-Scoped Context with contextvars
# Use When:
# - Attaches correlation IDs to logs without passing params everywhere
# - Works correctly across async tasks
# - Supports tracing and debugging in distributed systems

import logging
from contextlib import contextmanager
from contextvars import ContextVar

request_id_var: ContextVar[str] = ContextVar("request_id", default="-")
logger = logging.getLogger("svc")

@contextmanager
def request_context(request_id: str):
    token = request_id_var.set(request_id)
    try:
        yield
    finally:
        request_id_var.reset(token)

def do_work() -> None:
    rid = request_id_var.get()
    logger.info("event=work request_id=%s", rid)

if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO, format="%(levelname)s %(message)s")

    do_work()
    with request_context("req-123"):
        do_work()
```

44_sqlite_transactions.py

```
# Technique: Transaction Boundaries (sqlite3)
# Use When:
# - Guarantees data consistency (all-or-nothing changes)
# - Makes failure handling predictable
# - Centralizes commit/rollback logic

import sqlite3
from contextlib import contextmanager

@contextmanager
def transaction(conn: sqlite3.Connection):
    try:
        yield
        conn.commit()
    except Exception:
        conn.rollback()
        raise

def init_schema(conn: sqlite3.Connection) -> None:
    conn.execute("CREATE TABLE IF NOT EXISTS accounts (id INTEGER PRIMARY KEY, balance INTEG

def transfer(conn: sqlite3.Connection, from_id: int, to_id: int, amount: int) -> None:
    if amount <= 0:
        raise ValueError("amount must be > 0")

    with transaction(conn):
        from_bal = conn.execute("SELECT balance FROM accounts WHERE id=?", (from_id,)).fetch
        to_bal = conn.execute("SELECT balance FROM accounts WHERE id=?", (to_id,)).fetchone(
        if from_bal is None or to_bal is None:
            raise ValueError("account not found")
        if from_bal[0] < amount:
            raise ValueError("insufficient funds")

        conn.execute("UPDATE accounts SET balance = balance - ? WHERE id=?", (amount, from_i
        conn.execute("UPDATE accounts SET balance = balance + ? WHERE id=?", (amount, to_id)

if __name__ == "__main__":
    conn = sqlite3.connect(":memory:")
    init_schema(conn)
    conn.execute("INSERT INTO accounts(id, balance) VALUES (1, 100)")
    conn.execute("INSERT INTO accounts(id, balance) VALUES (2, 0)")
    conn.commit()

    transfer(conn, 1, 2, 50)
```

```
print(conn.execute("SELECT id, balance FROM accounts ORDER BY id").fetchall())
```

45_idempotency_keys.py

```
# Technique: Idempotency Keys for Safe Retries
# Use When:
# - Prevents double-processing when clients retry requests
# - Makes APIs safe under timeouts and network failures
# - Improves correctness in payment/order flows

import sqlite3
from dataclasses import dataclass

@dataclass(frozen=True)
class Result:
    ok: bool
    message: str

def init_schema(conn: sqlite3.Connection) -> None:
    conn.execute(
        "CREATE TABLE IF NOT EXISTS idempotency (key TEXT PRIMARY KEY, ok INTEGER NOT NULL,
    )
    conn.commit()

def get_cached(conn: sqlite3.Connection, key: str) -> Result | None:
    row = conn.execute("SELECT ok, message FROM idempotency WHERE key=?", (key,)).fetchone(
    if row is None:
        return None
    return Result(ok=bool(row[0]), message=row[1])

def put_cached(conn: sqlite3.Connection, key: str, result: Result) -> None:
    conn.execute(
        "INSERT INTO idempotency(key, ok, message) VALUES (?, ?, ?)",
        (key, int(result.ok), result.message),
    )
    conn.commit()

def process_order(conn: sqlite3.Connection, idem_key: str, order_id: int) -> Result:
    cached = get_cached(conn, idem_key)
    if cached is not None:
        return cached

    # Simulated side effect.
    result = Result(ok=True, message=f"processed order {order_id}")
    put_cached(conn, idem_key, result)
    return result

if __name__ == "__main__":
```

```
conn = sqlite3.connect(":memory:")
init_schema(conn)

print(process_order(conn, "k1", 100))
print(process_order(conn, "k1", 100))
```

46_circuit_breaker.py

```python
# Technique: Circuit Breaker (Protect Dependencies)
# Use When:
# - Prevents cascading failures when a downstream service is unhealthy
# - Gives the dependency time to recover
# - Improves overall system stability

import time
from dataclasses import dataclass
from typing import Callable

class CircuitOpenError(RuntimeError):
    pass

@dataclass
class CircuitBreaker:
    failure_threshold: int
    cooldown_s: float

    _failures: int = 0
    _opened_at: float | None = None

    def call(self, func: Callable[[], str]) -> str:
        now = time.time()
        if self._opened_at is not None:
            if now - self._opened_at < self.cooldown_s:
                raise CircuitOpenError("circuit is open")
            # half-open: allow a probe call
            self._opened_at = None
            self._failures = 0

        try:
            out = func()
        except Exception:
            self._failures += 1
            if self._failures >= self.failure_threshold:
                self._opened_at = time.time()
            raise

        self._failures = 0
        return out

def flaky_factory() -> Callable[[], str]:
    state = {"n": 0}
    def f() -> str:
        state["n"] += 1
        if state["n"] <= 3:
            raise RuntimeError("downstream timeout")
        return "ok"

    return f

if __name__ == "__main__":
    cb = CircuitBreaker(failure_threshold=2, cooldown_s=1.0)
    f = flaky_factory()

    for i in range(1, 7):
        try:
            print(i, cb.call(f))
        except Exception as exc:
            print(i, type(exc).__name__, exc)
        time.sleep(0.3)
```

47_rate_limiter_token_bucket.py

```python
# Technique: Token Bucket Rate Limiter
# Use When:
# - Protects services from overload
# - Enforces fair usage across callers
# - Smooths burst traffic

import time
from dataclasses import dataclass

@dataclass
class TokenBucket:
    capacity: float
    refill_rate_per_s: float

    _tokens: float | None = None
    _last: float | None = None

    def allow(self, cost: float = 1.0) -> bool:
        now = time.time()
        if self._tokens is None:
            self._tokens = self.capacity
            self._last = now

        assert self._last is not None
        elapsed = now - self._last
        self._tokens = min(self.capacity, self._tokens + elapsed * self.refill_rate_per_s)
        self._last = now

        if cost <= self._tokens:
            self._tokens -= cost
            return True
        return False

if __name__ == "__main__":
    bucket = TokenBucket(capacity=3, refill_rate_per_s=1)
    for i in range(10):
        allowed = bucket.allow()
        print(i, "allowed" if allowed else "blocked")
        time.sleep(0.2)
```

48_feature_flags_rollout.py

```python
# Technique: Feature Flags + Percentage Rollout
# Use When:
# - Gradual rollout reduces risk
# - Enables fast rollback without redeploy
# - Supports A/B tests and staged deployments

import hashlib
from dataclasses import dataclass

@dataclass(frozen=True)
class FeatureFlag:
    name: str
    enabled: bool
    rollout_percent: int = 100

def is_enabled(flag: FeatureFlag, subject_id: str) -> bool:
    if not flag.enabled:
        return False
    if flag.rollout_percent >= 100:
        return True
    if flag.rollout_percent <= 0:
        return False

    # Stable hash -> 0..99
    digest = hashlib.sha256(f"{flag.name}:{subject_id}".encode("utf-8")).digest()
    bucket = digest[0] % 100
    return bucket < flag.rollout_percent

if __name__ == "__main__":
    flag = FeatureFlag(name="new_checkout", enabled=True, rollout_percent=30)
    for user in ["u1", "u2", "u3", "u4", "u5"]:
        print(user, is_enabled(flag, user))
```

49_simple_di_container.py

```python
# Technique: Simple Dependency Injection Container
# Use When:
# - Centralizes wiring of services
# - Makes components testable (swap implementations)
# - Helps scale applications as dependencies grow

from collections.abc import Callable

class Container:
    def __init__(self) -> None:
        self._providers: dict[str, Callable[[], object]] = {}

    def register(self, name: str, provider: Callable[[], object]) -> None:
        self._providers[name] = provider

    def resolve(self, name: str) -> object:
        if name not in self._providers:
            raise KeyError(f"unknown service: {name}")
        return self._providers[name]()

class Clock:
    def now(self) -> str:
        return "2026-02-26T00:00:00Z"

class Greeter:
    def __init__(self, clock: Clock) -> None:
        self._clock = clock

    def greet(self, who: str) -> str:
        return f"hello {who} at {self._clock.now()}"

if __name__ == "__main__":
    c = Container()
    c.register("clock", lambda: Clock())
    c.register("greeter", lambda: Greeter(c.resolve("clock")))

    greeter = c.resolve("greeter")
    print(greeter.greet("dev"))  # type: ignore[attr-defined]
```

50_health_checks.py

```
# Technique: Health Checks (Liveness vs Readiness)
# Use When:
# - Orchestrators (Kubernetes, ECS) rely on health endpoints
# - Liveness: process is running
# - Readiness: process can serve traffic (deps available)

import sqlite3
from dataclasses import dataclass

@dataclass(frozen=True)
class Health:
    ok: bool
    detail: str

def liveness() -> Health:
    return Health(ok=True, detail="alive")

def readiness(conn: sqlite3.Connection) -> Health:
    try:
        conn.execute("SELECT 1").fetchone()
    except Exception as exc:
        return Health(ok=False, detail=f"db_error={exc}")
    return Health(ok=True, detail="ready")

if __name__ == "__main__":
    conn = sqlite3.connect(":memory:")
    print("liveness:", liveness())
    print("readiness:", readiness(conn))
```

51_url_parsing.py

```
# Technique: URL Parsing with urllib.parse
# Use When:
# - Correct URL parsing avoids subtle bugs (encoding, query params, fragments)
# - Safer than manual string splitting

from urllib.parse import parse_qs, urlencode, urlsplit, urlunsplit

def add_query_param(url: str, key: str, value: str) -> str:
    parts = urlsplit(url)
    query = parse_qs(parts.query, keep_blank_values=True)
    query.setdefault(key, []).append(value)
    new_query = urlencode(query, doseq=True)
    return urlunsplit((parts.scheme, parts.netloc, parts.path, new_query, parts.fragment))

if __name__ == "__main__":
    u = "https://example.com/search?q=python#top"
    print(add_query_param(u, "page", "2"))
```

52_html_parsing_htmlparser.py

```
# Technique: HTML Parsing with html.parser (Standard Library)
# Use When:
# - Avoids brittle regex parsing for HTML
# - Good for simple scraping/extraction tasks

from html.parser import HTMLParser

class LinkExtractor(HTMLParser):
    def __init__(self) -> None:
        super().__init__()
        self.links: list[str] = []

    def handle_starttag(self, tag: str, attrs: list[tuple[str, str | None]]) -> None:
        if tag != "a":
            return
        href = dict(attrs).get("href")
        if href:
            self.links.append(href)

if __name__ == "__main__":
    html = "<p>See <a href='https://example.com'>Example</a> and <a href='/local'>Local</a><
    parser = LinkExtractor()
    parser.feed(html)
    print(parser.links)
```

53_html_entity_unescape.py

```python
# Technique: HTML Entity Handling with html.unescape
# Use When:
# - Web pages often contain entities (&amp;, &lt;, &quot;)
# - You typically want to decode entities before processing text

import html

def normalize_text(raw: str) -> str:
    return " ".join(html.unescape(raw).split())

if __name__ == "__main__":
    s = "Tom &amp; Jerry &lt;3  \n  "
    print(normalize_text(s))
```

54_json_parsing_validation.py

```python
# Technique: JSON Parsing + Lightweight Validation
# Use When:
# - Most web APIs are JSON
# - Parsing is easy; validating shape/types is where production bugs happen

import json
from dataclasses import dataclass
from typing import Any

@dataclass(frozen=True)
class UserDTO:
    id: int
    email: str

def parse_user(payload: str) -> UserDTO:
    data: Any = json.loads(payload)
    if not isinstance(data, dict):
        raise ValueError("expected object")
    if not isinstance(data.get("id"), int):
        raise ValueError("id must be int")
    if not isinstance(data.get("email"), str):
        raise ValueError("email must be str")
    return UserDTO(id=data["id"], email=data["email"])

if __name__ == "__main__":
    print(parse_user('{"id": 1, "email": "dev@example.com"}'))
```

55_xml_parsing_elementtree.py

```python
# Technique: XML Parsing with xml.etree.ElementTree
# Use When:
# - XML is still common in enterprise integrations (feeds, legacy APIs)
# - ElementTree provides a safe, structured parser

import xml.etree.ElementTree as ET

def extract_titles(xml_text: str) -> list[str]:
    root = ET.fromstring(xml_text)
    return [el.text or "" for el in root.findall("./item/title")]

if __name__ == "__main__":
    xml_text = """
    <feed>
      <item><title>First</title></item>
      <item><title>Second</title></item>
    </feed>
    """.strip()
    print(extract_titles(xml_text))
```

56_csv_parsing.py

