ChatGPT

Below is a comprehensive expert reference for advanced **pandas** (v2.x) usage, focusing on high-impact features beyond the basics. It assumes you're fluent in DataFrame fundamentals and dives into extension types, multi-dimensional indexing, window functions, expression evaluation, performance tuning, time-series intricacies, and more.

---

0) Snapshot & Context

- **Latest (PyPI):** `pandas 2.3.3` released Sep 29, 2025. Requires **Python ≥3.9** [1]. Notable 2.x changes include optional **Apache Arrow** backing for many data types and a new **Copy-on-Write** memory model (off by default until pandas 3.0) [2].
- **Key 2.x features:** Arrow-backed dtypes for faster I/O and interoperability, experimental copy-on-write (eliminates unintended view modifications [3] [4]), expanded NULL support (NA for all dtypes), and numerous performance tweaks.
- **Scope:** This guide skips basic usage (indexing, simple I/O, etc.) and centers on advanced patterns: custom dtypes and accessors, MultiIndex operations, rolling/expanding windows, vectorized string/date/category methods, `eval()` / `query()`, custom aggregation, memory optimizations, time zone handling, file formats, and Arrow interop.

---

1) Extension Arrays & Custom Dtypes (and Subclassing Pandas)

Extension types: Pandas lets you define custom 1-D array types that integrate with its APIs (no silent conversion to `object` dtype). You implement a subclass of `pandas.api.extensions.ExtensionDtype` (describes the type metadata) and `pandas.api.extensions.ExtensionArray` (the actual data container) [5] [6]. Pandas uses this for internal types like **Categorical**, **Period**, **DatetimeTZ**. By creating an ExtensionArray, you get compatibility with pandas ops: e.g. `isna()`, `fillna()`, `astype()` will delegate to your implementation [5] [7]. You can even **register** the dtype so that `pd.Series(..., dtype="your_dtype_name")` works [8].

- *Example:* An IP address type might define `IPv4Dtype(ExtensionDtype)` and `IPv4Array(ExtensionArray)`. The `ExtensionDtype.name` (e.g. `"ipv4"`) can be registered to allow `dtype="ipv4"` [9]. The ExtensionArray holds the data (maybe as packed integers or as a structured dtype) and implements methods like `__getitem__`, `__len__`, `.isna()`, `.take()`, etc. There are mixins to help implement operators: e.g. `ExtensionScalarOpsMixin` provides vectorized arithmetic if your scalar elements have those operators [10] [11].

- **Operators:** By default, custom arrays have no arithmetic/comparison operators. You can either implement them one by one, or mix in pandas' `ExtensionScalarOpsMixin` and call `_add_arithmetic_ops()` / `_add_comparison_ops()` to auto-generate ops based on element-wise scalar ops [10] [11].

Series/DataFrame subclassing: In most cases, pandas' extension mechanism or custom accessors (next section) are preferable to subclassing. But if you need a specialized subclass (e.g. GeoPandas' GeoDataFrame), pandas supports it with some effort. Key points when subclassing `pd.Series` or `pd.DataFrame`:

- Override the `_constructor` properties so pandas operations return your subclass. For a DataFrame subclass, define `_constructor` (for DataFrame outputs) and `_constructor_sliced` (for Series outputs). For a Series subclass, define `_constructor` (Series outputs) and `_constructor_expanddim` (if the Series is transformed into a DataFrame) [12] [13]. This ensures, for example, slicing a subclassed DataFrame or calling `.to_frame()` on a subclassed Series yields your custom classes [14] [15].

- Use the `_metadata` attribute to list custom attributes that should propagate on copy. Attributes listed in `_metadata` will survive operations like `copy()` or slicing, whereas attributes in `_internal_names` are for internal use and won't propagate [16] [17]. For example, you might store a `schema` or `unit` metadata in a Series subclass; add its name to `_metadata` so that when pandas creates a new Series from it, the metadata is carried over.

When to use: Try to achieve your goals with extension dtypes or accessors first. Subclassing pandas objects is powerful but can be brittle across versions. Pandas itself cautions that **accessors or composition** are often safer than subclassing [18] [19].

---

2) Custom Accessors (`.str`, `.dt`, `.cat`, and your own)

Pandas' built-in accessors `.str`, `.dt`, `.cat` are examples of namespaced accessors: they hang custom methods off Series of certain types. You can register your own accessors to extend pandas objects with domain-specific methods, **without** subclassing.

- **Registering an accessor:** Use the decorators `@pandas.api.extensions.register_series_accessor("name")` (or `"dataframe_accessor"`, `"index_accessor"`) on a class that takes a pandas object in its `__init__`. The string you provide (e.g. `"geo"`) becomes the attribute name available on Series (or DataFrames, etc.) [20] [21]. For example, adding a `.geo` namespace for DataFrames:

```
@pd.api.extensions.register_dataframe_accessor("geo")
class GeoAccessor:
    def __init__(self, pandas_obj):
        self._obj = pandas_obj
        # (Optionally validate the DataFrame structure here)
    def center(self):
        lat = self._obj.latitude; lon = self._obj.longitude
        return (float(lon.mean()), float(lat.mean()))
    def plot(self): ...
```

Now any DataFrame `df` with columns `"latitude"` and `"longitude"` has `df.geo.center` and `df.geo.plot()` available [22] [23]. Accessor classes often validate that the underlying data meets prerequisites (raising AttributeError if not) [24] [25].

- **Why use accessors:** They let you **encapsulate custom logic** (e.g. geospatial transforms, financial analysis methods) and make it feel like a first-class pandas feature (`df.geo.method()`). This avoids monkey-patching and is easier to maintain than subclassing. Many libraries (GeoPandas, pandas-profiling, etc.) use this to add functionality.

- **Built-in accessors:** `.str` is automatically available for string or object-dtype Series, `.dt` for datetime-like Series (or columns in DataFrame via `.dt` accessor on the Series), and `.cat` for categorical Series. They provide vectorized methods:

- `series.str`: **text vectorization** – e.g. `s.str.contains("foo")`, `s.str.upper()`, `s.str.extract(r'(\d+)')`. These operations run C-optimized loops or use vectorized libraries under the hood, far faster than Python loops. They also **skip NA values** by design. Under pandas' nullable string dtype, `.str` methods that return numeric/bool will produce pandas nullable dtypes (e.g. `Int64`, `boolean`) instead of pure NumPy types [26] [27]. For example, counting occurrences:

  ```
  s = pd.Series(["a", None, "b"], dtype="string")
  s.str.count("a")  # -> Series(dtype: Int64) with <NA> for the None [28] [29]
  ```

  (By contrast, object dtype would give float output for NA.) Most usual string ops (length, case, find/replace, regex extract, slicing, etc.) are available.

- `series.dt`: **date/time properties and methods** – only for datetime64, timedelta64, or period Series. This offers properties like `.year`, `.weekday`, `.days` (for timedeltas), and methods like `.floor("H")`, `.tz_localize(tz)`, `.tz_convert(tz)`, `.to_period()`, etc. These enable vectorized time transformations without manually iterating. E.g. `df["order_date"].dt.strftime("%Y-%m")` yields a vector of formatted year-month strings. If you use **Arrow-backed timestamp arrays** (see §11), most `.dt` methods are supported as of pandas 2.0+ [30].

- `series.cat`: **categorical operations** – appears for dtype `"category"`. Allows getting/setting the category **levels** (`.cat.categories`) and order (`.cat.ordered`) [31] [32], adding or removing categories (`.cat.add_categories(new_categories)` or `.cat.remove_unused_categories()` to drop unused ones), and renaming or reordering categories. You can also use `CategoricalDtype` to define categories and sort order up front [33] [34]. Categorical accessors are crucial when working with pandas' **Categorical** type for memory and speed benefits (especially for object/string data with many repeats).

Advanced: You can even **override built-in accessors** by registering a custom one with the same name, but do so with caution (it may break expected pandas behavior). Typically, choose a unique namespace for your accessors.

---

3) MultiIndex and Hierarchical Data

A **MultiIndex** (hierarchical index) allows multiple levels of labels on a single axis, enabling you to represent higher-dimensional data in 2D tabular form. For example, a Series with a MultiIndex can behave like a 2D table (row+col in the index), and a DataFrame with MultiIndex columns can hold matrix-like data with row and column keys.

- **Creating MultiIndex:** The simplest way is via pandas factory methods:
- `pd.MultiIndex.from_arrays([arr1, arr2, ...], names=[...])` – combine parallel arrays of labels [35].
- `pd.MultiIndex.from_tuples([(...), (...)])` – from a list of index tuple labels [35].
- `pd.MultiIndex.from_product([iter1, iter2, ...])` – Cartesian product of iterables [36] [37].
- Setting an index on a DataFrame with multiple columns: `df.set_index(["col1","col2"])` directly gives a MultiIndex on rows.

You can also pass a list of tuples to a Series/DF index and pandas will infer a MultiIndex [38] [39]. For columns, you might get a MultiIndex after certain operations (like stacking, or groupby with multiple keys).

