PANDAS OVERFLOW

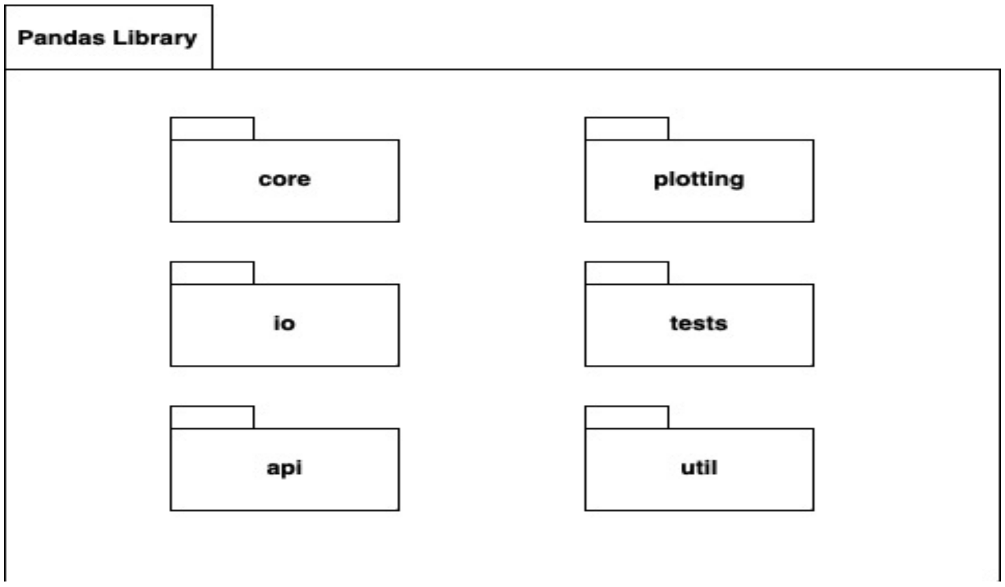
CSCD01 Group Project

Deliverable 2

Zhiqi Chen
Anika Sultana
Christina Ma
Gabriela Esquivel Gaghi
Hyun Woo (Eddie) Shin
Arthur Lu
Man Hei Ho

Table of Contents

Pandas Library Architecture Overview



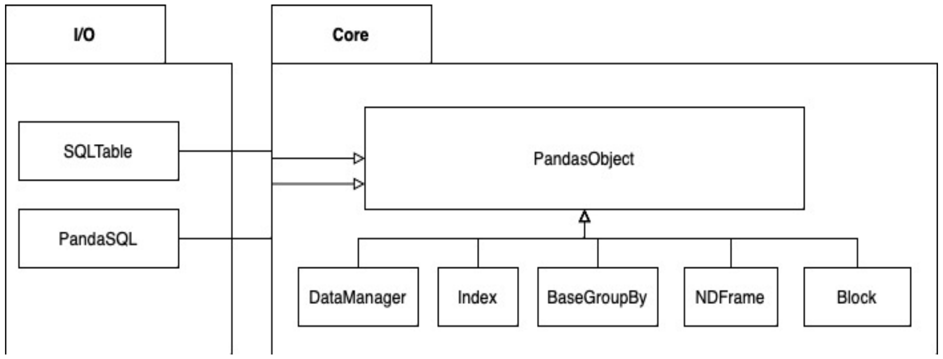
The above diagram roughly describes the high-level architecture of Pandas library. Pandas library contains Core, I/O, API, Plotting, tests and util packages which are essential packages to perform heavy computations, data processing and data visualizations.

- **Core**: The core package provides basics files for various operations in the library and implements key data structures, Series and DataFrames.
- **I/O**: The I/O package provides the basic functionalities of reading and writing data from external sources.
- **API**: The API package handles external API imports which are necessary for the library to be functional.
- **Plotting**: The Plotting package works with matplotlib, data visualization package in Python, to plot the data processed in the library.
- **Tests**: The Tests package contains a suite of test cases which are used to test the code changes made by contributors since the library is maintained by the open-source community.
- **Utils**: The Utils package contains various helper functions and development tools used in various purposes such as testing and debugging.