```python
# Technique: CSV Parsing with csv.DictReader
# Use When:
# - Web exports and integrations often deliver CSV
# - DictReader handles headers and quoting correctly

import csv
import io

def parse_rows(csv_text: str) -> list[dict[str, str]]:
    f = io.StringIO(csv_text)
    reader = csv.DictReader(f)
    rows: list[dict[str, str]] = []
    for row in reader:
        if row.get("email") is None:
            raise ValueError("missing email column")
        rows.append({k: (v or "").strip() for k, v in row.items() if k is not None})
    return rows

if __name__ == "__main__":
    csv_text = "id,email\n1, dev@example.com \n2,admin@example.com\n"
    print(parse_rows(csv_text))
```

57_robots_txt_parsing.py

```python
# Technique: robots.txt Parsing with urllib.robotparser
# Use When:
# - Basic politeness and compliance for scraping
# - Quickly checks whether a URL is allowed for a user-agent

from urllib.robotparser import RobotFileParser

def can_fetch(robots_txt: str, user_agent: str, url: str) -> bool:
    rp = RobotFileParser()
    rp.parse(robots_txt.splitlines())
    return rp.can_fetch(user_agent, url)

if __name__ == "__main__":
    robots = """
    User-agent: *
    Disallow: /private
    """.strip()
    print(can_fetch(robots, "mybot", "https://example.com/"))
    print(can_fetch(robots, "mybot", "https://example.com/private"))
```

58_http_headers_parsing.py

```
# Technique: HTTP Header Parsing with email.message
# Use When:
# - Headers are structured but messy (case-insensitive, folding, duplicates)
# - Standard library provides robust parsing

from email.parser import Parser

def parse_headers(raw: str) -> dict[str, str]:
    msg = Parser().parsestr(raw)
    return {k: v for k, v in msg.items()}

if __name__ == "__main__":
    raw = """Host: example.com\nContent-Type: text/html; charset=utf-8\nX-Test: a\nX-Test: b
    print(parse_headers(raw))
```

59_http_responses_gzip.py

```python
# Technique: Handling gzip-compressed HTTP bodies
# Use When:
# - Many web servers compress responses
# - You need to decode correctly based on Content-Encoding

import gzip

def decode_gzip_body(body: bytes, charset: str = "utf-8") -> str:
    decompressed = gzip.decompress(body)
    return decompressed.decode(charset, errors="replace")

if __name__ == "__main__":
    original = "hello gzip"
    compressed = gzip.compress(original.encode("utf-8"))
    print(decode_gzip_body(compressed))
```

60_sitemap_parsing.py

```python
# Technique: Parsing XML Sitemaps (sitemap.xml)
# Use When:
# - Sitemaps are a common, crawl-friendly way to discover URLs
# - Namespaces are the main gotcha when parsing

import xml.etree.ElementTree as ET

def extract_sitemap_urls(xml_text: str) -> list[str]:
    root = ET.fromstring(xml_text)
    ns = {"sm": "http://www.sitemaps.org/schemas/sitemap/0.9"}
    return [el.text or "" for el in root.findall("./sm:url/sm:loc", ns)]

if __name__ == "__main__":
    xml_text = """
    <urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
      <url><loc>https://example.com/</loc></url>
      <url><loc>https://example.com/about</loc></url>
    </urlset>
    """.strip()
    print(extract_sitemap_urls(xml_text))
```

Pandas

01_dtypes_and_memory.py

```python
# Technique: Dtypes and Memory Discipline (Trading Data at Scale)
# Use When:
# - Market/position data gets large quickly; dtype choices drive latency and cost
# - Use nullable dtypes for clean missing handling without object columns
# - Use categoricals for repeated strings (tickers, venues, sectors)

import pandas as pd

def demo() -> None:
    df = pd.DataFrame(
        {
            "ticker": ["AAPL", "AAPL", "MSFT", "MSFT"],
            "venue": ["NASDAQ", "NASDAQ", "NASDAQ", "NASDAQ"],
            "qty": [100, 200, None, 50],
            "price": [189.1, 190.2, 412.0, None],
        }
    )

    print("before dtypes:\n", df.dtypes)
    print("before mem:", int(df.memory_usage(deep=True).sum()), "bytes")

    df["ticker"] = df["ticker"].astype("category")
    df["venue"] = df["venue"].astype("category")
    df["qty"] = df["qty"].astype("Int64")
    df["price"] = df["price"].astype("Float64")

    print("\nafter dtypes:\n", df.dtypes)
    print("after mem:", int(df.memory_usage(deep=True).sum()), "bytes")
    print("\nnormalized:\n", df)

if __name__ == "__main__":
    demo()
```

02_reading_large_csv.py

```python
# Technique: Reading Large CSVs (dtype, parse_dates, chunks)
# Use When:
# - You ingest vendor dumps (ticks, bars, ref data) that can be GBs
# - Correct `dtype` prevents object columns and speeds parsing
# - `chunksize` enables streaming and incremental processing

import io
import pandas as pd

def read_trades(csv_text: str) -> pd.DataFrame:
    return pd.read_csv(
        io.StringIO(csv_text),
        usecols=["ts", "ticker", "qty", "price"],
        dtype={"ticker": "string", "qty": "Int64", "price": "Float64"},
        parse_dates=["ts"],
    )

def read_trades_in_chunks(csv_text: str, chunk_rows: int = 2) -> pd.DataFrame:
    chunks = pd.read_csv(
        io.StringIO(csv_text),
        usecols=["ts", "ticker", "qty", "price"],
        dtype={"ticker": "string", "qty": "Int64", "price": "Float64"},
        parse_dates=["ts"],
        chunksize=chunk_rows,
    )
    out = pd.concat(chunks, ignore_index=True)
    return out

if __name__ == "__main__":
    csv_text = """ts,ticker,qty,price,ignored
2026-02-26T09:30:00Z,AAPL,100,189.1,x
2026-02-26T09:31:00Z,AAPL,200,190.2,x
2026-02-26T09:32:00Z,MSFT,,412.0,x
"""

    df1 = read_trades(csv_text)
    df2 = read_trades_in_chunks(csv_text, chunk_rows=2)

    print(df1)
    print(df1.dtypes)
    print("chunks equal:", df1.equals(df2))
```

03_datetime_timezone.py

```python
# Technique: Datetime + Timezones (Market Data Correctness)
# Use When:
# - Mixing naive/aware timestamps silently breaks joins and resampling
# - Normalize to UTC for storage; convert to local exchange tz for session logic

import pandas as pd

def demo() -> None:
    df = pd.DataFrame(
        {
            "ts": [
                "2026-02-26 09:30:10-05:00",
                "2026-02-26 09:30:40-05:00",
                "2026-02-26 09:31:05-05:00",
            ],
            "price": [100.0, 100.2, 100.1],
        }
    )

    df["ts"] = pd.to_datetime(df["ts"], utc=True)
    df["minute"] = df["ts"].dt.floor("min")
    df["ts_ny"] = df["ts"].dt.tz_convert("America/New_York")

    print(df)

if __name__ == "__main__":
    demo()
```

04_indexing_best_practices.py

```python
# Technique: Indexing Best Practices (DatetimeIndex, MultiIndex)
# Use When:
# - Time series operations (resample/rolling) are easiest with DatetimeIndex
# - Panel data (date x asset) is natural with MultiIndex

import pandas as pd

def demo() -> None:
    dates = pd.date_range("2026-02-24", periods=3, freq="D", tz="UTC")
    tickers = ["AAPL", "MSFT"]

    idx = pd.MultiIndex.from_product([dates, tickers], names=["ts", "ticker"])
    df = pd.DataFrame({"close": [100, 200, 101, 201, 102, 202]}, index=idx).sort_index()

    print("panel:\n", df)
    print("\nAAPL slice:\n", df.loc[(slice(None), "AAPL"), :])

if __name__ == "__main__":
    demo()
```

05_merge_join_basics.py

```python
# Technique: merge vs join (Reference Data Enrichment)
# Use When:
# - Enrich trades with security master fields (sector, currency, lot size)
# - Bad joins create silent row explosions or dropped observations

import pandas as pd

def demo() -> None:
    trades = pd.DataFrame(
        {
            "trade_id": [1, 2, 3],
            "ticker": ["AAPL", "MSFT", "AAPL"],
            "qty": [100, 50, 200],
        }
    )

    secmaster = pd.DataFrame(
        {
            "ticker": ["AAPL", "MSFT"],
            "sector": ["Tech", "Tech"],
            "currency": ["USD", "USD"],
        }
    )

    enriched = trades.merge(
        secmaster,
        on="ticker",
        how="left",
        validate="many_to_one",
        indicator=True,
    )

    print(enriched)

if __name__ == "__main__":
    demo()
```

06_merge_asof_market_alignment.py

```python
# Technique: merge_asof (Align Trades to Quotes/Bars)
# Use When:
# - You often need the last known quote before a trade (or a bar close)
# - Regular merge fails because timestamps rarely match exactly

import pandas as pd

def demo() -> None:
    quotes = pd.DataFrame(
        {
            "ts": pd.to_datetime(
                [
                    "2026-02-26T14:30:00Z",
                    "2026-02-26T14:30:30Z",
                    "2026-02-26T14:31:00Z",
                ],
                utc=True,
            ),
            "bid": [99.9, 100.0, 100.1],
            "ask": [100.1, 100.2, 100.3],
        }
    ).sort_values("ts")

    trades = pd.DataFrame(
        {
            "trade_id": [1, 2],
            "ts": pd.to_datetime(["2026-02-26T14:30:35Z", "2026-02-26T14:31:10Z"], utc=True)
            "price": [100.15, 100.25],
        }
    ).sort_values("ts")

    aligned = pd.merge_asof(
        trades,
        quotes,
        on="ts",
        direction="backward",
        tolerance=pd.Timedelta("45s"),
    )

    aligned["mid"] = (aligned["bid"] + aligned["ask"]) / 2
    print(aligned)

if __name__ == "__main__":
    demo()
```

07_concat_alignment_pitfalls.py

```python
# Technique: concat + Alignment Pitfalls
# Use When:
# - Combining signals/returns from different sources can misalign dates
# - Pandas aligns on index by default, which is powerful but can surprise

import pandas as pd

def demo() -> None:
    idx1 = pd.date_range("2026-02-24", periods=3, freq="D")
    idx2 = pd.date_range("2026-02-25", periods=3, freq="D")

    s1 = pd.Series([0.01, -0.02, 0.03], index=idx1, name="strat_a")
    s2 = pd.Series([0.00, 0.01, -0.01], index=idx2, name="strat_b")

    wide_outer = pd.concat([s1, s2], axis=1)  # union of dates
    wide_inner = pd.concat([s1, s2], axis=1, join="inner")  # intersection of dates

    print("outer:\n", wide_outer)
    print("\ninner:\n", wide_inner)

if __name__ == "__main__":
    demo()
```

08_groupby_agg_named.py

```python
# Technique: groupby + Named Aggregations
# Use When:
# - Summarize exposures by sector, region, book, PM
# - Named aggregations produce clean column names (no MultiIndex columns)

import pandas as pd

def demo() -> None:
    positions = pd.DataFrame(
        {
            "pm": ["A", "A", "B", "B"],
            "sector": ["Tech", "Energy", "Tech", "Tech"],
            "mv": [10_000, -2_000, 5_000, 7_000],
            "dv01": [120, -10, 40, 55],
        }
    )

    summary = (
        positions.groupby(["pm", "sector"], as_index=False)
        .agg(mv_sum=("mv", "sum"), dv01_sum=("dv01", "sum"), n=("mv", "size"))
        .sort_values(["pm", "sector"])
    )

    print(summary)

if __name__ == "__main__":
    demo()
```

09_transform_vs_apply.py

```python
# Technique: groupby transform vs apply (Performance + Correctness)
# Use When:
# - Cross-sectional normalization (z-scores) should keep original shape
# - `transform` returns same length as input; `apply` can change shape

import pandas as pd

def cross_section_zscore(df: pd.DataFrame) -> pd.Series:
    # z-score of returns per date across tickers
    def z(x: pd.Series) -> pd.Series:
        return (x - x.mean()) / x.std(ddof=0)

    return df.groupby("date")["ret"].transform(z)

if __name__ == "__main__":
    data = pd.DataFrame(
        {
            "date": ["2026-02-24", "2026-02-24", "2026-02-25", "2026-02-25"],
            "ticker": ["AAPL", "MSFT", "AAPL", "MSFT"],
            "ret": [0.01, -0.02, 0.03, 0.01],
        }
    )
    data["date"] = pd.to_datetime(data["date"])
    data["z"] = cross_section_zscore(data)
    print(data)
```

10_pipe_and_clean_pipelines.py