- **MultiIndex in use:** Suppose you have:

```
tuples = [("bar","one"),("bar","two"),("baz","one"),("baz","two")]
idx = pd.MultiIndex.from_tuples(tuples, names=["first","second"])
s = pd.Series(np.random.randn(4), index=idx)
```

Then `s` has two-level row index ("first" level and "second" level). This lets you store data like:

```
first  second
bar    one      1.23
       two      4.56
baz    one      ...
       two      ...
dtype: float64
```

You can then **slice** or **select** by partial keys: e.g. `s.loc["bar"]` returns a sub-Series with index `second` = {"one","two"} under "bar". `s.loc[("bar","two")]` returns a scalar. DataFrame `.loc` similarly accepts tuples for multi-indexed axes. Pandas also provides `IndexSlice` to make fancy slicing easier (e.g. `s.loc[pd.IndexSlice["bar":"foo", "two":], :]` for ranged slicing on each level).

- **Alignment and arithmetic:** Operations on MultiIndexed objects align on the full combination of index keys. If two Series have MultiIndexes, they will align by the tuple of keys (like a join on multi-column primary key). This means missing combinations become NaN. You can use `s.add(s2, fill_value=0)` to fill missing with 0 as usual.

- **Hierarchical indexing gotchas:** By default, partial indexing on the **outer level** is straightforward (`s.loc["bar"]`). For inner levels, you may need to pass `slice(None)` for outer level to select across all. Alternatively, use the `xs` method:

- `df.xs(key, level=...)` extracts a cross-section at a given level [40]. For example, `df.xs("one", level="second")` would select all rows where second level == "one", dropping that level [41]. You can even do `xs` on columns if they're multi-indexed (pass `axis=1` and a level) [42]. The `drop_level` argument controls whether the sliced-out level is removed from the result index [43].

- To iterate level-wise: `df.groupby(level="first")` treats all "bar" and "baz" groups together, etc. Or use `df.sum(level="first")` for a quick aggregation by one level.

- **Reshaping with MultiIndex:** MultiIndex enables powerful reshape operations:

- **Stack:** turn columns into an inner index level. `df.stack()` takes a DataFrame with (possibly) MultiIndex columns and stacks a level into the rows, returning a Series or DataFrame with a MultiIndex index [44] [45]. It's like pivoting **longer** (columns → rows). By default, it stacks the innermost column level. Each column value becomes associated with an extra index level in the result. If you stack a single-level column index, you get a Series with a two-level index (original index + former column labels) [46].

- **Unstack:** inverse of stack, pivot a row level out into columns [44] [47]. `df.unstack(level)` on a Series with MultiIndex (or DataFrame with MI index) will produce a DataFrame, with the specified index level turned into column labels. Unstacking increases dimensionality (e.g. Series → DataFrame). If the MultiIndex has N levels and you unstack one, the result will have N-1 index levels and one new column level. Example: if `s` has index levels ["first","second"] and you do `s.unstack("second")`, you'll get a DataFrame with index "first" and columns "second".

Stacking: The diagram (from pandas docs) illustrates `stack()`. The DataFrame with columns "A" and "B" is stacked into a Series with a two-level index ("row label" and original column name) [48]. This is useful to **melt** data or combine sparse columns.

Unstacking: Similarly, `unstack()` pivots the lower index level into new columns [44] [49]. Here a Series with MultiIndex (levels "first", "second") is unstacked on "second", creating a DataFrame where "second" values become column labels under A/B.

- **Pivot tables:** While `stack/unstack` work on index levels, `pivot_table` is a higher-level method to aggregate and reshape in one go (like Excel pivot tables). Example: `pd.pivot_table(df, values="X", index=["key1","key2"], columns="category", aggfunc="mean")` will group by key1&key2, aggregate X, and spread "category" values across columns. It produces a DataFrame with a MultiIndex index (key1/key2) and one level of columns = categories. The result is essentially `df.groupby(keys).agg({"X":"mean"}).unstack("category")`. You can add `margins=True` to get an "All" subtotal [50] [51]. Pivot tables handle missing combos by default (filling with NaN). For the inverse (long to wide without aggregation), use `df.pivot(index, columns, values)` if you know each index/column pair has at most one value.

In summary, MultiIndex unlocks multi-dimensional data manipulations in pandas. It's powerful for representing panel data, hierarchical indices (e.g. geographic regions within categories), or the result of groupby combinations. Just beware of the added complexity in indexing and the potential for `MultiIndex` to make code harder to read if overused.

---

4) Windowing, Rolling, and Expanding Transformations

Pandas supports a rich set of **window functions** for rolling calculations, expanding accumulations, and exponentially-weighted moving averages. These are critical for time series analysis (moving averages, rolling sums, etc.) and other contexts where you need to aggregate over sliding windows of data.

Window API basics: You call a **window method** on a Series/DataFrame, which returns a special Window object, then you call an aggregation function. For example, `df['x'].rolling(7).mean()` computes a 7-point moving average. The API mimics groupby: you do `obj.rolling(...).agg()` or `.sum()`, etc. Key window types: - **Rolling window:** fixed-size window sliding over values (based on count or time span). - **Expanding window:** grows from the start of the data up to each point (cumulative). - **EWM (Exponentially Weighted):** expanding window with exponentially decaying weights (a.k.a. EWMA/EMA).

Basic rolling example:

```
s = pd.Series(range(5))
s.rolling(window=2).sum()
# 0 NaN; 1 1.0; 2 3.0; 3 5.0; 4 7.0  (sum of each 2-length window) 52 53
```

By default, a window of size N produces the first N-1 results as NaN (since there isn't a full window). You can adjust `min_periods` to allow earlier results with partial windows 54 55 . For example, `s.rolling(2, min_periods=1).sum()` would yield `[0,1,3,5,7]` (treat single values as window until two are available).

Time-based rolling: Instead of a count, you can specify a time offset like `"7D"` or `"90s"` for the window, if the index is datetime or you use the `on=` parameter with a datetime column. This creates variable-length windows covering that duration. It's crucial for irregular time-series data where 7 observations ≠ 7 days.

- If your Series/DataFrame index is a `DatetimeIndex`, you can do `series.rolling('30T').mean()` for a 30-minute window, for example 56 . Pandas will include all points within the last 30 minutes for each timestamp.
- If your datetime is in a column (not index), use `df.rolling('30T', on='timestamp_column')[...]`. This usage was introduced to avoid resetting the index. For example:

  ```
  df["rev_7d"] = df.rolling("7D", on="date")["revenue"].sum()
  ```

  will compute a 7-day rolling sum on the "revenue" column, treating gaps in "date" properly 57 58 .

- By default, the window is *inclusive of the current observation* and looks backward. You can adjust the window's **closed-end** using `closed='right'` or `'left'` to control whether the interval is inclusive/exclusive on its endpoints [59] (useful for edge alignment).

Rolling + grouping: You can combine groupby with rolling or expanding to do window calcs *within each group*. This is an advanced but powerful pattern:

```
df.groupby("id")["value"].rolling(3).mean()
```

This won't immediately return a DataFrame; it's a "grouped rolling" object. If you call `.mean()`, you'll get a Series with a MultiIndex: the first level is the group key, second level the original index [60] [61]. This effectively computes a 3-point moving average *per id*, independently. Similarly, `df.groupby("id").expanding().sum()` would give cumulative sum per id [61]. (The result index is multi-level: id and the index within group.)

Expanding: Use `.expanding(min_periods=…)` for cumulative window from start. E.g. `df['cum_avg'] = df['metric'].expanding().mean()` gives a running average. This is equivalent to `df['metric'].rolling(len(df), min_periods=1).mean()`, but more efficient and semantically clearer. You can group expand too (e.g. `df.groupby('key').expanding().count()` to count observations per group over time).

EWM (Exponential Weighted Mean): Use `.ewm(span=..., halflife=..., alpha=..., adjust=...)`. This does not produce NaNs at start (by default it treats the first value as starting point or uses `min_periods`). `halflife` is a convenient way to specify the decay: e.g. `halflife=3` means weight drops by ~50% every 3 observations. `adjust=False` yields a recursive formula (good for streaming) [62], whereas `adjust=True` (default) normalizes weights to actual EWMA definition. EWM can also operate on **time gaps**: you can supply a `times=` parameter (name of a datetime column) so that the weighting accounts for uneven spacing [62]. For example, `df.ewm(alpha=0.5, times='timestamp').mean()` will apply half-life based on actual time differences.

Custom window functions: If built-in reductions (`mean(), sum(), max(), std(), etc.`) don't suffice, use `.apply(func, raw=False)`. For example, rolling range:

```
df['range_4'] = df['value'].rolling(4).apply(lambda x: x.max()-x.min(),
raw=True)
```

Setting `raw=True` passes a NumPy array to the function for speed [63]. Note that custom Python functions will be slower than pandas' Cythonized ones, and `raw=True` only accepts functions that operate on a NumPy array and return a single number.

- *Tip:* If your rolling function can be expressed as a composition of pandas ops, try to avoid `.apply` for performance. E.g. a rolling z-score can be done as `(x - x.rolling(10).mean()) / x.rolling(10).std()` rather than using `.apply`.