Pandas Core

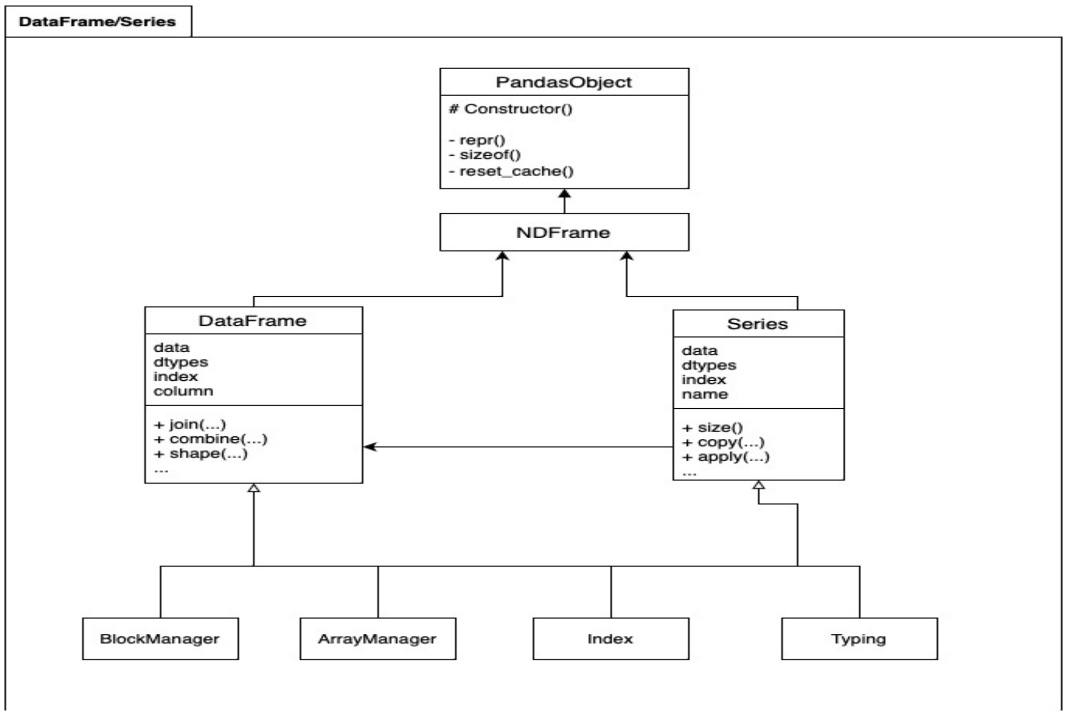
In Pandas, the core module defines the fundamental data structures and their operations, and algorithms for working with the data. In our high-level UML diagram, we chose to dive deeper into the Series, DataFrame, Index, Internals, Groupby, and Dtypes modules.

PandasObject



The PandasObject is the base class for many data structures within Pandas like Series, DataFrame, and Index.

Series and DataFrame



Series and DataFrame are the core Data Structure for storing data within pandas. Series is a one-dimensional data structure, while DataFrame is 2D. Notably, it is mutable, and can have multiple different data types including time series. DataFrames have attributes like `data` (block manager), `dtypes` (for type of data that is in the object), `index` (names of the rows), `columns` (names of the columns).
Ie.

```
   c  a
0  3  1
1  6  4
2  9  7
```

Would have *indexes*: 0, 1, 2 and *columns*: ‘c’, ‘a’.
Series have similar attributes except columns and includes `name` for the name of the object.

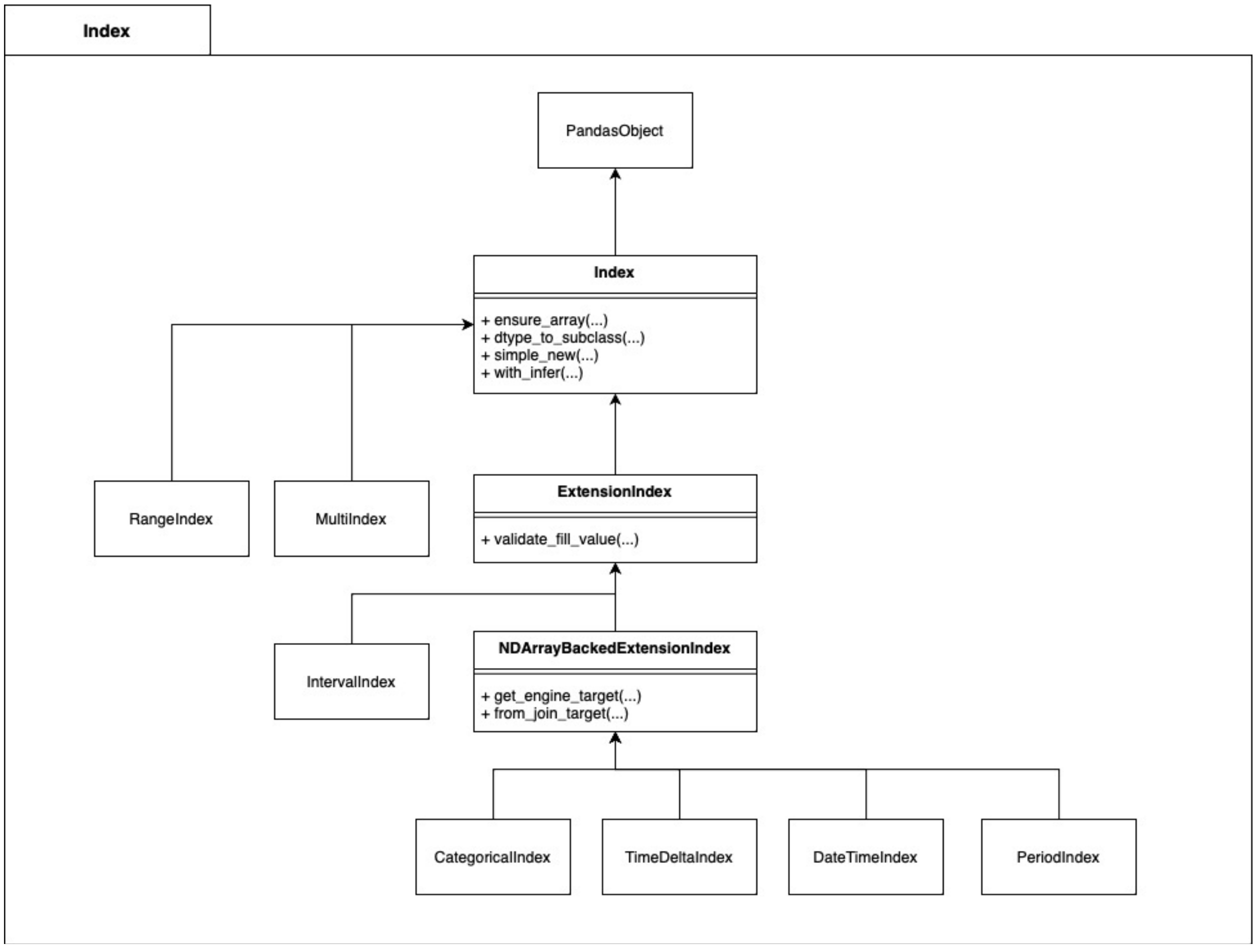
They both have a host of dependencies.