```python
# Technique: pipe for Clean Data Pipelines
# Use When:
# - Research code becomes production-like when pipelines are explicit
# - `pipe` reduces intermediate variables and encourages reusable steps

import pandas as pd

def to_utc(df: pd.DataFrame) -> pd.DataFrame:
    out = df.copy()
    out["ts"] = pd.to_datetime(out["ts"], utc=True)
    return out

def add_mid(df: pd.DataFrame) -> pd.DataFrame:
    out = df.copy()
    out["mid"] = (out["bid"] + out["ask"]) / 2
    return out

def filter_spread(df: pd.DataFrame, max_spread: float) -> pd.DataFrame:
    out = df.copy()
    out["spread"] = out["ask"] - out["bid"]
    return out.loc[out["spread"] <= max_spread, ["ts", "mid", "spread"]]

if __name__ == "__main__":
    quotes = pd.DataFrame(
        {
            "ts": ["2026-02-26T14:30:00Z", "2026-02-26T14:31:00Z"],
            "bid": [99.9, 100.1],
            "ask": [100.2, 100.3],
        }
    )

    cleaned = quotes.pipe(to_utc).pipe(add_mid).pipe(filter_spread, max_spread=0.25)
    print(cleaned)
```

11_rolling_and_ewm.py

```python
# Technique: Rolling Windows and EWM (Volatility, Trends, Risk)
# Use When:
# - Rolling stats drive signals (moving averages) and risk (rolling vol)
# - EWM (exponentially weighted) reacts faster, common in risk models

import pandas as pd

def demo() -> None:
    idx = pd.date_range("2026-02-01", periods=10, freq="D")
    prices = pd.Series([100, 101, 102, 101, 103, 104, 103, 105, 106, 107], index=idx, name='

    rets = prices.pct_change()
    out = pd.DataFrame({"px": prices, "ret": rets})

    out["ma_3"] = out["px"].rolling(3, min_periods=3).mean()
    out["vol_5"] = out["ret"].rolling(5, min_periods=5).std(ddof=0)

    out["ewm_ma"] = out["px"].ewm(span=3, adjust=False).mean()
    out["ewm_vol"] = out["ret"].ewm(span=5, adjust=False).std(bias=True)

    print(out)

if __name__ == "__main__":
    demo()
```

12_resample_ohlc_and_vwap.py

```python
# Technique: Resampling to Bars (OHLC) and VWAP
# Use When:
# - You often convert irregular trades to fixed interval bars
# - OHLC bars and VWAP are standard downstream inputs

import pandas as pd

def demo() -> None:
    ts = pd.to_datetime(
        [
            "2026-02-26T14:30:10Z",
            "2026-02-26T14:30:20Z",
            "2026-02-26T14:30:50Z",
            "2026-02-26T14:31:10Z",
            "2026-02-26T14:31:40Z",
        ],
        utc=True,
    )
    trades = pd.DataFrame({"ts": ts, "price": [100.0, 100.1, 99.9, 100.2, 100.3], "qty": [10
    trades = trades.set_index("ts").sort_index()

    ohlc = trades["price"].resample("1min").ohlc()
    vol = trades["qty"].resample("1min").sum().rename("volume")

    vwap_num = (trades["price"] * trades["qty"]).resample("1min").sum()
    vwap_den = trades["qty"].resample("1min").sum()
    vwap = (vwap_num / vwap_den).rename("vwap")

    bars = pd.concat([ohlc, vol, vwap], axis=1)
    print(bars)

if __name__ == "__main__":
    demo()
```

13_pivot_melt_wide_long.py

```python
# Technique: Wide vs Long (pivot, pivot_table, melt)
# Use When:
# - Research often needs wide matrices (date x ticker returns)
# - Storage/ETL often prefers long format (tidy records)

import pandas as pd

def demo() -> None:
    df = pd.DataFrame(
        {
            "date": ["2026-02-24", "2026-02-24", "2026-02-25", "2026-02-25"],
            "ticker": ["AAPL", "MSFT", "AAPL", "MSFT"],
            "ret": [0.01, -0.02, 0.03, 0.01],
        }
    )
    df["date"] = pd.to_datetime(df["date"])

    wide = df.pivot(index="date", columns="ticker", values="ret")
    long = wide.reset_index().melt(id_vars=["date"], var_name="ticker", value_name="ret")

    print("wide:\n", wide)
    print("\nlong:\n", long.sort_values(["date", "ticker"]))

if __name__ == "__main__":
    demo()
```

14_multiindex_stack_unstack.py

```python
# Technique: MultiIndex, stack/unstack (Panel Data)
# Use When:
# - A common representation is (date, ticker) -> fields
# - stack/unstack reshapes between panel (MultiIndex) and matrix forms

import pandas as pd

def demo() -> None:
    df = pd.DataFrame(
        {
            "date": ["2026-02-24", "2026-02-24", "2026-02-25", "2026-02-25"],
            "ticker": ["AAPL", "MSFT", "AAPL", "MSFT"],
            "close": [100.0, 200.0, 101.0, 201.0],
        }
    )
    df["date"] = pd.to_datetime(df["date"])

    panel = df.set_index(["date", "ticker"]).sort_index()
    wide = panel["close"].unstack("ticker")
    back = wide.stack("ticker").rename("close").to_frame()

    print("panel:\n", panel)
    print("\nwide:\n", wide)
    print("\nback:\n", back)

if __name__ == "__main__":
    demo()
```

15_missing_data_imputation.py

```python
# Technique: Missing Data (dropna, fillna, ffill) for Time Series
# Use When:
# - Corporate actions, trading halts, and vendor gaps create missing values
# - You need explicit rules per dataset (prices vs fundamentals vs signals)

import pandas as pd

def demo() -> None:
    idx = pd.date_range("2026-02-24", periods=5, freq="D")
    px = pd.Series([100.0, None, None, 103.0, None], index=idx, name="px")

    df = px.to_frame()
    df["px_ffill"] = df["px"].ffill(limit=2)
    df["ret"] = df["px"].pct_change()
    df["ret_safe"] = df["px_ffill"].pct_change()

    print(df)

if __name__ == "__main__":
    demo()
```

16_categoricals_for_ref_data.py

```python
# Technique: Categoricals for Reference Data (Speed + Memory)
# Use When:
# - Repeated strings (ticker, sector, venue, book) are common and expensive
# - Categoricals compress memory and speed groupby operations

import pandas as pd

def demo() -> None:
    df = pd.DataFrame(
        {
            "ticker": ["AAPL", "MSFT", "AAPL", "TSLA", "MSFT"],
            "sector": ["Tech", "Tech", "Tech", "Auto", "Tech"],
            "pnl": [10.0, -3.0, 5.0, 7.0, 2.0],
        }
    )

    print("before mem:", int(df.memory_usage(deep=True).sum()))
    df["ticker"] = df["ticker"].astype("category")
    df["sector"] = pd.Categorical(df["sector"], categories=["Tech", "Auto"], ordered=True)
    print("after mem:", int(df.memory_usage(deep=True).sum()))

    print(df.groupby("sector", observed=True).agg(pnl_sum=("pnl", "sum")))

if __name__ == "__main__":
    demo()
```

17_query_and_eval.py

```
# Technique: query/eval for Fast, Readable Filtering
# Use When:
# - Analysts write lots of filters (liquidity, borrow cost, universe rules)
# - `query` reads closer to a DSL and can be faster than chained masks

import pandas as pd

def demo() -> None:
    df = pd.DataFrame({"ticker": ["AAPL", "MSFT", "TSLA"], "price": [190.2, 412.0, 210.0], '
    df = df.eval("notional = price * qty")
    filt = df.query("price > 200 and qty >= 50")
    print(df)
    print("\nfiltered:\n", filt)

if __name__ == "__main__":
    demo()
```

18_groupby_rolling_panel.py

```python
# Technique: groupby + rolling (Per-Asset Rolling Metrics)
# Use When:
# - Compute rolling vol per ticker across dates
# - Avoid Python loops by using groupby+rolling

import pandas as pd

def demo() -> None:
    df = pd.DataFrame(
        {
            "date": ["2026-02-24", "2026-02-25", "2026-02-26", "2026-02-24", "2026-02-25", '
            "ticker": ["AAPL", "AAPL", "AAPL", "MSFT", "MSFT", "MSFT"],
            "ret": [0.01, -0.02, 0.03, 0.00, 0.01, -0.01],
        }
    )
    df["date"] = pd.to_datetime(df["date"])
    df = df.sort_values(["ticker", "date"]).reset_index(drop=True)

    df["vol_2"] = (
        df.groupby("ticker")["ret"].rolling(2, min_periods=2).std(ddof=0).reset_index(level=
    )

    print(df)

if __name__ == "__main__":
    demo()
```

19_covariance_and_correlation.py

```python
# Technique: Covariance/Correlation Matrices (Risk + Diversification)
# Use When:
# - Portfolio risk models start with cov/corr of returns
# - Proper alignment and missing handling are critical

import pandas as pd

def demo() -> None:
    rets = pd.DataFrame(
        {
            "date": pd.to_datetime(["2026-02-24", "2026-02-25", "2026-02-26"]),
            "AAPL": [0.01, -0.02, 0.03],
            "MSFT": [0.00, 0.01, -0.01],
            "TSLA": [0.02, -0.03, 0.01],
        }
    ).set_index("date")

    print("corr:\n", rets.corr())
    print("\ncov:\n", rets.cov())

if __name__ == "__main__":
    demo()
```

20_vectorized_pnl_backtest.py

```python
# Technique: Vectorized PnL (Signals -> Positions -> Returns)
# Use When:
# - Fast research iteration requires vectorized backtests
# - A common pattern: signal -> position (shift to avoid lookahead) -> PnL

import pandas as pd

def demo() -> None:
    df = pd.DataFrame(
        {
            "date": pd.date_range("2026-02-20", periods=6, freq="D"),
            "ret": [0.01, -0.02, 0.015, 0.00, -0.005, 0.02],
            "signal": [1, 1, -1, -1, 1, 1],
        }
    ).set_index("date")

    df["pos"] = df["signal"].shift(1).fillna(0)  # trade tomorrow based on today's signal
    df["strat_ret"] = df["pos"] * df["ret"]
    df["equity"] = (1 + df["strat_ret"]).cumprod()

    print(df)

if __name__ == "__main__":
    demo()
```

21_calendar_alignment.py

```
# Technique: Calendar Alignment (Trading Days vs Calendar Days)
# Use When:
# - Equity markets have holidays; crypto trades 24/7; rates have different calendars
# - Misaligned calendars break backtests and risk

import pandas as pd

def demo() -> None:
    # Missing a "holiday" on 2026-02-25 (simulated).
    px = pd.Series(
        [100.0, 101.0, 103.0],
        index=pd.to_datetime(["2026-02-24", "2026-02-26", "2026-02-27"]),
        name="px",
    )

    cal = pd.date_range("2026-02-24", "2026-02-27", freq="B")  # business days
    aligned = px.reindex(cal)
    aligned_ffill = aligned.ffill()

    out = pd.DataFrame({"px": aligned, "px_ffill": aligned_ffill})
    out["ret"] = out["px"].pct_change()
    out["ret_safe"] = out["px_ffill"].pct_change()

    print(out)

if __name__ == "__main__":
    demo()
```

22_outliers_winsorize_clip.py

```python
# Technique: Outliers (clip / winsor-like handling)
# Use When:
# - Bad prints or corporate action errors can create huge outliers
# - Outliers can dominate z-scores, correlations, and risk estimates

import pandas as pd

def clip_by_quantile(s: pd.Series, q: float = 0.01) -> pd.Series:
    lo = s.quantile(q)
    hi = s.quantile(1 - q)
    return s.clip(lower=lo, upper=hi)

if __name__ == "__main__":
    s = pd.Series([0.01, 0.02, -0.03, 0.5, -0.4, 0.015], name="ret")
    print("raw:\n", s)
    print("clipped:\n", clip_by_quantile(s, q=0.1))
```

23_cross_sectional_ranking.py

```python
# Technique: Cross-Sectional Ranking (Signals)
# Use When:
# - Many strategies rank assets daily (momentum, value, quality)
# - Ranks are robust to outliers compared to raw values

import pandas as pd

def demo() -> None:
    df = pd.DataFrame(
        {
            "date": ["2026-02-24"] * 4 + ["2026-02-25"] * 4,
            "ticker": ["A", "B", "C", "D"] * 2,
            "signal": [1.0, 2.0, 2.0, -1.0, 0.5, 0.2, 1.5, 1.5],
        }
    )
    df["date"] = pd.to_datetime(df["date"])

    df["rank_pct"] = df.groupby("date")["signal"].rank(pct=True, method="average")
    print(df.sort_values(["date", "rank_pct"]))

if __name__ == "__main__":
    demo()
```

24_long_short_construction.py

```python
# Technique: Long/Short Portfolio Construction (Toy)
# Use When:
# - Most equity L/S flows: rank -> pick top/bottom -> equal weight or score weight
# - Important: shift positions to avoid lookahead bias

import pandas as pd

def make_weights(df: pd.DataFrame, long_n: int, short_n: int) -> pd.Series:
    df = df.copy()
    df["rank"] = df.groupby("date")["signal"].rank(method="first", ascending=False)

    def bucket(g: pd.DataFrame) -> pd.Series:
        longs = g["rank"] <= long_n
        shorts = g["rank"] > (len(g) - short_n)

        w = pd.Series(0.0, index=g.index)
        if longs.any():
            w.loc[longs] = 1.0 / longs.sum()
        if shorts.any():
            w.loc[shorts] = -1.0 / shorts.sum()
        return w

    return df.groupby("date", group_keys=False).apply(bucket)

if __name__ == "__main__":
    df = pd.DataFrame(
        {
            "date": ["2026-02-24"] * 4,
            "ticker": ["A", "B", "C", "D"],
            "signal": [1.0, 2.0, 0.0, -1.0],
        }
    )
    df["date"] = pd.to_datetime(df["date"])

    df["w"] = make_weights(df, long_n=1, short_n=1)
    print(df)
```

25_event_driven_backtest_basic.py