Window parameters and performance: - `min_periods`: minimum non-NA count in window to compute a value (otherwise result is NA). Defaults to window size for fixed windows (meaning skip until window filled) or 1 for time windows [64] [65]. Set `min_periods=1` to get values from the first point onward. - `center=True`: label results at the window center (by shifting the output index). - `win_type`: specify a window function (from SciPy) for weighted rolling (e.g. Gaussian). - Many window ops gained a `method='table'` option (since pandas 1.3) to compute entire DataFrame windows in one go for performance. This requires `engine='numba'` and allows operations using all columns at once [66] [67]. It's an advanced feature: for example, a weighted rolling mean using two columns as weight and value could be done via `.rolling(window, method='table').apply(...)` on the 2D array.

Online/streaming calculation: New in pandas 1.3, certain window objects have an `.online()` method to continue a window calc with new data. For example, `online_ewm = df.head(100).ewm(alpha=0.5).online(); online_ewm.mean(); online_ewm.mean(update=new_data)` will update the EWM with additional points without recomputing from scratch [68] [69]. This is niche but useful for iterative processing on streaming data.

Rolling vs Resampling: If your data is time-series and you want to normalize to a regular frequency before rolling (especially if data is irregular), consider using `.resample()` to a fixed frequency with an appropriate fill or interpolation, then rolling. Alternatively, use time-based rolling directly with `on=` or a datetime index as mentioned.

Summary: Pandas window functions let you replace explicit Python loops for common patterns like "moving average over last N observations or last T time units" and "percent change over window" or "cumulative sum". These are optimized in C/NumPy and often use multiple CPU cores (especially true if `numexpr` is used for simple aggregations). They integrate with groupby and time-series naturally, making pandas a strong tool for rolling statistics. Always prefer vectorized window ops over manual looping for performance and clarity.

---

5) Vectorized String, Date, and Categorical Operations

Beyond basic arithmetic, pandas provides vectorized operations tailored to specific data types, leveraging the accessors and dtype-specific methods.

String data (text): As mentioned, the `.str` accessor gives a multitude of vectorized string operations: - **Pattern matching:** `s.str.contains("foo", case=False, regex=True)` returns a boolean Series (supports regex) [70]. Similarly `str.match()`, `str.fullmatch()`, `str.startswith/endswith` etc. - **Substitution:** `s.str.replace(pat, repl)` can handle regex patterns and even use backrefs. There's also `str.translate` for char maps. - **Extraction:** `s.str.extract(r'(...)', expand=False)` to capture regex groups (returns DataFrame of captures), or `str.get(i)` to get i-th element from each (for lists in strings). - **Transformation:** `s.str.upper()`, `s.str.lower()`, `s.str.title()`, `s.str.strip()` (whitespace removal), `s.str.pad()`, `s.str.center()` etc. All apply elementwise. - **Misc:** Length (`s.str.len()`), find index of substring (`s.str.find('x')`), count occurrences (`s.str.count('sub')`), splitting (`s.str.split(',')` which yields a list Series, or `str.split(expand=True)` to get a DataFrame of splits), and joining (`s.str.join('-')` if each

element is a list of strings). Also `s.str.get_dummies()` to create dummy/indicator columns from a delimited string.

These string methods are quite efficient. Under the hood, operations may use compiled C regex (via `re` or `regex` library) or loop in C. That said, certain complex regex on large text can still be slow (in which case consider vectorized `re` via `df.apply` with `regex` library or even port to a faster text tool). Pandas 2.x introduced an optional **Arrow string dtype** which can speed up some operations and use less memory; with `string[pyarrow]`, many `.str` methods dispatch to Arrow's highly optimized UTF8 routines (e.g. `contains`, `startswith`) and can be significantly faster for large data [71] [72].

Datetime data: The `.dt` accessor consolidates many Python datetime attributes and more: - **Properties:** Year, month, day, hour, minute, second, week/weekday, dayofyear, quarter, etc. Example: `df['ts'].dt.dayofweek` yields 0–6. - **Round/floor/ceil:** `df['ts'].dt.floor('H')` floors each timestamp to hour boundary (similarly `.ceil` and `.round` for rounding to nearest freq). - **Time deltas:** If you have a `Timedelta` Series, `.dt` gives properties like `.days`, `.seconds`, and you can also floor/ceil on timedeltas. - **Time zone handling:** With tz-aware datetimes, use `dt.tz_convert(tz)` to change timezone (must already have tz), or `dt.tz_localize(tz)` to attach a timezone to naïve times. These are vectorized. If times are ambiguous or nonexistent due to DST transitions, you **must** specify a strategy (otherwise `tz_localize` will raise). For example, `series.dt.tz_localize("Europe/Paris", ambiguous='NaT', nonexistent='shift_forward')` might be one approach. Options: `ambiguous='infer'` tries to guess based on order (for fall DST) [73] [74], or provide a boolean array marking which timestamps are DST. For spring-forward gaps, `nonexistent='shift_forward'` or `'shift_backward'` will move times into the valid range, or `'NaT'` to mark as missing [75] [76]. These parameters give fine-grained control to handle tricky timezone adjustments. - **Conversion:** `dt.to_pydatetime()` yields Python `datetime` objects (beware: heavy on memory), `dt.to_period(freq)` converts timestamps to Period (see below), and `dt.to_timezone` (alias for tz_convert). - **Components:** `dt.date` gives Python date objects for each, `dt.time` for time objects, `dt.timetz` (time with tz). - **Intervals:** If dealing with Period or Interval data, pandas has PeriodIndex and IntervalIndex with their own accessors (e.g. `.to_timestamp()` on a PeriodIndex converts to actual Timestamp).

Period vs Timestamp: A **Timestamp** is a specific point in time (usually stored as nanoseconds since epoch). A **Period** is a time span (e.g. May 2023, or Q4 2025). Periods are useful for calendar algebra (adding 1 to a period moves to next period, etc.). Pandas supports `PeriodIndex` and `PeriodDtype`. You can obtain periods by using `pd.period_range` or converting timestamps: e.g. `df['month'] = df['date'].dt.to_period('M')`. This yields a Series of type `period[M]`. Periods have a `.to_timestamp()` method (often picking start or end of the period) to go back to exact times. Grouping by period can be more intuitive for certain analyses (e.g. all data by year/month regardless of exact day). Under the hood, periods are stored as int64 with metadata of frequency. They can simplify date arithmetic with non-day frequencies (like adding 1 month even if month lengths differ).

Categorical data: We touched on `.cat` methods for renaming and reordering categories. Some additional notes: - Categorical comparisons and groupings use the **codes** (integer labels) internally, which makes them very fast and memory-efficient. If you have repetitive string values (like categories), converting a column to `astype('category')` can massively reduce memory and speed up `groupby` and `merge` because comparisons become integer operations. - You can specify an order for categories (ordered=True).

Then operations like min/max use that order, and you can sort according to the categorical order (not lexicographically). - **Codes access:** `cat.codes` property gives the integer code for each element (0-based, with `-1` for NA). You can use this for numerical operations or to feed into algorithms, then map back to categories via the `.categories` list. - **Efficiency:** If a categorical has many categories relative to data length, the benefits diminish. But if you have e.g. 1e6 values with only 20 unique categories, `category` dtype is ideal. Conversions: `Series.astype('category')` infers categories from data (may sort them unless you set `ordered=False` and give a specific list). Use `CategoricalDtype(categories=[...], ordered=...)` if you need control.

Deferred execution vs eager: Generally, pandas operations are eager (happen immediately). The phrase "deferred methods" in the context of vectorized ops might refer to how some operations on certain extension types defer to an underlying library. For instance, with `string[pyarrow]` dtype, `s.str.contains("a")` offloads the work to the Arrow library, which might not execute Python bytecode for each element but does the work in a native routine [77] [78] . From a user perspective, it's still "eager" (the result is computed immediately), but the computation happens at C/C++ level, often in parallel.

In summary, **use these accessors and dtype-specific methods instead of Python loops**. They are more readable and leverage optimized code. Whether it's splitting a column of strings on a delimiter, rounding hundreds of timestamps to the nearest hour, or adding new categories to an enum, the vectorized approach will be shorter and orders of magnitude faster.

---

6) Expression Evaluation: `eval()` and `query()`

Pandas offers an **expression evaluation** system to compute or query data using string expressions, potentially speeding up large computations by avoiding intermediate objects and using optimized engines (like NumExpr). These come in two flavors: - `DataFrame.eval(expr)` for computing new columns or scalar expressions involving columns. - `DataFrame.query(expr)` for filtering the DataFrame based on a boolean expression.

DataFrame.eval: This lets you write formulas referring to column names directly, similar to R's dataframe syntax. For example:

```
df.eval("total = quantity * price")
df.eval("new_col = col1 + col2 - col3 / 2", inplace=False)
```

In the expression, you refer to `col1`, `col2` as variables instead of using `df['col1']`. The evaluation happens all at once internally. By default, `df.eval(...)` returns a new DataFrame (it won't modify original unless you pass `inplace=True`, which is supported). In fact, assigning to a column in an eval expression (as in the examples) is allowed; it will assign on a copy and return it (leaving original untouched unless inplace) [79] [80] . This is a **side-effect-free** approach: you can chain multiple evals and capture the result. For example:

```
df2 = df.eval("c = a + b").eval("d = c / 100")
```

Original `df` remains unchanged [81] [82].

- **Multiple assignments:** You can assign multiple columns in one `eval` by separating with semicolons or line breaks. E.g.

```
df.eval("""
  c = a + b
  d = c * 2
""")
```

would create `c` and `d` in one pass [83] [84].