- Typing that includes various types of data that pandas accept and basic data like Scalor that these data structure use.
- Block and Array Manager for various methods for manipulation of data are found within them.
- Index for indexing and searching data.
- And many more that provide functionalities to these data structures.

Finally, `NDFrame` facilitates numerous backend and basic functionalities like setting label (names) of index and columns, comparing 2 objects for the same elements, returning an absolute value of the DataFrame/Series, etc.

It's worth noting that they are well decoupled. DataFrame and Series facilitates but the core implementation of functionalities are divided and handled by the Manager, Indexes, or NDFrame.

Index



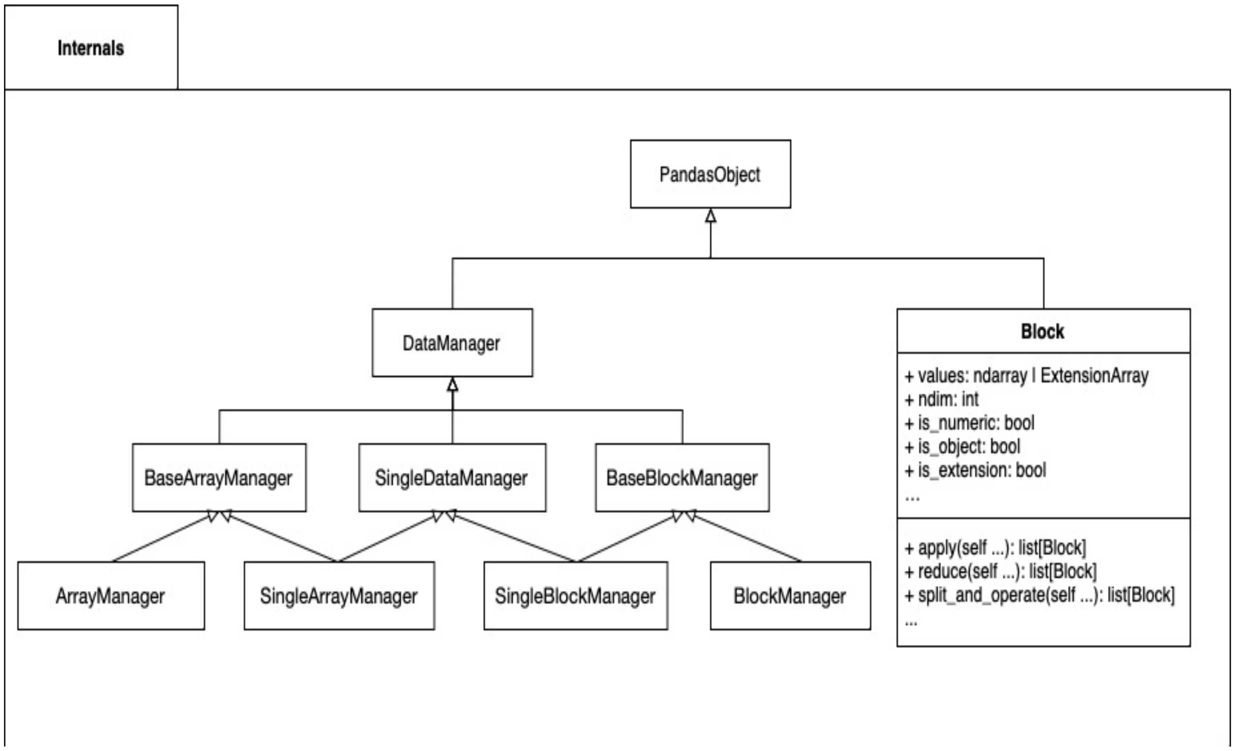
`Index` is an immutable sequence that is used to index DataFrame and Series. This object stores all axis labels for all pandas objects. An index can be thought of as a label or identifier for each row or column in a pandas DataFrame or Series.

There are 8 distinct instances of Index:

- `RangeIndex`: the default instance that represents a range of integers
- `CategoricalIndex`: represents categorical data
- `MultiIndex`: represents multi-levels, or hierarchy
- `IntervalIndex`: represents one-dimensional intervals or ranges
- `DatetimeIndex`: represents n-dimensional arrays of datetime64 data
- `TimedeltaIndex`: represents n-dimensional arrays of timedelta64 data
- `PeriodIndex`: represents n-dimensional arrays holding sequences of time periods

All of these instances inherit from the `Index` class.