```
# Technique: Event-Driven-ish Backtest with Vectorization
# Use When:
# - Many real systems are event-driven, but research often approximates with

import pandas as pd

def demo() -> None:
    df = pd.DataFrame(
        {
            "date": pd.date_range("2026-02-20", periods=6, freq="D"),
            "signal": [0, 1, 1, -1, -1, 0],
            "ret": [0.01, -0.02, 0.015, 0.00, -0.005, 0.02],
        }
    ).set_index("date")

    df["pos"] = df["signal"].shift(1).fillna(0)
    df["turnover"] = df["pos"].diff().abs().fillna(0)

    df["strat_ret"] = df["pos"] * df["ret"]
    df["equity"] = (1 + df["strat_ret"]).cumprod()

    print(df)

if __name__ == "__main__":
    demo()
```

26_timeseries_joins_many_assets.py

```python
# Technique: Time Series Joins for Many Assets (Panel Merge)
# Use When:
# - Combine returns + signals + fundamentals keyed by (date, ticker)
# - MultiIndex joins prevent accidental cartesian products

import pandas as pd

def demo() -> None:
    idx = pd.MultiIndex.from_product(
        [pd.to_datetime(["2026-02-24", "2026-02-25"]), ["AAPL", "MSFT"]],
        names=["date", "ticker"],
    )

    rets = pd.DataFrame({"ret": [0.01, -0.02, 0.03, 0.01]}, index=idx)
    sig = pd.DataFrame({"signal": [1, 0, -1, 1]}, index=idx)

    panel = rets.join(sig, how="inner")
    print(panel)

if __name__ == "__main__":
    demo()
```

27_windowed_features_lagging.py

```python
# Technique: Lagging + Feature Engineering (Avoid Lookahead)
# Use When:
# - Feature engineering often uses rolling stats
# - You must lag features to ensure they are known at decision time

import pandas as pd

def demo() -> None:
    idx = pd.date_range("2026-02-01", periods=6, freq="D")
    px = pd.Series([100, 101, 102, 101, 103, 104], index=idx, name="px")

    df = px.to_frame()
    df["ret"] = df["px"].pct_change()
    df["ma3"] = df["px"].rolling(3, min_periods=3).mean()
    df["ma3_lag1"] = df["ma3"].shift(1)

    # signal: price above lagged MA
    df["signal"] = (df["px"] > df["ma3_lag1"]).astype("Int64")

    print(df)

if __name__ == "__main__":
    demo()
```

28_performance_copy_vs_view.py

```python
# Technique: Performance Hygiene (copies, chained assignment)
# Use When:
# - Research code becomes slow or wrong due to chained assignment
# - In production, you want explicit `.copy()` boundaries

import pandas as pd

def demo() -> None:
    df = pd.DataFrame({"a": [1, 2, 3, 4], "b": [10, 20, 30, 40]})

    # Good: explicit loc assignment
    mask = df["a"] % 2 == 0
    df.loc[mask, "b"] = df.loc[mask, "b"] * 2

    # Good: explicit copy boundary for derived frame
    even = df.loc[mask, ["a", "b"]].copy()
    even["c"] = even["a"] + even["b"]

    print(df)
    print(even)

if __name__ == "__main__":
    demo()
```

29_style_of_validation_checks.py

```python
# Technique: Data Validation Checks (Cheap Guardrails)
# Use When:
# - Silent data issues (duplicates, missing keys) destroy PnL credibility
# - Lightweight assertions catch issues early in notebooks/ETL

import pandas as pd

def validate_trades(df: pd.DataFrame) -> None:
    required = {"trade_id", "ts", "ticker", "qty", "price"}
    missing = required - set(df.columns)
    if missing:
        raise ValueError(f"missing columns: {sorted(missing)}")

    if df["trade_id"].duplicated().any():
        raise ValueError("trade_id must be unique")

    if df[["ts", "ticker", "qty", "price"]].isna().any().any():
        raise ValueError("ts/ticker/qty/price must be non-null")

    if (df["qty"] <= 0).any():
        raise ValueError("qty must be > 0")

    if (df["price"] <= 0).any():
        raise ValueError("price must be > 0")

if __name__ == "__main__":
    df = pd.DataFrame(
        {
            "trade_id": [1, 2],
            "ts": pd.to_datetime(["2026-02-26T14:30:00Z", "2026-02-26T14:30:01Z"], utc=True
            "ticker": ["AAPL", "MSFT"],
            "qty": [100, 50],
            "price": [190.2, 412.0],
        }
    )
    validate_trades(df)
    print("ok")
```

30_rank_normalize_by_group.py

```python
# Technique: Normalize Within Groups (Sector-Neutral Signals)
# Use When:
# - Sector-neutral ranking reduces unintended factor exposure
# - You frequently normalize within sector, industry, or region

import pandas as pd

def zscore(s: pd.Series) -> pd.Series:
    return (s - s.mean()) / s.std(ddof=0)

def demo() -> None:
    df = pd.DataFrame(
        {
            "date": ["2026-02-24"] * 6,
            "ticker": ["A", "B", "C", "D", "E", "F"],
            "sector": ["Tech", "Tech", "Tech", "Energy", "Energy", "Energy"],
            "signal": [1.0, 2.0, -1.0, 0.2, 0.1, 0.5],
        }
    )
    df["date"] = pd.to_datetime(df["date"])

    df["z_sector"] = df.groupby(["date", "sector"])["signal"].transform(zscore)
    df["rank_sector"] = df.groupby(["date", "sector"])["signal"].rank(pct=True)

    print(df.sort_values(["sector", "ticker"]))

if __name__ == "__main__":
    demo()
```

31_vectorization_vs_apply.py

```python
# Technique: Vectorization vs apply (Speed for Research -> Prod)
# Use When:
# - The difference between vectorized ops and row-wise apply can be minutes vs seconds
# - Prefer NumPy/pandas vector ops; reserve apply for unavoidable Python logic

import pandas as pd

def demo() -> None:
    df = pd.DataFrame({"price": [100.0, 50.0, 200.0], "qty": [10, 0, 5]})

    # Vectorized notional
    df["notional"] = df["price"] * df["qty"]

    # Vectorized safe notional (qty must be > 0)
    df["notional_safe"] = df["notional"].where(df["qty"] > 0)

    print(df)

if __name__ == "__main__":
    demo()
```

32_numpy_interop.py

```python
# Technique: NumPy Interop (Fast Math, Stable Outputs)
# Use When:
# - Many quant transforms are expressed naturally in NumPy
# - Use NumPy for heavy numeric kernels, pandas for alignment/indexing

import numpy as np
import pandas as pd

def demo() -> None:
    s = pd.Series([100.0, 101.0, 99.0, 103.0], name="px")

    arr = s.to_numpy()
    logret = np.log(arr[1:] / arr[:-1])

    out = pd.Series([np.nan] + logret.tolist(), index=s.index, name="logret")
    print(pd.concat([s, out], axis=1))

if __name__ == "__main__":
    demo()
```

33_time_series_split_train_test.py

```
# Technique: Time Series Train/Test Split (No Leakage)
# Use When:
# - Random splits leak future information
# - Always split by time (and often embargo around events)

import pandas as pd

def time_split(df: pd.DataFrame, split_date: str) -> tuple[pd.DataFrame, pd.DataFrame]:
    split_ts = pd.Timestamp(split_date)
    train = df.loc[df.index < split_ts].copy()
    test = df.loc[df.index >= split_ts].copy()
    return train, test

if __name__ == "__main__":
    df = pd.DataFrame({"ret": [0.01, -0.02, 0.03, 0.01]}, index=pd.date_range("2026-02-01",
    train, test = time_split(df, "2026-02-03")
    print("train:\n", train)
    print("\ntest:\n", test)
```

34_factor_exposure_regression_toy.py

```python
# Technique: Factor Exposure Regression (Toy, No Statsmodels)
# Use When:
# - Exposure estimation (beta) and factor neutralization show up everywhere
# - Production uses robust stats tooling; here we show a minimal matrix approach

import numpy as np
import pandas as pd

def ols_beta(y: pd.Series, x: pd.Series) -> float:
    Y = y.to_numpy()
    X = np.column_stack([np.ones(len(x)), x.to_numpy()])  # intercept + factor
    coef, *_ = np.linalg.lstsq(X, Y, rcond=None)
    return float(coef[1])

if __name__ == "__main__":
    df = pd.DataFrame(
        {
            "mkt": [0.01, -0.02, 0.015, 0.0, 0.005],
            "asset": [0.012, -0.025, 0.020, -0.001, 0.006],
        },
        index=pd.date_range("2026-02-01", periods=5, freq="D"),
    )
    print("beta:", ols_beta(df["asset"], df["mkt"]))
```

35_neutralize_signal_by_sector.py

```python
# Technique: Sector Neutralization (Demean within Group)
# Use When:
# - Removes sector tilts from signals
# - Often used as a cheap, robust neutralization step

import pandas as pd

def demo() -> None:
    df = pd.DataFrame(
        {
            "date": ["2026-02-24"] * 6,
            "ticker": list("ABCDEF"),
            "sector": ["Tech", "Tech", "Tech", "Energy", "Energy", "Energy"],
            "signal": [1.0, 2.0, -1.0, 0.2, 0.1, 0.5],
        }
    )
    df["date"] = pd.to_datetime(df["date"])

    grp_mean = df.groupby(["date", "sector"])["signal"].transform("mean")
    df["signal_neutral"] = df["signal"] - grp_mean
    print(df)

if __name__ == "__main__":
    demo()
```

36_quantile_binning.py

```python
# Technique: Quantile Binning (Deciles/Quintiles)
# Use When:
# - Standard in factor research: bucket assets into deciles by signal
# - Used for long/short portfolio formation and performance attribution

import pandas as pd

def demo() -> None:
    df = pd.DataFrame({"ticker": list("ABCDEFGH"), "signal": [1, 2, 3, 4, 5, 6, 7, 8]})
    df["bucket"] = pd.qcut(df["signal"], q=4, labels=False, duplicates="drop")
    print(df.sort_values("signal"))

if __name__ == "__main__":
    demo()
```

37_turnover_and_costs_toy.py

```python
# Technique: Turnover and Simple Transaction Cost Model (Toy)
# Use When:
# - Gross returns without costs are misleading
# - Turnover is a first-order proxy for costs and capacity

import pandas as pd

def demo() -> None:
    df = pd.DataFrame(
        {
            "date": pd.date_range("2026-02-01", periods=5, freq="D"),
            "w": [0.0, 1.0, 1.0, -1.0, 0.0],
            "ret": [0.0, 0.01, -0.02, 0.015, 0.0],
        }
    ).set_index("date")

    df["turnover"] = df["w"].diff().abs().fillna(0)
    k = 0.001  # 10 bps per 1.0 turnover (toy)
    df["cost"] = k * df["turnover"]

    df["gross"] = df["w"].shift(1).fillna(0) * df["ret"]
    df["net"] = df["gross"] - df["cost"]
    df["equity_net"] = (1 + df["net"]).cumprod()

    print(df)

if __name__ == "__main__":
    demo()
```

38_drawdown_and_sharpe.py

```python
# Technique: Performance Metrics (Sharpe, Max Drawdown)
# Use When:
# - Sharpe and drawdown are baseline reporting metrics
# - Always define your conventions (annualization factor, risk-free rate)

import numpy as np
import pandas as pd

def sharpe(returns: pd.Series, ann_factor: float = 252.0) -> float:
    r = returns.dropna()
    if r.std(ddof=0) == 0:
        return 0.0
    return float(np.sqrt(ann_factor) * r.mean() / r.std(ddof=0))

def max_drawdown(equity: pd.Series) -> float:
    peak = equity.cummax()
    dd = equity / peak - 1.0
    return float(dd.min())

if __name__ == "__main__":
    ret = pd.Series([0.01, -0.02, 0.015, 0.0, -0.005, 0.02])
    equity = (1 + ret).cumprod()
    print("sharpe:", sharpe(ret))
    print("max_dd:", max_drawdown(equity))
```

39_vectorized_position_limits.py

