piso

Riley Clement

Jan 29, 2022

CONTENTS

Date: Jan 29, 2022 **Version**: 0.8.0

Useful links: Source Repository | Issues & Ideas

Pandas Interval Set Operations: methods for set operations, analytics, lookups and joins on pandas' Interval, IntervalArray and IntervalIndex

CHAPTER

ONE

GETTING STARTED

1.1 Installation

piso can be installed from PyPI or Anaconda.

To install the latest version from PyPI:

```
python -m pip install piso
```

To install the latest version through conda-forge:

```
conda install -c conda-forge piso
```

1.2 Package overview

piso exists to bring set operations (union, intersection, difference + more), analytical methods, and lookup and join functionality to `pandas` interval classes, specifically

- `pandas.Interval`
- `pandas.arrays.IntervalArray`
- `pandas.IntervalIndex`

Currently, there is a lack of such functionality in *pandas*, although it has been earmarked for development. Until this eventuates, *piso* aims to fill the void. Many of the methods can be used via accessors, which can be registered to `pandas.arrays.IntervalArray` and `pandas.IntervalIndex` classes.

An array of intervals can be interpreted in two different ways. It can be seen as a container for intervals, which are sets, or if the intervals are disjoint it may be seen as a set itself. Both interpretations are supported by the methods introduced by *piso*.

The domain of the intervals can be either numerical, `pandas.Timestamp` or `pandas.Timedelta`. Currently, most of the set operaitons in *piso* are limited to intervals which:

- have a non-zero length
- have a finite, length
- are left-closed right-open, or right-closed left-open

To check if these restrictions apply to a particular method, please consult the *API reference*.

Several *case studies* using *piso* can be found in the *user guide*. Further examples, and a detailed explanation of functionality, are provided in the *API reference*.

1.3 Versioning

SemVer is used by *piso* for versioning releases. For versions available, see the tags on this repository.

1.4 License

This project is licensed under the MIT License:

1.5 Acknowledgments

Currently, *piso* is a pure-python implentation which relies heavily on `staircase` and `pandas`. It is clearly designed to operate as part of the *pandas ecosystem*. The colours for the piso logo have been assimilated from pandas as a homage, and is not to intended to imply and affiliation with, or endorsement by, pandas.

Additionally, two classes have been borrowed, almost verbatim, from the pandas source code:

- `pandas.util._decorators.Appender`
- `pandas.core.accessor.CachedAccessor`

CHAPTER

TWO

USER GUIDE

2.1 Intervals and sets

Below is a brief discussion on the mathematical definition of intervals and sets, and how they relate to *piso* - aside from making up half of the acronym!

2.1.1 Sets

A set is a collection of elements where each element, or member, of the set is unique, i.e. a set does not contain duplicated elements. There may be an infinite number of members, such as the set of positive integers, or it could be empty. There is both standard notation and standard operations for sets.

For example:

- $A = \{1, 2, 3\}$ is a set containing three numbers
- $B = \{3, 4\}$ is a set containing two numbers

The union of these sets, $A \cup B$, is the set containing all elements in A and B. That is, $A \cup B = \{1, 2, 3, 4\}$.

The intersection of these sets, $A \cap B$, is the set containing elements both in A and B. That is, $A \cap B = \{3\}$.

The difference of A and B = $A \setminus B$, is the set containing all elements in A but not in B. That is, $A \setminus B = \{1, 2\}$.

The symmetric difference of these sets, $A \Delta B$, is the set containing all elements in A and in B, but not in both. That is, $A \Delta B = \{1, 2, 4\}$. Symmetric difference of sets is equivalent to the difference between the union and the intersection.

Python is no stranger to set operations, with the `set` class being a built-in data structure (for sets with a finite number of elements) with the following methods:

- `set.union`
- `set.intersection`
- `set.difference`
- `set.symmetric_difference`
- `set.isdisjoint`
- `set.issubset`
- `set.issuperset`

Set operations, for sets with a finite number of elements, also exist for `pandas.Index`:

- `pandas.Index.union()`
- `pandas.Index.intersection()`

- pandas.Index.difference()
- pandas.Index.symmetric_difference()

To continue a gentle introduction to sets, please refer to math is fun.

2.1.2 Intervals

An interval is a set of (real) numbers that contains all real numbers lying between any two numbers (endpoints). The notion of an interval can be applied to other domains in which a "total ordering" exists, such as time-related domains modelled by pandas.Timestamp and pandas.Timedelta.

Intervals are classified as being open, closed, or half-open. An open interval does not include its endpoints. For example, the set of numbers between 0 and 1 (but not including 0 and 1) is an open interval. In set notation it is written as $\{x|0 < x < 1\}$. In interval notation it is written as $(0, 1)$. A closed interval contains both of its endpoints, while a half-closed interval contains only one of its endpoints. The notation is as follows:

- $[0, 1] = \{x|0 \leq x \leq 1\}$ (closed)
- $[0, 1) = \{x|0 \leq x < 1\}$ (left-closed)
- $(0, 1] = \{x|0 < x \leq 1\}$ (right-closed)

The length of an interval is defined by subtracting the smaller end point from the larger. Intervals could have an infinite length, such as the set of numbers greater than zero, or they could have zero length such as the set containing a single number (known as a degenerate interval).

The definitions of set operations, outlined above, follow through to intervals, however the result of set operations with intervals may not be an interval - but it will be a set! For example,

- $[0, 2] \cup [1, 3] = \{x|1 \leq x \leq 3\} = [0, 3]$
- $[0, 2] \cap [1, 3] = \{x|1 \leq x \leq 2\} = [1, 2]$
- $[0, 2] \setminus [1, 3] = \{x|0 \leq x < 1\} = [0, 1)$
- $[0, 2] \Delta [1, 3] = \{x|0 \leq x < 1 \text{ or } 2 < x \leq 3\} = [0, 1) \cup (2, 3]$

The result in the last of these examples above cannot be expressed as an interval. It can however be expressed as the union of two disjoint (non-overlapping) intervals. Modelling intervals in pandas is facilitated through pandas.Interval, while representing the union of disjoint intervals can be achieved through an interval array such as pandas.arrays.IntervalArray or pandas.IntervalIndex. The intervals contained in one of these interval arrays do not have to be disjoint, so with respect to set operations an interval array can be interpreted in one of two ways:

a) a collection of intervals, which become operands in a set operation, or

b) a set itself, formed by the union of disjoint intervals, and used as an operand in a set operation.

An example of a) is applying an intersection operation to a interval array containing the intervals

$$[0, 5), [4, 6), [7, 9), [8, 12)$$

which results in an in interval array containing the intervals

$$[4, 5), [8, 9)$$

An example of b) is applying an intersection operation to two interval arrays (of disjoint intervals)

$$[0, 5), [7, 9) \quad \text{and} \quad [2, 3), [6, 8)$$

which results in an in interval array containing

$$[0, 2), [3, 5), [8, 9)$$

Both of these interpretations are supported by methods in *piso*. The methods will switch interpretations depending on the number of interval array arguments supplied to the methods. Note that if a interval array is used as an operand (as shown in the example for b) above) then any overlapping intervals will be merged to create a set of disjoint intervals before the operation is applied.

It is important to note that *piso* does not support all types of intervals. Specifically, intervals must have a length which is non-zero and finite. It must be either left-closed, or right-closed. Any `pandas.Interval`, `pandas.IntervalIndex` and `pandas.array.IntervalArray` arguments supplied to *piso* methods must have the same value for their *closed* attribute (either "left" or "right").

For code examples involving *piso* set operations please see the small *case study* or examples provided in the *API reference*.

2.2 The piso accessors

Applying set operations to interval array objects: `pandas.arrays.Interval` and `pandas.IntervalIndex` can be done with top-level functions:

- *piso.union()*
- *piso.intersection()*
- *piso.difference()*
- *piso.symmetric_difference()*

The exact same functionality is available through accessors on these classes. Using accessors allows us to extend the functionality associated with these classes, without adding the methods directly. Before the accessors can be used they must be registered:

```
In [1]: import piso

In [2]: piso.register_accessors()
```

Registering the accessors adds a *piso* property to the classes which can be used like so:

```
In [3]: arr = pd.arrays.IntervalArray.from_tuples(
   ...:     [(1,5), (3,6), (2,4)]
   ...: )
   ...:

In [4]: arr.piso.intersection()
Out[4]:
<IntervalArray>
[(3, 4]]
Length: 1, dtype: interval[int64, right]
```

Further examples using the *piso* accessors can be found in the *API reference*.

2.3 Case studies

Finding common gaps in daily calendars

This case study introduces the use of *piso* for set operations such as *piso.intersection()* and *piso.union()* and applies it to an example where personal calendars are represented by interval arrays.

Verifying a maintenance schedule

This case study introduces the use of *piso* for analysis with functions that return scalars, such as *piso.issuperset()* and *piso.coverage()*. In this example maintenance schedules and windows of opportunity are represented by interval arrays.

Estimating tax payable

This case study demonstrates the use of *piso.lookup()* where tax brackets are represented by a pandas.DataFrame, indexed by a pandas.IntervalIndex. The tax payable for an array of income values is calculated by efficiently finding the corresponding tax brackets.

