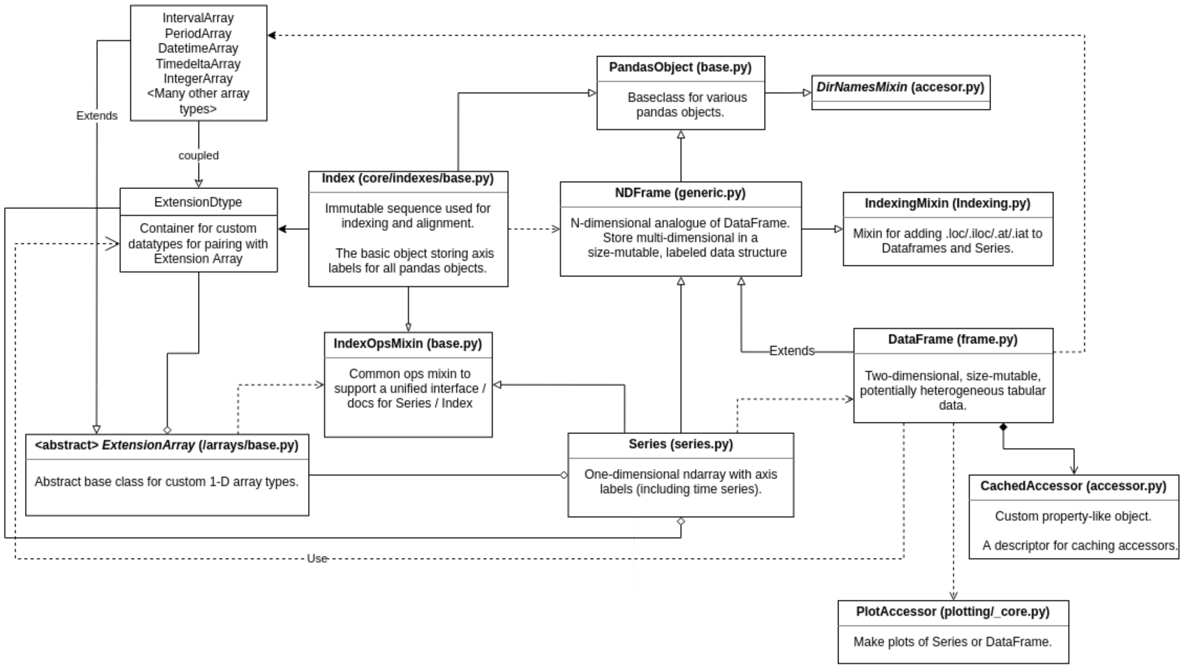
Deliverable 2

Sunday February 19, 2023

Members
Ali Abdoli
Seyon Kuganesan
Kyle Lewis
Youzhang Sun
Leo (Si) Wang
Glenn Ye
Daniel Zhang

System Architecture

Class Interactions



Detailed Class Diagrams

DataFrame (frame.py)

Attributes: data, index, columns, dtype, copy

Methods
+ dot(self, DataFrame | Index | ArrayLike): DataFrame
+ insert(self, int, Hashable, Scalar | AnyArrayLike, bool | lib.NoDefault)
+ align(self, DataFrame, AlignJoin, Axis, Level, bool, None, FillnaOption, int, Axis, Axis)
+ bfill(self, *, Axis | None, bool, int | None, *)

Description
- conversion to and from other arraylike / dict / dataframe objects from various libraries and other various file type objects (e.g. HTML, MarkDown, XML) (inherited)
- indexing/iteration (inherited)
- dot product and other matrix multiplications with Series, DataFrame or arraylike structures
- data querying, insertions and other various SQL like operations (inherited)
- data reindexing & alignment
- handling for missing and duplicate data
- dataframe combining and updating using another
- statistical methods upon data

IndexOpsMixin (base.py)

