University of Toronto

CSCD01H3 Engineering Large Software System

Deliverable #2: Pandas Codebase Design

Final Version

Winter 2023

Instructor: Prof. Rawad Abou Assi

Group Name: Timbits

Group Members:

Xuen Shen

Xu Zheng

Lingfeng Su

Yawen Zhang

Megan Mujia Liu

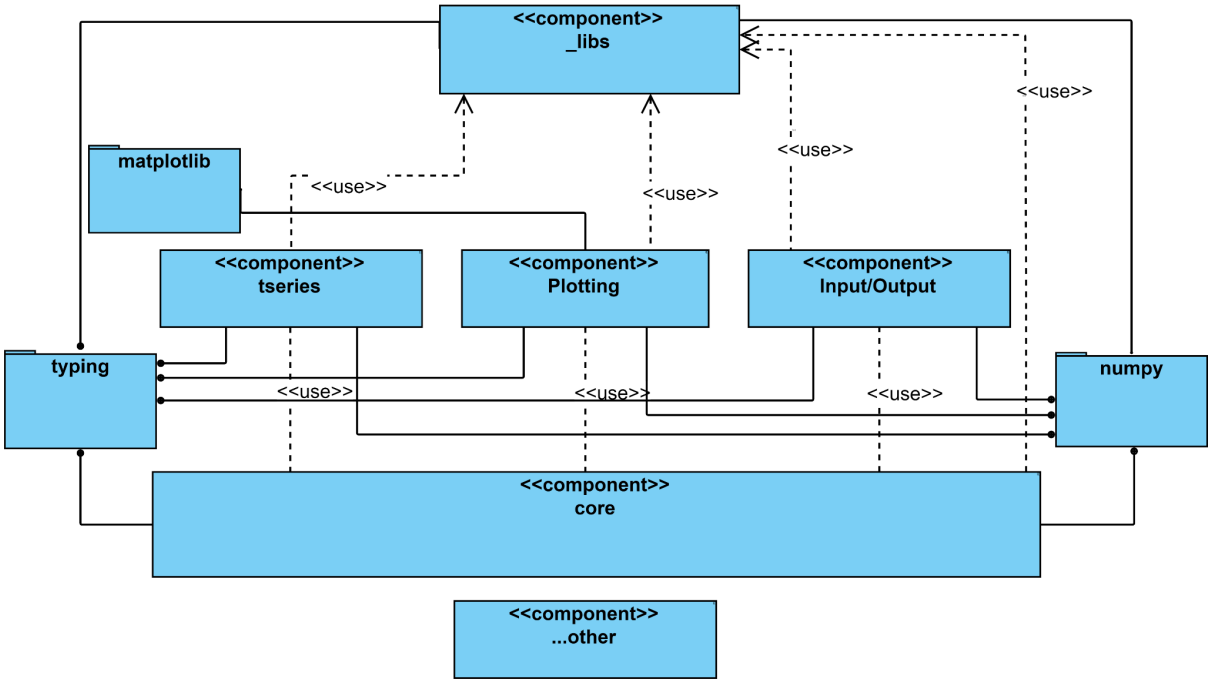
Shaopeng Lin

Runyu Yue

Feb.19, 2023

2.1 System Architecture

Highest level Architecture



The Pandas library can be interpreted as two separate components which are the core data structures and the toolings that enhances its capabilities. We have presented a few of the tooling modules in the UML that are widely used such as:

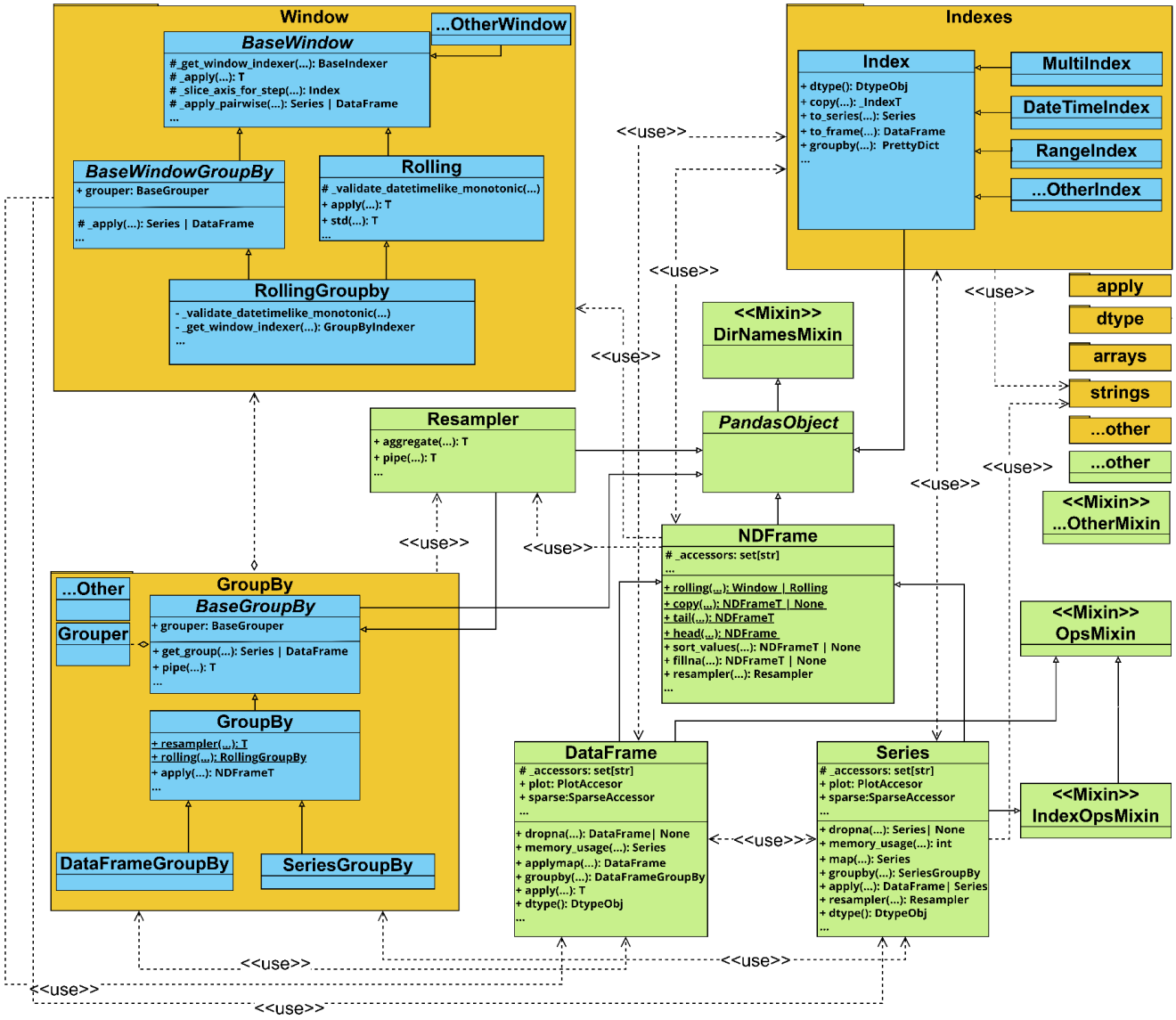
- ***pandas.core***: Core implementation of the user APIs such as DataFrame and Series. The Core Architecture section will cover the details of this module.
- ***pandas._libs***: contains important optimizations implemented in Cython/C that contributes to the high performance of the library. The module provides much faster implementations of operations such as arithmetic operations, sorting, time-series, indexing, and string manipulation.
- ***pandas.plotting***: This module provides a simpler interface of ***matplotlib*** that is specialized to be used with the core data structures. The specialized customizations and functionality allows pandas users to plot their data with little knowledge of ***matplotlib***.
- ***pandas.io***: This module provides a huge variety of input output capabilities that allow users to read/write CSV, Excel, SQL table, and JSON into/from the core data structures. It is designed to provide Pandas with a wide range of data interoperability.
- ***pandas.tseries***: uses ***pandas._libs*** to create utility methods for time series frequency analysis.

Other tooling modules such as pandas.compat or pandas._config are not listed on the UML diagram as they are not essential for us to understand the pandas library at a higher level.

We will also have to talk about the importance of ***numpy*** and ***typing***, as they are an integral part of the library:

- ***numpy***: The core functionality of Pandas is built on top of ***numpy***, such as the Series and DataFrame data structure, which can be seen as extensions of 1-D and 2-D numpy arrays.Though ***numpy*** is not designed for tabular data, it has the reputation of having robust multi-dimensional array data structures, efficient computation, and great interoperability and compatibility with many other python data analytic libraries. For those reasons, it became a part of the most important ***pandas.core***, ***pandas._libs***, ***pandas.io***, and is used extensively in almost all parts of the Pandas library .
- ***typing***: The module introduced in Python3.5 is now an integral part of the Pandas library. It provided type hints but most importantly generics to the core data structures and allowed more flexibility in the design. The Union capability is especially important for the amazing compatibility Pandas has with numpy. Pandas defines and uses a lot of extended types and it is good to note some common generics such as NDFrameT, AnyArrayLike, and Dtype.