- **Speed:** If the expression is numeric and NumExpr is installed, `eval` will try to use the **'numexpr' engine** which computes the whole expression in chunks (to limit memory usage) and can use multiple cores [85] [86]. This can be much faster than doing the equivalent Python operations column by column, especially for large arrays. If NumExpr isn't available or the expression is too complex, pandas can fall back to the **'python' engine**, which basically does `eval()` in pure Python (no speed gain) [85] [87]. You can choose engine and parser via params: `df.eval(expr, engine='numexpr', parser='pandas')` (the default parser 'pandas' supports some pandas-specific syntax like `and/or` and benefit from pandas name resolution).

- **Allowed expressions:** You can use any column names (even ones that are Python keywords by backtick quoting them), arithmetic operators `+ - * / ** // %`, bitwise ops `| & ^` (for booleans), comparisons, and math functions via `@` prefix for calling Python functions. There is also a notion of "speedups" for certain ops (like chained comparisons). **Note:** `eval` does not support assignment to Python variables (only to DataFrame columns), and it doesn't magically avoid Python for anything not supported by NumExpr (e.g. string operations or method calls will likely use the Python engine).

- **Example:** Instead of

```
df['speed'] = df['dist'] / df['time']
df['energy'] = 0.5 * df['mass'] * (df['speed']**2)
```

you could do

```
df = df.eval("speed = dist / time; energy = 0.5 * mass * speed**2")
```

to get both columns in one go. This avoids creating the intermediate `speed` Python-level before computing energy, and leverages numexpr to evaluate `0.5 * mass * (dist/time)**2` efficiently.

DataFrame.query: This provides SQL-like row filtering using boolean expressions:

```
res = df.query("quantity > 5 and origin == 'JFK'")
```

This returns a filtered DataFrame, equivalent to `df[(df['quantity']>5) & (df['origin']=='JFK')]`. The advantages are conciseness and potential speed-up for large frames.

- **Syntax:** Inside `query()`, you refer to columns by name (as variables). You can use `and/or/not` (which pandas internally maps to `& | ~` properly, with correct precedence [88] [89]), and comparison ops. String literals should be in quotes as usual. If your column name has spaces or isn't a valid Python var, wrap it in backticks, e.g. `df.query("`Flight Number` == 1001")`.

- **Using variables:** To use external Python variables in the expression, prefix them with `@`. For example:

  ```
  cutoff = 100
  df.query("colA < @cutoff or colB == @user_id")
  ```

  will substitute the Python variable values. This is safer than string formatting the query, and it's efficient (pandas will map the @variable into a placeholder that the engine can handle).

- **Engine:** Like eval, query can use numexpr. It will by default for large data if possible. You usually don't need to specify `engine='numexpr'` because pandas will choose it if available. One caveat: numexpr has some reserved names (like `sin`, `max`) and may fail if your column names conflict (e.g. a column named "max"). In such cases, you can force `engine="python"` or rename the column.

- **Return value:** `query` returns a DataFrame view or copy depending on setting. As of pandas 1.4+, it tries to return a **view** if no copy is needed (similar to `.loc` filtering), but you shouldn't rely on that for mutation. Just treat the result as a new frame (and note that Copy-on-Write in future will make it so that even if it's a view, writing to it won't affect original).

- **Security:** Be careful using `query` or `eval` on strings that come from an untrusted source (e.g. user input in a web app), as it can execute arbitrary code if not sanitized. Pandas doesn't sandbox the expression – it's executed with `numexpr` or Python `eval`. You can mitigate by controlling allowed names or using the `parser='pandas'` which is more strict than Python's.

Under the hood: When using the pandas expression parser, it will **compile** the expression into an abstract syntax tree and then either evaluate it via numexpr (which vectorizes it) or via Python. The speed gains come from two things: avoiding large temporary arrays (e.g. computing `df.A + df.B + df.C` in one pass instead of making a temporary for A+B then adding C) and leveraging multi-threaded, CPU-optimized operations in `numexpr`. The threshold for when it's beneficial is usually when your data has at least a few million elements; for small DataFrames, the overhead of parsing may outweigh any gain.

When to use: `eval` is great for complex expressions especially if they reuse sub-expressions, and when you want to keep your code compact. `query` is great for readability of filters. However, in many cases the speed difference might not be huge if you're doing simple things on moderately sized data. Always favor clarity first; use these tools when they make code cleaner or when you've identified a performance bottleneck on a large DataFrame.

---

7) Function Application Patterns: `agg`, `transform`, `apply`, `pipe`, `applymap`

Pandas provides several ways to apply custom functions to your data, each with different purposes and shapes of output. Understanding which to use is crucial for writing clear, efficient code:

- **Aggregate (`.agg` or `.aggregate`):** Used to reduce values *within a group* (or entire axis) to a single (or a few) values. Commonly used after `groupby` or on a DataFrame/Series directly. You can pass a function (or list/dict of functions) to `.agg`. For example:

  ```
  df.groupby("category").agg({"value": "sum", "price": "mean"})
  ```

  will produce one row per category with the sum of value and mean of price. The function can be a string name of a NumPy/pandas aggregation (`"sum"`, `"mean"`, etc.), or a callable (like a `lambda x: ...` that returns a scalar), or a list of such functions (to produce multiple output columns). Aggregation always yields **reduced dimensions** – e.g. a Series -> scalar or DataFrame -> Series per group. In DataFrame.agg, if you provide a list of functions, you'll get a MultiIndex column (function names level) by default, unless you provide custom names.

You can also aggregate without grouping: `df.agg({"col1": ["min","max"], "col2": ["mean"]})` will compute those stats on the whole DataFrame for specified columns.

- **Transform (`.transform`):** This is for **broadcasting a computation back to the shape of the original data**. It is primarily used with `groupby`: it must return an output of the *same length* as the input chunk. For example:

  ```
  df['val_ratio'] = df.groupby('category')['value'].transform(lambda x: x / x.sum())
  ```

  Here the lambda computes the sum per group, and transform broadcasts the result to each row in the group, so `val_ratio` is each row's value divided by the total of its category. The result has the same number of rows as original. `transform` is limited: the function can't change the length or order of data, it should operate column-by-column. If you pass a built-in reduction to transform (like `"cumsum"`), pandas optimizes it [90] [91].

In short, use transform when you need an **element-wise group operation** (e.g. fill missing with group mean, rank within group, standardize within group). If the transform function returns a scalar per group, pandas will broadcast it (e.g. group mean -> that mean for every row in group).

- **Apply (`.apply`):** This is the most flexible but often the slowest. It allows you to take each group (as a DataFrame or Series) and return a transformed object. For DataFrameGroupBy, your function receives a DataFrame for each group and can return either a DataFrame, Series, or scalar. Pandas will then try to combine these into an appropriate output:
- If each group's function returns a DataFrame, pandas concatenates them (keys become a new index level by default unless you use `group_keys=False` in groupby).
- If each returns a Series, they get concatenated into a DataFrame (with group keys as index).
- If each returns a scalar, they get combined into a Series indexed by group key (similar to agg).

Because `.apply` can do anything, pandas can't optimize it as well. It's basically a Python loop over groups. **Only use apply if neither agg nor transform can do what you need.** Many things that people historically did with apply (like custom aggregations) can be done with agg by passing a named lambda or using `.pipe` (see below).

Example: Suppose you want for each group, the three largest values of column X. You could do:

```
top3 = df.groupby('group')['X'].apply(lambda s: s.nlargest(3))
```

This returns a Series with a MultiIndex (group, original index) containing the top 3 values per group. This is something transform can't handle (transform must return same length as input), and agg can't either (agg gives one row per group). Apply is appropriate here. But note, `s.nlargest(3)` is a built-in method that is already optimized in C for Series, so this combination is okay. If your lambda did a pure Python loop, it'd be slow.

Performance note: The pandas docs explicitly warn that using apply with a Python function can be **orders of magnitude slower** than using vectorized operations or built-in aggregations [92] [93]. Always check if a canned method exists (e.g. `groupby().rank()` instead of apply + rank, etc.). Also, in many cases you can rewrite an apply in terms of a merge or boolean indexing. For instance, the above "top3 per group" could also be done by sorting df by X within group and taking head(3) per group (which can be achieved via `groupby().head(3)` in recent pandas, which is much faster than apply in C).

- **Applymap (`.applymap` on DataFrame, not to be confused with apply on groupby):** This is an element-wise operation for DataFrames (similar to a double loop). `df.applymap(func)` applies the function to every single value in the DataFrame (non-NA). It's like doing `df.replace(v, func(v))` for each cell. This is rarely needed because it's essentially Python-level looping. If you find yourself wanting to do this, consider if you can use vectorized NumPy ops or a ufunc across the whole DataFrame (possibly by casting to object numpy array, but that's also often slow). Applymap might be acceptable for small DataFrames or for formatting outputs (e.g. `df.applymap("{:.2f}".format)` to format all cells as strings with 2 decimals).