__array_priority__ : int

+ size(): int
+ transpose(): IndexOpsMixin
+ T(): IndexOpsMixin
+ shape(): tuple
+ ndim(): int
+ array(): ExtensionArray
+ max(bool) : any
+ min(bool): any
+ to_list(): list
# __iter__(): Iterator
+ nunqiue(): int
+ value_counts (bool, bool, bool, any, bool): Series
+ unique (): ExtensionArray

<abstract> *ExtensionArray* (/arrays/base.py)

+ dtype:ExtensionDType
+ nbytes: int
+ ndim: int
+ shape: tuple
+ size: int

- _from_sequence (Sequence, Union(bool, int, str, complex, numpy.dtype, ExtensionDType) | None, bool): ExtensionArray
- _from_factorized (numpy.ndarray, ExtensionArray)
__getitem__(ExtensionArray, tuple) : ExtensionArray |any
__len__(): int
__eq__(any): Union(ExtensionArray, np.ndarray)
+ dtype(): ExtensionDType
+ nbytes():int
+ isna (): np.ndarray | ExtensionArray
+ take (Sequence, bool, any) : ExtensionArray
+ copy (ExtensionArray): ExtensionArray
- _concat_same_type (Sequence[ExtensionArray] ExtensionArray
+ insert (int, any): ExtensionArray
+ unique (): ExtensionArray

Series (series.py)

+ data: array-like | Iterable | dict | scalar value
- _name - Hashable
+ dtype: numpy.dtype | ExtensionDType
+ copy: bool
- _values: ExtensionArray

Methods
+ sort_values(Union([str, int], bool | int | Sequence[bool] | Sequence[int], bool, str, str, bool, any): Series | None
+unqiue (): Union[ExtensionArray, np.ndarray]
+ apply (any, bool, tuple): Series | DataFrame
+ map (Callable | Mapping | Series, Literal["ignore"] | None): Series
+ drop_duplicates(Literal["first", "last", False], bool, bool): Series
- _reduce(any, str, [str, int], bool, bool): any

Description
- One-dimensional array with axis label
- Integer / label-based indexing
- Statistical methods modified to cope with missing data

IndexingMixin (Indexing.py)

Description: Mixin for adding .loc/.iloc /.at/.iat to Dataframes and Series.

Methods

+iloc() - integer based indexing for selection by position

+loc() - Access a group of rows and/or columns by label(s) or a boolean array

+at() - Access a single value for a row/column label pair

+iat() - Access a single value for a row/column pair by integer position

NDFrame (generic.py)

__hash__ : ClassVar
+ axes: List[Index]
+ attrs: dict[Hashable : any]

+ drop(Hashable, Union([str,int]), Hashable | Sequence[Hashable], Hashable | Sequence[Hashable], Union([Hashable,List]), Literal[True])
+ get(object, Union([ExtensionDType, np.dtype] | None): Union([ExtensionDType, np.dtype])
+ head(): NDFrame
+ tail(): NDFrame
# __len__(): int
+ pop(): NDFrame
# __contains__(any) : bool
+ describe(any,any,any) : NDFrame
+ filter(any, str | None, str | None, [str,int] | None): NDFrame
- _min_count_stat_function(str, any, [str,int] | None, bool, bool, int):
+ sum([str, int] | None, bool | None, bool | None, int):

Index (core/indexes/base.py)

+ data: Union[ExtensionArray, np.ndarray]
+dtype: object
+name:object
+copy: bool
tupleized_col: bool

+ append(Index | Sequence[Index]): Index
- _concat(List[Index], Hashable): Index
+ delete(Index, int | List[int]): Index
+ insert (int, object): Index
+ unqiue(Hashable | None): Index
+ sort_values (bool, bool, str, Callable): Index

***DirNamesMixin* (accesor.py)**

__dir__(): any

PandasObject (base.py)

__size_of__(): int
#__repr__(): str

PlotAccessor (plotting/_core.py)

+ data: Series | DataFrame
+ kind : str

- _load_backend(str): ModuleType
- _get_plot_backend(str): ModuleType
__call__(): any (calls _get_plot_backend() to load module, then gets that module to plot)

CachedAccessor (accessor.py)

+ field: _name, _accessor

+ method(): __get__

Array Like types

DatetimeArray, ExtensionArray, PeriodArray, TimedeltaArray

Commentary

Although the system architecture of Pandas seems complex at first glance, after spending some time mapping out the interactions and behaviours of each class, we get to see various architectural patterns arise.

