PYTHON ADVANCED & DATA SCIENCE

EXHAUSTIVE SCENARIO-MAPPED TECHNICAL COMPENDIUM

PART 1: ADVANCED OBJECT-ORIENTED PROGRAMMING & INTERNAL ARCHITECTURE

1. Encapsulation Matrix & Data Protection Mechanics

- **Public Scope:** Attributes declared without leading underscores (e.g., `self.title`). Fully accessible externally. Use exclusively if fields have zero restrictions or validation constraints.
- **Private Scope & Name Mangling:** Fields prefixed with double underscores (e.g., `self.__height`). Triggers a compiler transformation changing the internal slot handle to `_ClassName__attribute`. Protects internal system mechanics against accidental overrides during class inheritance loops.
- **Protected Scope:** Fields prefixed with a single underscore (e.g., `self._height`). An explicit design API contract signaling to downstream engineers that the underlying data layout is internal and subject to breaking modifications.

```
class Wall:
    def __init__(self, height):
        self._height = height          # Protected instance attribute convention
        self.__internal_id = 9912      # Private attribute; name-mangled to _Wall__internal_id

    def get_height(self):
        return self._height            # Regulated public entry point routine
```

2. Smart Control Properties (Uniform Access Principle Getter/Setter Matrix)

Properties decouple how class fields are stored in hardware from how callers access them using dot notation, eliminating bulky custom getter/setter method calls.

```
class Book:
    def __init__(self, title, isbn):
        self.title = title            # Routes directly through the title setter method validat
        self._isbn = isbn

    @property
    def title(self):
        return self._title            # Read-only public view interface wrapper (Getter)

    @title.setter
    def title(self, value):
        if not value or not isinstance(value, str):
            raise RuntimeError("Validation Fault: Title must be a non-empty string struc
        self._title = value           # Active rule enforcement gateway (Setter)

    @property
    def isbn(self):
        return self._isbn             # Immutable read-only class property
```

3. Finite Memory Processing: Iterables, Generators, & Comprehension Memory Footprints

- **Iterables:** Structural data collections (Lists, Tuples, Dictionaries, Sets, Strings) containing multi-element states. They implement the iterator protocol loop interface and can be traversed multiple times via standard `for` blocks.
- **Generators:** Lazy-evaluated iterator blocks declared using `yield` statements. Instead of allocating memory for an entire collection at once, generators process items on demand and pause execution state between yields. *Critical Exam Fact:* Once a generator loop finishes, it is "spent" and empty; to reuse it, you must instantiate a fresh generator object.
- **Generator Expressions:** High-performance lazy iterators declared using parentheses: `(expr for x in xs)`. They run operations item-by-item to keep application memory footprint at $O(1)$.

```
def languages():
    yield "Python"
    yield "Java"
    yield "Haskell"

# Spent state trap:
g = languages()
for lang in g: print(lang) # Prints Python, Java, Haskell
for lang in g: print(lang) # Sinks silently. Generator is completely spent!

# Memory Allocation Contrast:
lowest_weight_clean = min(person.weight for person in people) # Optimized: Streams records seque
total_cost = sum(item.price for item in shopping_basket)
passed = all(student.grade(c) > 10 for c in student.courses)
```

4. Recursive Context Engines & Context Call Stacks

Recursive architectures solve large computations by breaking them down into simpler, self-referential tracking steps inside the system frame execution context stack.

- **Base Case:** A strict conditional code path that prevents further loops and stops execution. Without it, the program crashes with a `RecursionError`.
- **Recursive Case:** Performs a processing step and triggers a self-referential call, moving arguments closer to the base case boundaries.

```
def factorial_recursive(n):
    if n == 1:                    # Absolute structural terminal condition (Base Case)
        return 1
    return n * factorial_recursive(n - 1) # Context reduction call step (Recursive Case)
```

5. High-Order Iterative Built-ins & Advanced Key Sort Multi-Tie-Breaking

- **Boolean Matrix Diagnostics:** `all()` evaluates if every single iterable element checks out as truthy. `any()` verifies if at least one record tests truthy.
- **Sequence Alignment Mapping:** `zip()` locks parallel tracking matrices into synchronized multi-element tuple sequences.
- **Functional Key Extractor Sorting:** `sort()` and `sorted()` accept an optional tracking callback parameter: `key=lambda x: ...`. This lets you sort complex object properties without altering the original structural layout.
- **Multi-Dimensional Tie-Breaking:** When processing non-scalar types like tuples, Python compares indexes sequentially from left to right (e.g., if $a == c$ in $(a, b) < (c, d)$, the tie-breaker scales directly to assess if $b < d$).

```
cards.sort(key=lambda card: (card.suit, card.value)) # Primary suit, secondary value tie-breaki
oldest_person = max(people, key=lambda person: person.age)
```