- **DataFrame.apply:** When not using groupby, `df.apply(func, axis=...)` can apply a function along an axis:

- `axis=0` (default) will pass each column as a Series to `func` (so it applies column-wise).
- `axis=1` will pass each row as a Series.
  This is basically a shortcut to do something like: *for each column, do X* or *for each row, do Y*. For column-wise operations, prefer using vectorized methods if possible (or DataFrame methods that operate on multiple columns). For row-wise, because each row is a Series (with mixed dtypes potentially), a Python loop is hard to avoid. If you need to compute a complex function of multiple columns for each row, `df.apply(axis=1)` is often the simplest expression (though not the fastest). Alternatively, one might vectorize by breaking the formula into vector operations using NumPy if possible.

Example: `df.apply(lambda col: col.dtype, axis=0)` would produce a Series of dtypes for each column (but you could also just do `df.dtypes`). Row example: `df.apply(lambda row: row['a']*row['b'] - row['c'], axis=1)` calculates a custom formula per row (but this could be done as `df['a']*df['b'] - df['c']` without apply, which is better).

Keep in mind, `df.apply` with axis=1 constructs a Series for each row (index as column names), which is quite slow for large frames (lots of small Series objects). Whenever possible, express your row operation in a vectorized way.

- **pipe (`.pipe`):** This isn't about applying an elementwise function but rather about improving method chaining and functional composition. `df.pipe(func, *args, **kwargs)` will call `func(df, *args, **kwargs)`. This allows usage like:

  ```
  (df.pipe(func1)
     .pipe(func2, arg=5)
     .pipe(lambda d: d.assign(newcol = d.colA - d.colB))
     .pipe(func3)
  )
  ```

  so that you can build a pipeline of transformations where each step gets the DataFrame and returns a new DataFrame. It's especially useful when you want to use functions that aren't DataFrame/Series methods in a chained expression. `pipe` doesn't provide performance benefits; it's for cleaner code. It can also make user-defined functions easier to incorporate: if you have a function that expects a DataFrame and returns one, you can insert it in a chain with `.pipe(my_func, extra_param=...)`. Another trick: with pipe, you can swap argument order using lambda if needed (since pipe always passes the object as first arg).

Using these in groupby or windowed contexts: Note that `GroupBy` objects themselves have `.agg`, `.transform`, `.apply`, and even `.pipe`. GroupBy.agg and GroupBy.transform correspond to applying those functions per group and then combining results. E.g. `df.groupby('key').agg(sum)` returns one row per key, whereas `transform(sum)` would give a Series of same length as df with group sum repeated. GroupBy.apply works as described above for arbitrary group-wise operations, but it's slow if used in Python. Similarly, `Rolling` and `Expanding` objects have an `.apply` method for custom window functions, as we saw. There is no `.transform` for rolling (not needed since rolling itself already keeps output aligned), but there is a `.agg` which can compute multiple window stats in one go.

Pitfalls: - Avoid using `apply` when an existing method can do it. Common newbie pattern: `df.apply(np.sum, axis=0)` – this is just `df.sum()`. Or `df.apply(lambda x: x.max() - x.min())` – use `df.max()-df.min()` or if per column, maybe `df.max(axis=0) - df.min(axis=0)`. - If you use DataFrame.apply with axis=1 and inside the lambda you perform Series lookups (like row['col'] each time), realize that's Python overhead per cell. If possible, break the computation into column arrays. For example, instead of `df.apply(lambda r: expensive_func(r['x'], r['y']), axis=1)`, try to vectorize `expensive_func` to accept arrays `df['x']` and `df['y']` at once (this might require writing `expensive_func` to handle arrays, or using NumPy if it's mathematical). - When aggregating multiple functions on multiple columns, consider the **DataFrameGroupBy.agg** with a dict of column -> func or list of funcs, rather than separate groupby calls or apply. It's concise and optimized in pandas (it'll compute in one pass if possible).

In conclusion, **choose the right tool**: use `agg` for reducing results, `transform` for same-length outputs (especially after groupby for adding back columns), and `apply` only for weird cases not covered by those. Use `pipe` to keep code tidy and to integrate custom functions smoothly into method chains. And remember: the more you can rely on pandas' internal vectorization (rather than Python loops in apply), the better your code will perform.

---

8) Performance Tuning: Memory Use, Sparse Data, and Copy-on-Write

For large datasets, pandas can become memory-bound or slow if not used carefully. Here are advanced tips to diagnose and improve performance:

Memory usage diagnostics: Use `df.memory_usage(deep=True)` to see memory used by each column (in bytes) [94]. By default, `memory_usage()` doesn't count object-dtype element sizes, but `deep=True` does a full scan. You can also do `df.info(memory_usage="deep")` to get total memory including objects. This helps identify big columns (especially object dtype text which may dominate RAM). If you see a column of long strings taking 500 MB, you might consider converting it to categorical or a more efficient dtype.

`convert_dtypes()`**:** This DataFrame method attempts to "upgrade" the dtypes of each column to better (often pandas extension) types. For example, integer columns with NAs (which would be `object` or `float64`) can become `Int64` nullable integers; object columns that are all numbers could become Int/Float; object booleans to BooleanDtype, etc. It will also use pandas StringDtype for text. In pandas 2.0+, `convert_dtypes(dtype_backend="pyarrow")` will even convert to Arrow-backed dtypes [95] [96]. Running this on a DataFrame can drastically reduce memory and improve performance if many columns were generic types. It's an easy first step after reading data, to ensure you're using efficient types (e.g. get away from Python objects).

Categorical compression: As discussed, convert categorical-like data to `category`. Example: a DataFrame of 1 million rows with a column of 5 possible strings – as Python strings this might be ~50 million bytes; as a categorical, maybe ~1 million bytes + a tiny amount for 5 category strings (which might be interned). That's a huge savings. Categorical operations (comparisons, groupby, joins) are also faster on

many repeated values because they work on the integer codes. The trade-off is a little overhead in maintaining the categories and the need to handle the categories if you append new data (categories won't automatically expand unless you use `unsafe` settings or update categories explicitly).

Sparse data structures: If you have a DataFrame mostly full of zeros or NaNs (common in one-hot encoding or certain signal processing), consider using sparse dtypes. Pandas supports `pd.SparseDtype(base_dtype, fill_value)` for Series. You can convert a dense column to sparse by `series.astype(pd.SparseDtype(np.float64, fill_value=0))` for example. Under the hood, a SparseSeries stores only the non-fill-value entries and their index locations. This can save a lot of memory if, say, 99% of entries are zero. There's also `pd.DataFrame.sparse.from_spmatrix(scipy_matrix)` to directly get a sparse DataFrame from a SciPy sparse matrix. Operating with sparse data has some limitations (not all methods preserve sparsity, some ops might materialize data). But basic math and selection generally preserve the sparse structure. Use `df.sparse.to_coo()` or `.to_dense()` to convert if needed [97] [98]. Sparse data in pandas is mostly beneficial for memory; computations might still be a bit slower than on dense (due to overhead), unless you go to SciPy.

Arrow-backed arrays: Pandas 2.0 introduced the option to use **pyarrow** for column storage. This can improve memory usage (Arrow uses binary columnar format, and can be zero-copy shared between processes or libraries) and sometimes speed (especially for string data, as Arrow has efficient string ops). You can request Arrow dtypes by, for instance, `pd.read_csv(..., dtype_backend="pyarrow")` [95] [96]. This will make all suitable columns Arrow types (e.g. int64→ `int64[pyarrow]`, string→ `string[pyarrow]`). You can also convert an existing DataFrame via `df.convert_dtypes(dtype_backend="pyarrow")` [96]. Under Arrow backing, certain operations may use Arrow's C++ implementation. Example: a `string[pyarrow]` Series uses Arrow's UTF8 storage; calling `Series.str.lower()` on it will use Arrow's compute kernel (potentially faster). Using Arrow can also make passing data to **PyArrow**, **Polars**, **Spark**, etc. instantaneous (zero-copy), since they all understand the Arrow memory format.

- One caveat: Not all pandas operations fully support Arrow dtypes yet. If an operation isn't implemented, pandas may convert to numpy under the hood, so test your use-case. But basic ops, sorting, filtering, and many string and datetime methods are supported. The benefit is especially seen in large string columns – Arrow strings use much less overhead per value compared to Python strings and can be processed in parallel in C++.

Copy-on-Write (CoW): Pandas historically had the quirk that slicing a DataFrame sometimes gave a view and sometimes a copy, which could lead to unexpected "silent" propagation of changes (or the infamous SettingWithCopy warnings). The new Copy-on-Write mode (opt-in in 2.x, on by default in 3.0) changes this behavior: **views are not exposed**. Slicing returns a view that is marked "read-only until mutated", and if you try to mutate it, it will automatically copy to avoid affecting the original. CoW thus guarantees that you never accidentally write to the original when you think you are writing to a subset [3] [4]. This gives more predictable behavior at the cost of a bit of overhead on writes.

- To enable CoW now, do `pd.options.mode.copy_on_write = True`. With it on, the warning about chained assignment becomes an error (since chained assignment always works on a copy, it just outright fails if you try to set with two indexes). For example:

```
pd.options.mode.copy_on_write = True
df = pd.DataFrame({"A":[1,2,3], "B":[4,5,6]})
subset = df["A"]         # previously this was a view of df's series
subset.iloc[0] = 100     # with CoW, this writes to subset only
df.iloc[0,0]            # remains 1, not 100 [99] [100]
```

CoW prevented modifying `df` via the subset. In pandas 3.0, this will be standard behavior.