One such pattern that Pandas uses is the Layered pattern, where the software is divided into layers and each class is assigned to a layer. Furthermore, Pandas uses an open layered architecture, thus classes are able to use services from any of the lower layers.

It was also observed that the Pandas library has a relatively low degree of coupling. We found this to be the case since the library is modular and many of its components can be used independently of one another. For example, it is possible to use data structures in Pandas (i.e. DataFrames, Series, etc.) in conjunction with other data analysis libraries and tools, or to use the Pandas I/O functions to read and write data to and from files without necessarily using other parts of the library.

One improvement that could be made to Pandas is increased extensibility. There could be new interfaces that define various functionalities that open-source developers would like to implement, which would allow their code to integrate with the core Pandas functionality seamlessly. This would have several benefits, such as allowing users to easily add new data sources or data manipulation functions to Pandas, without having to modify the core library code.

Design Patterns

1. Factory Method

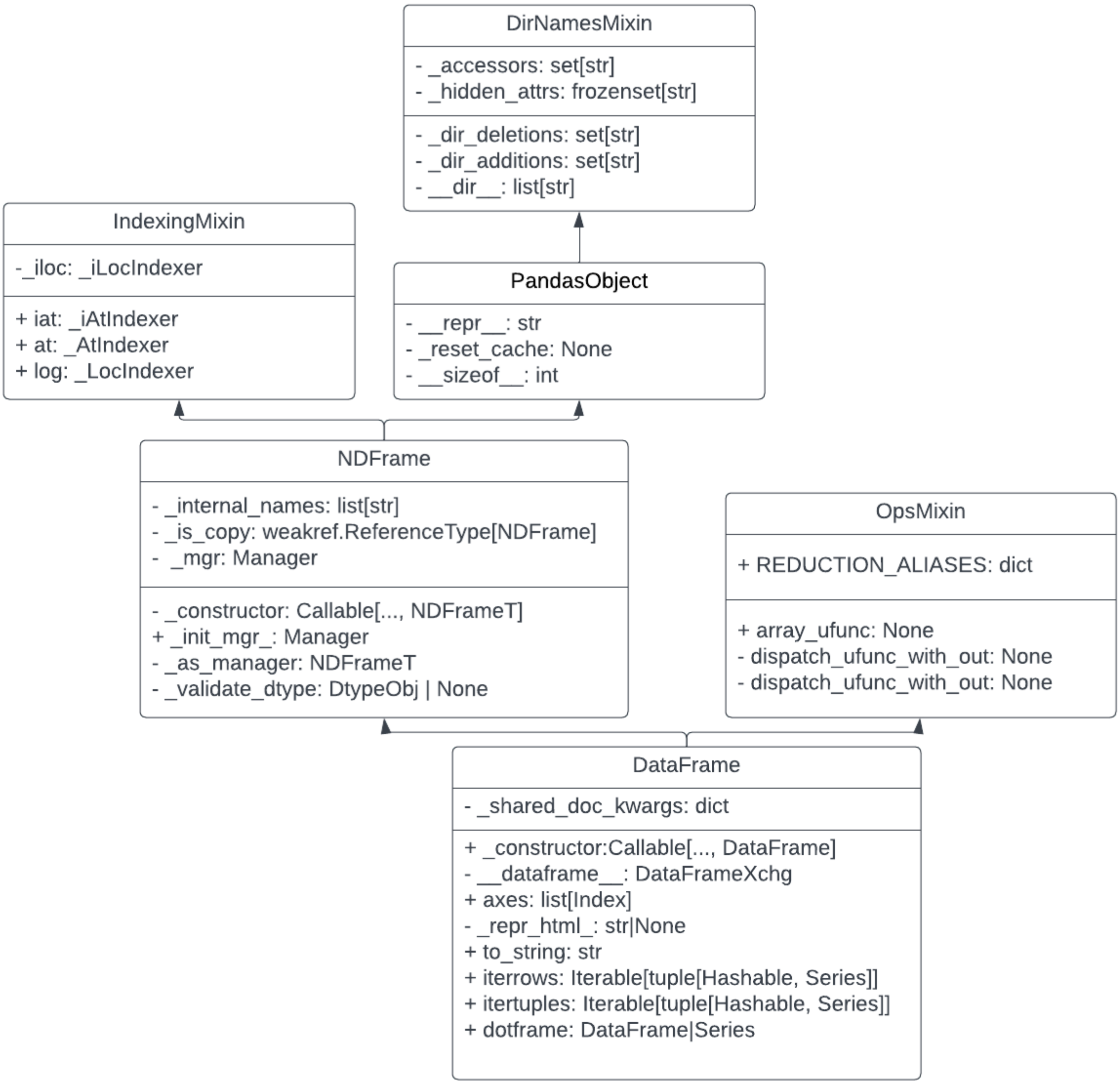
Path to file: pandas/core/frame.py

Example

This pattern can be found in the `Dataframe` constructor, which was between lines 641-846, within the file specified above. This constructor is one of the main entry points for creating a `Dataframe` object in Pandas, and it acts as a factory method that creates `Dataframe` objects from various data structures, like `NumPy` arrays, dictionaries, and lists. The constructor uses the **Factory Method** pattern by providing a common interface for creating `Dataframe` objects, while allowing the underlying implementation to change based on the type of data structure being passed to it (which fits the definition of the factory design pattern). This can be seen in the following lines of code within the constructor (lines 693-710), although similar conditionals can be seen throughout the entirety of the constructor:

```
if isinstance(data, (BlockManager, ArrayManager)):

   mgr = self._init_mgr(data, axes={"index": index, "columns": columns},
dtype=dtype, copy=copy)

elif isinstance(data, dict):

   mgr = dict_to_mgr(data, index, columns, dtype=dtype, copy=copy, typ=manager)

elif isinstance(data, ma.MaskedArray):

   from numpy.ma import mrecords

   # masked recarray

   if isinstance(data, mrecords.MaskedRecords):

       raise TypeError(

           "MaskedRecords are not supported. Pass "

           "{name: data[name] for name in data.dtype.names} "

           "instead"

       )
```

UML Diagram & Explanation



The UML diagram shows the `Dataframe` class, which acts as the superclass, and the `__init__` method, which acts as the factory method. The `__init__` method serves as the bridge between the client and the object creation code, allowing the client to create `Dataframe` objects without knowing the specific implementation that will be used, which helps inform our decision on why this is an example of the factory design pattern. The `__init__` method uses the `isinstance` function to check the type of data being passed to the constructor, and based on the type, it creates a `Dataframe` object using the appropriate implementation.

2. Builder Pattern

Path to file: pandas/io/formats/latex.py

Example

The builder design pattern is used to differentiate between table formats that could be displayed in LaTeX. `GenericTableBuilder` builds the `header`, `top_separator`, `middle_separator` and `env_body`, which are the common elements of all the tables. `LongTableBuilder`, `RegularTableBuilder`, and `TabularBuilder` implement the `env_begin`, `bottom_separator`, and `env_end` of the table. `LongTableBuilder` also overwrites the `middle_separator`.