Core Architecture



As we have mentioned previously in the highest level, the core data structures lies in the module ***pandas.core*** and is the heart and soul of the Pandas library. This module contains the implementation of all the Pandas data structures and majority of the operations that can be performed on them. There is a lot of content inside ***pandas.core***, but we believe understanding the three major components, **NDFrames(Series, DataFrame)**, **GroupBy** and **Window**, are enough to understand the system and grasp the architecture. The rest of the classes act more as utility, however, we explicitly bring up some classes and modules that will need your attention before we start:

- The **Index** is an important set of classes that gave Pandas the power to navigate data through different structures in ways ***numpy*** could not. However, there are too many of them, used in a fixed number of ways, but are used in a very broad range. This means a

lot of relationships are present on the UML diagram but carry redundant information. We believe it would be best to ignore the majority of the relationships and just highlight their two way dependency with the **NDFrames** which is that **NDFrames** use indexes to operate on data and indexes can be used to create **NDFrames.**

- The **accessor** classes such as **string** contain important classes and methods that helps developers extend Pandas. They are explained in more detail in the architecture style section.
- The set of **apply** classes are an integral part in achieving the functional programming style in Pandas. It has the same problem as **Index** which is why we will simply mention it.
- **dtypes** and **arrays** contain essential type definition classes and methods used in Pandas. They have the same problem as **Index** and **apply** which is why we will simply mention it.
- **Mixin** style classes are used extensively in classes of Pandas and we will explain them in more detail in the architecture style section. It also has the same problem as **Index** which is why we will only show a few obvious ones as an example.

Now that we have gotten the noisy classes taken care off, we explain the architecture between the three major components of Pandas, **NDFrames(Series, DataFrame)**, **GroupBy** and **Window**.

First of all, we should identify from UML that most of the class inherits the **PandasObject**. Note that though **Window** classes do not have a direct inheritance relationship with **PandasObject**, they are generic **NDFrames** classes who are **PandasObjects**. The most important takeaway from this class is it gives its subclasses the ability to cache certain methods and report that memory usage.

In a brief summary, **DataFrame** and **Series** can transform themselves to each other, be grouped by some index into a GroupBy object, or be converted to a Rolling/Window object for time series analysis. These three data types use their own methods and aggregation methods from **apply** to convert between each other.
Now, let's discuss the **NDFrame**, **Series**, **DataFrame**, **GroupBy** and **Window** one by one:

- **NDFrame** is the abstract base class for Series and DataFrame. With added dependency on the **Index** class and incorporating **Window** and **Resampler**(a time-series specialized GroupBy), it helps define the basis of **Series** and **DataFrame**'s well known ability of Indexing on different types of data, remove missing data, data Transformation, alignment, merge, filter, selection, etc. With the help of ***pandas.io*** classes, we also provide the basics for outputting data inside a frame.
- **Series** is a concrete class extended from **NDFrame** that aims to specialize at a fixed size, same type, 1-D array operations. It depends on **GroupBy** to obtain further capabilities in