- **Performance and memory with CoW:** A side benefit is that operations that created views can be faster (no immediate copy), and only incur a copy if a write happens. This can save memory in workflows that slice data without modifying it. However, if you do many small edits on slices, CoW might copy repeatedly whereas the old behavior might have inadvertently edited in place. Overall, CoW is aimed at correctness and easier reasoning. For performance, note that many pandas methods have been internally optimized with CoW in mind by returning views when possible (since it's "safe" now). The release notes for 2.1 list many DataFrame/Series methods that now return views under CoW, improving speed and memory [101] [102] (for example, selecting a single column now doesn't copy data if CoW is on, whereas before it often returned a view anyway but with uncertainty).

Other tips: - **Use NumPy vectorization when appropriate:** If you need to do a custom numeric operation, writing a NumPy ufunc or using existing ones can be faster than apply. You can also use `np.vectorize` or Numba for certain tasks, but often pandas built-ins cover a lot. - **Avoid Python object dtype when possible:** If you have mixed types, see if you can separate them or use ExtensionDtypes. Python objects kill vectorization. For example, if you have a column of `datetime.datetime` objects, convert it to pandas `datetime64[ns]` dtype – it will be much faster to filter, etc. - **Chunk large file processing:** When reading a gigantic CSV, use `chunksize=` to iterate in pieces rather than reading all into memory. For example:

```
iter_tbl = pd.read_csv("huge.csv", chunksize=100000)
for chunk in iter_tbl:
    process(chunk)
```

This way, only 100k rows are in memory at once (plus whatever processed output you keep). Many pandas I/O functions support iteration via chunks (CSV, SQL queries, etc.).

- **Scaling beyond memory:** If pandas isn't enough (data too large for RAM or needing distributed computing), consider tools like Dask, or out-of-core techniques. But that's beyond this scope – just note that efficient use of in-memory pandas (as above) can often handle surprisingly large data on a single machine if you optimize types and avoid Python-level loops.

In summary, to tune performance: **reduce memory footprint** (use appropriate dtypes, categorical, sparse, arrow), **enable CoW** for safer slicing (and possibly performance gains in slicing-heavy workflows), and **vectorize your computations** as much as possible. Measure using tools like `df.memory_usage()` and Python's profiling to find bottlenecks.

---

9) Time Series Features: Custom Frequencies, Offsets, and Time Zone Handling

Pandas was originally built with time series in mind, and it includes a wealth of functionality for dates, times, and frequencies:

Date offsets and frequencies: Pandas uses a system of frequency strings and **DateOffset** objects to represent calendar-aligned intervals (business days, month ends, etc.). - A **frequency string** like `"D"`, `"M"`, `"1H30T"` (90 minutes), `"Q"` (quarter end), `"W-WED"` (weekly on Wednesday) maps to a `DateOffset` subclass internally [103] [104]. For example, `"M"` is MonthEnd, `"MS"` is MonthStart, `"B"` is BusinessDay, etc. - **DateOffset objects:** e.g. `pd.DateOffset(days=3)` or `pd.offsets.BDay(2)` (2 business days). These can be added to timestamps or used in `date_range`. Offsets are smart about calendar rules (BusinessDay will skip weekends, etc.). You can also combine offsets: adding 1 DayOffset and 2 HourOffset yields a single offset of 26 hours. - Pandas defines many offsets: Week, BusinessWeek, MonthEnd, MonthBegin, BMonthEnd (business month end), QuarterEnd, YearEnd, *custom* business days (you can define a Holiday calendar and use `CustomBusinessDay(calendar=my_cal)`), and more. See `pd.offsets` module for list. - Using offsets: If `ts` is a Timestamp, `ts + MonthEnd(1)` will roll it to month end (if ts is Jan 10, +MonthEnd gives Jan 31). If `ts` is already a month end, MonthEnd usually rolls to *next* month end unless used carefully. Offsets have `rollforward()` / `rollback()` methods to adjust a date to the next/previous valid date for that offset [105]. - **Holiday/Custom Calendar:** Pandas allows constructing a `CustomBusinessDay` or `CustomBusinessMonth` with your own holidays. This involves making a `pd.tseries.holiday.HolidayCalendar` and using `pd.offsets.CustomBusinessDay(calendar=cal)`. This is advanced but useful if you need country-specific business days, etc. The frequency string for custom business day is "C" (stands for "custom business day") [106], but the actual holidays are attached via the calendar object.

Generating ranges: `pd.date_range(start, end, freq='...')` is the workhorse to create a DatetimeIndex. It accepts those freq strings (including combos like "2H" for every 2 hours). Similarly `pd.period_range` for PeriodIndex. - If you specify `freq='B'` (business daily), it will exclude weekends. `freq='W-MON'` means weekly, ending on Monday of each week. - `pd.timedelta_range` exists for TimedeltaIndex generation.

Resampling: This is the time-series equivalent of groupby for fixed frequency conversion. `df.resample('M').agg({...})` will group data into monthly bins (according to the DatetimeIndex or specified `on=` column) and aggregate. Commonly, `series.resample('D').mean()` or `.sum()` to upsample or downsample. Resample has its own syntax like `.resample('15T').ffill()` for forward-filling to a higher freq. Resampling is great for interpolation, frequency normalization, etc.

Time zone handling: Pandas datetime (`Timestamp`) can be tz-aware. Key points: - When you localize a naive timestamp to a timezone (e.g. convert naive times to UTC or US/Eastern), use `tz_localize`. This does not change the numeric time, just tags it with a zone. If your times were in local time originally, you might localize to that local zone. If you want to go from naive (implicitly e.g. US/Eastern) to UTC, you'd do `tz_localize('US/Eastern').tz_convert('UTC')`. - **Ambiguous times (fall back DST):** as discussed, e.g. 1:30 AM might occur twice. Pandas requires you to handle this with `ambiguous` param [73] [74]. `'infer'` works if your data is a sequence covering the DST transition (and it will infer by order which is

DST). If you have a single ambiguous timestamp or out-of-order data, `'infer'` may not work – then you might supply a boolean array of same length marking which ones are DST (True) vs standard (False) [107]. `'NaT'` will mark ambiguous times as Not-a-Time so you know they were problematic. - **Nonexistent times (spring forward gap):** e.g. in many zones the clock jumps from 1:59 to 3:00, so 2:30 doesn't exist on that day. If you attempt to localize such a time, pandas again asks what to do via `nonexistent` param [75] [76]. Options: `'shift_forward'` (e.g. 2:30 becomes 3:00), `'shift_backward'` (2:30 → 2:00), `'NaT'` (mark as missing), or provide a timedelta like `pd.Timedelta(hours=1)` to add an hour and thus move it into existence. Default is 'raise' which will throw an error so you don't silently get a wrong time. - **tz_convert:** Once tz-aware, you can convert between zones easily: `ts.tz_convert("Australia/Sydney")`. This changes the numeric timestamp to the new zone equivalent moment (so the displayed time changes). - **Normalization:** Sometimes you have tz-aware times and want to drop tz info (e.g. to do value comparisons without worrying about tz). `ts.tz_localize(None)` will remove tz and give you naive local times [108]. Conversely, `ts.tz_localize('UTC')` on naive times will assume they were UTC originally (common to do before tz_convert).

Time arithmetic: Pandas Timestamps support direct subtraction/addition of Python `datetime.timedelta` or pandas `Timedelta` or `DateOffset`. If you do `ts1 - ts2` you get a Timedelta. Adding an integer to a Timestamp is interpreted as nanoseconds (not often useful; rather use offsets or Timedelta). For Periods, addition/subtraction moves by that period frequency.

Rolling with time offsets: We covered rolling with offset like `'7D'`. One more detail: if your data index has an **irregular frequency** or gaps, rolling by time will include those gaps up to the window length. If you want to ensure a fixed number of observations as well, you can set `min_periods` accordingly. Also, `pd.Window` supports a `on=` param for time weight in EWM.

Seasonal and business utils: Pandas has week/quarter/year frequencies (and SEMI, MIN, etc.). For business year (financial year) you can specify month in freq like `'A-JUN'` for Annual freq ending in June. Also `date_range` supports `periods` param to get a certain number of periods instead of stop.

Example DST scenario: If you have a daily timestamp and you want same wall-clock time in another TZ that has DST shifts, consider using `dt.tz_localize` then `tz_convert`. Pandas has logic to align the times properly. If something goes wrong (e.g. you end up an hour off on some days), check ambiguous flags.

PeriodIndex vs DatetimeIndex use-case: If you want to do period-specific calculations, e.g. growth quarter-over-quarter, PeriodIndex can be handy since adding 1 to a Q4 2022 Period gives Q1 2023, whereas adding 3 months to a timestamp Oct 31 might not give Jan 31 (it gives Jan 30 depending on year). Periods handle "end of month" consistently. However, many pandas methods (like rolling, resample) work only on fixed frequency DatetimeIndex, not PeriodIndex (though you can convert period to timestamp at start or end of period easily for rolling).

Time zone aware merging/aligning: If you have two time series in different time zones, aligning them (in a DataFrame or join) will implicitly convert one to the other's tz (actually, it will raise if they are not the same tz, to avoid confusion). Best practice is to convert everything to a common tz (like UTC) for analysis, then convert to local for presentation.