PART 2: REGULAR EXPRESSIONS ENGINE (RE) SYNTAX BLUEPRINT

Pattern	Structural Match Expression	Pattern	Structural Match Expression
`.`	Any character except newline	`*`	0 or more continuous repetitions
`\d`	Decimal digits `[0-9]`	`+`	1 or more continuous repetitions
`\w`	Alphanumeric characters `[a-zA-Z0-9_]`	`?`	0 or 1 optional match instance
`\s`	Whitespace character (spaces, tabs, newlines)	`{n,m}`	Bounded structural interval loop: minimum *n*, maximum *m*
`^` or `\A`	Start of text string alignment	`[...]`	Custom set matching single character inside brackets
`$` or `\Z`	End of text string alignment	`[^...]`	Inverted matching set rejects characters in brackets

```
import re
clean_isbn = re.sub(r'[-\s]', '', '978-1779501127') # Strip noise formatting dashes/whitespace
```

PART 3: THE PYTEST SOFTWARE TESTING ECOSYSTEM

1. Arrange-Act-Assert (AAA) Architectural Pattern

Maintains strict deterministic bounds by separating test environments, target executions, and outcomes verification.

```
def test_task_becomes_overdue():
    # Arrange: Setup prerequisites, dates, mocks, and target system.
    today = date(2000, 1, 1)
    calendar = CalendarStub(today)
    task = Task('Bake cake', date(2000, 1, 2))
    sut = TaskList(calendar)
    sut.add_task(task)

    # Act: Trigger system modifications
    calendar.today = date(2000, 1, 8)

    # Assert: Verify matching requirements
    assert [task] == sut.overdue_tasks
```

2. Unbundled Parametrization Isolation Matrices

Combining multiple asserts within one test causes failures to cascade and masks bugs. Isolate datasets to separate execution frames via `@pytest.mark.parametrize`.

```
import pytest

@pytest.mark.parametrize('interval1, interval2, expected', [
    ((1, 5), (3, 6), True),
    ((1, 2), (3, 4), False),
    ((1, 5), (5, 6), True),
    ((4, 0), (1, 2), False)
])
def test_intervals(interval1, interval2, expected):
    assert overlapping_intervals(interval1, interval2) == expected
```

3. Decoupled Context Fixtures & Parameter Injection

Avoid global `setup_function()` declarations that run indiscriminately. Use dependency-injected `@pytest.fixture` components to manage state life cycles cleanly.

```
@pytest.fixture
def today():
    return date(2000, 1, 1)

@pytest.fixture
def calendar(today):
    return CalendarStub(today)

@pytest.fixture
def sut(calendar):
    return TaskList(calendar)

def test_creation(sut):
    assert len(sut) == 0
```

4. Numerical Tolerance Limits & Exception Verification

- **Binary Floating-Point Approximation:** Decimal values converted to binary floating point yield minor rounding anomalies (e.g., $0.1 + 0.2 != 0.3$). Wrap lookups in `pytest.approx()`.
- **Exception Testing:** Use `with pytest.raises(...)` context manager block to verify execution faults. Keep blocks minimal.

```
from pytest import approx
import pytest

assert approx(0.1 + 0.2) == 0.3
assert approx(10.0, abs=0.001) == 10.002

def test_creation_with_invalid_title():
    with pytest.raises(RuntimeError):
        Book(title=' ', isbn='978-1779501127')
```

5. Differential Twin-System Testing Matrix Patterns

Compare complex algorithms' outputs against a slower, highly verified reference implementation (e.g., built-in `sorted()`).

```
@pytest.mark.parametrize('ns', [[3, 1, 2], [5, 8, 1, 9, 2], [10, 20, 30]])
def test_differential_algorithmic_sort(ns):
    assert complex_fast_merge_sort(ns) == sorted(ns)
```

> **CRITICAL EXAM ALERT: RELEASE OPTIMIZATION TRAP (python -O)**
> Interpreters compiled inside optimized performance releases automatically eliminate global script execution constraints wrapped by internal assert checkpoints. **Never use production assertions for validation blocks.**

PART 4: COMPLETE HOME AUTOMATION EXAM SCENARIO

1. System Architecture: ABC, Classes, and Relationships

```
from abc import ABC, abstractmethod

class Location:
    def __init__(self, room, floor):
        self.room = room
        self.floor = floor

class Device(ABC):
    def __init__(self, name, location, energy_consumption):
        self._name = name
        self._location = location
        self._energy_consumption = energy_consumption

    @property
    def name(self):
        return self._name

    @property
    def location(self):
        return self._location

    @property
    def energy_consumption(self):
        return self._energy_consumption

    @property
    @abstractmethod
    def is_on(self):
        pass

    @abstractmethod
    def turn_on(self):
        pass

    @abstractmethod
    def turn_off(self):
        pass

    def __lt__(self, other):
        return self.energy_consumption < other.energy_consumption
    def __gt__(self, other):
        return self.energy_consumption > other.energy_consumption
    def __eq__(self, other):
        return self.energy_consumption == other.energy_consumption
```