Internals

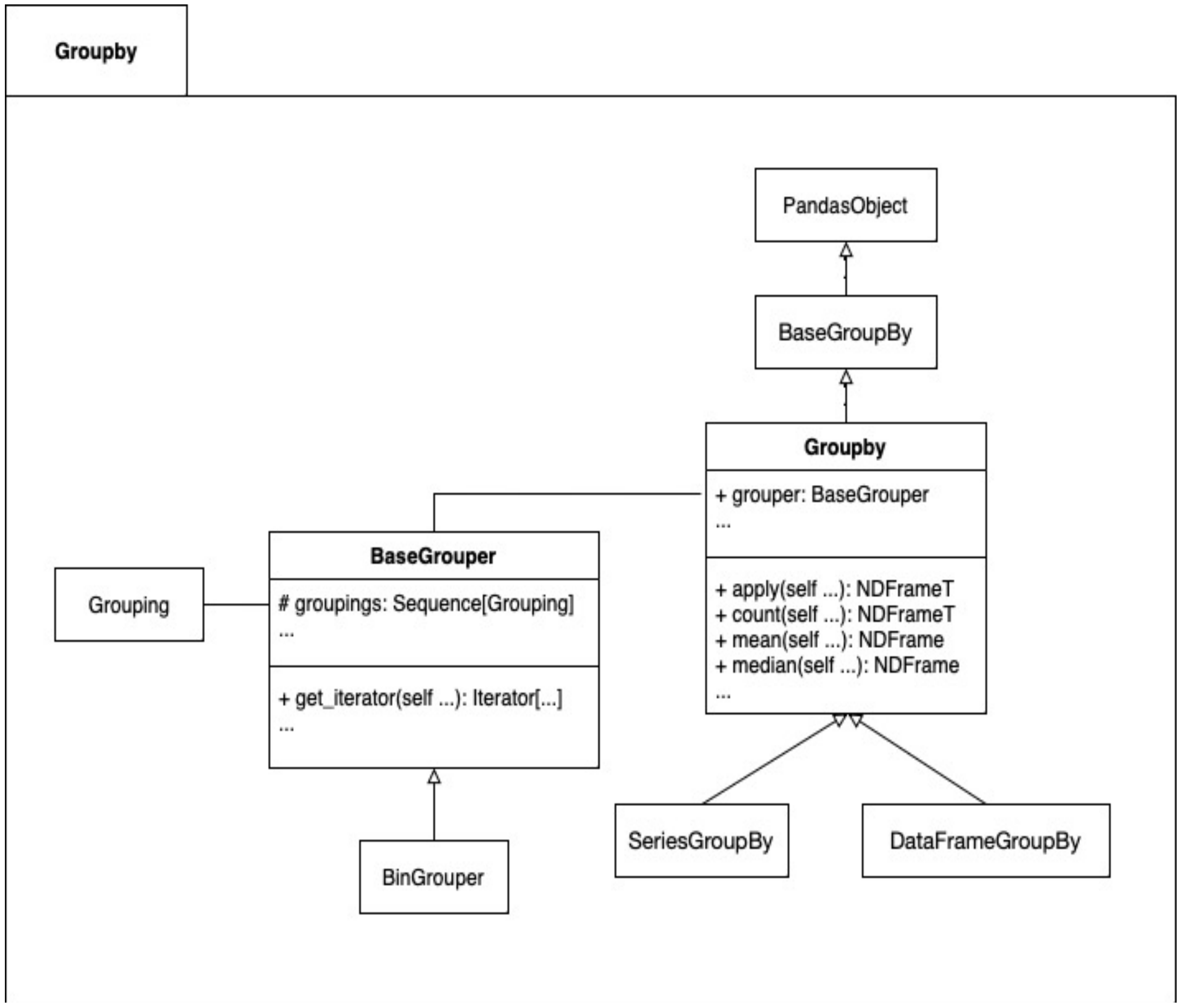


The `internals` module implements the internal data structures and functions underlying the Pandas objects.

The `DataManager` class implements the low-level architecture for storing and managing data. The `BaseBlockManager` extends `DataManager` and is the core internal data structure that implements DataFrame and Series. Its two children, `BlockManager` and `SingleBlockManager`, hold 2-dimensional and 1-dimensional blocks of data respectively; they are typically used to hold rows or columns of DataFrame and Series. The `BaseArrayManager` along with its children, `ArrayManager` and `SingleArrayManager`, is also the core internal data structure to implement DataFrame and Series. It is an alternative to the `BaseBlockManager`, storing a list of arrays instead of blocks.

The `Block` class is the base class of classes used internally by Pandas' data structure to store and manage a canonical n-dimensional unit of homogeneous data. The subclasses include `ExtensionBlock`, `NumpyBlock`, `NumericBlock`, etc. Note that `Block` is index-ignorant, meaning that the container (like DataFrame) should take care of indexing information itself.