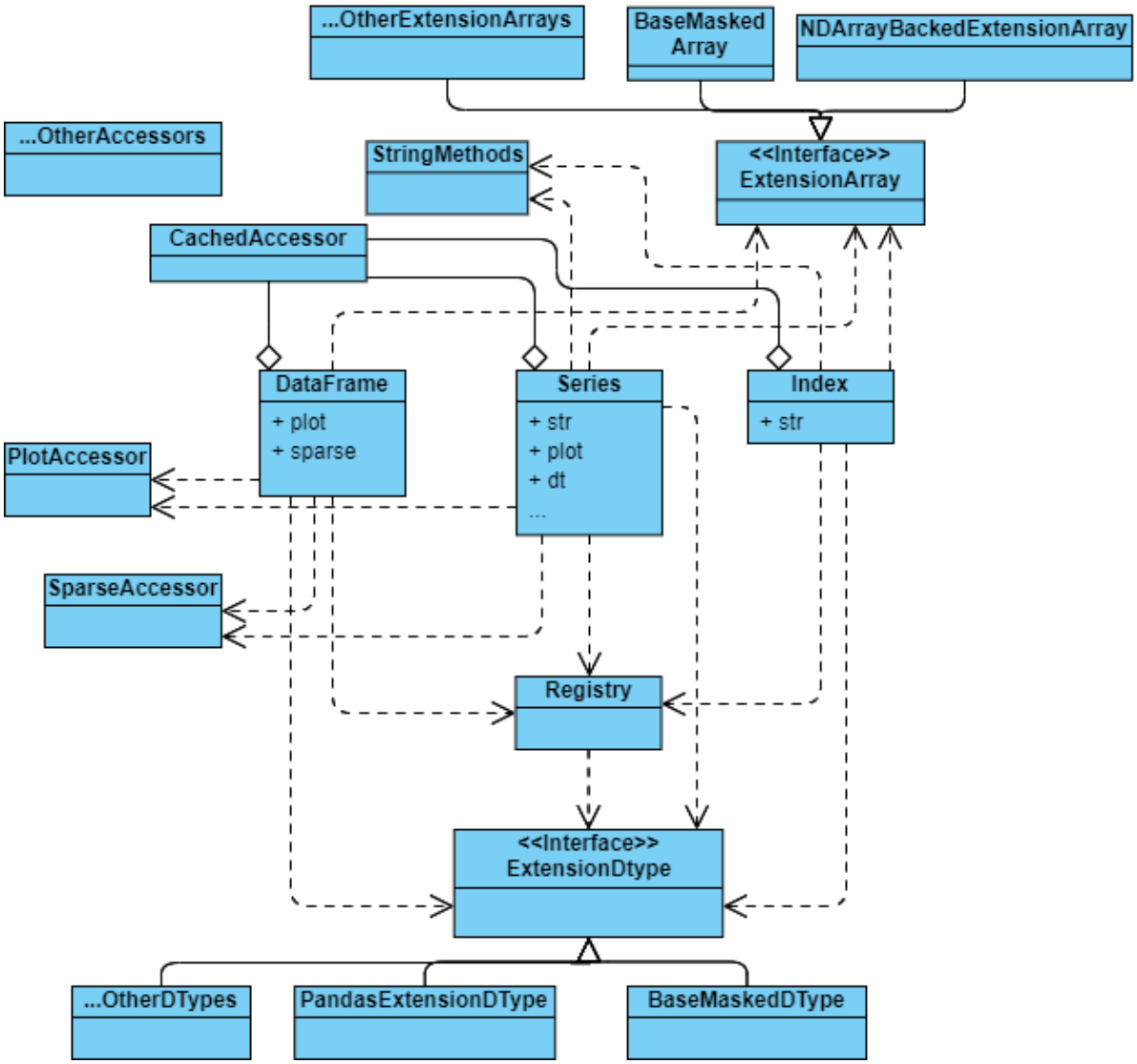
data transformation similar to SQL and using **apply** as a dependency, it is able to create a multitude of data manipulation capabilities from functional programming such as sum() and mean(). **Series** also started aggregating **accessor** to extend its functionality, for example, **StringMethod** accessor allows easy broadcasted string manipulation like lower() if we have a Series of strings.

- **DataFrame** is another concrete class extended from **NDFrame** but aims to specialize in tabular data or 2-D array operations. Different from **Series** is that as we have 2-D data, there exist a lot more complications. First of all, since we have multiple columns/rows, there could be different types instead of one fixed type in **Series**. Due to the type inconsistency, a lot of accessors are not available for **DataFrame**. We are also size-mutable as we can remove rows or columns but **Series** are designed to be fixed size. The general **apply** and aggregations have to be separated into frame_apply which you can specify row/column and applymap which operates the whole frame.
- **GroupBy** module contains two main concrete classes **DataFrameGroupBy** and **SeriesrameGroupBy** that in general aims to group **DataFrame** or **Series** on one or more indexes and perform aggregations to get back another data structure. Similar to **DataFrame** and **Series**, they also depend on the functionalities in **apply** to perform aggregation. They both inherit the GroupBy generic class that provides majority of these functionalities but in a generic sense and also the method resampler() and rolling() to be converted to a time series object. Finally, the GroupBy generic class is the subclass of BaseGroupBy which is a generic abstract base class that provides a very basic interface to apply functions with pipe() and to convert back to a NDFrame using get_group(). It contains the **grouper** attribute which defines how a NDFrame should be grouped.
- **Window** contains many classes of time series analysis but they all follow a similar inheritance pattern using **BaseWindow** and **BaseWindowGroupBy**. **BaseWindow** is a generic abstract class that contains protected methods that allow for windowing time series operation such as _slice_axis_for_step() and _apply_pair_wise(). **BaseWindowGroupBy** is also a generic abstract class but its purpose is to be able to facilitate a grouping operation. It then stores the **grouper** attribute and its protected _apply method is tuned toward **DataFrame** and **Series**. The class **Rolling** then inherits from **BaseWindow** and similarly depends on **apply** to perform analysis but using a rolling window on a DataFrame or Series data. A class like this is usually paired with a **RollingGroupby** class that inherits from **BaseWindowGroupBy** and **Rolling** that aims to do the same rolling window analysis on the grouped data.