2. Specialized Devices: Lamp, Heating, Washing Machine

```
class Lamp(Device):
    def __init__(self, name, location, energy_consumption):
        super().__init__(name, location, energy_consumption)
        self._brightness = 0

    @property
    def brightness(self):
        return self._brightness

    @property
    def is_on(self):
        return self._brightness > 0

    def turn_on(self):
        self._brightness = 100

    def turn_off(self):
        self._brightness = 0


class Heating(Device):
    def __init__(self, name, location, energy_consumption):
        if not isinstance(location, Location): raise TypeError("Must be Location instanc
        if not isinstance(name, str): raise TypeError("Name must be string")
        if name.strip() == "": raise ValueError("Name cannot be empty")
        if not isinstance(energy_consumption, int): raise TypeError("Energy must be int"
        if energy_consumption < 0: raise ValueError("Energy cannot be negative")
        super().__init__(name, location, energy_consumption)
        self._temperature = 0
        self._preferred_temperature = 20

    @property
    def temperature(self):
        return self._temperature

    @property
    def is_on(self):
        return self._temperature > 0

    def turn_on(self):
        self._temperature = self._preferred_temperature

    def turn_off(self):
        self._temperature = 0

    @property
    def preferred_temperature(self):
        return self._preferred_temperature

    @preferred_temperature.setter
    def preferred_temperature(self, temperature):
        if not isinstance(temperature, (int, float)): raise TypeError("Must be number")
        if temperature < 0 or temperature > 100: raise ValueError("Must be between 0 and
        self._preferred_temperature = temperature
```

```
class WashingMachine(Device):
    def __init__(self, name, location, energy_consumption):
        super().__init__(name, location, energy_consumption)
        self._is_on = False

    @property
    def is_on(self):
        return self._is_on

    def turn_on(self):
        self._is_on = True

    def turn_off(self):
        self._is_on = False
```

3. Central Automation Engine Management & Filtering

```
class HomeAutomation:
    def __init__(self):
        self.devices = []

    def add_device(self, device):
        if device is None: raise ValueError("Device cannot be None.")
        self.devices.append(device)

    def turn_on_all(self):
        for device in self.devices: device.turn_on()

    def turn_off_all(self):
        for device in self.devices: device.turn_off()

    def get_status(self):
        lines = []
        for device in self.devices:
            state = "on" if device.is_on else "off"
            loc_str = f"{device.location.room} (Floor {device.location.floor})"
            lines.append(f"Device {device.name} with energy {device.energy_consumption}
        return "
".join(lines)

    def filter(self, condition):
        return [device for device in self.devices if condition(device)]

    def filter_by_location(self, location):
        return self.filter(lambda device: device.location == location)

    def filter_by_energy_consumption(self, energy_consumption):
        return self.filter(lambda device: device.energy_consumption < energy_consumption
```

4. Exam-Grade Pytest Suite for Home Automation

```
import pytest
from datetime import date

def test_successful_heating_initialization():
    loc = Location("Living room", 1)
    heating = Heating("Heater", loc, 3000)
    assert heating.name == "Heater"
    assert heating.location == loc
    assert heating.energy_consumption == 3000
    assert heating.temperature == 0
    assert heating.preferred_temperature == 20

@pytest.mark.parametrize("name, location, energy, error", [
    ("heating", "living room", 3000, TypeError),
    (696969, Location("living room", 1), 3000, TypeError),
    ("", Location("living room", 1), 3000, ValueError),
    ("heating", Location("living room", 1), "3000", TypeError),
    ("heating", Location("living room", 1), 3000, ValueError)
])
def test_exception_heating_constructor(name, location, energy, error):
    with pytest.raises(error):
        Heating(name, location, energy)

def test_device_rich_comparisons():
    loc = Location("Kitchen", 0)
    low_p = Lamp("Eco", loc, 15)
    high_p = Heating("Furnace", loc, 3000)
    assert low_p < high_p
    assert high_p > low_p
```

PART 5: DATA MANIPULATION & ANALYSIS (PANDAS)