```
# Technique: Vectorized Position Limits (Clipping Weights)
# Use When:
# - Constraints (per-name, gross/net exposure) are central in real portfolios
# - First pass can clip per-name weights before more advanced optimization

import pandas as pd

def demo() -> None:
    w = pd.Series({"AAPL": 0.12, "MSFT": -0.08, "TSLA": 0.20, "XOM": -0.15}, name="w")
    w_limited = w.clip(lower=-0.10, upper=0.10)

    gross = w_limited.abs().sum()
    net = w_limited.sum()

    print("raw:\n", w)
    print("\nlimited:\n", w_limited)
    print("\ngross:", float(gross), "net:", float(net))

if __name__ == "__main__":
    demo()
```

40__stability__reproducibility.py

```python
# Technique: Stability + Reproducibility (Determinism)
# Use When:
# - Research must be reproducible to be trusted
# - Deterministic outputs reduce debugging time and deployment risk

import numpy as np
import pandas as pd

def demo() -> None:
    rng = np.random.default_rng(42)

    df = pd.DataFrame(
        {
            "date": ["2026-02-24"] * 5,
            "ticker": ["B", "A", "E", "C", "D"],
            "signal": rng.normal(size=5),
        }
    )
    df["date"] = pd.to_datetime(df["date"])

    # Stable order to avoid surprises when breaking ties.
    df = df.sort_values(["date", "ticker"], kind="mergesort").reset_index(drop=True)

    df["rank"] = df.groupby("date")["signal"].rank(method="first", ascending=False)
    print(df)

if __name__ == "__main__":
    demo()
```

41_wide_returns_matrix_and_alignment.py

```python
# Technique: Wide Returns Matrix + Alignment Discipline
# Use When:
# - Many risk/signal ops assume a (date x ticker) matrix
# - Alignment errors are the #1 silent bug in matrix-based work

import pandas as pd

def demo() -> None:
    long = pd.DataFrame(
        {
            "date": ["2026-02-24", "2026-02-24", "2026-02-25", "2026-02-25"],
            "ticker": ["AAPL", "MSFT", "AAPL", "MSFT"],
            "ret": [0.01, -0.02, 0.03, 0.01],
        }
    )
    long["date"] = pd.to_datetime(long["date"])

    wide = long.pivot(index="date", columns="ticker", values="ret").sort_index().sort_index
    print(wide)

    # Example alignment-safe operation: cross-sectional mean per date
    cs_mean = wide.mean(axis=1)
    print("\ncs_mean:\n", cs_mean)

if __name__ == "__main__":
    demo()
```

42_cross_sectional_neutralization_matrix.py

```
# Technique: Cross-Sectional Neutralization (Demean Matrix)
# Use When:
# - A cheap neutralization step is to remove the per-date mean
# - Helps reduce unwanted net exposure when signals are biased

import pandas as pd

def demean_by_date(wide: pd.DataFrame) -> pd.DataFrame:
    row_mean = wide.mean(axis=1)
    return wide.sub(row_mean, axis=0)

if __name__ == "__main__":
    wide = pd.DataFrame(
        {
            "A": [0.1, 0.2, -0.1],
            "B": [0.0, 0.1, 0.0],
            "C": [-0.1, -0.3, 0.2],
        },
        index=pd.date_range("2026-02-24", periods=3, freq="D"),
    )
    print("raw:\n", wide)
    print("\ndemeaned:\n", demean_by_date(wide))
```

43_vectorized_ic_information_coefficient.py

```python
# Technique: Information Coefficient (IC) per Date (Toy)
# Use When:
# - IC measures signal predictive power (corr(signal_t, return_{t+1}))
# - Common for factor evaluation and monitoring

import pandas as pd

def ic_by_date(df: pd.DataFrame) -> pd.Series:
    # df columns: date, ticker, signal, ret
    df = df.copy()
    df["ret_fwd"] = df.groupby("ticker")["ret"].shift(-1)

    def corr(g: pd.DataFrame) -> float:
        return float(g["signal"].corr(g["ret_fwd"]))

    return df.groupby("date").apply(corr)

if __name__ == "__main__":
    df = pd.DataFrame(
        {
            "date": pd.to_datetime(["2026-02-24"] * 3 + ["2026-02-25"] * 3 + ["2026-02-26"]
            "ticker": ["A", "B", "C"] * 3,
            "signal": [1.0, 0.5, -0.5, 0.2, 0.1, -0.1, -0.3, 0.0, 0.4],
            "ret": [0.01, -0.02, 0.00, 0.03, 0.01, -0.01, -0.01, 0.02, 0.01],
        }
    )
    print(ic_by_date(df))
```

44_weighted_average_and_exposure.py

```python
# Technique: Weighted Averages + Exposure Attribution
# Use When:
# - Portfolio-level metrics are weighted by positions
# - Exposure decomposition by sector/book is core reporting

import pandas as pd

def weighted_avg(values: pd.Series, weights: pd.Series) -> float:
    w = weights.astype(float)
    v = values.astype(float)
    if w.abs().sum() == 0:
        return 0.0
    return float((v * w).sum() / w.sum())

if __name__ == "__main__":
    df = pd.DataFrame(
        {
            "ticker": ["AAPL", "MSFT", "XOM"],
            "sector": ["Tech", "Tech", "Energy"],
            "w": [0.05, -0.02, 0.03],
            "beta": [1.1, 1.0, 0.8],
        }
    )

    df["beta_contrib"] = df["w"] * df["beta"]
    by_sector = df.groupby("sector", as_index=False).agg(
        net_w=("w", "sum"),
        beta_exposure=("beta_contrib", "sum"),
    )

    print(by_sector)
```

45_multioutput_agg_and_named_columns.py

```
# Technique: Multi-Output Aggregations with Named Columns
# Use When:
# - Reporting often needs multiple metrics per group (PnL, vol, hit rate)
# - Named aggs keep outputs tidy and avoid MultiIndex columns

import pandas as pd

def hit_rate(s: pd.Series) -> float:
    s = s.dropna()
    if len(s) == 0:
        return 0.0
    return float((s > 0).mean())

if __name__ == "__main__":
    pnl = pd.DataFrame(
        {
            "pm": ["A", "A", "A", "B", "B"],
            "day": pd.to_datetime(["2026-02-24", "2026-02-25", "2026-02-26", "2026-02-24", "
            "pnl": [10.0, -3.0, 5.0, 2.0, -1.0],
        }
    )

    rpt = pnl.groupby("pm", as_index=False).agg(
        pnl_sum=("pnl", "sum"),
        pnl_mean=("pnl", "mean"),
        hit=("pnl", hit_rate),
        n=("pnl", "size"),
    )

    print(rpt)
```

46_duplicate_handling_primary_keys.py

```
# Technique: Duplicate Handling (Primary Keys) in Market Data
# Use When:
# - Vendor files sometimes duplicate records
# - You must decide a deterministic rule (last, first, max volume)

import pandas as pd

def demo() -> None:
    df = pd.DataFrame(
        {
            "ts": pd.to_datetime(["2026-02-26T14:30:00Z", "2026-02-26T14:30:00Z", "2026-02-2
            "ticker": ["AAPL", "AAPL", "AAPL"],
            "price": [100.0, 100.1, 100.2],
            "seq": [1, 2, 3],
        }
    )

    # Deterministic: keep the record with the highest seq.
    df = df.sort_values(["ts", "ticker", "seq"]).drop_duplicates(["ts", "ticker"], keep="las
    print(df)

if __name__ == "__main__":
    demo()
```

47_merge_validation_and_audit.py

```python
# Technique: Merge Validation + Audit (Avoid Row Explosions)
# Use When:
# - Joining positions to ref data can silently explode if keys aren't unique
# - Use `validate` and `_merge` indicators to catch errors early

import pandas as pd

def demo() -> None:
    pos = pd.DataFrame({"ticker": ["AAPL", "MSFT"], "qty": [100, 50]})
    ref = pd.DataFrame({"ticker": ["AAPL", "MSFT"], "sector": ["Tech", "Tech"]})

    merged = pos.merge(ref, on="ticker", how="left", validate="one_to_one", indicator=True)
    print(merged)
    print("merge counts:\n", merged["_merge"].value_counts())

if __name__ == "__main__":
    demo()
```

48_fast_string_ops.py

```python
# Technique: Fast String Ops (Cleaning Symbol IDs)
# Use When:
# - Identifiers arrive messy (spaces, case, vendor suffixes)
# - Vectorized string methods keep it fast and consistent

import pandas as pd

def demo() -> None:
    s = pd.Series([" AAPL ", "msft", "BRK.B ", None], dtype="string")
    cleaned = s.str.strip().str.upper().str.replace(".", "-", regex=False)
    print(pd.DataFrame({"raw": s, "cleaned": cleaned}))

if __name__ == "__main__":
    demo()
```

49_benchmark_relative_returns.py

```python
# Technique: Benchmark-Relative Returns (Alpha)
# Use When:
# - Many reports are relative to a benchmark (SPX, sector index)
# - Requires careful date alignment

import pandas as pd

def demo() -> None:
    idx = pd.date_range("2026-02-01", periods=5, freq="D")
    strat = pd.Series([0.01, -0.02, 0.015, 0.0, 0.005], index=idx, name="strat")
    bench = pd.Series([0.008, -0.01, 0.010, 0.001, 0.002], index=idx, name="bench")

    aligned = pd.concat([strat, bench], axis=1, join="inner")
    aligned["excess"] = aligned["strat"] - aligned["bench"]
    print(aligned)

if __name__ == "__main__":
    demo()
```

50_signal_monitoring_checks.py

```python
# Technique: Signal Monitoring Checks (Production Guardrails)
# Use When:
# - Once a signal is live, you monitor stability: missing rates, drift, extremes
# - Cheap checks catch upstream data breaks before trading losses

import pandas as pd

def signal_checks(df: pd.DataFrame, col: str = "signal") -> pd.DataFrame:
    s = df[col]
    return pd.DataFrame(
        {
            "n": [int(s.size)],
            "null_rate": [float(s.isna().mean())],
            "min": [float(s.min())],
            "p01": [float(s.quantile(0.01))],
            "p50": [float(s.quantile(0.50))],
            "p99": [float(s.quantile(0.99))],
            "max": [float(s.max())],
        }
    )

if __name__ == "__main__":
    df = pd.DataFrame({"signal": [0.1, 0.2, None, -0.3, 10.0, -10.0]})
    print(signal_checks(df))
```

NumPy

01_01_arrays_and_dtypes.py

```
# Technique: Arrays + dtypes (why dtype discipline matters)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    a = np.array([1, 2, 3])
    b = np.array([1, 2, 3], dtype=np.int32)
    c = np.array([1, 2, 3], dtype=np.float64)

    print('a', a.dtype, a.nbytes)
    print('b', b.dtype, b.nbytes)
    print('c', c.dtype, c.nbytes)

if __name__ == '__main__':
    demo()
```

02_02_views_vs_copies.py

```
# Technique: Views vs copies (slicing can alias memory)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.arange(10)
    view = x[2:6]       # view
    copy = x[2:6].copy()

    view[:] = 99
    print('x after view write:', x)

    x[:] = np.arange(10)
    copy[:] = 77
    print('x after copy write:', x)

if __name__ == '__main__':
    demo()
```

03__03__broadcasting.py

```python
# Technique: Broadcasting (vectorize across dimensions)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    m = np.arange(6).reshape(2, 3)
    v = np.array([10, 20, 30])
    print(m)
    print(m + v)  # v broadcasts across rows

if __name__ == '__main__':
    demo()
```

04_04_ufuncs_and_where.py

```python
# Technique: Ufuncs + where (branchless selection)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([-2.0, -0.5, 0.0, 0.5, 2.0])
    y = np.where(x >= 0, x * x, -x)  # example piecewise
    print(y)

if __name__ == '__main__':
    demo()
```

05_05_reductions_axis_keepdims.py

```python
# Technique: Reductions with axis/keepdims
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.arange(12).reshape(3, 4)
    print('sum all:', x.sum())
    print('sum axis=0:', x.sum(axis=0))
    print('sum axis=1:', x.sum(axis=1))
    print('mean axis=1 keepdims:', x.mean(axis=1, keepdims=True))

if __name__ == '__main__':
    demo()
```

06_06_boolean_masking.py

```python
# Technique: Boolean masking (filter + assign)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([1, 2, 3, 4, 5, 6])
    mask = x % 2 == 0
    print('evens:', x[mask])

    y = x.copy()
    y[mask] *= 10
    print('assigned:', y)

if __name__ == '__main__':
    demo()
```

07_07_advanced_indexing_pitfalls.py

```python
# Technique: Advanced indexing (creates copies)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.arange(10)
    idx = np.array([2, 5, 7])
    sub = x[idx]
    sub[:] = 99
    print('x unchanged:', x)