All the tables are built the same way as they inherit `TableBuilderAbstract` `get_result()` method which returns the string format of a Latex table.

```
def get_result(self) -> str:
        """String representation of LaTeX table."""
        elements = [
                self.env_begin,
                self.top_separator,
                self.header,
                self.middle_separator,
                self.env_body,
                self.bottom_separator,
                self.env_end,
        ]
        result = "\n".join([item for item in elements if item])
        trailing_newline = "\n"
        result += trailing_newline
        return result
```

Lines 353-365

`LatexFormatter` creates the correct builder depending on the options given to it in its constructor, and returns it on a `self.builder` call. `get_result()` can then be called on said builder to get the string output of the table LaTeX code. Hence this is why the `to_string` method of `LatexFormatter` is simply:

```
def to_string(self) -> str:
    return self.builder.get_result()
```

Line 713, 718

Builder can be easily used here since all the table formats have the same structure, but they simply output a different kind of string.

The client can call an individual builder directly:

```
"""
>>> from pandas.io.formats import format as fmt
>>> df = pd.DataFrame({"a": [1, 2], "b": ["b1", "b2"]})
>>> formatter = fmt.DataFrameFormatter(df)
>>> builder = TabularBuilder(formatter, column_format='lrc')
>>> table = builder.get_result()
>>> print(table)
"""
```

