THE ULTIMATE "WHEN TO USE WHAT" CHEATSHEET

ARCHITECTURAL DECISION MATRIX FOR PANDAS & OBJECT-ORIENTED PYTHON

🐼 Section 1: Pandas Data Decisions

1. Row Filtering & Subsetting

Syntax Pattern	When to Use It	The Hidden Gotcha / Why
`df[df['col'] > val]`	Quick, read-only temporary viewing or assigning to a completely new DataFrame variables.	Throws a `SettingWithCopyWarning` if you try to mutate or update the rows downstream.
`df.loc[df['col'] > val, 'target_col'] = new_val`	**Mandatory rule** whenever you intend to modify, overwrite, or create a column based on a row condition.	Guarantees operations occur safely on the original underlying block memory structure layout instead of a view copy.
`df.query("col > @val")`	Complex programmatic filter strings with multiple logical variables or dynamic chains.	Drastically increases readability by removing structural brackets. Reference local Python variables using `@` prefix.

> ⚠️ **CRITICAL PANDAS FILTER TRAP**
>
> Standard Python operators (`and` , `or` , `not`) do not work on Pandas sequences. You **must** use the bitwise operators: `&` (and), `|` (or), `~` (not). Always wrap each sub-condition strictly inside parenthetical blocks: `df[(df['Quantity'] > 2) & (df['article'] == 'BAGUETTE')]`

2. Transforming Element Data

Operation Paradigm	When to Use It	Performance Level
Vectorization `df['total'] = df['qty'] * df['price']`	Default baseline logic. Arithmetic, math boundaries, basic text concatenation arrays.	**Blazing Fast.** Executes compiled C under the hood via vectorized numpy routines.
`.map()` Lookup Map `df['day'] = df['day_fr'].map(d_lookup)`	Translating categorical keys, processing lookups via key-value dictionaries across a single series.	**Optimized.** Efficient execution paths explicitly targeted for index/ hash mappings.
`.apply(lambda x: ...)` `df.apply(custom_func, axis=1)`	Complex logical branches (like nested `if-elif-else` scripts) matching separate columns.	**Extremely Slow.** Acts as a hidden pure Python loop. Avoid on production large-scale data rows.

3. Conversions & Structural Casting

- Use `pd.to_datetime(df['date'])` to ingest unstable text timestamps. It successfully converts dirty formats, extracts dates safely, and translates blank items into standard `NaT` values.
- Use `df['col'].astype('int')` or `.astype('category')` to lock down pure system data tracking, or optimize data structures for repetitive textual values (like day titles or product categories).

⚙️ Section 2: Object-Oriented Blueprint Rules

1. Abstract Classes vs. Concrete Implementation

- **Make a template abstract (via `from abc import ABC, abstractmethod`) when:** It represents a conceptual architecture model (e.g., `Device`) that should never be instantiated standalone in the runtime stack.
- **Mark a method with `@abstractmethod` when:** Every inherited concrete subclass *must* feature this routine block, but each implementation behaves completely differently (e.g., `turn_on()` behaves differently for a `Lamp` versus a `Heating` unit).

2. Method Decorator Blueprint Matrix

Method Type	Implicit First Arg	When to Use It
Instance Method	`self`	Standard routine behavior that must fetch or update instance state fields specific to that unique instance object.
Class Method `@classmethod`	`cls`	For **Alternative Constructors** (instantiating from specialized formats like string lines or loaded JSON trees) or parsing global class context attributes.
Static Method `@staticmethod`	None	For structural isolation utilities that naturally belong inside the namespace block domain of the class, but read/write **zero** instance data or class attributes.

```
class Device:
    def __init__(self, name):
        self.name = name

    # 1. Instance Method
    def update_name(self, new_name):
        self.name = new_name

    # 2. Class Method (Alternative Constructor)
    @classmethod
    def from_string(cls, input_str):
        name = input_str.split(",")[0]
        return cls(name)

    # 3. Static Method (Pure utility)
    @staticmethod
    def kw_to_watts(kw):
        return kw * 1000
```

3. Encapsulation Access Controls

- `self.name` **(Public)**: Directly exposed fields. Anyone outside can read, overwrite, or mutate tracking variables at will.
- `self._name` **(Protected)**: Design communication marker alerting downstream developers: *"Internal class framework asset. Do not alter out of context."*
- `self.__name` **(Private / Name Mangling)**: Enforced structural containment protection. Hard-locks properties out of exterior reach. Provide managed endpoints via `@property` fields:

```
@property
def preference(self):
    return self.__preference

@preference.setter
def preference(self, value):
    if not (0 <= value <= 100):
        raise ValueError("Value out of valid ranges!")
    self.__preference = value
```

🛑 Section 3: Fatal Mistakes Elimination Checklist

- ☐ **Inherited Constructor Amnesia:** Ensure you include `super().__init__(...)` inside child classes. If omitted, parent properties fail to map, rendering future attribute references broken.
- ☐ **The Shared Mutable Defaults Trap:** Never construct routines matching `def __init__(self, items=[])`. Python constructs default objects exactly once at memory import load time, which shares a state across separate instances. Use `items=None` paired with an inline safe instantiation check.
- ☐ **The Ghost Mutation operations in Pandas:** Methods like `df.drop_duplicates()`, `df.dropna()`, or `df.rename()` do not perform edits dynamically in place unless explicitly declared with `inplace=True` or overwritten via explicit assignment variable definitions: `df = df.operation()`.