Analysis of scores in a football match

This case study introduces the idea of *joins* using pandas.IntervalIndex. Using *piso.join()* a dataframe is constructed, indexed by intervals for unique score combinations in the 2009 Chelsea vs Liverpool Champions League quarter-final.

2.3.1 Finding common gaps in daily calendars

In this example we are given the following scenario:

> Alice, Bob, and Carol are busy people, after all they appear in their fair share of computing examples. They wish to meet on the 5th of October 2021 to discuss their next appearance. Given calendar data which details their scheduled meetings, find the gaps during business hours (9am to 5pm) that they can meet.

We start by importing pandas and *piso*

```
In [1]: import pandas as pd

In [2]: import piso
```

Running the *piso.register_accessors()* function will add "piso" accessors to pandas.IntervalIndex and pandas.arrays.IntervalArray. Using accessors allows us to extend the functionality associated with these classes, without adding the methods directly.

```
In [3]: piso.register_accessors()
```

Next we load the data from a csv file and store it into a pandas.DataFrame. Each row of the dataframe corresponds to a meeting that has been booked for the 5th of October, and is characterised by the attendee, a start time and an end time.

```
In [4]: data = pd.read_csv("./data/calendar.csv", parse_dates=["start", "end"])

In [5]: data
Out[5]:
     name               start                 end
0   Alice 2021-10-05 09:00:00 2021-10-05 10:00:00
1   Alice 2021-10-05 10:00:00 2021-10-05 11:00:00
2   Alice 2021-10-05 11:30:00 2021-10-05 12:30:00
3   Alice 2021-10-05 14:30:00 2021-10-05 15:30:00
4     Bob 2021-10-05 10:00:00 2021-10-05 10:30:00
5     Bob 2021-10-05 10:30:00 2021-10-05 11:30:00
6     Bob 2021-10-05 11:00:00 2021-10-05 11:30:00
7     Bob 2021-10-05 16:00:00 2021-10-05 17:00:00
8   Carol 2021-10-05 09:30:00 2021-10-05 10:00:00
9   Carol 2021-10-05 09:30:00 2021-10-05 10:30:00
10  Carol 2021-10-05 12:30:00 2021-10-05 13:30:00
11  Carol 2021-10-05 16:30:00 2021-10-05 17:30:00
```

This data is reasonably readable however to work with *piso* we need it in interval arrays. The following code creates a `pandas.Series`, indexed by the names Alice, Bob and Carol, and where the values are instances of `pandas.arrays.IntervalArray` and hold the data corresponding to each person.

```
In [6]: meetings = (
   ...:     data
   ...:     .groupby("name")
   ...:     .apply(
   ...:         lambda df: pd.arrays.IntervalArray.from_arrays(
   ...:             df["start"],
   ...:             df["end"],
   ...:             closed="left",
   ...:         ),
   ...:     )
   ...: )
   ...:

In [7]: meetings
Out[7]:
name
Alice    [[2021-10-05 09:00:00, 2021-10-05 10:00:00), [...
Bob      [[2021-10-05 10:00:00, 2021-10-05 10:30:00), [...
Carol    [[2021-10-05 09:30:00, 2021-10-05 10:00:00), [...
dtype: object
```

We define a method *print_intervals* which is designed to make it easy for us to display interval array data. It prints a heading, then each interval in an array.

```
In [8]: def print_intervals(header, interval_array):
   ...:     print(header)
   ...:     print("--------------")
   ...:     for interval in interval_array:
   ...:         print(interval)
   ...:     print()
   ...:
```

Let's see this method in action by printing the meeting times for each person.

```
In [9]: for person in ("Alice", "Bob", "Carol"):
   ...:     print_intervals(person, meetings[person])
   ...:
Alice
-------------
[2021-10-05 09:00:00, 2021-10-05 10:00:00)
[2021-10-05 10:00:00, 2021-10-05 11:00:00)
[2021-10-05 11:30:00, 2021-10-05 12:30:00)
[2021-10-05 14:30:00, 2021-10-05 15:30:00)

Bob
-------------
[2021-10-05 10:00:00, 2021-10-05 10:30:00)
[2021-10-05 10:30:00, 2021-10-05 11:30:00)
[2021-10-05 11:00:00, 2021-10-05 11:30:00)
[2021-10-05 16:00:00, 2021-10-05 17:00:00)

Carol
-------------
[2021-10-05 09:30:00, 2021-10-05 10:00:00)
[2021-10-05 09:30:00, 2021-10-05 10:30:00)
[2021-10-05 12:30:00, 2021-10-05 13:30:00)
[2021-10-05 16:30:00, 2021-10-05 17:30:00)
```

Where are the overlaps in meetings for each person?

You may notice that there seems to be some overlaps in the individual calendars for each person. Who amongst us can say they've never double booked?

We will examine these overlaps using *piso.intersection()* - but we will use it via the piso accessor. We will not supply any additional array arguments, so the sets are those intervals belonging to the IntervalArray. The *min_overlaps* parameter value of 2 indicates that we are looking for overlaps between two or more intervals. If we do not specify this parameter then the default behaviour is to find regions where every interval in the interval array overlaps (there are no such cases in this data).

```
In [10]: print("******** Individual Meeting Clashes ********\n")
******** Individual Meeting Clashes ********


In [11]: for person in ("Alice", "Bob", "Carol"):
   ....:     print_intervals(
   ....:         person,
   ....:         meetings[person].piso.intersection(min_overlaps=2),
   ....:     )
   ....:
Alice
-------------

Bob
-------------
[2021-10-05 11:00:00, 2021-10-05 11:30:00)

Carol
```

(continues on next page)

(continued from previous page)

```
-------------
[2021-10-05 09:30:00, 2021-10-05 10:00:00)
```

As you can see Bob and Carol each have an interval of time where they have meeting clashes.

What are the busy times for each person?

Let's not worry about the meeting clashes, they are irrelevant for finding the schedule gaps shared by Alice, Bob and Carol. We can simplify the "busy" times in each calendar with the *piso.union()* method (via the piso accessor).

```
In [12]: print("*************** Busy periods ***************\n")
*************** Busy periods ***************


In [13]: for person in ("Alice", "Bob", "Carol"):
   ....:     print_intervals(
   ....:         person,
   ....:         meetings[person].piso.union(),
   ....:     )
   ....:
Alice
-------------
[2021-10-05 09:00:00, 2021-10-05 11:00:00)
[2021-10-05 11:30:00, 2021-10-05 12:30:00)
[2021-10-05 14:30:00, 2021-10-05 15:30:00)

Bob
-------------
[2021-10-05 10:00:00, 2021-10-05 11:30:00)
[2021-10-05 16:00:00, 2021-10-05 17:00:00)

Carol
-------------
[2021-10-05 09:30:00, 2021-10-05 10:30:00)
[2021-10-05 12:30:00, 2021-10-05 13:30:00)
[2021-10-05 16:30:00, 2021-10-05 17:30:00)
```

So these are the disjoint intervals, in each person's calendar, in which they are busy. But we are interested in the complement of these intervals. That is, the times (between 9am and 5pm) that each person is free.

Where are the gaps in the schedule for each person?

We'll create an interval array holding a single *pandas.Interval* which represents the business day. For each person we can use the *piso.difference()* method (via the piso accessor), to remove the busy intervals from the business day interval. We do this using `pandas.Series.map()` and a lambda function but there are more verbose ways to perform this calculation. The result will be a `pandas.Series` called *gaps* which is indexed by the names, and whose values are interval arrays containing the "free" intervals in each person's calendar.

```
In [14]: business_day = pd.arrays.IntervalArray.from_breaks(
   ....:     [pd.Timestamp("2021-10-5 9:00"), pd.Timestamp("2021-10-5 17:00")],
   ....:     closed="left",
   ....: )
   ....:
```

(continues on next page)

(continued from previous page)

```
In [15]: gaps = meetings.map(lambda ia: business_day.piso.difference(ia))

In [16]: print("************* Gaps in schedule *************\n")
************* Gaps in schedule *************


In [17]: for person in ("Alice", "Bob", "Carol"):
   ....:     print_intervals(person, gaps[person])
   ....:
Alice
-------------
[2021-10-05 11:00:00, 2021-10-05 11:30:00)
[2021-10-05 12:30:00, 2021-10-05 14:30:00)
[2021-10-05 15:30:00, 2021-10-05 17:00:00)

Bob
-------------
[2021-10-05 09:00:00, 2021-10-05 10:00:00)
[2021-10-05 11:30:00, 2021-10-05 16:00:00)

Carol
-------------
[2021-10-05 09:00:00, 2021-10-05 09:30:00)
[2021-10-05 10:30:00, 2021-10-05 12:30:00)
[2021-10-05 13:30:00, 2021-10-05 16:30:00)
```

Where can we schedule a meeting between Alice, Bob and Carol?