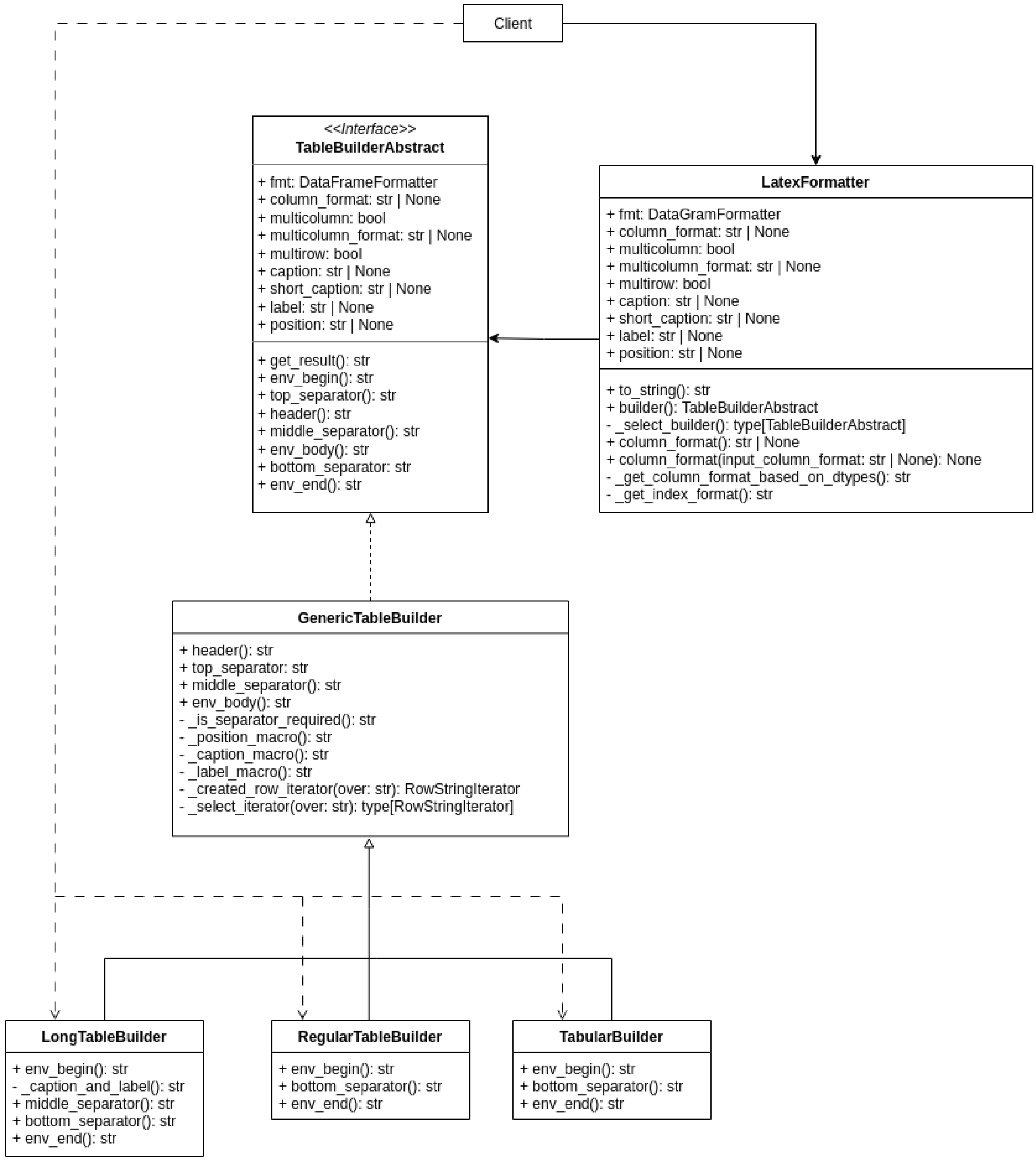
Lines 627-632

They can also use `LatexFormatter` with the option of longtable for `LongTableBuilder` or giving caption, label or position for `RegularTableBuilder`:

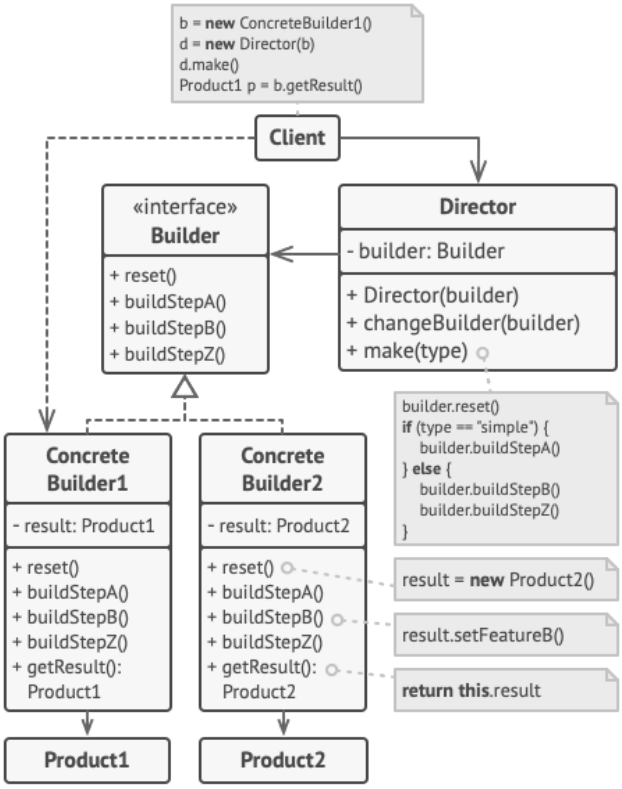
```
"""
>>> from pandas.io.formats import format as fmt
>>> df = pd.DataFrame({"a": [1, 2], "b": ["b1", "b2"]})
>>> formatter = fmt.DataFrameFormatter(df)
>>> latexFormatter = LatexFormatter(formatter, column_format='lrc',
longtable=True)
>>> table = latexFormatter.to_string()
>>> print(table)
"""
```

Self-created

UML Diagram & Explanation



The `TableBuilderAbstract` is an **interface** that defines all the variables a LaTeX table builder must have, and any functions it must implement, except for `get_result()`, which is already implemented. GenericTableBuilder inherits from `TableBuilderAbstract` and implements all necessary interface functions plus additional functions that are common to all **concrete builder** classes. Concrete builders provide different implementations of TableBuilderAbstract, and can be called by the **client** to be used. In this case, these concrete builders are `LongTableBuilder`, `RegularTableBuilder`, and `TabularBuilder`, all of which inherit from `GenericTableBuilder`. `LatexFormatter` is the **director** class, since it allows the client to choose any builder inherited from `TableBuilderAbstract` and get the result: LaTeX code of the table they want to make.



Generic builder design pattern (https://refactoring.guru/design-patterns/builder)

Overall, the existence of the bolded class types in the explained structure demonstrates the existence of the builder design pattern.