Groupby

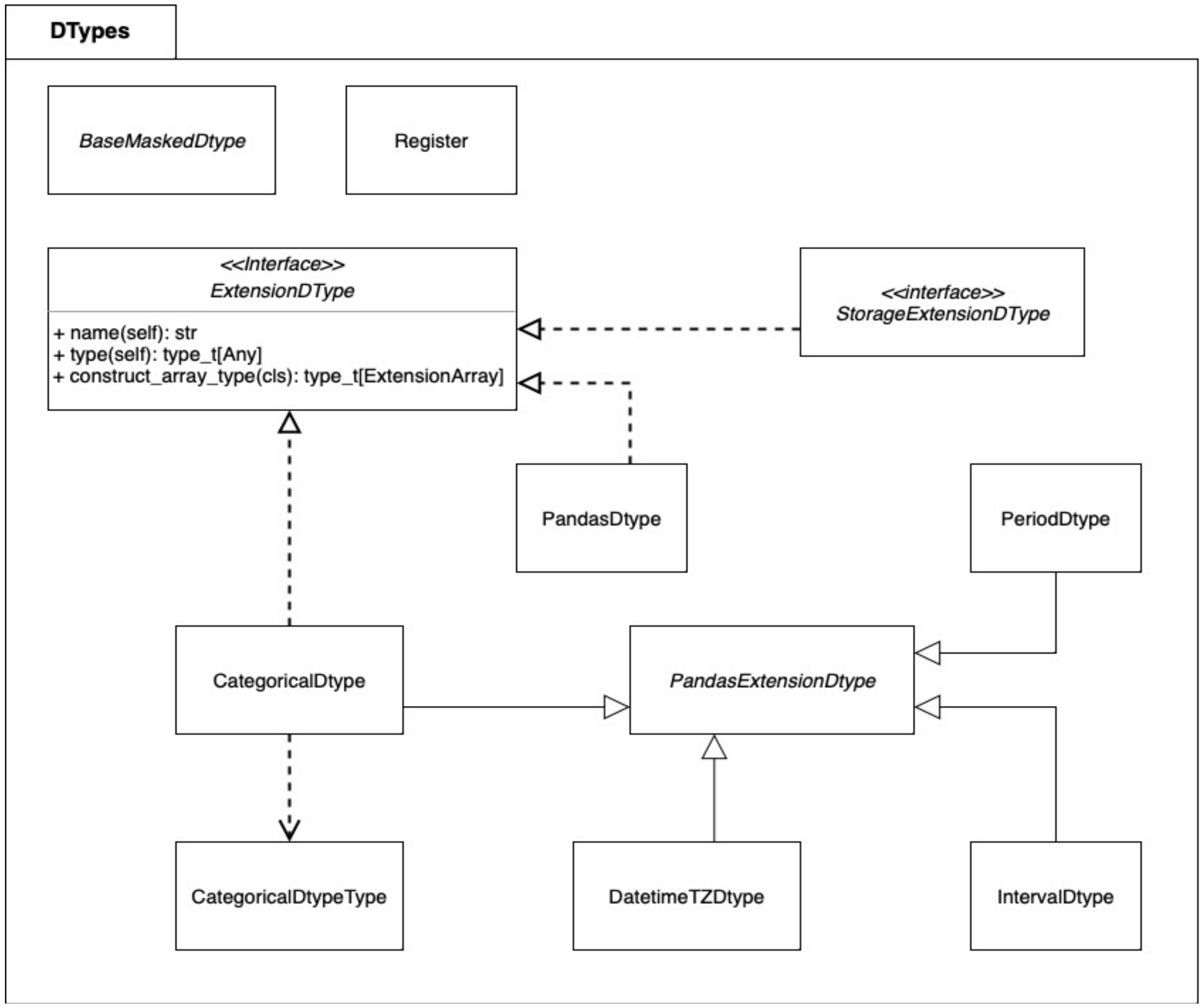


Groupby
PandasObject
BaseGroupBy
Groupby
+ grouper: BaseGrouper
...
+ apply(self ...): NDFrameT
+ count(self ...): NDFrameT
+ mean(self ...): NDFrame
+ median(self ...): NDFrame
...
BaseGrouper
# groupings: Sequence[Grouping]
...
+ get_iterator(self ...): Iterator[...]
...
Grouping
BinGrouper
SeriesGroupBy
DataFrameGroupBy

The `groupby` module implements grouping and aggregating functionalities for DataFrame and Series. The `groupby()` function returns a `Groupby` object (`DataFrameGroupBy` or `SeriesGroupBy`, subclasses of `Groupby`) which allows users to work with grouped data like aggregating, transforming or applying operations on the grouped data.

Internally, the `Groupby` class is implemented using the `BaseGrouper` class which actually holds the generated groups. The `BaseGrouper` class uses the `Grouping` class to hold the grouping information for each key.

DTypes



In the `dtypes/base.py` file, with the purpose of extending Pandas with custom array types, three classes are defined. The first one, *`ExtensionDType`*, is an interface which represents a custom data type to be paired with an `ExtensionArray` (which is an abstract base class for custom 1-D array types). It includes the following abstract methods that must be implemented by its subclasses:

- *`type(self): type_t[Any]`*
- *`name(self): str`*
- *`construct_array_type(cls): type_t[ExtensionArray]`*

Secondly, *`StorageExtensionDtype`* extends *`ExtensionDType`* to allow for an *`ExtensionDType`* that is backed by multiple implementations. Lastly, `Registry` is a class responsible for dtype inference. According to the Pandas Documentation, it allows one to map a string representation of an *`ExtensionDType`* to an *`ExtensionDType`*.