Architectural Styles

Extension using Accessor/Decorator and Registry Pattern Style

We have discovered that the Pandas library provides some interesting ways for users to extend its functionalities. These extensions can be roughly separated into two types, decorating data structures with accessors and type extension with registry.



Accessors in Pandas can be seen as decorated namespaces that hold methods designated to a data structure. As you have seen in the Core Architecture diagram, classes hold a list of **_accessors** and as seen in the above diagram, the accessors are attributes in the data structures. They are seen as *decorations* as the adding the accessors do not need any change in the data structure's code. It allows for a more user-friendly, pythonic interface. With these accessors, you can add a bunch of methods into DataFrame or Series without breaking the original functionality. In addition, **CachedAccessor in** [accessor] is an interesting design that accessor method calls are

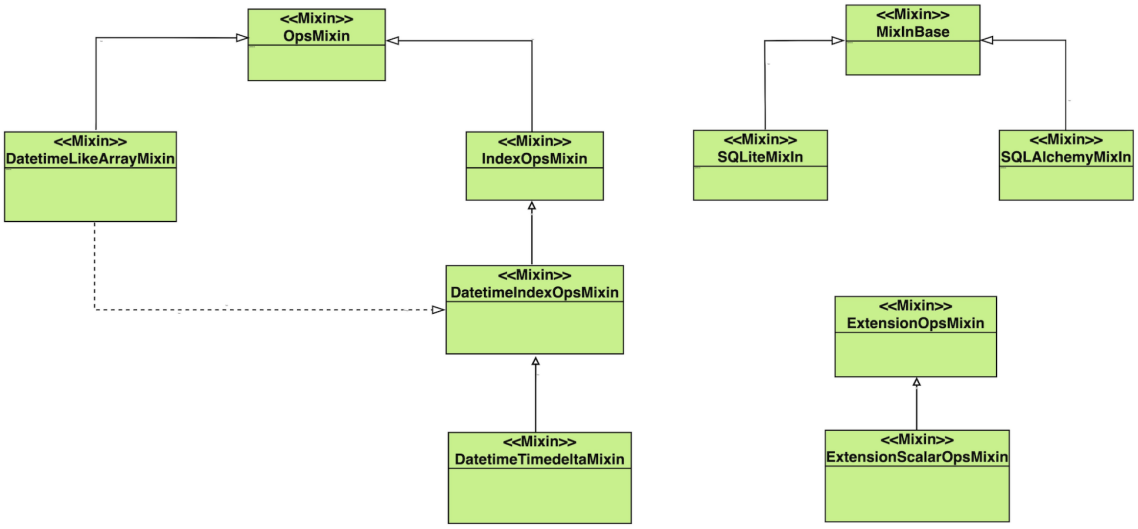
converted into new accessors/namespaces in the class, thus decreasing the computational cost of expensive methods. As of current, all accessors used in DataFrame, Series, and Index are wrapped by a **CachedAccessor** . The custom accessors from users using decorators such as **register_dataframe_accessor** are also converted to attributes using **CachedAccessor**. The UML showcases some common accessors that can be used as reference.

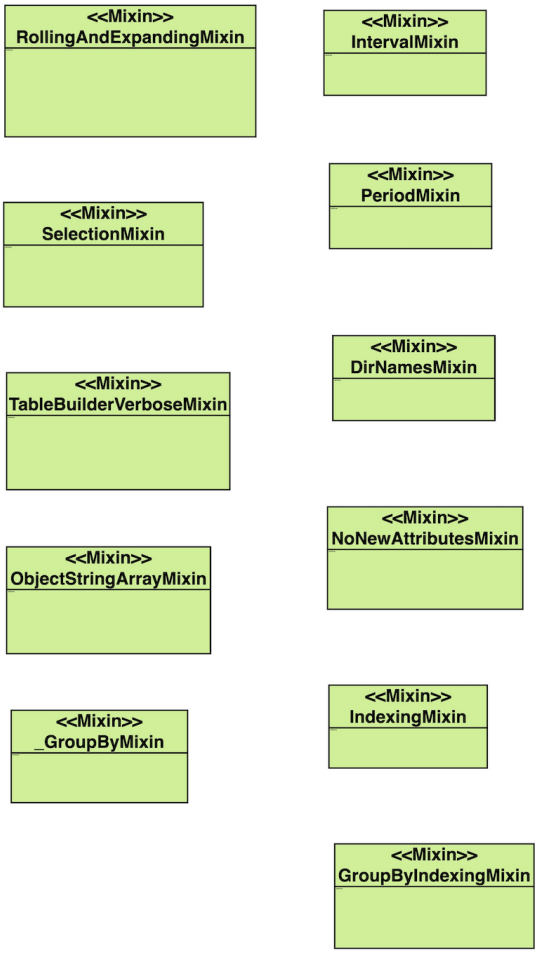
Type extension is an integral part of Pandas as data can take many forms and it is not always the case that Pandas have a standardized type that can handle a complex data entry. Creating your own type was difficulty as DataFrame and Series has a lot of dependencies and some might require certain operators on such type to be implemented. Therefore, Pandas provided **ExtensionDtype** and **ExtensionArray** to solve the issue. They are minimal interfaces that the users need to implement to avoid undefined behaviors. For example, if Series does not recognize an input type as ExtensionDtype, it will try to infer or convert it into simple object data, meaning the outcomes of some operations will not be what you expect. Additionally, by using **register_extension_dtype**, your class that is extending **ExtensionDtype** will be added to a singleton registry that holds all the extended types. This registry is often used by Series or Index methods find data types using strings. On the other hand, **ExtensionArray** does not need this extra step of hustle but it does have more required methods to implement. The UML showcases some common extensions that can be used as reference.