    # Use direct assignment with idx to mutate.
    x[idx] = 88
    print('x mutated:', x)

if __name__ == '__main__':
    demo()
```

08_08_sort_argsort_argpartition.py

```python
# Technique: Sorting: sort/argsort/argpartition (top-k)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([5, 1, 9, 2, 7, 3])
    print('sorted:', np.sort(x))
    print('argsort:', np.argsort(x))

    k = 3
    idx = np.argpartition(x, -k)[-k:]
    topk = x[idx]
    print('topk (unordered):', topk)
    print('topk sorted:', np.sort(topk))

if __name__ == '__main__':
    demo()
```

09_09_unique_and_counts.py

```python
# Technique: unique + counts (frequency tables)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array(['AAPL', 'MSFT', 'AAPL', 'TSLA', 'MSFT', 'AAPL'])
    vals, counts = np.unique(x, return_counts=True)
    order = np.argsort(-counts)
    for v, c in zip(vals[order], counts[order]):
        print(v, int(c))

if __name__ == '__main__':
    demo()
```

10_10_nan_handling.py

```python
# Technique: NaN-aware stats: isnan, nanmean, nanstd
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([1.0, np.nan, 2.0, 3.0, np.nan])
    print('mean:', np.mean(x))
    print('nanmean:', np.nanmean(x))
    print('nanstd:', np.nanstd(x))

if __name__ == '__main__':
    demo()
```

11_11_random_generator_api.py

```python
# Technique: Random numbers: Generator API (reproducible)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    rng = np.random.default_rng(42)
    x = rng.normal(size=5)
    y = rng.integers(0, 10, size=5)
    print(x)
    print(y)

if __name__ == '__main__':
    demo()
```

12_12_linear_algebra_basics.py

```python
# Technique: Linear algebra: solve vs inverse
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    A = np.array([[3.0, 1.0], [1.0, 2.0]])
    b = np.array([9.0, 8.0])

    x = np.linalg.solve(A, b)
    print('solve:', x)

    inv = np.linalg.inv(A)
    x2 = inv @ b
    print('inv@b:', x2)

if __name__ == '__main__':
    demo()
```

13_13_cholesky_psd.py

```
"""
Technique: Cholesky psd
Use When:
- You are working in numerical computing with arrays
- You need cholesky psd as a reliable building block
"""

"""Lecture 13: Cholesky (SPD matrices) and checks

Focus:
- Intermediate/advanced NumPy techniques with a quant/finance bias.
- Each file is runnable and uses only NumPy + stdlib.

"""

import numpy as np

def demo() -> None:
    A = np.array([[4.0, 2.0], [2.0, 3.0]])  # SPD
    L = np.linalg.cholesky(A)
    recon = L @ L.T
    print('L:
', L)
    print('recon close:', np.allclose(A, recon))

if __name__ == '__main__':
    demo()
```

14_14_svd_and_pca_toy.py

```python
# Technique: SVD / PCA toy (demean then SVD)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    X = np.array([[1.0, 2.0], [2.0, 1.0], [3.0, 0.0], [4.0, -1.0]])
    Xc = X - X.mean(axis=0, keepdims=True)
    U, S, Vt = np.linalg.svd(Xc, full_matrices=False)
    print('singular values:', S)
    pc1 = Vt[0]
    print('pc1:', pc1)

if __name__ == '__main__':
    demo()
```

15_15_cov_corr_from_returns.py

```python
"""
Technique: Cov corr from returns
Use When:
- You are working in numerical computing with arrays
- You need cov corr from returns as a reliable building block
"""

"""Lecture 15: Covariance/correlation from returns matrix

Focus:
- Intermediate/advanced NumPy techniques with a quant/finance bias.
- Each file is runnable and uses only NumPy + stdlib.

"""

import numpy as np

def demo() -> None:
    # rows = time, cols = assets
    R = np.array(
        [
            [0.01, 0.00, 0.02],
            [-0.02, 0.01, -0.03],
            [0.03, -0.01, 0.01],
        ]
    )

    cov = np.cov(R, rowvar=False, ddof=0)
    std = np.sqrt(np.diag(cov))
    corr = cov / np.outer(std, std)

    print('cov:
', cov)
    print('corr:
', corr)

if __name__ == '__main__':
    demo()
```

16_16_einsum_for_speed.py

```python
# Technique: einsum (explicit tensor contractions)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    A = np.arange(6).reshape(2, 3)
    B = np.arange(12).reshape(3, 4)

    m1 = A @ B
    m2 = np.einsum('ij,jk->ik', A, B)
    print(np.allclose(m1, m2))

if __name__ == '__main__':
    demo()
```

17_17_strides_and_rolling_window.py

```
# Technique: Stride tricks: rolling window view (advanced)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def rolling_view(x: np.ndarray, window: int) -> np.ndarray:
    if window <= 0 or window > x.size:
        raise ValueError('bad window')
    shape = (x.size - window + 1, window)
    strides = (x.strides[0], x.strides[0])
    return np.lib.stride_tricks.as_strided(x, shape=shape, strides=strides)

def demo() -> None:
    x = np.arange(10, dtype=np.float64)
    w = rolling_view(x, 4)
    ma = w.mean(axis=1)
    print('rolling mean:', ma)

if __name__ == '__main__':
    demo()
```

18_18_memmap_large_arrays.py

```python
# Technique: memmap (out-of-core arrays)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

import os
import numpy as np

def demo() -> None:
    path = '/tmp/pymaster_memmap.dat'
    if os.path.exists(path):
        os.remove(path)

    m = np.memmap(path, dtype='float64', mode='w+', shape=(1000,))
    m[:] = np.linspace(0.0, 1.0, num=1000)
    m.flush()

    m2 = np.memmap(path, dtype='float64', mode='r', shape=(1000,))
    print(float(m2[:5].sum()))

if __name__ == '__main__':
    demo()
```

19_19_structured_arrays.py

```python
# Technique: Structured arrays (typed records)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    dt = np.dtype([('ticker', 'U8'), ('qty', 'i4'), ('price', 'f8')])
    rec = np.array([('AAPL', 100, 190.2), ('MSFT', 50, 412.0)], dtype=dt)
    notionals = rec['qty'] * rec['price']
    print(notionals)

if __name__ == '__main__':
    demo()
```

20_20_frombuffer_and_bytes.py

```python
# Technique: frombuffer (zero-copy decode of binary)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([1, 2, 3, 4], dtype=np.int32)
    b = x.tobytes()
    y = np.frombuffer(b, dtype=np.int32)
    print('y shares bytes:', y)

if __name__ == '__main__':
    demo()
```

21_21_vectorized_string_like_ops.py

```
# Technique: Vectorized operations over encoded IDs (use integers)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    # Replace slow string ops by mapping symbols -> integer codes.
    symbols = np.array(['AAPL', 'MSFT', 'AAPL', 'TSLA'])
    uniq, codes = np.unique(symbols, return_inverse=True)
    print('uniq:', uniq)
    print('codes:', codes)

if __name__ == '__main__':
    demo()
```

22__22__bincount__histograms.py

```python
# Technique: bincount / histograms (fast counting)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([0, 1, 1, 2, 2, 2, 4])
    counts = np.bincount(x, minlength=6)
    print(counts)

if __name__ == '__main__':
    demo()
```

23_23_searchsorted_time_alignment.py

```python
# Technique: searchsorted (align events to grid)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    grid = np.array([10, 20, 30, 40, 50])
    events = np.array([9, 10, 11, 35, 60])
    idx = np.searchsorted(grid, events, side='right') - 1  # last grid <= event
    idx = np.clip(idx, 0, len(grid) - 1)
    print('aligned:', grid[idx])

if __name__ == '__main__':
    demo()
```

24_24_pad_and_clip.py

```python
# Technique: clip and pad patterns
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([-5, -1, 0, 1, 10])
    print(np.clip(x, -2, 3))
    y = np.array([1, 2, 3])
    print(np.pad(y, (2, 1), mode='constant', constant_values=0))

if __name__ == '__main__':
    demo()
```

25_25_take_along_axis.py

```python
# Technique: take_along_axis (gather with indices)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([[10, 11, 12], [20, 21, 22]])
    idx = np.array([[2, 0, 1], [1, 1, 0]])
    y = np.take_along_axis(x, idx, axis=1)
    print(y)

if __name__ == '__main__':
    demo()
```

26_26_putmask_and_place.py

```python
# Technique: putmask / place (in-place conditional updates)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.arange(6, dtype=np.float64)
    np.putmask(x, x % 2 == 0, -1)
    print(x)

if __name__ == '__main__':
    demo()
```

27_27_diff_and_cumsum.py

```python
# Technique: diff/cumsum (PnL-style transforms)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    px = np.array([100.0, 101.0, 99.0, 103.0])
    ret = np.diff(px) / px[:-1]
    eq = np.cumprod(np.concatenate([[1.0], 1.0 + ret]))
    print('ret:', ret)
    print('equity:', eq)

if __name__ == '__main__':
    demo()
```

28__28__percentile__quantile.py

```python
# Technique: percentile/quantile (risk percentiles)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    pnl = np.array([1.0, -2.0, 0.5, 3.0, -1.0, -4.0])
    print('p01:', np.percentile(pnl, 1))
    print('p50:', np.percentile(pnl, 50))
    print('p99:', np.percentile(pnl, 99))

if __name__ == '__main__':
    demo()
```

29_29_softmax_stable.py

```python
# Technique: Stable softmax (numerical stability)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def softmax(x: np.ndarray) -> np.ndarray:
    x = x - np.max(x)
    e = np.exp(x)
    return e / e.sum()

def demo() -> None:
    x = np.array([1000.0, 1001.0, 999.0])
    print(softmax(x))

if __name__ == '__main__':
    demo()
```

30_30_logsumexp_stable.py

```
# Technique: Stable logsumexp
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def logsumexp(x: np.ndarray) -> float:
    m = float(np.max(x))
    return m + float(np.log(np.exp(x - m).sum()))

def demo() -> None:
    x = np.array([1000.0, 1001.0, 999.0])
    print(logsumexp(x))

if __name__ == '__main__':
    demo()
```

31_31_float_precision.py

```python
# Technique: Float precision: float32 vs float64
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x64 = np.array([1e8, 1, -1e8], dtype=np.float64)
    x32 = np.array([1e8, 1, -1e8], dtype=np.float32)
    print('sum float64:', float(x64.sum()))
    print('sum float32:', float(x32.sum()))

if __name__ == '__main__':
    demo()
```

32_32_kahan_like_sum.py

```python
# Technique: Mitigating cancellation with sorted sum
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([1e8] + [1.0] * 10000 + [-1e8], dtype=np.float64)
    naive = x.sum()
    better = np.sort(x).sum()
    print('naive:', float(naive))
    print('sorted:', float(better))

if __name__ == '__main__':
    demo()
```

33_33_block_matrix_ops.py

```python
# Technique: Block operations with reshape
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.arange(24).reshape(2, 3, 4)
    # Treat last two dims as a matrix per batch.
    s = x.sum(axis=(1, 2))
    print(s)

if __name__ == '__main__':
    demo()
```

34_34_concatenate_stack.py

```python
# Technique: concatenate vs stack
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    a = np.array([1, 2, 3])
    b = np.array([4, 5, 6])
    print('concat:', np.concatenate([a, b]))
    print('stack:', np.stack([a, b], axis=0))

if __name__ == '__main__':
    demo()
```

35_35_repeat_tile.py

```python
# Technique: repeat vs tile (be careful with memory)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([1, 2, 3])
    print('repeat:', np.repeat(x, 2))
    print('tile:', np.tile(x, 2))

if __name__ == '__main__':
    demo()
```

36_36_meshgrid.py

```python
# Technique: meshgrid (parameter grids)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    a = np.array([0.1, 0.2, 0.3])
    b = np.array([1, 2])
    A, B = np.meshgrid(a, b, indexing='xy')
    print(A)
    print(B)

if __name__ == '__main__':
    demo()
```

37_37_histogram_bin_edges.py

```
# Technique: histogram and bin edges
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    rng = np.random.default_rng(0)
    x = rng.normal(size=1000)
    counts, edges = np.histogram(x, bins=10)
    print(counts)
    print(edges)

if __name__ == '__main__':
    demo()
```

38__38__vectorize__is__not__vectorized.py

```python
# Technique: np.vectorize is convenience, not speed
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def f(x: float) -> float:
    return x * x + 1

def demo() -> None:
    x = np.arange(5, dtype=np.float64)
    vf = np.vectorize(f)
    print(vf(x))
    print('Prefer ufunc-style math when possible')

if __name__ == '__main__':
    demo()
```

39_39_outer_products.py

```
# Technique: outer products and factor models
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    a = np.array([1.0, 2.0, 3.0])
    b = np.array([10.0, 20.0])
    print(np.outer(a, b))

if __name__ == '__main__':
    demo()
```

40_40_kronecker_product.py

```python
# Technique: Kronecker product (block structure)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    A = np.array([[1, 2], [3, 4]])
    B = np.array([[0, 5], [6, 7]])
    print(np.kron(A, B))

if __name__ == '__main__':
    demo()
```

41__41__linalg__eig.py

```python
"""
Technique: Linalg eig
Use When:
- You are working in numerical computing with arrays
- You need linalg eig as a reliable building block
"""

"""Lecture 41: Eigenvalues (risk PCA intuition)

Focus:
- Intermediate/advanced NumPy techniques with a quant/finance bias.
- Each file is runnable and uses only NumPy + stdlib.