All that remains to do is find the intersection between the interval array of gaps calculated above. We do this with *piso.intersection()*, but we will provide it with multiple `pandas.arrays.IntervalArray` operands, which indicates that each IntervalArray is interpreted as a set (as opposed to the intervals contained within.). We use python's "*" unpack operator to transform the values of the *gaps* series - which is a `numpy` array of `pandas.arrays.IntervalArray` - into the arguments in the method call.

```
In [18]: print_intervals(
   ....:     "Potential meetings times",
   ....:     piso.intersection(*gaps.values)
   ....: )
   ....:
Potential meetings times
-------------
[2021-10-05 13:30:00, 2021-10-05 14:30:00)
[2021-10-05 15:30:00, 2021-10-05 16:00:00)
```

So there we have it. There is a one-hour opportunity at 1:30pm and a half-hour opportunity at 3:30pm.

This has not been an exhaustive illustration of the functions in *piso*. There are many methods and parameters which have not been demonstrated above, but hopefully it has whet your appetite. For details of all the full functionality offered by *piso* refer to the *API reference*.

2.3.2 Verifying a maintenance schedule

In this example we are given the following scenario:

> There are three identical assets *X*, *Y* and *Z* which require periodic maintenance. No more than one asset should be under maintenance at any time, in order to handle the workload. Futhermore any maintenance should occur within windows of opportunity which represent when maintenance will be least disruptive. Given a proposed schedule for 2021, verify these rules are respected, and analyse time usage.

We start by importing `pandas` and *piso*

```
In [1]: import pandas as pd

In [2]: import piso
```

Running the *piso.register_accessors()* function will add "piso" accessors to `pandas.IntervalIndex` and `pandas.arrays.IntervalArray`. Using accessors allows us to extend the functionality associated with these classes, without adding the methods directly.

```
In [3]: piso.register_accessors()
```

Next we load the data from a csv file and store it into a `pandas.DataFrame`. Each row of the dataframe corresponds to an interval of maintenance for a particular asset.

```
In [4]: data = pd.read_csv("./data/asset_maintenance.csv", parse_dates=["start", "end"],␣
   ↪dayfirst=True)

In [5]: data
Out[5]:
   asset               start                 end
0      X 2021-12-14 00:00:00 2021-12-17 07:00:00
1      X 2021-01-31 23:00:00 2021-02-01 00:00:00
2      X 2021-03-15 05:00:00 2021-03-21 17:00:00
3      X 2021-09-07 13:00:00 2021-09-13 22:00:00
4      X 2021-05-02 00:00:00 2021-05-07 10:00:00
5      X 2021-08-03 14:00:00 2021-08-05 00:00:00
6      Y 2021-11-14 00:00:00 2021-11-16 22:00:00
7      Y 2021-09-01 00:00:00 2021-09-02 13:00:00
8      Y 2021-06-23 05:00:00 2021-06-30 11:00:00
9      Y 2021-01-18 19:00:00 2021-01-27 08:00:00
10     Y 2021-05-28 00:00:00 2021-05-31 16:00:00
11     Y 2021-03-23 05:00:00 2021-03-25 00:00:00
12     Z 2021-07-06 21:00:00 2021-07-10 00:00:00
13     Z 2021-04-11 07:00:00 2021-04-18 05:00:00
14     Z 2021-10-03 00:00:00 2021-10-06 14:00:00
15     Z 2021-02-25 00:00:00 2021-02-26 22:00:00
16     Z 2021-11-21 09:00:00 2021-11-26 00:00:00
17     Z 2021-06-03 06:00:00 2021-06-04 00:00:00
```

To work with *piso* we need the data in interval arrays. The following code creates a `pandas.Series`, indexed by the assets *X*, *Y* and *Z*, where the values are instances of `pandas.arrays.IntervalArray`.

```
In [6]: maintenance = (
   ...:     data
   ...:     .groupby("asset")
```

(continues on next page)

(continued from previous page)

```
   ...:     .apply(
   ...:         lambda df: pd.arrays.IntervalArray.from_arrays(
   ...:             df["start"],
   ...:             df["end"],
   ...:             closed="left",
   ...:         ),
   ...:     )
   ...: )
   ...:

In [7]: maintenance
Out[7]:
asset
X    [[2021-12-14, 2021-12-17 07:00:00), [2021-01-3...
Y    [[2021-11-14, 2021-11-16 22:00:00), [2021-09-0...
Z    [[2021-07-06 21:00:00, 2021-07-10), [2021-04-1...
dtype: object
```

Checking that no more than one asset is under maintenance at any time is equivalent to checking that the sets corresponding to each interval array are disjoint. This is as simple as the following code, where we unpack the values of the maintenance *Series* as arguments to *piso.isdisjoint()*.

```
In [8]: piso.isdisjoint(*maintenance.values)
Out[8]: True
```

The windows in which maintenance is preferred is described by the following data

```
In [9]: window_df = pd.read_csv(
   ...:     "./data/maintenance_windows.csv",
   ...:     parse_dates=["start", "end"],
   ...:     dayfirst=True,
   ...: )
   ...:

In [10]: window_df
Out[10]:
        start        end
0  2021-01-18 2021-02-01
1  2021-02-25 2021-03-05
2  2021-03-15 2021-03-25
3  2021-04-10 2021-04-20
4  2021-05-02 2021-05-09
5  2021-05-28 2021-06-04
6  2021-06-20 2021-07-10
7  2021-08-01 2021-08-05
8  2021-09-01 2021-09-14
9  2021-10-03 2021-10-08
10 2021-11-14 2021-11-26
11 2021-12-14 2021-12-22
```

As before, we transform this to an interval array

```
In [11]: windows = pd.arrays.IntervalArray.from_arrays(
   ....:     window_df["start"],
   ....:     window_df["end"],
   ....:     closed="left",
   ....: )
   ....:

In [12]: windows
Out[12]:
<IntervalArray>
[[2021-01-18, 2021-02-01), [2021-02-25, 2021-03-05), [2021-03-15, 2021-03-25), [2021-04-
↪10, 2021-04-20), [2021-05-02, 2021-05-09) ... [2021-08-01, 2021-08-05), [2021-09-01,␣
↪2021-09-14), [2021-10-03, 2021-10-08), [2021-11-14, 2021-11-26), [2021-12-14, 2021-12-
↪22)]
Length: 12, dtype: interval[datetime64[ns], left]
```

Checking that the maintenance occurs within the preferred windows can be done by checking that the set corresponding to the *windows* interval array is a superset of each of the sets corresponding to the asset interval arrays. Instead of doing this for each asset we can check against the union of these sets.

```
In [13]: combined_maintenance = piso.union(*maintenance.values)

In [14]: windows.piso.issuperset(combined_maintenance, squeeze=True)
Out[14]: True
```

Now let's answer some questions using *piso*, specifically *piso.coverage()* and its accessor counterpart.

What fraction of the year 2021 constitutes maintenance window opportunities?

```
In [15]: windows.piso.coverage(pd.Interval(pd.Timestamp("2021"), pd.Timestamp("2022")))
Out[15]: 0.3232876712328767
```

How many days in each month in 2021 constitute maintenance window opportunities?

For this we'll create a pandas.IntervalIndex for the months, then construct a pandas.Series with a monthly pandas.PeriodIndex.

```
In [16]: months = pd.IntervalIndex.from_breaks(pd.date_range("2021", "2022", freq="MS"))

In [17]: pd.Series(
   ....:     [windows.piso.coverage(month)*month.length for month in months],
   ....:     index = months.left.to_period()
   ....: )
   ....:
Out[17]:
2021-01   14 days
2021-02    4 days
2021-03   14 days
2021-04   10 days
2021-05   11 days
2021-06   14 days
2021-07    9 days
2021-08    4 days
2021-09   13 days
```

(continues on next page)

(continued from previous page)

```
2021-10    5 days
2021-11   12 days
2021-12    8 days
Freq: M, dtype: timedelta64[ns]
```

What fraction of the time in window opportunities is utilised by the combined maintenance?

```
In [18]: combined_maintenance.piso.coverage(windows)
Out[18]: 0.5903954802259888
```

What fraction of the combined maintenance is occupied by each asset

```
In [19]: maintenance.apply(piso.coverage, domain=combined_maintenance)
Out[19]:
asset
X    0.330742
Y    0.369019
Z    0.300239
dtype: float64
```

2.3.3 Estimating tax payable

In this example we are given the following scenario:

> Personal tax (before deductions) in Australia is based on the table below. The tax payable at the end of the financial year depends on the individual's income. The higher the income, the higher the tax rate, as defined by tax brackets (or tiers). Given a list of incomes, calculate the corresponding tax payable for each income.

Income Thresholds	Rate	Tax payable
$0 - $18,200	0%	Nil
$18,200 - $45,000	19%	19c for each $1 over $18,200
$45,000 - $120,000	32.5%	$5,092 plus 32.5c for each $1 over $45,000
$120,000 - $180,000	37%	$29,467 plus 37c for each $1 over $120,000
$180,000 and over	45%	$51,667 plus 45c for each $1 over $180,000

We start by importing `pandas`, `numpy` and *`piso`*, and creating an interval index for the tax brackets.

```
In [1]: import pandas as pd

In [2]: import numpy as np
```

(continues on next page)

(continued from previous page)

```
In [3]: import piso

In [4]: tax_brackets = pd.IntervalIndex.from_breaks(
   ...:     [0,18200,45000,120000,180000,np.inf],
   ...:     closed="left",
   ...: )
   ...:

In [5]: tax_brackets
Out[5]: IntervalIndex([[0.0, 18200.0), [18200.0, 45000.0), [45000.0, 120000.0), [120000.
↪0, 180000.0), [180000.0, inf)], dtype='interval[float64, left]')
```

With each interval in the tax bracket, we'll associate three values:

1) the lower threshold for the tax bracket
2) the fixed amount payable
3) the tax rate for each dollar above the threshold (as a fraction)

We describe this data as a `pandas.DataFrame` indexed by *tax_brackets*.