Mixin Style Inheritance

The mixin pattern is used to add additional reusable functionality across different classes without inheritance. In each mixin of Pandas, one or more properties are defined so that users can use them when passing mixin into other classes. Mixin pattern mainly extends functionalities for the

core data structures, such as the **Series**, **Indexing**, and **DataFrame** classes.





We discovered several mixins in Pandas, the relations and details are shown in the picture above. Because of the great redundancy that may appear if we were to mention all Mixins in Pandas, we will put emphasis on and introduce one of the most important and commonly used mixin: **OpsMixin**.

OpsMixin class is a fundamental part of the Pandas library's architecture and plays a critical role in its development. defines a set of methods that implement arithmetic and comparison operations between pandas objects. These methods are designed to work with different data types, and they provide a lot of flexibility in how data can be combined and compared. Below are some methods defined in **OpsMixin** that we have discovered so far:

- **__add__, __radd__**: Implement addition between two objects.
- **__sub__, __rsub__**: Implement subtraction between two objects.
- **__mul__, __rmul__**: Implement multiplication between two objects.
- **__truediv__, __rtruediv__**: Implement true division between two objects.

- **__floordiv__, __rfloordiv__**: Implement floor division between two objects.
- **__mod__, __rmod__**: Implement modulo between two objects.
- **__pow__, __rpow__**: Implement exponentiation between two objects.
- **__eq__, __ne__**: Implement equality and inequality between two objects.
- **__lt__, __le__, __gt__, __ge__**: Implement less than, less than or equal to, greater than, and greater than or equal to between two objects.

To apply these methods to other classes, Mixins can be passed in when defining a class. The methods defined in **OpsMixin** class are used to implement the binary arithmetic and comparison operations between pandas objects.

In addition, there are some inheritance relationships between several mixin classes. For example, in the picture below, the creation of class **IndexOpsMixin** requires **OpsMixin**, and the creation of **DatetimeIndexOpsMixin** requires **IndexOpsMixin**. We take **IndexOpsMixin** and **OpsMixin** as an example.

OpsMixin is a parent class that defines the binary arithmetic and comparison methods for pandas objects like **Series** and **DataFrame**. It is designed to be a mixin class, meaning it is intended to be used as a base class to provide a set of methods that can be mixed in with other classes.

IndexOpsMixin is a mixin class that provides the binary operations and comparison methods for pandas index objects like **Index**, **MultiIndex**, and **DatetimeIndex**. It is designed to be mixed in with other classes, similar to **OpsMixin**.

The **IndexOpsMixin** class provides methods for performing binary operations between index objects, such as union, intersection, and difference. It also provides methods for comparison operations between index objects, such as checking for equality, inequality, and containment.

Both **OpsMixin** and **IndexOpsMixin** provide a flexible framework for performing binary operations and comparison methods on pandas objects. They are designed to work together to provide a consistent and efficient set of methods for pandas users.

Functional Programming Style / Method Chaining

Pandas also uses a functional programming approach. It seems the main goal is to reduce side effects, allow for low coupling, and allow for test cases to be written easier and faster. Both groupby() and rolling() are examples of this style of functional programming. Pandas also uses pipes (it implements its own pipes() function) when wanting to chain methods together instead of using functions as arguments. Pandas' rule of using .pipe(), .apply(), and .map() for chaining together methods that expect Series, DataFrames or GroupBy objects means that all data frames

are immutable as a result, and generally adhere to functional programming philosophies. Another interesting observation is that Pandas does not only implement a functional programming style as Pandas also allows for subclassing of its data structure when needing to implement new functionalities. However, while allowed, subclassing is not encouraged as Pandas tend to use composition instead of inheritance (subclassing) to reduce coupling. Another interesting property of the pandas library is that its data structures are separate from its operators.