Furthermore, the `dtypes/dtypes.py` file defines extension dtypes using the *`ExtensionDType`* interface described above. Initially, the class

`PandasExtensionDtype` (which inherits from *`ExtensionDType`*) is defined for holding a custom `dtype`. Then, `CategoricalDtypeType` is a metaclass to represent the type of the `CategoricalDtype` class. Afterwards, the file defines the following classes for different dtypes using `PandasExtensionDtype:`

- `CategoricalDtype` (which inherits from *`PandasExtensionDtype`* and *`ExtensionDtype`*) is used as a type for categorical data, and defines two attributes:
    - `categories` (an `Index` containing the categories allowed)
    - `ordered` (a `boolean` indicating whether it is an ordered Categorical).
- `DatetimeTZDtype` (which inherits from *`PandasExtensionDtype`*) is an *`ExtensionDtype`* for timezone-aware datetime data. This class defines two attributes:
    - `unit` (a `str` representing the precision of the datetime data)
    - `tz` (the timezone, represented as a `str`, `int`, or `datetime.tzinfo`)
- `PeriodDtype` (which inherits from `PeriodDtypeBase` and `PandasExtensionDtype`) is an *`ExtensionDtype`* for Period data. It defines one attribute:
    - `freq`, (which is a `str` or `DateOffset` indicating the frequency of the `PeriodDType`).
- `IntervalDtype` (which inherits from `PandasExtensionDtype`) is an *`ExtensionDtype`* for Interval data. This class has one attribute:
    - `subtype` (which can be a `str` or a `np.dtype` representing the dtype of the Interval bounds)

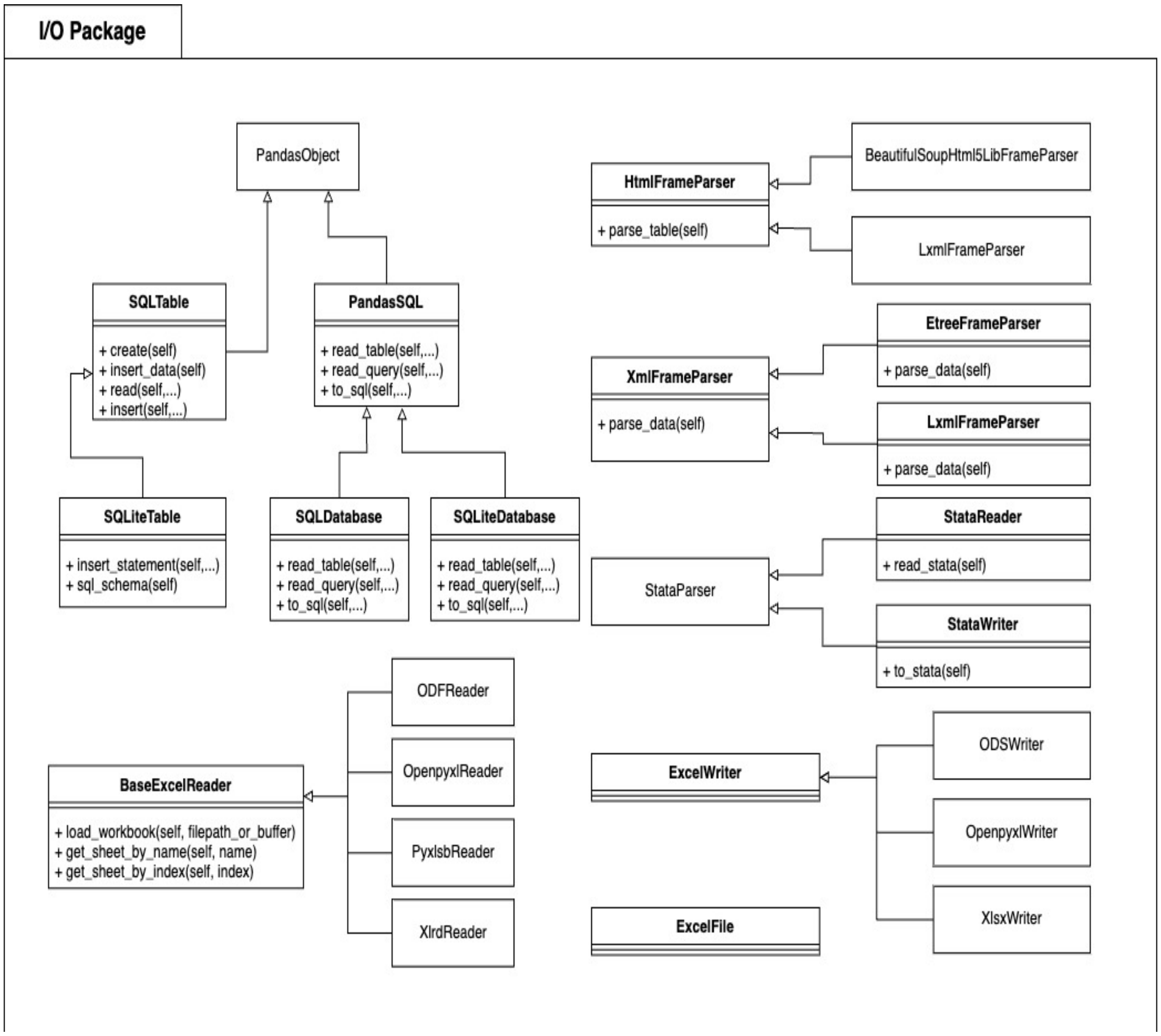
Next, the class `PandasDtype` (which implements the *`ExtensionDtype`* interface) is a Pandas *`ExtensionDtype`* for NumPy dtypes, used mostly for internal compatibility. This class provides implementations for the abstract methods *`type(self)`*, *`name(self)`*, and *`construct_array_type(cls)`*, as required by the interface it implements. Lastly, `BaseMaskedDtype` (which inherits from *`ExtensionDtype`*) is defined as a base class for the dtypes for `BaseMaskedArray` subclasses. `BaseMaskedDtype` requires its subclasses to implement the following class methods:

- `from_numpy_dtype(cls, dtype: np.dtype): BaseMaskedDtype`
- `construct_array_type(cls): type_t[BaseMaskedArray]`

Other

Other modules within the Pandas core module define other operations for the Pandas data structures. As examples, the `computation` module defines the operators and provides a set of functions for performing mathematical and statistical operations on the Pandas data structure like DataFrame and Series. The `ops` module provides a set of functions for arithmetic and logical operation on Pandas data structure like add, subtract, kleen and, etc. The `reshape` module provides functions to transform a Pandas data structure including pivoting and merging.

Pandas I/O



One of the key responsibilities of the I/O package in Pandas library is handling different types of input data formats. Some of the main data formats that Pandas can process are Excel, JSON, SQL, HTML and XML. Based on the high level structure of the I/O package, we were able to identify key modules and classes which contain key attributes and methods for the primary I/O operations for each data format. The publicly available methods widely used by Pandas users are read_[data_format] and to_[data_format]. These methods are implemented differently depending on data formats, and Pandas provide an interface for these methods within the I/O package. More detailed information about IO tools in Pandas can be found here. In the following, we explained how I/O functionalities of some of the major data formats are implemented.

XML Format

XML files can be parsed into DataFrames using the internal class called `XMLFrameParser`. Inside the same module, `EtreeFrameParser` inherits from `XMLFrameParser` to override some parent functions for the Python standard XML module library. `LxmlFrameParser` also inherits from `XMLFrameParser` to override some parent functions to incorporate the use of a third-party lxml library.

HTML Format

HTML files can be parsed into DataFrames using the internal class called `HtmlFrameParser`. `BeautifulSoupHtml5LibFrameParser` inherits from `HtmlFrameParser` to use BeautifulSoup under the hood and `LxmlFrameParser` inherits from `HtmlFrameParser` to use lxml under the hood.

Excel Format

There are three base classes which are `BaseExcelReader`, `ExcelWriter` and `ExcelFile`. `BaseExcelReader` is an abstract class which provides method signatures which can be implemented when it is inherited. Currently, `BaseExcelReader` is inherited by four different classes which are `ODFReader`, `OpenpyxlReader`, `PyxlsbReader` and `XlrdReader`. They handle different file extensions which make them have different method implementations from the parent class. `ExcelWriter` is responsible for writing DataFrame objects into excel sheets.

Currently, `ExcelWriter` is inherited by `ODSWriter`, `OpenpyxlWriter` and `XlsxWriter`. Lastly, `ExcelFile` is a general class which parses tabular excel sheets into DataFrame objects.

Stata Format

There is a base class called `StataParser` which provides the binary conversion functionality. The base class is inherited by `StataReader` and `StataWriter`. `StataReader` is responsible for reading Stata binary files and `StataWriter` is responsible for writing Stata binary files.

SQL Format

In `sql.py`, there are `SQLTable` class and `PandasSQL` class which are the two base classes for handling I/O operations from SQL. Both of the base classes inherit from `PandasObject` is the base class for various pandas objects. `PandasObject` class is located in `base.py` inside the core package, and this is one example of how two different packages are related to each other.

Based on the codebase, `PandasSQL` is an abstract class providing only the method signatures which must be implemented by child classes. `SQLDatabase` is a child class of `PandasSQL` which implements the methods defined in `PandasSQL`. In the end, `SQLDatabase` enables conversion between DataFrame and SQL databases. `SQLTable` is responsible for mapping pandas tables to SQL tables.