"""

import numpy as np

def demo() -> None:
    A = np.array([[2.0, 1.0], [1.0, 2.0]])
    w, v = np.linalg.eig(A)
    print('eigvals:', w)
    print('eigvecs:
', v)

if __name__ == '__main__':
    demo()
```

42_42_linalg_qr.py

```python
"""
Technique: Linalg qr
Use When:
- You are working in numerical computing with arrays
- You need linalg qr as a reliable building block
"""

"""Lecture 42: QR decomposition (numerical methods)

Focus:
- Intermediate/advanced NumPy techniques with a quant/finance bias.
- Each file is runnable and uses only NumPy + stdlib.

"""

import numpy as np

def demo() -> None:
    A = np.array([[1.0, 1.0], [1.0, -1.0], [1.0, 0.0]])
    Q, R = np.linalg.qr(A)
    print('Q:
', Q)
    print('R:
', R)
    print('recon close:', np.allclose(A, Q @ R))

if __name__ == '__main__':
    demo()
```

43_43_batched_matmul.py

```python
# Technique: Batched matmul with @
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    A = np.arange(12).reshape(2, 2, 3)  # batch=2
    B = np.arange(18).reshape(2, 3, 3)
    C = A @ B
    print(C.shape)

if __name__ == '__main__':
    demo()
```

44_44_datetime64_timedelta64.py

```
# Technique: datetime64/timedelta64 basics
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    t = np.array(['2026-02-26', '2026-02-27'], dtype='datetime64[D]')
    dt = np.diff(t).astype('timedelta64[D]')
    print(t)
    print(dt)

if __name__ == '__main__':
    demo()
```

45_45_masked_arrays.py

```python
# Technique: Masked arrays (explicit missingness)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([1.0, 2.0, -999.0, 3.0])
    mx = np.ma.masked_equal(x, -999.0)
    print('mean:', float(mx.mean()))

if __name__ == '__main__':
    demo()
```

46_46_piecewise_polynomials.py

```python
# Technique: polyfit (toy regression)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([0.0, 1.0, 2.0, 3.0])
    y = np.array([1.0, 2.1, 3.9, 6.2])
    coef = np.polyfit(x, y, deg=1)
    print('coef:', coef)
    pred = np.polyval(coef, x)
    print('pred:', pred)

if __name__ == '__main__':
    demo()
```

47_47_fft_basics.py

```
# Technique: FFT basics (toy signal)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    n = 64
    t = np.arange(n)
    x = np.sin(2 * np.pi * t / 8) + 0.5 * np.sin(2 * np.pi * t / 16)
    X = np.fft.rfft(x)
    mag = np.abs(X)
    print('peak bins:', np.argsort(-mag)[:5])

if __name__ == '__main__':
    demo()
```

48_48_save_load_npy_npz.py

```
# Technique: Save/load .npy/.npz
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

import os
import numpy as np

def demo() -> None:
    x = np.arange(5)
    y = np.linspace(0, 1, 5)
    path = '/tmp/pymaster_arrays.npz'
    if os.path.exists(path):
        os.remove(path)
    np.savez(path, x=x, y=y)

    data = np.load(path)
    print(data['x'])
    print(data['y'])

if __name__ == '__main__':
    demo()
```

49_49_errstate_and_floating.py

```python
# Technique: errstate (control warnings for invalid ops)
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([1.0, 0.0, -1.0])
    with np.errstate(divide='ignore', invalid='ignore'):
        y = 1.0 / x
    print(y)

if __name__ == '__main__':
    demo()
```

50_50_testing_numerical_close.py

```python
# Technique: Testing numerics with allclose
# Use When:
# - Intermediate/advanced NumPy techniques with a quant/finance bias
# - Each file is runnable and uses only NumPy + stdlib

import numpy as np

def demo() -> None:
    x = np.array([0.1, 0.2, 0.3])
    y = np.array([0.1, 0.2000000001, 0.3])
    print('equal:', np.array_equal(x, y))
    print('allclose:', np.allclose(x, y, rtol=1e-8, atol=1e-12))

if __name__ == '__main__':
    demo()
```

SciPy

01_optimize_minimize_basics.py

```python
# Technique: scipy.optimize.minimize (basics)
# Use When:
# - Unconstrained optimization with explicit objective
# - Common in calibration and parameter estimation

import numpy as np
from scipy import optimize

def f(x: np.ndarray) -> float:
    # Simple convex bowl.
    return float((x[0] - 2) ** 2 + (x[1] + 1) ** 2)

if __name__ == '__main__':
    res = optimize.minimize(f, x0=np.array([0.0, 0.0]))
    print(res.x, res.fun, res.success)
```

02_optimize_constraints.py

```python
# Technique: Constrained Optimization (bounds + constraints)
# Use When:
# - Bounds and linear/nonlinear constraints
# - Typical for portfolio constraints and calibration

import numpy as np
from scipy import optimize

def f(x: np.ndarray) -> float:
    return float((x[0] - 1) ** 2 + (x[1] - 2) ** 2)

if __name__ == '__main__':
    bounds = optimize.Bounds([0, 0], [2, 3])
    cons = ({'type': 'eq', 'fun': lambda x: x[0] + x[1] - 3},)
    res = optimize.minimize(f, x0=np.array([0.5, 2.5]), bounds=bounds, constraints=cons)
    print(res.x, res.success)
```

03_optimize_root_finding.py

```python
# Technique: Root Finding (scipy.optimize.root)
# Use When:
# - Solve systems f(x)=0
# - Appears in implied volatility and equilibrium conditions

import numpy as np
from scipy import optimize

def g(x: np.ndarray) -> np.ndarray:
    # Solve: x0^2 + x1 = 4, x0 + x1^2 = 4
    return np.array([x[0] ** 2 + x[1] - 4, x[0] + x[1] ** 2 - 4])

if __name__ == '__main__':
    res = optimize.root(g, x0=np.array([1.0, 1.0]))
    print(res.x, res.success)
```

04_optimize_least_squares.py

```python
# Technique: Nonlinear Least Squares (least_squares)
# Use When:
# - Fit model parameters by minimizing residuals
# - Used for curve fitting and calibration

import numpy as np
from scipy import optimize

def residuals(p: np.ndarray, x: np.ndarray, y: np.ndarray) -> np.ndarray:
    a, b = p
    return a * x + b - y

if __name__ == '__main__':
    x = np.array([0.0, 1.0, 2.0, 3.0])
    y = np.array([1.0, 2.1, 3.9, 6.2])
    res = optimize.least_squares(residuals, x0=np.array([1.0, 0.0]), args=(x, y))
    print(res.x)
```

05_interpolate_interp1d.py

```
# Technique: 1D Interpolation (interp1d)
# Use When:
# - Interpolate curves (yield curves, vol surfaces slices)
# - Choose kind and bounds behavior

import numpy as np
from scipy.interpolate import interp1d

if __name__ == '__main__':
    x = np.array([0.0, 1.0, 2.0, 3.0])
    y = np.array([0.0, 1.0, 0.0, 1.0])
    f = interp1d(x, y, kind='linear', fill_value='extrapolate')
    xs = np.linspace(0, 3, 7)
    print(f(xs))
```

06_interpolate_splines.py

```python
# Technique: Splines (UnivariateSpline)
# Use When:
# - Smooth noisy observations with a smoothing parameter
# - Useful for curve smoothing in noisy data

import numpy as np
from scipy.interpolate import UnivariateSpline

if __name__ == '__main__':
    x = np.linspace(0, 10, 20)
    rng = np.random.default_rng(0)
    y = np.sin(x) + 0.1 * rng.normal(size=x.size)
    sp = UnivariateSpline(x, y, s=0.5)
    xs = np.linspace(0, 10, 5)
    print(sp(xs))
```

07_integrate_quad.py

```
# Technique: Numerical Integration (quad)
# Use When:
# - Integrate functions with error estimates
# - Used in pricing and probability calculations

import numpy as np
from scipy import integrate

def f(x: float) -> float:
    return float(np.exp(-x * x))

if __name__ == '__main__':
    val, err = integrate.quad(f, 0.0, 1.0)
    print(val, err)
```

08_integrate_solve_ivp.py

```
# Technique: ODE Solving (solve_ivp)
# Use When:
# - Solve initial value problems
# - Shows up in term structure models and dynamics

import numpy as np
from scipy.integrate import solve_ivp

def rhs(t: float, y: np.ndarray) -> np.ndarray:
    # dy/dt = -y
    return -y

if __name__ == '__main__':
    sol = solve_ivp(rhs, t_span=(0.0, 5.0), y0=np.array([1.0]), t_eval=np.linspace(0, 5, 6))
    print(sol.y[0])
```

09_stats_distributions.py

```python
# Technique: Distributions (scipy.stats)
# Use When:
# - PDF/CDF/PPF, random variates
# - Core for risk metrics and simulations

from scipy import stats

if __name__ == '__main__':
    dist = stats.norm(loc=0.0, scale=1.0)
    print('cdf(0):', dist.cdf(0.0))
    print('ppf(0.95):', dist.ppf(0.95))
```

10_stats_ttest_and_nonparametric.py

```
# Technique: Hypothesis Tests (t-test, Mann-Whitney)
# Use When:
# - Basic statistical tests for research sanity checks
# - Beware multiple testing and dependence

import numpy as np
from scipy import stats

if __name__ == '__main__':
    x = np.array([1.0, 2.0, 1.5, 2.1])
    y = np.array([0.9, 1.8, 1.2, 2.0])
    print('ttest:', stats.ttest_ind(x, y, equal_var=False))
    print('mw:', stats.mannwhitneyu(x, y, alternative='two-sided'))
```

11_signal_filtering_butter.py

```python
# Technique: Signal Filtering (Butterworth)
# Use When:
# - Low/high-pass filtering with scipy.signal

import numpy as np
from scipy import signal

if __name__ == '__main__':
    fs = 100.0
    t = np.arange(0, 1, 1/fs)
    x = np.sin(2*np.pi*5*t) + 0.5*np.sin(2*np.pi*30*t)
    b, a = signal.butter(4, 10, btype='low', fs=fs)
    y = signal.filtfilt(b, a, x)
    print(float(y[:5].mean()))
```

12_signal_spectrogram.py

```python
# Technique: Spectrogram (time-frequency)
# Use When:
# - Inspect frequency content over time

import numpy as np
from scipy import signal

if __name__ == '__main__':
    fs = 100.0
    t = np.arange(0, 2, 1/fs)
    x = np.sin(2*np.pi*(5 + 10*t)*t)
    f, tt, Sxx = signal.spectrogram(x, fs=fs)
    print(Sxx.shape)
```

13_fft_rfft.py

```python
# Technique: FFT (scipy.fft)
# Use When:
# - Real FFT and frequency bins

import numpy as np
from scipy.fft import rfft, rfftfreq

if __name__ == '__main__':
    fs = 64.0
    t = np.arange(0, 1, 1/fs)
    x = np.sin(2*np.pi*8*t)
    X = np.abs(rfft(x))
    f = rfftfreq(t.size, d=1/fs)
    print(f[int(np.argmax(X))])
```

14_sparse_csr_basics.py

```python
# Technique: Sparse Matrices (CSR)
# Use When:
# - Build and multiply sparse matrices

import numpy as np
from scipy import sparse

if __name__ == '__main__':
    A = sparse.csr_matrix([[1, 0, 0], [0, 2, 0], [0, 0, 3]])
    x = np.array([1.0, 1.0, 1.0])
    print(A @ x)
```

15_sparse_spsolve.py

```
# Technique: Sparse Linear Solve (spsolve)
# Use When:
# - Solve Ax=b when A is sparse

import numpy as np
from scipy import sparse
from scipy.sparse.linalg import spsolve

if __name__ == '__main__':
    A = sparse.csr_matrix([[4.0, 1.0], [1.0, 3.0]])
    b = np.array([1.0, 2.0])
    x = spsolve(A, b)
    print(x)
```

16_linalg_solve_triangular.py

```python
# Technique: Triangular Solve (solve_triangular)
# Use When:
# - Efficient solves after Cholesky/QR

import numpy as np
from scipy.linalg import solve_triangular

if __name__ == '__main__':
    L = np.array([[2.0, 0.0], [1.0, 1.0]])
    b = np.array([2.0, 2.0])
    y = solve_triangular(L, b, lower=True)
    print(y)
```

17_spatial_kdtree.py

```python
# Technique: KDTree Nearest Neighbors
# Use When:
# - Fast nearest neighbor queries

import numpy as np
from scipy.spatial import KDTree

if __name__ == '__main__':
    pts = np.array([[0.0, 0.0], [1.0, 1.0], [2.0, 2.0]])
    tree = KDTree(pts)
    d, i = tree.query([1.2, 1.2])
    print(d, i)
```

18_cluster_kmeans.py