Possible Improvements

As we have seen in our previous discussions and diagrams in the core architecture, Pandas library exhibits quite a high level of coupling between the core classes. This means subclassing the DataFrame or Series can be very difficult for new developers since their core behaviours are relied on by other classes such as GroupBy and Window. However, it might not be worth to improve it as Pandas also aims to provide easy user api experience, which means changing the coupling can cause the user api to be less cohesive and harder to use.

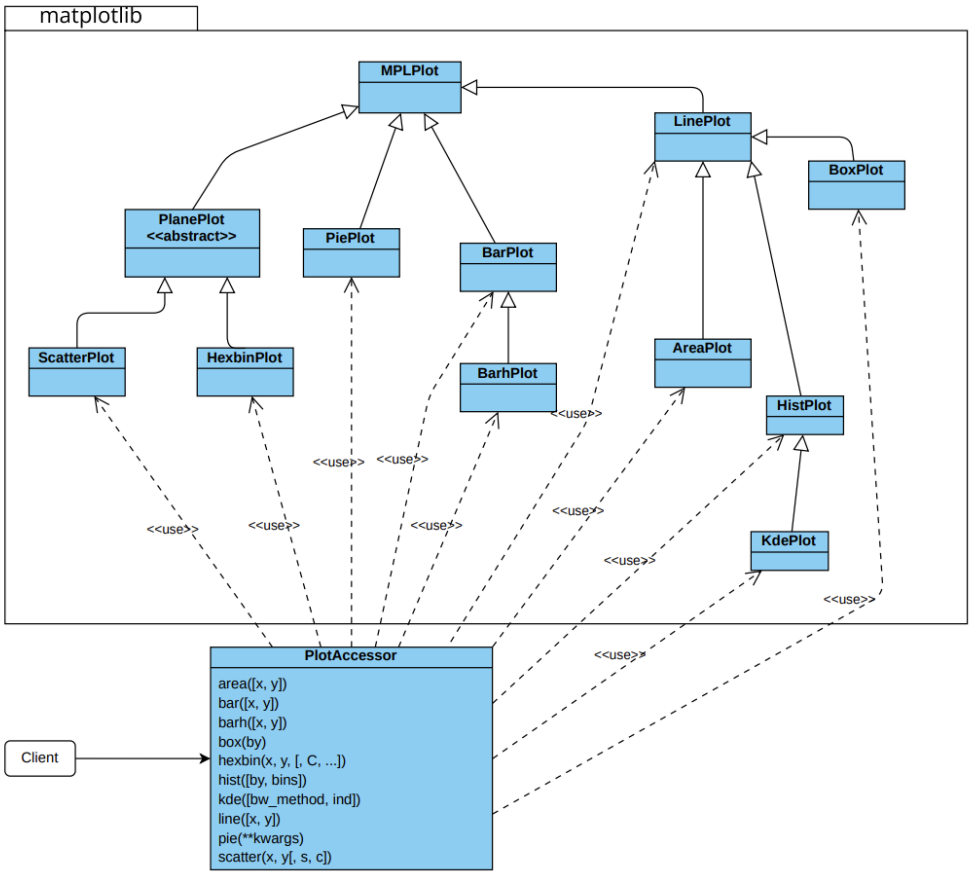
The Caching as talked about in the architectural style section is a big part of what made Pandas so fast, but it is also what made pandas consume an incredible amount of memory. Similar to Indexing, each dataframe needs its own set of caches and this can blow the memory usage up to huge proportions given a large dataset. We think this design should not be turned on by default of the DataFrame or Series. Accessor Extensions to the class should have the option to not use the CachedAccessor to avoid clumsy computation management.