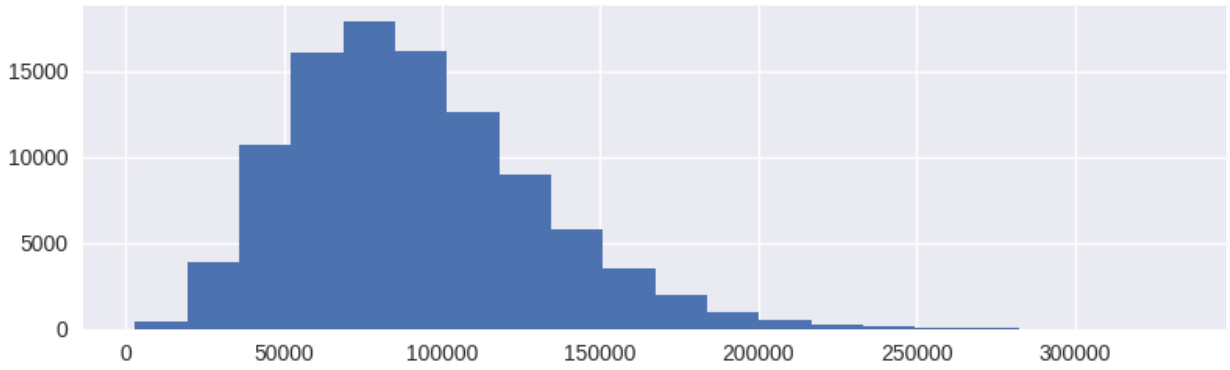
```
In [6]: tax_rates = pd.DataFrame(
   ...:     {
   ...:         "threshold":tax_brackets.left,
   ...:         "fixed":[0, 0, 5092, 29467, 51667],
   ...:         "rate":[0, 0.19, 0.325, 0.37, 0.45],
   ...:     },
   ...:     index = tax_brackets,
   ...: )
   ...:

In [7]: tax_rates
Out[7]:
                     threshold  fixed   rate
[0.0, 18200.0)             0.0      0  0.000
[18200.0, 45000.0)     18200.0      0  0.190
[45000.0, 120000.0)    45000.0   5092  0.325
[120000.0, 180000.0)  120000.0  29467  0.370
[180000.0, inf)       180000.0  51667  0.450
```

For the income, we'll generate some random integers (and plot the distribution) corresponding to 100,000 individuals.

```
In [8]: income = pd.Series(np.random.beta(5,50, size=100000)*1e6).astype(int)

In [9]: income.plot.hist(bins=20);
```



We are now in a position to use *piso.lookup()*, which take two parameters:

1) a pandas.DataFrame or pandas.Series which is indexed by a pandas.IntervalIndex
2) the values which are will be compared to the interval index

```
In [10]: tax_params = piso.lookup(tax_rates, income)

In [11]: tax_params
Out[11]:
        threshold  fixed   rate
87018     45000.0   5092  0.325
114404    45000.0   5092  0.325
39719     18200.0      0  0.190
88310     45000.0   5092  0.325
163758   120000.0  29467  0.370
...           ...    ...    ...
99841     45000.0   5092  0.325
29619     18200.0      0  0.190
88065     45000.0   5092  0.325
47199     45000.0   5092  0.325
159500   120000.0  29467  0.370

[100000 rows x 3 columns]
```

The result is a dataframe, indexed by the values of *income*, sharing the same columns as *tax_rates*.

We can then use a vectorised calculation for the tax payable:

```
In [12]: tax_params["fixed"] + (tax_params.index-tax_params["threshold"])*tax_params[
↪"rate"]
Out[12]:
87018     18747.850
114404    27648.300
39719      4088.610
88310     19167.750
163758    45657.460
             ...
99841     22915.325
29619      2169.610
```

(continues on next page)

(continued from previous page)

```
88065     19088.125
47199      5806.675
159500    44082.000
Length: 100000, dtype: float64
```

Alternative approaches

There are a couple of alternative, straightforward solutions which do not require *piso* which we detail below.

Alternative 1: pandas.cut

The *tax_params* dataframe that was produced above by *piso.lookup()* can be reproduced using pandas.cut() which can be used to assign bins to data with an interval index.

```
In [13]: tax_params = tax_rates.loc[pd.cut(income, tax_brackets)].set_index(income)

In [14]: tax_params
Out[14]:
        threshold  fixed   rate
87018     45000.0   5092  0.325
114404    45000.0   5092  0.325
39719     18200.0      0  0.190
88310     45000.0   5092  0.325
163758   120000.0  29467  0.370
...           ...    ...    ...
99841     45000.0   5092  0.325
29619     18200.0      0  0.190
88065     45000.0   5092  0.325
47199     45000.0   5092  0.325
159500   120000.0  29467  0.370

[100000 rows x 3 columns]
```

This approach however runs approximately 20 times slower than *piso.lookup()*.

Alternative 2: applying function

The second approach involves writing a function which takes a single value (an income for an individual) and returns the tax payable. The function can then used with *pandas.Series.apply*

```
In [15]: def calc_tax(value):
   ....:     if value <= 18200:
   ....:         tax = 0
   ....:     elif value <= 45000:
   ....:         tax = (value-18200)*0.19
   ....:     elif value <= 120000:
   ....:         tax = 5092 + (value-45000)*0.325
   ....:     elif value <= 180000:
   ....:         tax = 29467 + (value-120000)*0.37
   ....:     else:
   ....:         tax = 51667 + (value-180000)*0.45
   ....:     return tax
   ....:
```

(continues on next page)

(continued from previous page)

```
In [16]: income.apply(calc_tax)
Out[16]:
0        18747.850
1        27648.300
2         4088.610
3        19167.750
4        45657.460
            ...
99995    22915.325
99996     2169.610
99997    19088.125
99998     5806.675
99999    44082.000
Length: 100000, dtype: float64
```

This approach runs approximately 3 times slower than *piso.lookup()*. It also requires a function to be defined which is relatively cumbersome to implement. This approach becomes increasingly unattractive, and error prone, as the number of tax brackets increases.

2.3.4 Analysis of scores in a football match

In this example we will look at a football match from 2009:

> The Champions League quarter-final between Chelsea and Liverpool in 2009 is recognised as among the best games of all time. Liverpool scored twice in the first half in the 19th and 28th minute. Chelsea then opened their account in the second half with three unanswered goals in the 51st, 57th and 76th minute. Liverpool responded with two goals in the 81st and 83rd minute to put themselves ahead, however Chelsea drew with a goal in the 89th minute and advanced to the next stage on aggregate.

We start by importing `pandas` and *piso*

```
In [1]: import pandas as pd

In [2]: import piso
```

For the analysis we will create a `pandas.Series`, indexed by a `pandas.IntervalIndex` for each team. The values of each series will be the team's score and the interval index, defined by `pandas.Timedelta`, will describe the durations corresponding to each score. We define the following function which creates such a Series, given the minute marks for each score.

```
In [3]: def make_series(goal_time_mins):
   ...:     breaks = pd.to_timedelta([0] + goal_time_mins + [90], unit="min")
   ...:     ii = pd.IntervalIndex.from_breaks(breaks)
   ...:     return pd.Series(range(len(ii)), index = ii, name="score")
   ...:
```

We can now create each Series.

```
In [4]: chelsea = make_series([51,57,76,89])

In [5]: liverpool = make_series([19,28,81,83])
```

For reference, the Series corresponding to *chelsea* is

```
In [6]: chelsea
Out[6]:
(0 days 00:00:00, 0 days 00:51:00]    0
(0 days 00:51:00, 0 days 00:57:00]    1
(0 days 00:57:00, 0 days 01:16:00]    2
(0 days 01:16:00, 0 days 01:29:00]    3
(0 days 01:29:00, 0 days 01:30:00]    4
Name: score, dtype: int64
```

To enable analysis for separate halves of the game we'll define a similar Series which defines the time intervals for each half

```
In [7]: halves = pd.Series(
   ...:     ["1st", "2nd"],
   ...:     pd.IntervalIndex.from_breaks(pd.to_timedelta([0, 45, 90], unit="min")),
   ...:     name="half",
   ...: )
   ...:

In [8]: halves
Out[8]:
(0 days 00:00:00, 0 days 00:45:00]    1st
(0 days 00:45:00, 0 days 01:30:00]    2nd
Name: half, dtype: object
```

We can now perform a join on these three Series. Since *chelsea* and *liverpool* Series have the same name it will be necessary to provide suffixes to differentiate the columns in the result. The *halves* Series does not have the same name, but a suffix must be defined for each of the join operands if there are any overlaps.

```
In [9]: CvsL = piso.join(chelsea, liverpool, halves, suffixes=["_chelsea", "_liverpool",
↪""])

In [10]: CvsL
Out[10]:
                                  score_chelsea  score_liverpool half
(0 days 00:00:00, 0 days 00:19:00]             0                0  1st
(0 days 00:19:00, 0 days 00:28:00]             0                1  1st
(0 days 00:28:00, 0 days 00:45:00]             0                2  1st
(0 days 00:45:00, 0 days 00:51:00]             0                2  2nd
(0 days 00:51:00, 0 days 00:57:00]             1                2  2nd
(0 days 00:57:00, 0 days 01:16:00]             2                2  2nd
(0 days 01:16:00, 0 days 01:21:00]             3                2  2nd
(0 days 01:21:00, 0 days 01:23:00]             3                3  2nd
(0 days 01:23:00, 0 days 01:29:00]             3                4  2nd
(0 days 01:29:00, 0 days 01:30:00]             4                4  2nd
```

By default, the *piso.join()* function performs a left-join. Since every interval index represents the same domain, that is *(0', 90']*, all join types - *left*, *right*, *inner*, *outer* - will give the same result.