1. Data Initialization & I/O Pipeline

```
import pandas as pd
import numpy as np

my_series = pd.Series([10, 100, 10], index=['London', 'Tripoli', 'Cairo'])

data_dict = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [60000, 75000, 80000],
    'City': ['London', 'Paris', 'Tokyo']
}
df = pd.DataFrame(data_dict)

df.to_csv('output.csv', index=False) # Export file without index values
df_csv = pd.read_csv('soccer_data.csv', sep=',', index_col=0)
```

2. Structural Diagnosis, Slicing, Conditional Filtering & Mutations

- `df.head(8)` - First 8 target metrics rows.
- `df.tail(3)` - Trailing 3 entries.
- `df.shape` - Dimensional tracker tuple: `(rows, columns)`.
- `df.info()` - Column data types schema inventory metrics and non-null status checks.
- `df.describe()` - Summary statistics engine parameters (mean, standard deviation, quartiles).

```
df['Name']             # Extract single column as Series
df[['Name', 'Salary']] # Filter multi-columns
df.iloc[0:3]           # Extract rows based on position index

df.rename(index={'London': 'London-Hub'}, columns={'Age': 'Years'}, inplace=True)
df.drop(labels=['London-Hub'], axis=0, errors='ignore', inplace=True)

# Boolean Mask Combining (CRITICAL: Always use bitwise operators & or | with round brackets)
london_high_earners = df[(df['Salary'] > 65000) & (df['City'] == 'London')]
complex_filter = df[(df['Years'] <= 30) | (df['Salary'] == 80000)]
```

3. Grouping, Aggregations & Datetime Conversion

```
city_means = df.groupby('City')['Salary'].mean()
top_routes = df.groupby(['Route'])['Passengers'].sum().sort_values(ascending=False).head
completion_rates = df['Booking_Complete'].value_counts(normalize=True) * 100

df['date'] = pd.to_datetime(df['date'], dayfirst=False)
df['Year'] = df['date'].dt.year
df['Month'] = df['date'].dt.month
```

4. Row Transformations via .apply() and .explode()

- `.apply(axis=1)`: Maps custom evaluation parsing functions row-by-row across raw data matrix tracks.
- `.explode()`: Flattens cells containing lists or strings into distinct independent rows while duplicating matching index handles.

```
def evaluate_stock(row):
    return 'downwards' if row['open'] > row['close'] else 'upwards'
df['trend'] = df.apply(evaluate_stock, axis=1)

df['genres'] = df['genres'].apply(lambda x: x.split(',') if isinstance(x, str) else [])
df_exploded = df.explode('genres')
```

PART 6: VISUALIZATIONS & PLOTTING (MATPLOTLIB)

1. Line Plot Blueprint

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.plot(df['year'], df['Pigeons'], color='blue', linestyle='', marker='o', linewidth=2,
plt.title('Pigeon Population Dynamics over Time', fontsize=12, fontweight='bold')
plt.xlabel('Observational Year Metric')
plt.ylabel('Quantified Bird Count')
plt.grid(True, linestyle='', alpha=0.6)
plt.legend(loc='upper left')
plt.show()
```

2. Unaligned Offset Grouped Bar Configuration

To prevent multiple bar series from overlapping on the same chart, shift their positions along the X-axis manually using exact width steps.

```
bar_width = 0.2
x_positions = range(len(df['year']))

plt.figure(figsize=(10, 6))
plt.bar([pos - bar_width for pos in x_positions], df['Pigeons'], bar_width, color='blue'
plt.bar(x_positions, df['Sparrows'], bar_width, color='green', label='Sparrows')
plt.bar([pos + bar_width for pos in x_positions], df['others'], bar_width, color='orange

plt.xticks(x_positions, df['year'])
plt.legend()
plt.show()
```

3. Horizontal Stacked Progression Bars

```
fig, ax = plt.subplots(figsize=(8, 6))
bar_thickness = 0.35
categorical_labels = ['Adventure', 'RPG', 'Shooter']

# Implement the left tracking parameters to stack values incrementally
ax.barh(categorical_labels, dataset_value_a, bar_thickness, label='Completed Metrics', c
ax.barh(categorical_labels, dataset_value_b, bar_thickness, left=dataset_value_a, label=

ax.set_xlabel('Number of Players')
ax.legend()
plt.show()
```

4. Multi-Axis Layout Subplots Mosaic & Grid Subplot Configurations

```
# Define canvas positioning layout using semantic mapping text strings
fig, axs = plt.subplot_mosaic([['left', 'right']], figsize=(12, 6))

axs['left'].plot(df['year'], df['Pigeons'], color='blue', marker='o')
axs['left'].set_title('Linear Trajectory tracking vector path')

axs['right'].bar(df['year'], df['Total birds'], color='purple', width=0.4)
axs['right'].set_title('Volumetric Column Breakdown')

plt.tight_layout()
plt.show()
```