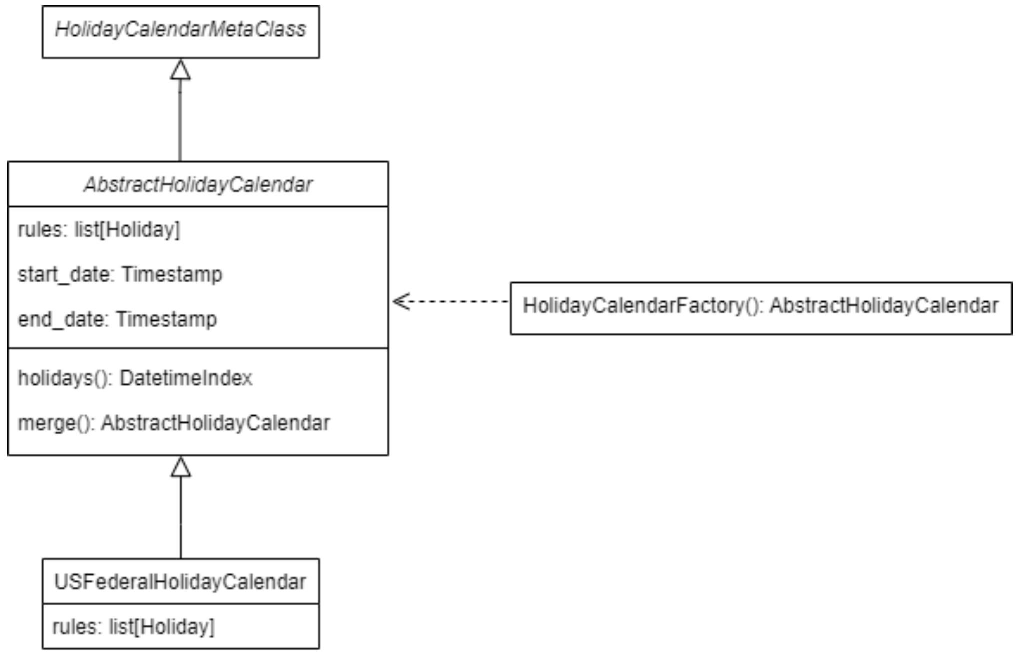
Given the fact that SQL has a lighter version called SQLite, `SQLiteTable` inherits from `SQLTable` and `SQLiteDatabase` inherits from `PandasSQL`. The basic functionalities are basically equivalent to their parent classes but small modifications are made to be used with SQLite.

JSON Format

There are three key classes to handle JSON files in Pandas library. `Writer` class provides a backbone structure of writing in JSON files. `SeriesWriter` and `FrameWriter` inherit from the Writer class. The only difference between SeriesWriter and `FrameWriter` is how they work with axes as Series and DataFrames have different dimensions. `JSONTableWriter` subsequently inherits from `FrameWriter` to convert the data into JSONTable format.
`JsonReader` class handles reading JSON files by providing an interface for read operations in JSON. The `JsonReader` class contains many overloaded method signatures which can be implemented differently by its subclasses.

Lastly, it's important for Pandas to correctly parse JSON files, and the `Parser` class provides a backbone structure of parsing operations in JSON. `SeriesParser` and `FrameParser` inherit from the `Parser` class, and the difference in terms of their implementations is subtle due to the difference in dimensionality of Series and DataFrame.

Design Pattern #1 – Factory Method

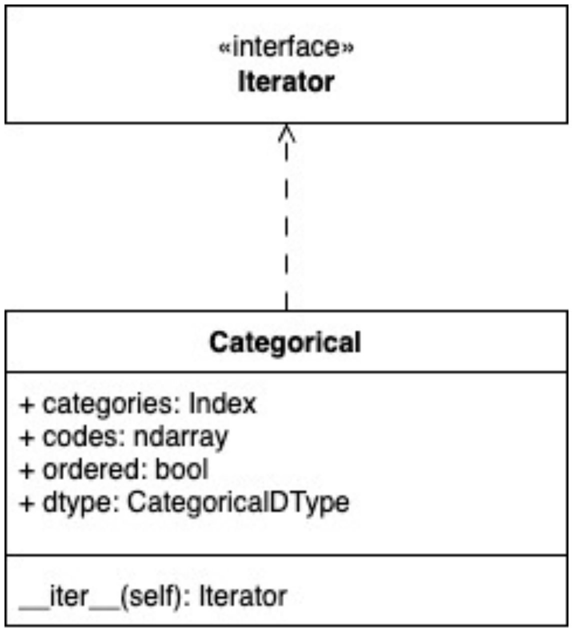


The above UML diagram highlights an implementation of the factory method pattern in the Pandas codebase. `AbstractHolidayCalendar` (line 389) is an abstract class based on the metaclass `HolidayCalendarMetaClass` that stores a list of `Holiday`s (line 146) over a period of time defined by `start_date` and `end_date`. It is given one concrete implementation in the predefined `USFederalHolidayCalendar` class (line 556), an example calendar that contains federal holidays as specified by the American government. Pandas uses these calendars to perform data analysis over periods of time while accounting for holidays (i.e; skipping over them when calculating work days).

The factory method `HolidayCalendarFactory()` (line 583) provides an interface to create new `AbstractHolidayCalendar` type objects or combine existing calendars. Since this design pattern was used, the client does not have to specify what class of calendar will be created, instead receiving whatever type object (as in `PyTypeObject`) is returned by the factory method. Though this implementation differs slightly in that the factory method is declared outside of a class, the goal of using this method to generate different subclasses of `AbstractHolidayCalendar` remains the same.

Code Location: pandas/tseries/holiday.p, line 382–586

Design Pattern #2 – Iterator Pattern



`Categorical` is a data type in pandas and all values of categorical data are either in *categories* or *np.nan*. `Categorical` is not a numpy array but it is a python object. An iterator design pattern is used to iterate over a collection or group of objects. This helps mask the specific implementations of the object. In pandas, a specific example is inside Categorical.py, where a method is declared in **line 1786**, which implements an iterator for `Categorical` (the return type is an iterator). Since `Categorical` has values, we need to iterate over them and therefore an iterator has been declared. The method makes a call to `iter()`, which converts an `iterable` to `iterator`.

Code Location: pandas/core/arrays/categorical.py lines 1786-1793