Using this dataframe we will now provide answers for miscellaneous questions. In particular we will filter the dataframe based on values in the columns, then sum the lengths of the intervals in the filtered index.

How much game time did Chelsea lead for?

```
In [11]: CvsL.query("score_chelsea > score_liverpool").index.length.sum()
Out[11]: Timedelta('0 days 00:05:00')
```

How much game time did Liverpool lead for?

```
In [12]: CvsL.query("score_liverpool > score_chelsea").index.length.sum()
Out[12]: Timedelta('0 days 00:44:00')
```

How much game time were the teams tied for?

```
In [13]: CvsL.query("score_liverpool == score_chelsea").index.length.sum()
Out[13]: Timedelta('0 days 00:41:00')
```

How much game time in the first half were the teams tied for?

```
In [14]: CvsL.query("score_chelsea == score_liverpool and half == '1st'").index.length.
↪sum()
Out[14]: Timedelta('0 days 00:19:00')
```

For how long did Liverpool lead Chelsea by exactly one goal (split by half)?

```
In [15]: CvsL.groupby("half").apply(
   ....:     lambda df: df.query("score_liverpool - score_chelsea == 1").index.length.
↪sum()
   ....: )
   ....:
Out[15]:
half
1st   0 days 00:09:00
2nd   0 days 00:12:00
dtype: timedelta64[ns]
```

What was the score at the 80 minute mark?

```
In [16]: piso.lookup(CvsL, pd.Timedelta(80, unit="min"))
Out[16]:
                 score_chelsea  score_liverpool half
0 days 01:20:00              3                2  2nd
```

This analysis is also straightforward using `staircase`. For more information on this please see the corresponding example with staircase

2.4 Frequently asked questions

Can any interval be used with piso?

Unfortunately no. The intervals must

- have a non-zero length
- have a finite, length
- either be left-closed right-open, or right-closed left-open

Operations between Intervals, IntervalIndex and IntervalArray objects must have the same value for their *closed* attribute.

Are there plans to add support for intervals which are either degenerate (contain a single point), infinite length, or not half-closed?

At this stage no, but this may change depending on the popularity of the package and the demand for this functionality.

Are there existing set operations for intervals in pandas?

Yes, but currently there are very few:

`pandas.Interval`:

- `pandas.Interval.is_empty`
- `pandas.Interval.overlaps()`

`pandas.arrays.IntervalArray`:

- `pandas.arrays.IntervalArray.is_empty`
- `pandas.arrays.IntervalArray.is_non_overlapping_monotonic`
- `pandas.arrays.IntervalArray.contains()`
- `pandas.arrays.IntervalArray.overlaps()`

`pandas.IntervalIndex`:

- `pandas.IntervalIndex.is_empty`
- `pandas.IntervalIndex.is_non_overlapping_monotonic`
- `pandas.IntervalIndex.is_overlapping`
- `pandas.IntervalIndex.contains()`
- `pandas.IntervalIndex.overlaps()`

Additional set operations for intervals, like those implemented in *piso*, are earmarked for development in `pandas` at some time in the future.

Can I work with datetime/timestamp data?

Yes *piso* will work with `pandas.Timestamp` and `pandas.Timedelta` data. Users who wish to use `numpy.datetime64` and `datetime.datetime` (and timedelta counterparts) should be aware that:

- `pandas.Interval` can only be constructed with numeric, `pandas.Timestamp` or `pandas.Timedelta` data
- when using construction class methods, such as `pandas.IntervalIndex.from_arrays()`, any datetime objects from `numpy` or `datetime` modules will be converted by `pandas` to the `pandas` equivalent.

Why is there no *piso* accessor for *pandas.Interval*?

Objects of type `pandas.Interval` are immutable, meaning they cannot be changed (incuding the addition of an accessor).

Why use accessors?

Accessors provide a nice way of carving out a seperate namespace for *piso*, as opposed to monkey patching. This is particularly important for `pandas.IntervalIndex`, which inherits methods from `pandas.Index`, which are set based operations:

- `pandas.Index.union()`
- `pandas.Index.intersection()`
- `pandas.Index.difference()`

- pandas.Index.symmetric_difference()

however these methods consider the elements of the to be the intervals themselves - there is no notion as the intervals being sets.

What if I want to map intervals with a scalar?

This question may arise if, for example, a pandas.Series with a numerical dtype, was indexed with a pandas.IntervalIndex. Given two intervals, and their associated scalar values, a user may wish to find the overlap of these intervals, and map it to the minimum of the two scalar values - or perhaps the addition of the scalar values. These sorts of manipulations can be achieved via staircase. There is a one-to-one mapping between sets of disjoint intervals (with associated scalars) and step functions, which is what motivates the internal implementations of *piso*. staircase provides a comprehensive range of arithmetic, logical, relational and statistical methods for working with step functions. For related case studies see the *football case study with piso* and the football case study with staircase

CHAPTER

THREE

API REFERENCE

This page gives an overview of all public *piso* functionality. Classes and functions exposed in the *piso.** and *piso.interval.** namespaces are public. Other top-level modules should be considered **private** until specified otherwise.

3.1 Top level functions

register_accessors()	When called this function will register the "piso" ArrayAccessor on pandas.IntervalIndex and pandas.arrays.IntervalArray.
union(interval_array, *interval_arrays[, ...])	Performs a set union operation.
intersection(interval_array, *interval_arrays)	Performs a set intersection operation.
difference(interval_array, *interval_arrays)	Performs a set difference operation.
symmetric_difference(interval_array, ...[, ...])	Performs a set symmetric difference operation.
isdisjoint(interval_array, *interval_arrays)	Indicates whether one, or more, sets are disjoint or not.
issuperset(interval_array, *interval_arrays)	Indicates whether a set is a superset of one, or more, other sets.
issubset(interval_array, *interval_arrays[, ...])	Indicates whether a set is a subset of one, or more, other sets.
coverage(interval_array[, domain, bins, how])	Calculates the fraction of a domain (or possibly multiple domains) covered by a collection of intervals.
complement(interval_array[, domain])	Calculates the complement of a collection of intervals (in an array) over some domain.
contains(interval_array, x[, include_index, ...])	Evaluates the intersection of a set of intervals with a set of points.
split(interval_array, x)	Given a set of intervals, and break points, splits the intervals into pieces wherever the overlap a break point.
lookup(frame_or_series, x)	Given a pandas.DataFrame, or pandas.Series, indexed by a *pandas.IntervalIndex*, finds the intervals which contain each point in an array and returns the associated rows/elements.
join(*frames_or_series[, how, suffixes, sort])	Joins multiple dataframes or series by their pandas.IntervalIndex.
adjacency_matrix(interval_array[, edges, ...])	Returns a 2D array (or dataframe) of boolean values indicating edges between nodes in a graph.

3.1.1 piso.register_accessors

piso.**register_accessors**()

When called this function will register the "piso" ArrayAccessor on pandas.IntervalIndex and pandas.arrays.IntervalArray.

Examples

```
>>> import piso
>>> piso.register_accessors()
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5), (3, 6), (7, 8), (8, 9), (10, 12)],
... )
```

```
>>> arr.piso.union()
<IntervalArray>
[(0.0, 6.0], (7.0, 9.0], (10.0, 12.0]]
Length: 3, closed: right, dtype: interval[float64]
```

```
>>> arr = pd.IntervalIndex(arr)
>>> arr.piso.union()
IntervalIndex([(0.0, 2.0], (5.0, 6.0], (7.0, 9.0], (10.0, 12.0]],
              closed='right',
              dtype='interval[float64]')
```

3.1.2 piso.union

piso.**union**(*interval_array*, **interval_arrays*, *squeeze=False*, *return_type='infer'*)

Performs a set union operation.

What is considered a set is determined by the number of positional arguments used, that is, determined by the size of *interval_arrays*.

If *interval_arrays* is empty then the sets are considered to be the intervals contained in *interval_array*.

If *interval_arrays* is not empty then the sets are considered to be *interval_array* and the elements in *interval_arrays*. Each of these arrays is assumed to contain disjoint intervals (and satisfy the definition of a set). Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

Parameters

interval_array [pandas.IntervalIndex or pandas.arrays.IntervalArray] The first (and possibly only) operand to the union operation.