2.2 Design Patterns

Design Pattern 1: Facade

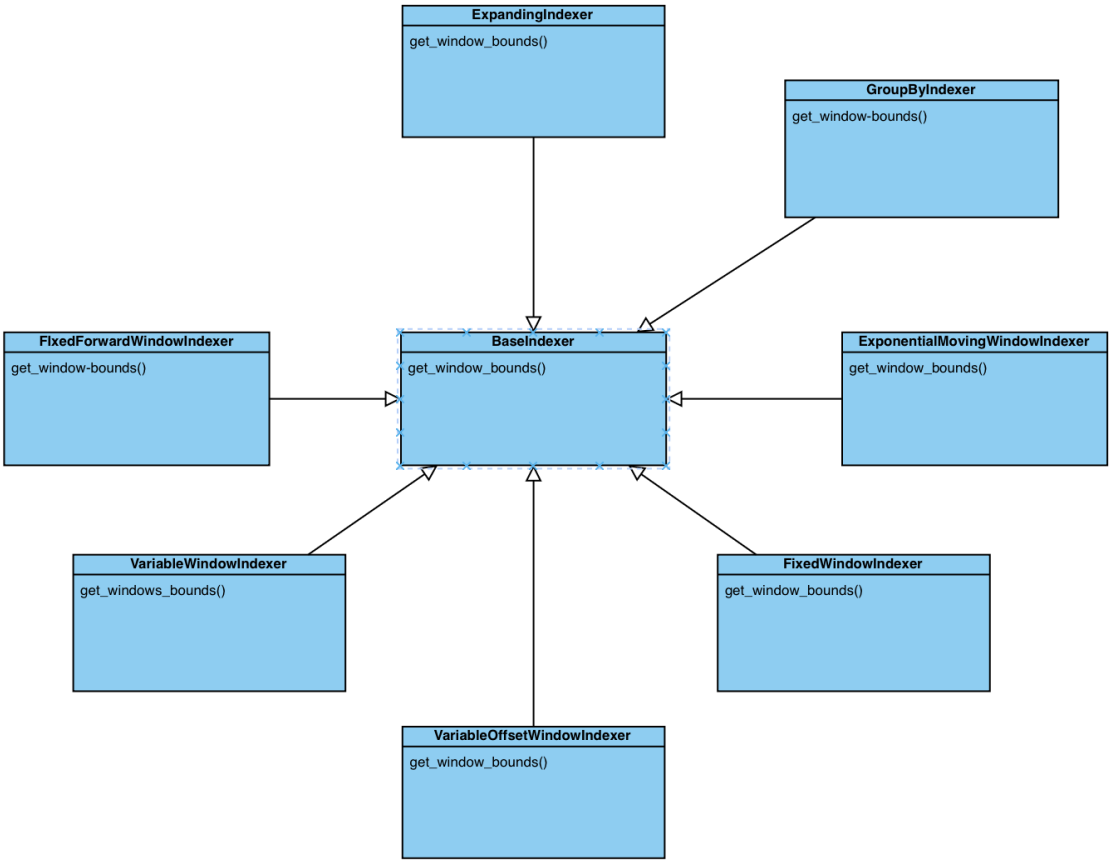
The class PlotAccessor is a facade that provides the plotting methods to a complex package matplotlib. A string attribute “kind” is used to differentiate which type of plot the client wants to draw, and this string will be passed in while calling package matplotib. Inside the folder _matplotlib, there is an _init_.py file that defines a global dictionary PLOT_CLASSES (line 44). Each plot class is a value that has a string key referred to it. These keys matched with the “kind” filed in PlotAccessor. For example, when operation line() (line 1050) is called by the client, “kind” filed will be set to string “line”. “line” is the key for LinePlot (line 45) in PLOT_CLASSES. Moreover, the parameter “**kwarg” is given by the client and passed into the package.

This is a classical facade data structure because the user only interacts with the PlotAccessor class in order to plot the graph without any knowledge of how matplotlib works internally.



Corresponding code:
pandas/pandas/plotting/_core.py: class PlotAccessor
pandas/pandas/plotting/_matplotlib: package matplotplib and all Plot classes

Design Pattern 2: Strategy



All classes mentioned above are located in pandas/core/indexers/objects.py

The class **"BaseIndexer"** defines the abstract method "get_window_bounds" that is needed to be implemented by all its subclasses. The class is defined in lines 43 - 71, and the method is defined in lines 61 - 71.

> The subclass **"FixedWindowIndexer"** is defined in lines 74 - 102, and the concrete implementation of the method "get_window_bounds" is defined in lines 77 - 102.
>
> The subclass **"VariableWindowIndexer"** is defined in lines 105 - 129, and the concrete implementation of the method "get_window_bounds" is defined in lines 108 - 129.
>
> The subclass **"VariableOffsetWindowIndexer"** is defined in lines 132 - 217, and the concrete implementation of the method "get_window_bounds" is defined in lines 147 - 217.
>
> The subclass **"ExpandingWindowIndexer"** is defined in lines 220 - 236, and the concrete implementation of the method "get_window_bounds" is defined in lines 223 - 236.

The subclass **"FixedForwardWindowIndexer"** is defined in lines 239 - 288, and the concrete implementation of the method "get_window_bounds" is defined in lines 264 - 288.

The subclass **"GroupByIndexer"** is defined in lines 291 - 382, and the concrete implementation of the method "get_window_bounds" is defined in lines 330 - 382.

The subclass **"ExponentialMovingWindowIndexer"** is defined in lines 385 - 398, and the concrete implementation of the method "get_window_bounds" is defined in lines 388 - 398.