Dateutil vs strict calendars: By default, pandas uses dateutil for parsing and handling time zones, which is quite flexible (e.g. it knows historical DST transitions). You can also use `pytz` time zones (pandas can work with either pytz or dateutil tzinfo objects).

Precision: pandas timestamps are nanosecond-resolved. If you need >ns precision or times beyond Timestamp bounds (~ 1677 to 2262 year range), you might need to use Python `datetime` or other libraries. For most applications, ns is plenty (also note numpy datetime64 max is year 2262).

Summary: Pandas time series tools let you **slice** by dates (e.g. `df["2025-01":"2025-02"]` to get Jan through Feb 2025 data if index is sorted datetime), **resample** to aggregate or interpolate, and handle real-world calendar quirks (business days, holidays, DST). Mastering these offsets and resampling allows you to avoid manual looping to, say, fill missing business days or compute monthly averages. Always be cautious with time zones – ensure you localize and convert appropriately, using the tools pandas provides to resolve ambiguous cases.

---

10) Advanced I/O (Feather/Parquet, HDF5, Excel, and Large CSVs)

Pandas supports a variety of file formats beyond basic CSV, with trade-offs in speed, size, and capabilities. Here's how to leverage them:

Feather and Parquet (Columnar formats): These are binary, column-oriented formats ideal for large datasets and fast read/write: - *Feather:* Essentially an Arrow IPC file. Very fast to write and read, but limited to storing raw data without much meta-information. Use `df.to_feather("data.feather")` and `pd.read_feather("data.feather")`. It requires pyarrow. Feather is fantastic for quick caching of DataFrames on disk (much faster than CSV, and no parsing needed on load). It's not splittable (one file) and not compressible by default (though newer versions allow compression). - *Parquet:* A more feature-rich columnar format. Use `df.to_parquet("data.parquet", engine="pyarrow")` (or engine="fastparquet") and `pd.read_parquet`. Parquet supports compression (snappy by default) and is splittable (you can have partitioned parquet files). It also supports storing pandas index and some metadata. It's a great choice for archiving data for analysis and is interoperable with big data tools (Hive, Spark, etc.). If you have categorical data, pandas will preserve it (as Parquet has dictionary encoding). One caution: if your DataFrame has mixed dtypes or complex objects, Parquet might not support them (e.g. it doesn't natively support Python objects or categorical with unpicklable categories, etc.). Stick to standard types.

Both Feather and Parquet use Arrow under the hood. If you write Arrow-backed pandas data, it can even zero-copy into Parquet (pyarrow will detect Arrow arrays). For analytic tasks, Parquet is often the go-to because of compression and schema evolution. Feather is useful for quick one-off data moving (e.g. between Python/R, or writing to disk cache).

HDF5 (PyTables): `df.to_hdf("file.h5", key="name", format="fixed" or "table")` and `pd.read_hdf`. HDF5 was the earlier solution for pandas archival. It can store multiple DataFrames (the `key` acts like a dict key in the HDF5 file). Two formats: - `"fixed"`: faster write, basically stores NumPy arrays for each column. Not queryable – you have to load entire dataset or use `start=/stop=` index offsets to read subsets. - `"table"`: stores data in a PyTables table format, which is queryable on disk. You

can do `pd.read_hdf("file.h5", key="name", where="col > 0 and col2 = 'XYZ'")` to perform on-disk filtering (indexing columns improves this). You can also append to a table-format dataset (`df.to_hdf(..., format="table", append=True)`). The downsides: HDF5 requires the PyTables library (not lightweight), .h5 files are not easily interoperable with other systems (mainly Python/PyTables), and reading with queries is not as fast as an actual database for complex conditions. Also, HDF5 isn't great with very large data (there is a 2GB per node limit in PyTables unless using 64-bit indices). That said, for medium data that you need to slice by date or category quickly without loading everything, it's very handy. Many legacy pandas workflows used HDF5 for its on-disk query ability. Nowadays, Parquet plus external tools may cover those needs.

Excel: Pandas can both read and write Excel files: - Reading: `pd.read_excel("file.xlsx", sheet_name=..., engine=...)`. It relies on either `openpyxl` or `xlrd`/`odf` etc., depending on format. It can read multiple sheets by passing `sheet_name=None` (returns a dict of DataFrames). - Writing: `df.to_excel("out.xlsx", sheet_name="Sheet1", engine="openpyxl")`. For multiple DataFrames or adding to an existing workbook, use `pd.ExcelWriter`:

```
with pd.ExcelWriter("out.xlsx") as writer:
    df1.to_excel(writer, sheet_name="Summary")
    df2.to_excel(writer, sheet_name="RawData")
```

This ensures both sheets go into one file [109] [110]. Note: writing to Excel is relatively slow (it's basically Python writing XML). For large data, prefer CSV or Parquet. But Excel is great for reports where end-users expect .xlsx.

- **Styling Excel:** Pandas supports exporting a Styler (from `df.style` or `Styler` object) to Excel with formatting. This allows applying conditional formatting, colors, etc., in Python and saving to Excel with those styles. E.g.,

  ```
  df.style.background_gradient().to_excel("styled.xlsx", engine="openpyxl")
  ```

  will color the cells. The styling isn't super fast for huge DataFrames, but it's useful for reports. The `Styler.to_excel` method can create multiple sheets as well [111] [112]. For custom formats or more complex xlsx writing, consider using openpyxl or xlsxwriter directly. But pandas covers a lot: you can freeze panes (`freeze_panes=(r,c)`), set column formats (via `float_format` or `df.style.format` before export) [113], merge cells for MultiIndex headers (`merge_cells=True` by default to span hierarchical index names across cells) [114].

Large CSVs (and TSVs, etc.): CSV is the lowest common denominator, but can be made more efficient: - **Reading in chunks:** As mentioned, use `chunksize`. For example, to process a 10M-row file without loading all:

```
iter_df = pd.read_csv("big.csv", chunksize=100000)
for chunk in iter_df:
    ...  # process each chunk
```

Each chunk is a DataFrame. You can concatenate or aggregate as you go. This prevents memory blow-up. - **Specifying dtypes:** `pd.read_csv` can infer dtypes, but for better memory use, explicitly pass `dtype={"col1": "int8", "col2": "category", ...}` when you know the data. This avoids up-front using larger types or object. Also use `parse_dates=[...]` for date columns so they come out as datetime64. If some columns should be categorical, you can either read as `object` then convert, or read as `pd.CategoricalDtype` by specifying a converter or dtype (though `dtype="category"` in read_csv will treat it as object then category – it may need two passes). - **Compression:** read_csv can handle compressed files (gzip, bz2, zip, xz) if filename ends with .gz, etc., or pass `compression=`. This is convenient but note that decompression is single-threaded and can be a bottleneck. If speed is critical, consider decompressing externally or using a different tool. - **Writing CSV:** Use `df.to_csv("file.csv", index=False)` for standard output. For very large output, writing in chunks can be done by:

```
with open("out.csv", "w") as f:
    for chunk in pd.read_csv("big.csv", chunksize=...):
        chunk.to_csv(f, header=f.tell()==0, index=False)
```

But often just writing directly is fine.

- **Other text formats:** `pd.read_table` is just read_csv with sep `\t`. For JSON, use `pd.read_json` (with lines=True for line-delimited JSON). For big JSON, consider `lines=True` to stream if each line is a JSON record.

Database I/O: (not in question, but worth noting) You can use `pd.read_sql` to query databases via SQLAlchemy and get DataFrames, and `df.to_sql` to write DataFrame to a SQL table. If you have truly large data, pushing computation into SQL (or using something like **DuckDB** within pandas, see the DuckDB reference) can be smarter.

Apache Arrow / Feather IPC streams: Pandas doesn't have a direct `to_arrow` file writer in user API (aside from Feather which is essentially arrow IPC). But you can get a pyarrow Table via `pa.Table.from_pandas(df)` and then use Arrow's API to stream or send over network (Flight, etc.). That's beyond pandas though.

FSSpec integration: Many pandas readers (`read_csv`, `read_parquet`, etc.) accept URLs or use `fsspec` to handle remote files (like S3, GCS). If you have `fsspec` and the appropriate filesystem installed (e.g. `s3fs` for S3), you can do `pd.read_parquet("s3://bucket/key.parquet")`. For large remote files, this can be slower due to network; some formats (parquet) allow partial reads which pandas/pyarrow will leverage (downloading only needed row groups or metadata). If reading many files, sometimes using `pyarrow.dataset` directly or tools like Dask is better.

Excel multi-index preservation: Pandas will write MultiIndex columns and index to Excel by expanding them into multiple rows. It merges cells for index names by default [114]. Reading that back via `read_excel` isn't guaranteed to reconstruct the exact MultiIndex unless you specify `header=[0,1,...]` for columns and `index_col=[0,1,...]` for index levels.

HDF5 performance note: If you go that route, know that "fixed" format is basically as fast as numpy save (but with compression option if you enable it), and "table" format is slower, especially for writes, but allows queries. Also, HDF5 files can become fragmented or slow if you do a lot of appends; you might need to periodically repack (copy to a new file).