***interval_arrays** [argument list of pandas.IntervalIndex or pandas.arrays.IntervalArray] May contain zero or more arguments.

squeeze [boolean, default False] If True, will try to coerce the return value to a single pandas.Interval. If supplied, must be done so as a keyword argument.

return_type [{"infer", pandas.IntervalIndex, pandas.arrays.IntervalArray}, default "infer"] If "infer" the return type will be the same as *interval_array*. If supplied, must be done so as a keyword argument.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso
```

Examples with *interval_arrays* empty:

```
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5), (3, 6), (7, 8), (8, 9), (10, 12)],
... )
```

```
>>> piso.union(arr)
<IntervalArray>
[(0.0, 6.0], (7.0, 9.0], (10.0, 12.0]]
Length: 3, closed: right, dtype: interval[float64]
```

```
>>> piso.union(arr.set_closed("left"))
<IntervalArray>
[[0, 4), [2, 5), [3, 6), [7, 8), [8, 9), [10, 12)]
Length: 6, closed: left, dtype: interval[int64]
```

```
>>> piso.union(pd.IntervalIndex(arr))
IntervalIndex([(0.0, 6.0], (7.0, 9.0], (10.0, 12.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> piso.union(arr, return_type=pd.IntervalIndex)
IntervalIndex([(0.0, 6.0], (7.0, 9.0], (10.0, 12.0]],
              closed='right',
              dtype='interval[float64]')
```

Examples with *interval_arrays* non empty:

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (5, 6), (7, 8), (10, 12)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(3, 5), (8, 9)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(6, 8), (9, 11)],
... )
```

```
>>> piso.union(arr1, arr2)
<IntervalArray>
[(0.0, 6.0], (7.0, 9.0], (10.0, 12.0]]
Length: 3, closed: right, dtype: interval[float64]
```

```
>>> piso.union(arr2, arr3, return_type=pd.IntervalIndex)
IntervalIndex([(3.0, 5.0], (6.0, 11.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> piso.union(arr1, arr2, arr3)
<IntervalArray>
[(0.0, 12.0]]
Length: 1, closed: right, dtype: interval[float64]
```

```
>>> piso.union(arr1, arr2, arr3, squeeze=True)
Interval(0.0, 12.0, closed='right')
```

3.1.3 piso.intersection

piso.**intersection**(*interval_array*, **interval_arrays*, *min_overlaps='all'*, *squeeze=False*, *return_type='infer'*)

Performs a set intersection operation.

What is considered a set is determined by the number of positional arguments used, that is, determined by the size of *interval_arrays*.

If *interval_arrays* is empty then the sets are considered to be the intervals contained in *interval_array*.

If *interval_arrays* is not empty then the sets are considered to be *interval_array* and the elements in *interval_arrays*. Each of these arrays is assumed to contain disjoint intervals (and satisfy the definition of a set). Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

Parameters

interval_array [pandas.IntervalIndex or pandas.arrays.IntervalArray] The first (and possibly only) operand to the intersection operation.

***interval_arrays** [argument list of pandas.IntervalIndex or pandas.arrays.IntervalArray] May contain zero or more arguments.

min_overlaps [int or "all", default "all"] Specifies the minimum number of intervals which overlap in order to define an *intersection*. If *min_overlaps* is an int then it must be no smaller than 2. If *min_overlaps* is all then an intersection is only defined where every interval overlaps. If supplied, must be done so as a keyword argument.

squeeze [boolean, default False] If True, will try to coerce the return value to a single pandas.Interval. If supplied, must be done so as a keyword argument.

return_type [{"infer", pandas.IntervalIndex, pandas.arrays.IntervalArray}, default "infer"] If "infer" the return type will be the same as *interval_array*. If supplied, must be done so as a keyword argument.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso
```

Examples with *interval_arrays* empty:

```
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5), (3, 6)],
... )
```

```
>>> piso.intersection(arr)
<IntervalArray>
[(3.0, 4.0]]
Length: 1, closed: right, dtype: interval[float64]
```

```
>>> piso.intersection(pd.IntervalIndex(arr))
IntervalIndex([(3.0, 4.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> piso.intersection(arr, return_type=pd.IntervalIndex)
IntervalIndex([(3.0, 4.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> piso.intersection(arr)
<IntervalArray>
[]
Length: 0, closed: right, dtype: interval[int64]
```

```
>>> piso.intersection(arr, min_overlaps=2)
<IntervalArray>
[(2.0, 5.0]]
Length: 1, closed: right, dtype: interval[float64]
```

Examples with *interval_arrays* not empty:

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (5, 6), (7, 8), (10, 12)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(3, 5), (8, 9)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(6, 8), (9, 11)],
... )
```

```
>>> piso.intersection(arr1, arr2)
<IntervalArray>
[(3.0, 4.0]]
Length: 1, closed: right, dtype: interval[float64]
```

```
>>> piso.intersection(arr1, arr2, squeeze=True)
Interval(3.0, 4.0, closed='right')
```

```
>>> piso.intersection(arr1, arr2, arr3)
<IntervalArray>
[]
Length: 0, closed: right, dtype: interval[float64]
```

```
>>> piso.intersection(arr1, arr2, arr3, min_overlaps=2)
<IntervalArray>
[(3.0, 4.0], (10.0, 11.0]]
Length: 2, closed: right, dtype: interval[float64]
```

3.1.4 piso.difference

piso.**difference**(*interval_array*, **interval_arrays*, *squeeze=False*, *return_type='infer'*)

Performs a set difference operation.

The argument *interval_array* and the array elements of *interval_arrays* are all considered to be the sets over which the operation is performed. Each of these arrays is assumed to contain disjoint intervals (and satisfy the definition of a set). Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

The list *interval_arrays* must contain at least one element. If *interval_arrays* contains a single element then the result is the set difference between *interval_array* and this single element. If *interval_arrays* contains multiple elements then the result is the set difference between *interval_array* and the union of the sets in *interval_arrays*. This is equivalent to iteratively applying a set difference operation with each array in *interval_arrays* as the second operand.

Parameters

interval_array [pandas.IntervalIndex or pandas.arrays.IntervalArray] The first operand to the difference operation.

***interval_arrays** [argument list of pandas.IntervalIndex or pandas.arrays.IntervalArray] Must contain at least one argument.

squeeze [boolean, default False] If True, will try to coerce the return value to a single pandas.Interval. If supplied, must be done so as a keyword argument.

return_type [{"infer", pandas.IntervalIndex, pandas.arrays.IntervalArray}, default "infer"] If "infer" the return type will be the same as *interval_array*. If supplied, must be done so as a keyword argument.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso
```

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5), (3, 6), (7, 8), (8, 9), (10, 12)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(4, 7), (8, 11)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(2, 5), (7, 13)],
... )
```

```
>>> piso.difference(arr1, arr2)
<IntervalArray>
[(0.0, 4.0], (7.0, 8.0], (11.0, 12.0]]
Length: 3, closed: right, dtype: interval[float64]
```

```
>>> piso.difference(arr1.set_closed("left"), arr2.set_closed("left"))
<IntervalArray>
[[0.0, 4.0), [7.0, 8.0), [11.0, 12.0)]
Length: 3, closed: left, dtype: interval[float64]
```

```
>>> piso.difference(arr1, arr2, return_type=pd.IntervalIndex)
IntervalIndex([(0.0, 4.0], (7.0, 8.0], (11.0, 12.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> piso.difference(arr1, arr2, arr3)
<IntervalArray>
[(0.0, 2.0]]
Length: 1, closed: right, dtype: interval[float64]
```

```
>>> piso.difference(arr1, arr2, arr3, squeeze=True)
Interval(0.0, 2.0, closed='right')
```

3.1.5 piso.symmetric_difference

piso.**symmetric_difference**(*interval_array*, **interval_arrays*, *min_overlaps=2*, *squeeze=False*, *return_type='infer'*)

Performs a set symmetric difference operation.

What is considered a set is determined by the number of positional arguments used, that is, determined by the size of *interval_arrays*.

If *interval_arrays* is empty then the sets are considered to be the intervals contained in *interval_array*.

If *interval_arrays* is not empty then the sets are considered to be *interval_array* and the elements in *interval_arrays*. Each of these arrays is assumed to contain disjoint intervals (and satisfy the definition of a set).

Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

The symmetric difference can be defined as the set difference, of the union and the intersection. The parameter *min_overlaps* in *piso.intersection()*, which defines the minimum number of intervals in an overlap required to constitute an intersection, follows through to symmetric difference under this definition.

Parameters

interval_array [pandas.IntervalIndex or pandas.arrays.IntervalArray] The first (and possibly only) operand to the symmetric difference operation.