```python
# Technique: KMeans Clustering
# Use When:
# - Basic clustering via scipy.cluster.vq

import numpy as np
from scipy.cluster.vq import kmeans2

if __name__ == '__main__':
    rng = np.random.default_rng(0)
    x = np.r_[rng.normal(0, 0.2, size=(50, 2)), rng.normal(2, 0.2, size=(50, 2))]
    centroids, labels = kmeans2(x, 2, minit='points')
    print(centroids.shape, labels[:5])
```

19_ndimage_gaussian_filter.py

```python
# Technique: ndimage Gaussian Filter
# Use When:
# - Smooth 2D arrays/images

import numpy as np
from scipy.ndimage import gaussian_filter

if __name__ == '__main__':
    x = np.zeros((5, 5), dtype=float)
    x[2, 2] = 1.0
    y = gaussian_filter(x, sigma=1.0)
    print(float(y.sum()))
```

20_special_erf.py

```python
# Technique: Special Functions (erf)
# Use When:
# - Common in normal CDF relationships

import numpy as np
from scipy import special

if __name__ == '__main__':
    x = np.array([-1.0, 0.0, 1.0])
    print(special.erf(x))
```

21_stats_kde.py

```python
# Technique: Kernel Density Estimation (gaussian_kde)
# Use When:
# - Nonparametric density estimation (risk tails exploration)

import numpy as np
from scipy.stats import gaussian_kde

if __name__ == '__main__':
    rng = np.random.default_rng(0)
    x = rng.normal(size=200)
    kde = gaussian_kde(x)
    xs = np.linspace(-3, 3, 5)
    print(kde(xs))
```

22__stats__qmc__sobol.py

```
# Technique: Quasi-Monte Carlo (Sobol)
# Use When:
# - Low-discrepancy sequences for faster convergence (when applicable)

from scipy.stats import qmc

if __name__ == '__main__':
    sampler = qmc.Sobol(d=2, scramble=True, seed=0)
    x = sampler.random_base2(m=3)
    print(x.shape)
```

23_optimize_dual_annealing.py

```python
# Technique: Global Optimization (dual_annealing)
# Use When:
# - Non-convex problems; global search heuristic

import numpy as np
from scipy.optimize import dual_annealing

def f(x: np.ndarray) -> float:
    return float(np.sin(x[0]) + (x[0] - 2) ** 2)

if __name__ == '__main__':
    res = dual_annealing(f, bounds=[(-5, 5)])
    print(res.x, res.fun)
```

24_optimize_linear_programming.py

```python
# Technique: Linear Programming (linprog)
# Use When:
# - LP problems show up in allocation and constraints

import numpy as np
from scipy.optimize import linprog

if __name__ == '__main__':
    # minimize c^T x subject to A_ub x <= b_ub, x>=0
    c = np.array([1.0, 2.0])
    A = np.array([[1.0, 1.0], [-1.0, 2.0]])
    b = np.array([4.0, 2.0])
    res = linprog(c, A_ub=A, b_ub=b, bounds=[(0, None), (0, None)])
    print(res.x, res.fun, res.success)
```

25_interpolate_griddata_2d.py

```
# Technique: 2D Interpolation (griddata)
# Use When:
# - Interpolate scattered points (toy vol surface points)

import numpy as np
from scipy.interpolate import griddata

if __name__ == '__main__':
    pts = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=float)
    vals = np.array([0.0, 1.0, 1.0, 0.0])
    xi = np.array([[0.5, 0.5]])
    print(griddata(pts, vals, xi, method='linear'))
```

26_integrate_simpson.py

```python
# Technique: Simpson Integration (samples)
# Use When:
# - Integrate sampled data

import numpy as np
from scipy.integrate import simpson

if __name__ == '__main__':
    x = np.linspace(0, 1, 101)
    y = np.exp(-x * x)
    print(float(simpson(y, x=x)))
```

27_linalg_expm.py

```python
# Technique: Matrix Exponential (expm)
# Use When:
# - Used in continuous-time models and Markov chains

import numpy as np
from scipy.linalg import expm

if __name__ == '__main__':
    A = np.array([[0.0, 1.0], [-1.0, 0.0]])
    print(expm(A))
```

28_sparse_eigs.py

```python
# Technique: Sparse Eigenvalues (eigs)
# Use When:
# - Large systems: compute a few eigenvalues

import numpy as np
from scipy import sparse
from scipy.sparse.linalg import eigs

if __name__ == '__main__':
    A = sparse.diags([1.0, 2.0, 3.0, 4.0], 0, format='csr')
    w, v = eigs(A, k=2)
    print(w)
```

29_signal_find_peaks.py

```python
# Technique: Peak Finding (find_peaks)
# Use When:
# - Detect peaks in a signal

import numpy as np
from scipy.signal import find_peaks

if __name__ == '__main__':
    x = np.array([0, 1, 0, 2, 0, 1, 0], dtype=float)
    peaks, props = find_peaks(x, height=0.5)
    print(peaks, props['peak_heights'])
```

30__spatial__distance__cdist.py

```python
# Technique: Pairwise Distances (cdist)
# Use When:
# - Distance matrices for clustering/NN

import numpy as np
from scipy.spatial.distance import cdist

if __name__ == '__main__':
    A = np.array([[0.0, 0.0], [1.0, 1.0]])
    B = np.array([[1.0, 0.0]])
    print(cdist(A, B))
```

31__stats__anderson__darling.py

```
# Technique: Normality Test (Anderson-Darling)
# Use When:
# - Quick check for distributional assumptions

import numpy as np
from scipy import stats

if __name__ == '__main__':
    rng = np.random.default_rng(0)
    x = rng.normal(size=200)
    print(stats.anderson(x, dist='norm'))
```

32__stats__bootstrap.py

```python
# Technique: Bootstrap Confidence Interval
# Use When:
# - Resampling-based CI; useful with unknown distributions

import numpy as np
from scipy import stats

if __name__ == '__main__':
    rng = np.random.default_rng(0)
    x = rng.normal(loc=0.1, scale=1.0, size=200)
    res = stats.bootstrap((x,), np.mean, n_resamples=500, method='basic', random_state=0)
    print(res.confidence_interval)
```

33_optimize_curve_fit.py

```python
# Technique: curve_fit (quick param fit)
# Use When:
# - Fit parametric curves; careful with extrapolation

import numpy as np
from scipy.optimize import curve_fit

def model(x: np.ndarray, a: float, b: float) -> np.ndarray:
    return a * x + b

if __name__ == '__main__':
    x = np.array([0.0, 1.0, 2.0, 3.0])
    y = np.array([1.0, 2.1, 3.9, 6.2])
    p, _ = curve_fit(model, x, y)
    print(p)
```

34_interpolate_pchip.py

```python
# Technique: Shape-Preserving Interpolation (PCHIP)
# Use When:
# - Monotone interpolation useful for curves (rates) to avoid overshoot

import numpy as np
from scipy.interpolate import PchipInterpolator

if __name__ == '__main__':
    x = np.array([0, 1, 2, 3], dtype=float)
    y = np.array([0, 1, 1.5, 1.6], dtype=float)
    f = PchipInterpolator(x, y)
    print(f(np.array([0.5, 1.5, 2.5])))
```

35_signal_resample_poly.py

```python
# Technique: Resampling Signals (resample_poly)
# Use When:
# - Rational resampling with anti-alias filtering

import numpy as np
from scipy.signal import resample_poly

if __name__ == '__main__':
    x = np.arange(10, dtype=float)
    y = resample_poly(x, up=2, down=1)
    print(y.shape)
```

36_sparse_cg_solver.py

```
# Technique: Iterative Solve (Conjugate Gradient)
# Use When:
# - Large SPD systems solved iteratively

import numpy as np
from scipy import sparse
from scipy.sparse.linalg import cg

if __name__ == '__main__':
    A = sparse.csr_matrix([[4.0, 1.0], [1.0, 3.0]])
    b = np.array([1.0, 2.0])
    x, info = cg(A, b, maxiter=50)
    print(x, info)
```

37_linalg_block_diag.py

```
# Technique: Block Diagonal Matrices
# Use When:
# - Build block structures for models

import numpy as np
from scipy.linalg import block_diag

if __name__ == '__main__':
    A = np.eye(2)
    B = 2 * np.eye(3)
    C = block_diag(A, B)
    print(C.shape)
```

38_spatial_convex_hull.py

```python
# Technique: Convex Hull (geometry)
# Use When:
# - Geometry utilities via scipy.spatial

import numpy as np
from scipy.spatial import ConvexHull

if __name__ == '__main__':
    pts = np.array([[0, 0], [1, 0], [1, 1], [0, 1], [0.2, 0.2]])
    hull = ConvexHull(pts)
    print(hull.vertices)
```

39_ndimage_label.py

```python
# Technique: Connected Components (ndimage.label)
# Use When:
# - Label connected regions in a binary matrix

import numpy as np
from scipy.ndimage import label

if __name__ == '__main__':
    x = np.array([[1, 0, 0], [1, 1, 0], [0, 0, 1]], dtype=int)
    lbl, n = label(x)
    print(n)
    print(lbl)
```

40_special_gamma.py

```python
# Technique: Special Functions (gamma)
# Use When:
# - Gamma appears in distributions and normalization constants

import numpy as np
from scipy import special

if __name__ == '__main__':
    x = np.array([0.5, 1.0, 2.5])
    print(special.gamma(x))
```

41_constants_physical.py

```python
# Technique: scipy.constants (sanity constants)
# Use When:
# - Standard constants (mostly not finance, but useful pattern)

from scipy import constants

if __name__ == '__main__':
    print(constants.pi)
```

42__stats__rankdata.py

```python
# Technique: Rank Data (rankdata)
# Use When:
# - Robust ranking for signals

import numpy as np
from scipy.stats import rankdata

if __name__ == '__main__':
    x = np.array([10.0, 20.0, 20.0, 5.0])
    print(rankdata(x, method='average'))
```

43_stats_winsorize.py

```python
# Technique: Winsorization (mstats.winsorize)
# Use When:
# - Cap extremes to reduce outlier impact

import numpy as np
from scipy.stats.mstats import winsorize

if __name__ == '__main__':
    x = np.array([-10.0, -1.0, 0.0, 1.0, 10.0])
    y = winsorize(x, limits=(0.2, 0.2))
    print(np.asarray(y))
```

44_optimize_brent_scalar.py

```python
# Technique: Scalar Minimization (Brent)
# Use When:
# - Fast 1D optimization (e.g., implied vol root/min)

import numpy as np
from scipy.optimize import minimize_scalar

def f(x: float) -> float:
    return float((x - 1.234) ** 2 + 0.1 * np.sin(10 * x))

if __name__ == '__main__':
    res = minimize_scalar(f, method='brent')
    print(res.x, res.fun)
```

45_optimize_bisect_root.py

```python
# Technique: 1D Root Finding (bisect)
# Use When:
# - Reliable bracketing root method

from scipy.optimize import bisect

def f(x: float) -> float:
    return x * x - 2.0

if __name__ == '__main__':
    r = bisect(f, 1.0, 2.0)
    print(r)
```

46_signal_correlate.py

```python
# Technique: Correlation (signal.correlate)
# Use When:
# - Cross-correlation for lag relationships

import numpy as np
from scipy.signal import correlate

if __name__ == '__main__':
    x = np.array([1, 2, 3])
    y = np.array([0, 1, 0.5])
    c = correlate(x, y, mode='full')
    print(c)
```

47_spatial_rotation.py

```
# Technique: Rotations (spatial.transform.Rotation)
# Use When:
# - Demonstrates clean rotation APIs

import numpy as np
from scipy.spatial.transform import Rotation

if __name__ == '__main__':
    r = Rotation.from_euler('z', 90, degrees=True)
    v = np.array([1.0, 0.0, 0.0])
    print(r.apply(v))
```

48_interpolate_bsplines.py

```python
# Technique: B-splines (BSpline)
# Use When:
# - Low-level spline representation

import numpy as np
from scipy.interpolate import BSpline

if __name__ == '__main__':
    t = np.array([0, 0, 0, 1, 2, 3, 3, 3], dtype=float)
    c = np.array([0.0, 1.0, 0.0, 1.0, 0.0])
    k = 2
    sp = BSpline(t, c, k)
    print(sp(np.array([0.5, 1.5, 2.5])))
```

49__stats__skew__kurtosis.py

```python
# Technique: Skew/Kurtosis
# Use When:
# - Tail shape metrics (be cautious; noisy estimates)

import numpy as np
from scipy.stats import skew, kurtosis

if __name__ == '__main__':
    x = np.array([1.0, 2.0, 3.0, 10.0, -5.0])
    print('skew:', float(skew(x)))
    print('kurt:', float(kurtosis(x)))
```

50_sparse_random_and_norm.py

```
# Technique: Sparse Random Matrices + Norms
# Use When:
# - Generate sparse random and compute norms

import numpy as np
from scipy import sparse
from scipy.sparse.linalg import norm

if __name__ == '__main__':
    A = sparse.random(100, 100, density=0.01, format='csr', random_state=0)
    print(float(norm(A)))
```