In general, **for analytical data storage**: use **Parquet** if you might use other tools or have very large data (columnar compression and predicate pushdown are big wins). Use **Feather** for quick, language-agnostic interchange (between Python/R or for caching between runs). Use **HDF5** if you need on-disk querying in a single-user setting or need to store multiple dataframes in one file with easy lookup. Use **Excel** for reporting or small data exchange with humans (never for very large data, as it's slow and limited to ~1M rows). And use **CSV** for simple, maximum compatibility needs (keeping in mind the size and parsing cost – e.g. a 1GB CSV might take 5-10x longer to read than a 1GB feather).

---

11) Interoperability: Arrow, NumPy, and DataFrame Exchange Protocol

Modern pandas (≥2.0) is increasingly built to interoperate with other dataframe libraries and computing frameworks:

Apache Arrow integration: As discussed, pandas can use Arrow as a storage backend for certain dtypes. But beyond that, converting between pandas and pyarrow is straightforward: - **pandas → Arrow:** `pyarrow.Table.from_pandas(df)` will produce an Arrow Table. This is essentially zero-copy for ArrowDtype columns, and for others it will convert (e.g. object strings to Arrow binary). Alternatively, use `df.to_parquet()` or `df.to_feather()` which under the hood go through Arrow libraries. - **Arrow → pandas:** If you have a `pyarrow.Table tbl`, do `tbl.to_pandas(types_mapper=pd.ArrowDtype)` [115]. The `types_mapper` ensures Arrow types become appropriate pandas extension types (like ArrowDtype or pandas nullable types) instead of degrading (e.g. without it, an Arrow string might become object dtype). In pandas 2.0+, Arrow's `to_pandas` will by default produce Arrow-backed columns when possible. You can also do `pd.DataFrame(tbl)` directly in some cases, or use `pd.api.interchange.from_dataframe` (if the table implements the interchange protocol, see below).

- **Why Arrow interop:** This allows zero-copy data exchange between pandas and libraries like **Polars**, **PySpark**, **Vaex**, **TensorFlow** (through tf.data), etc., that understand Arrow. It also means you can push a pandas DataFrame to Arrow, then use Arrow Flight RPC to send it to a remote system, or save to Parquet, all without serializing to Python objects.

NumPy vs PyArrow backing: - Traditional pandas uses NumPy ndarrays for storage. This is highly optimized in C, but limited in dtype flexibility (no missing for ints without separate mask, no string type besides object). Arrow backing offers a more uniform approach (each column is an Arrow array possibly with null bitmap). - In practice, you don't usually need to know which is which, except for performance: operations on Arrow arrays might have different performance characteristics than on NumPy. E.g. computing `series.sum()` on an Arrow int64 might internally use Arrow's SIMD routines (fast). But some pandas methods might convert Arrow to numpy under the hood if not supported (slower). The gap is closing with each version. - One difference: if you index a NumPy-backed Series with `.to_numpy()`, you get a view or copy of the numpy array. If you do the same on an Arrow-backed Series, `.to_numpy()` yields

an **object dtype numpy array** of Python scalars (because Arrow arrays can't be represented as a NumPy array directly except for numeric types). This is to be aware of if using `.to_numpy()` for performance or interoperability – better to use Arrow's own methods or leave as Series.

DataFrame Exchange Protocol (dataframe): This is a new protocol (defined by the Data API consortium) to allow zero-copy interchange between DataFrame libraries (pandas, Apache Arrow, cuDF, etc.) through a common interface. In pandas, a DataFrame has a method `df.__dataframe__(nan_as_null=False, allow_copy=False)` [116] [117] that returns an exchange object. Consuming libraries can call this without knowing it's pandas, then get buffers of data. - The returned object has methods like `.num_columns()`, `.num_rows()`, `.get_column()` that allow the consumer to retrieve columns in a standardized way. It basically exposes pointers to the memory and metadata about types. - There's also a convenience `pd.api.interchange.from_dataframe(other_df)` which takes any object supporting the protocol and returns a pandas DataFrame [118]. This is how, for example, you could take a Polars DataFrame (which implements **dataframe**) and get a pandas DataFrame without intermediate conversion (it will zero-copy if possible, or at least avoid Python loops). - The protocol is still stabilizing (it's marked as protocol draft). Pandas supports it since 2.0. In practice, for now, Arrow's to_pandas and from_pandas or Polars' built-in methods might be more direct, but long-term this protocol will mean you can pass a pandas DF to a library and it can operate on it without conversion if it supports it.

Example use-case: Suppose you have a pandas DataFrame `df` and you want to use a machine learning library that prefers Arrow or its own DF type:

```
# Using __dataframe__ protocol to go to cuDF, for instance
import cudf
gdf = cudf.from_dataframe(df)  # cudf will likely use the protocol internally
```

This would avoid going through CSV or other heavy serialization.

Another scenario is using Arrow Flight to send dataframes: you could do `pyarrow.Table.from_pandas(df)` then Flight to send.

Numpy interop: Many libraries accept NumPy arrays. You can get numpy arrays out of pandas easily (`df.values` or `df.to_numpy()`). But remember, `df.values` for mixed types will be a single numpy object array. If you need to interface with numeric libraries (Numba, SciPy), ensure each column is a numpy array of the appropriate dtype (e.g. `df['float_col'].to_numpy(dtype=np.float64)` if needed). Converting Arrow strings to numpy (as object dtype) will be slower — better if those libraries can accept Arrow or you process as pandas Series.

Matplotlib (plotting) works with pandas directly (it calls to_numpy internally), so nothing special needed.

R and others: There's `rpy2` to convert DataFrame to R data.frame, but a promising route is through Arrow: R's `arrow` package can directly read a Feather/Parquet written by pandas.

In summary, **pandas is no longer an island**: it plays well with Arrow for file and in-memory exchange, and implements a standard protocol so that data can flow between pandas and other DataFrame

implementations with minimal overhead. For the end user, this means you can use best-of-breed tools for each task (e.g. pandas for data cleaning, Arrow for serialization, a GPU dataframe for heavy lifting, etc.) without spending excessive time converting or copying data. Just keep your pandas DataFrames in modern dtypes (like nullable/Arrow types) when possible to maximize compatibility.

Sources: This reference is built on the official pandas documentation and community knowledge [22] [12] [44] [119] [120] [2] [96] [86] [73] [121], covering features as of pandas 2.3.x. Use it as a guide for advanced pandas techniques and as a starting point to delve deeper into each topic in the pandas docs. Happy data wrangling!

---

1 pandas · PyPI
https://pypi.org/project/pandas/

2 3 4 99 100 Copy-on-Write (CoW) — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/user_guide/copy_on_write.html

5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 Extending pandas — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/development/extending.html

26 27 28 29 70 Working with text data — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/user_guide/text.html

30 72 95 96 101 102 What's new in 2.0.0 (April 3, 2023) — pandas 2.3.3 documentation
https://pandas.pydata.org/pandas-docs/stable/whatsnew/v2.0.0.html

31 32 33 34 Categorical data — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/user_guide/categorical.html

35 36 37 38 39 40 41 42 43 MultiIndex / advanced indexing — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/user_guide/advanced.html

44 45 46 47 48 49 50 51 Reshaping and pivot tables — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/user_guide/reshaping.html

52 53 56 60 61 64 65 66 67 68 69 119 120 Windowing operations — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/user_guide/window.html

54 55 58 59 62 63 Pandas Rolling Window: Rolling, Expanding, and EWM – Kanaries
https://docs.kanaries.net/topics/Pandas/pandas-rolling-window

57 Pandas Window Functions You're Not Using Enough | by Nikulsinh Rajput | Medium
https://medium.com/@hadiyolworld007/pandas-window-functions-youre-not-using-enough-2ce73b77e2d5

71 Pandas with pyarrow Strings: Memory, Speed, and Gotchas - Medium
https://medium.com/@kaushalsinh73/pandas-with-pyarrow-strings-memory-speed-and-gotchas-f9f9773b02cb

73 74 75 76 107 108 pandas.Series.tz_localize — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/reference/api/pandas.Series.tz_localize.html

77 78 115 121 PyArrow Functionality — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/user_guide/pyarrow.html

79 80 81 82 83 84 85 86 87 88 89 Enhancing performance — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/user_guide/enhancingperf.html

90 91 Group by: split-apply-combine — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/user_guide/groupby.html

92 93 pandas.core.groupby.DataFrameGroupBy.apply — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/reference/api/pandas.core.groupby.DataFrameGroupBy.apply.html

94 Measuring the memory usage of a Pandas DataFrame
https://pythonspeed.com/articles/pandas-dataframe-series-memory-usage/

97 98 networkx.md
file://file_00000000be9c71f6b1a390076c7f9265

103 104 105 106 Time series / date functionality — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/user_guide/timeseries.html

109 110 111 112 113 114 pandas.io.formats.style.Styler.to_excel — pandas 3.0.0.dev0+2091.ge55d90783b documentation
https://pandas.pydata.org/pandas-docs/dev/reference/api/pandas.io.formats.style.Styler.to_excel.html?highlight=io%20excel%20read_excel

116 117 pandas.DataFrame.__dataframe__ — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.__dataframe__.html

118 pandas.api.interchange.from_dataframe - PyData |
https://pandas.pydata.org/docs/reference/api/pandas.api.interchange.from_dataframe.html