***interval_arrays** [argument list of pandas.IntervalIndex or pandas.arrays.IntervalArray] May contain zero or more arguments.

min_overlaps [int or "all", default "all"] Specifies the minimum number of intervals which overlap in order to define an *intersection*. If *min_overlaps* is an int then it must be no smaller than 2. If *min_overlaps* is all then an intersection is only defined where every interval overlaps. If supplied, must be done so as a keyword argument.

squeeze [boolean, default False] If True, will try to coerce the return value to a single pandas.Interval. If supplied, must be done so as a keyword argument.

return_type [{"infer", pandas.IntervalIndex, pandas.arrays.IntervalArray}, default "infer"] If "infer" the return type will be the same as *interval_array*. If supplied, must be done so as a keyword argument.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso
```

Examples with *interval_arrays* empty:

```
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5), (3, 6), (7, 8), (8, 9), (10, 12)],
... )
```

```
>>> piso.symmetric_difference(arr)
<IntervalArray>
[(0.0, 2.0], (5.0, 6.0], (7.0, 9.0], (10.0, 12.0]]
Length: 4, closed: right, dtype: interval[float64]
```

```
>>> piso.symmetric_difference(pd.IntervalIndex(arr))
IntervalIndex([(0.0, 2.0], (5.0, 6.0], (7.0, 9.0], (10.0, 12.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> piso.symmetric_difference(arr, return_type=pd.IntervalIndex)
IntervalIndex([(0.0, 2.0], (5.0, 6.0], (7.0, 9.0], (10.0, 12.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> piso.symmetric_difference(arr, min_overlaps=3)
<IntervalArray>
[(0.0, 3.0], (4.0, 6.0], (7.0, 9.0], (10.0, 12.0]]
Length: 4, closed: right, dtype: interval[float64]
```

```
>>> piso.symmetric_difference(arr, min_overlaps="all")
<IntervalArray>
[(0.0, 6.0], (7.0, 9.0], (10.0, 12.0]]
Length: 3, closed: right, dtype: interval[float64]
```

Examples with *interval_arrays* non-empty:

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (5, 6), (7, 8), (10, 12)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(3, 5), (8, 9)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(6, 8), (9, 11)],
... )
```

```
>>> piso.symmetric_difference(arr1, arr2)
<IntervalArray>
[(0.0, 3.0], (4.0, 6.0], (7.0, 9.0], (10.0, 12.0]]
Length: 4, closed: right, dtype: interval[float64]
```

```
>>> piso.symmetric_difference(arr1, arr2, arr3)
<IntervalArray>
[(0.0, 3.0], (4.0, 7.0], (8.0, 10.0], (11.0, 12.0]]
Length: 4, closed: right, dtype: interval[float64]
```

```
>>> piso.symmetric_difference(arr1, arr2, arr3, min_overlaps="all")
<IntervalArray>
[(0.0, 12.0]]
Length: 1, closed: right, dtype: interval[float64]
```

3.1.6 piso.isdisjoint

piso.**isdisjoint**(*interval_array*, **interval_arrays*)

Indicates whether one, or more, sets are disjoint or not.

interval_array must be left-closed or right-closed if *interval_arrays is non-empty. If *interval_array* is the only argument then this restriction does not apply.

What is considered a set is determined by the number of positional arguments used, that is, determined by the size of *interval_arrays*.

If *interval_arrays* is empty then the sets are considered to be the intervals contained in *interval_array*.

If *interval_arrays* is not empty then the sets are considered to be *interval_array* and the elements in *interval_arrays*. Each of these arrays is assumed to contain disjoint intervals (and satisfy the definition of a set).

Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

Parameters

interval_array [pandas.IntervalIndex or pandas.arrays.IntervalArray] The first (and possibly only) operand to the isdisjoint operation.

***interval_arrays** [argument list of pandas.IntervalIndex or pandas.arrays.IntervalArray] May contain zero or more arguments.

Returns

boolean

Examples

```
>>> import pandas as pd
>>> import piso
```

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 3), (2, 4)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(4, 7), (8, 11)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(2, 4), (7, 8)],
... )
```

```
>>> piso.isdisjoint(arr1)
False
```

```
>>> piso.isdisjoint(arr2)
True
```

```
>>> piso.isdisjoint(arr1, arr2)
True
```

```
>>> piso.isdisjoint(arr1, arr3)
False
```

3.1.7 piso.issuperset

piso.**issuperset**(*interval_array*, **interval_arrays*, *squeeze=True*)

Indicates whether a set is a superset of one, or more, other sets.

The argument *interval_array* and the array elements of *interval_arrays* are all considered to be the sets for the purposes of this set method. Each of these arrays is assumed to contain disjoint intervals (and satisfy the definition of a set). Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

The list *interval_arrays* must contain at least one element. The superset comparison is iteratively applied between *interval_array* and each array in *interval_arrays*. When *interval_arrays* contains multiple interval arrays, the

return type will be a numpy array. If it contains one interval array then the result can be coerced to a single boolean using the *squeeze* parameter.

Parameters

interval_array [pandas.IntervalIndex or pandas.arrays.IntervalArray] The first operand to which all others are compared operation.

***interval_arrays** [argument list of pandas.IntervalIndex or pandas.arrays.IntervalArray] Must contain at least one argument.

squeeze [boolean, default True] If True, will try to coerce the return value to a single pandas.Interval. If supplied, must be done so as a keyword argument.

Returns

boolean, or [class:*numpy.ndarray* of boolean]

Examples

```
>>> import pandas as pd
>>> import piso
```

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (3, 6), (7, 8), (10, 12)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(2, 5), (7, 8)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(3, 4), (10, 11)],
... )
```

```
>>> piso.issuperset(arr1, arr2)
True
```

```
>>> piso.issuperset(arr1, arr2, squeeze=False)
array([ True])
```

```
>>> piso.issuperset(arr1, arr2, arr3)
array([ True, True])
```

```
>>> piso.issuperset(arr2, arr3)
False
```

3.1.8 piso.issubset

piso.**issubset**(*interval_array*, **interval_arrays*, *squeeze=True*)

Indicates whether a set is a subset of one, or more, other sets.

The argument *interval_array* and the array elements of *interval_arrays* are all considered to be the sets for the purposes of this set method. Each of these arrays is assumed to contain disjoint intervals (and satisfy the definition of a set). Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

The list *interval_arrays* must contain at least one element. The subset comparison is iteratively applied between *interval_array* and each array in *interval_arrays*. When *interval_arrays* contains multiple interval arrays, the return type will be a numpy array. If it contains one interval array then the result can be coerced to a single boolean using the *squeeze* parameter.

Parameters

interval_array [pandas.IntervalIndex or pandas.arrays.IntervalArray] The first operand to which all others are compared operation.

***interval_arrays** [argument list of pandas.IntervalIndex or pandas.arrays.IntervalArray] Must contain at least one argument.

squeeze [boolean, default True] If True, will try to coerce the return value to a single pandas.Interval. If supplied, must be done so as a keyword argument.

Returns

boolean, or [class:*numpy.ndarray* of boolean]

Examples

```
>>> import pandas as pd
>>> import piso
```

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(2, 5), (7, 8)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (3, 6), (7, 8), (10, 12)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(3, 4), (10, 11)],
... )
```

```
>>> piso.issubset(arr1, arr2)
True
```

```
>>> piso.issubset(arr1, arr2, squeeze=False)
array([ True])
```

```
>>> piso.issubset(arr1, arr2, arr3)
array([ True, False])
```

```
>>> piso.issubset(arr1, arr3)
False
```

3.1.9 piso.coverage

piso.**coverage**(*interval_array*, *domain=None*, *bins=False*, *how='fraction'*)

Calculates the fraction of a domain (or possibly multiple domains) covered by a collection of intervals.

Calculation over multiple domains is only possible when *bins* = True.

Parameters

interval_array [pandas.IntervalIndex or pandas.arrays.IntervalArray] Contains the (possibly overlapping) intervals which partially, or wholly cover the domain. May be left-closed, right-closed, both, or neither.

domain [tuple, pandas.Interval, pandas.IntervalIndex or pandas.arrays.IntervalArray, optional] Specifies the domain over which to calculate the "coverage". If *domain* is *None*, then the domain is considered to be the extremities of the intervals contained in *interval_array* If *domain* is a tuple then it should specify lower and upper bounds, and be equivalent to a pandas.Interval. If *domain* is a pandas.IntervalIndex or pandas.arrays.IntervalArray then the intervals it contains define a possibly disconnected domain. If *bins* = True then *domain* must be pandas.IntervalIndex or pandas.arrays.IntervalArray with disjoint intervals.

bins [boolean, default False] If False, then the *domain* is interpreted as a single domain and returns one value. If True, then *domain* is interpreted as disjoint bins over which coverage is calculated for each.

how [{"fraction", "sum"}, default "fraction"] If *how* = "fraction" then the result is a fraction of the size of the domain. If *how* = "sum" then the result is the length of the domain covered.

New in version 0.8.0.

Returns

float or [class:*pandas.Series*]

Examples

```
>>> import pandas as pd
>>> import piso
```

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (3, 5), (7, 8)],
... )
```

```
>>> piso.coverage(arr1)
0.75
```

```
>>> piso.coverage(arr1, (0, 10))
0.6
```

```
>>> piso.coverage(arr1, pd.Interval(-10, 10))
0.3
```

```
>>> piso.coverage(arr1, pd.Interval(-10, 10), how="sum")
6
```

```
>>> domain = pd.arrays.IntervalArray.from_tuples(
...     [(4,6), (7, 10)],
... )
>>> piso.coverage(arr1, domain)
0.4
```

```
>>> piso.coverage(arr1, domain, bins=True)
(4, 6]     0.500000
(7, 10]    0.333333
dtype: float64
```

```
>>> piso.coverage(arr1, domain, bins=True, how="sum")
(4, 6]     1.0
(7, 10]    1.0
dtype: float64
```

3.1.10 piso.complement

piso.**complement**(*interval_array*, *domain=None*)

Calculates the complement of a collection of intervals (in an array) over some domain.

Equivalent to the set difference of the domain and the intervals in the array.

Parameters

interval_array [pandas.IntervalIndex or pandas.arrays.IntervalArray] Contains the (possibly overlapping) intervals. Must be left-closed or right-closed.

domain [tuple, pandas.Interval, pandas.IntervalIndex or pandas.arrays.IntervalArray, optional] Specifies the domain over which to calculate the "complement". If *domain* is *None*, then the domain is considered to be the extremities of the intervals contained in *interval_array* If *domain* is a tuple then it should specify lower and upper bounds, and be equivalent to a pandas.Interval. If *domain* is a pandas.IntervalIndex or pandas.arrays.IntervalArray then the intervals it contains define a possibly disconnected domain.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*] The return type will be the same as *interval_array*.

Examples

```
>>> import pandas as pd
>>> import piso
```

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (3, 5), (7, 8)],
... )
```

```
>>> piso.complement(arr1)
<IntervalArray>
[(5, 7]]
Length: 1, closed: right, dtype: interval[int64]
```

```
>>> piso.complement(arr1, (-5, 10))
<IntervalArray>
[(-5, 0], (5, 7], (8, 10]]
Length: 3, closed: right, dtype: interval[int64]
```

```
>>> piso.complement(arr1, pd.Interval(-5, 6))
<IntervalArray>
[(-5, 0], (5, 6]]
Length: 2, closed: right, dtype: interval[int64]
```

```
>>> domain = pd.arrays.IntervalArray.from_tuples(
...     [(-5,-2), (7,10)],
... )
```

```
>>> piso.complement(arr1, domain)
<IntervalArray>
[(-5, -2], (8, 10]]
Length: 2, closed: right, dtype: interval[int64]
```

3.1.11 piso.contains

piso.**contains**(*interval_array*, *x*, *include_index=True*, *result='cartesian'*, *how='any'*)

Evaluates the intersection of a set of intervals with a set of points.

The format of the result is dependent on the *result* parameter. If *result* = *"cartesian"* then the the function returns a 2-dimensional boolean mask *M* of shape *(m,n)* where *m* is the number of intervals, and *n* is the number of points. The element in the i-th row and j-th column is True if the i-th interval contains the j-th point.

If *result* = *"points"* then the result is a 1-dimensional boolean mask of length *n*. If *result* = *"intervals"* then the result is a 1-dimensional boolean mask of length *m*.

Parameters

interval_array [pandas.IntervalIndex or pandas.arrays.IntervalArray] Contains the intervals. May be left-closed, right-closed, both, or neither.

x [scalar, or array-like of scalars] Values in *x* should belong to the same domain as the intervals in *interval_array*.

include_index [boolean, default True] Indicates whether to return a `numpy.ndarray` or `pandas.DataFrame` indexed by *interval_array* and column names equal to *x*

result [{"cartesian", "points", "intervals"}, default "cartesian"] If *result* = "cartesian" then the result will be two dimensional, otherwise it will be one dimensional.

how [{"any", "all"}, default "any"] Only relevant if *result* is not "cartesian". This parameter indicates either: - a True value means any or all points are contained within an interval, or - a True value means any or all intervals contained a point. Which of these interpretations is dependent on the *result* parameter.

Returns

:class:`numpy.ndarray`, [class:*pandas.DataFrame* or `pandas.Series`] One, or two, dimensional and boolean valued. Return type dependent on *include_index* and *result*.

Examples

```
>>> import pandas as pd
>>> import piso
```

```
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5)],
... )
```

```
>>> piso.contains(arr, 1)
           1
(0, 4]   True
(2, 5]  False
```

```
>>> piso.contains(arr, [0, 1, 3, 4])
           0      1     3     4
(0, 4]  False   True  True  True
(2, 5]  False  False  True  True
```

```
>>> piso.contains(arr, [0, 1, 3, 4], include_index=False)
array([[False,  True,  True,  True],
       [False, False,  True,  True]])
```

```
>>> piso.contains(arr, [0, 1, 3, 4], result="points")
0    False
1     True
3     True
4     True
dtype: bool
```

```
>>> piso.contains(arr, [0, 1, 3, 4], result="points", how="all")
0    False
1    False
3     True
4     True
dtype: bool
```

```
>>> piso.contains(arr, [0, 1, 3, 4], result="intervals")
(0, 4]    True
(2, 5]    True
dtype: bool
```

```
>>> piso.contains(pd.IntervalIndex.from_tuples([(0,2)]), 1, include_index=False)
array([[ True]])
```

3.1.12 piso.split

piso.**split**(*interval_array*, *x*)

Given a set of intervals, and break points, splits the intervals into pieces wherever the overlap a break point.

Parameters

interval_array [pandas.IntervalIndex or pandas.arrays.IntervalArray] Contains the (possibly overlapping) intervals. May be left-closed, right-closed, both, or neither.

x [scalar, or array-like of scalars] Values in *x* should belong to the same domain as the intervals in *interval_array*. May contain duplicates and be unsorted.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*] Return type will be the same type as *interval_array*

Examples

```
>>> import pandas as pd
>>> import piso
```

```
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5)],
... )
```

```
>>> piso.split(arr, 3)
<IntervalArray>
[(0, 3], (3, 4], (2, 3], (3, 5]]
Length: 4, closed: right, dtype: interval[int64]
```

```
>>> piso.split(arr, [3,3,3,3])
<IntervalArray>
[(0, 3], (3, 4], (2, 3], (3, 5]]
Length: 4, closed: right, dtype: interval[int64]
```

```
>>> arr = pd.IntervalIndex.from_tuples(
...     [(0, 4), (2, 5)], closed="neither",
... )
```

```
>>> piso.split(arr, [1, 6, 4])
IntervalIndex([(0.0, 1.0), (1.0, 4.0), (2.0, 4.0), (4.0, 5.0)],
              closed='neither',
              dtype='interval[float64]')
```

3.1.13 piso.lookup

piso.**lookup**(*frame_or_series*, *x*)

Given a pandas.DataFrame, or pandas.Series, indexed by a *pandas.IntervalIndex*, finds the intervals which contain each point in an array and returns the associated rows/elements.

Parameters

frame_or_series [pandas.DataFrame or pandas.Series] Must be indexed by a pandas.IntervalIndex containing disjoint intervals

x [scalar, or array-like of scalars] Values in *x* should belong to the same domain as the intervals in the interval index.

Returns

:class:`pandas.DataFrame` or [class:*pandas.Series*] Will be the same type as *frame_or_series*

Examples

```
>>> import pandas as pd
>>> import piso
```

```
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (6, 8), (10, 12)],
... )
>>> df = pd.DataFrame({"A":[3, 2, 1],"B":["x", "y", "z"]}, index=arr)
>>> df
          A  B
(0, 4]    3  x
(6, 8]    2  y
(10, 12]  1  z
```

```
>>> piso.lookup(df, 1)
   A  B
1  3  x
```

```
>>> piso.lookup(df, [1, 6, 5, 12])
      A    B
1   3.0    x
6   NaN  NaN
5   NaN  NaN
12  1.0    z
```

```
>>> piso.lookup(df["A"], [1, 2, 6])
1    3.0
```

(continues on next page)

(continued from previous page)

```
2     3.0
6     NaN
Name: A, dtype: float64
```

3.1.14 piso.join

piso.**join**(**frames_or_series*, *how='left'*, *suffixes=None*, *sort=False*)

Joins multiple dataframes or series by their `pandas.IntervalIndex`.

Each interval in a `pandas.IntervalIndex` is considered a set, and the interval index containing them a set defined by their union. Join types are as follows:

- left: the set defined by the interval index of the result is the same as the set defined by the index of the first argument in *frames_or_series*
- right: the set defined by the interval index of the result is the same as the set defined by the index of the last argument in *frames_or_series*
- inner: the set defined by the interval index of the result is the intersection of sets defined by interval indexes from all join arguments
- outer: the set defined by the interval index of the result is the union of sets defined by interval indexes from all join arguments

Parameters

***frames_or_series** [argument list of `pandas.DataFrame` or `pandas.Series`] May contain two or more arguments, all of which must be indexed by a `pandas.IntervalIndex` containing disjoint intervals. The index can have any *closed* value. Every `pandas.Series` must have a name.

how [{"left", "right", "inner", "outer"}, default "left"] What sort of join to perform.

suffixes [list of str or None, default None] Suffixes to use for overlapping columns. If used then should be same length as *frames_or_series*.

sort [bool, default False] Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type.

Returns

`pandas.DataFrame` A dataframe containing columns from elements of *frames_or_series*

Examples

```
>>> import pandas as pd
>>> import piso
```

```
>>> df = pd.DataFrame(
...     {"A":[4,3], "B":["x","y"]},
...     index=pd.IntervalIndex.from_tuples([(1,3), (5,7)]),
... )
>>> s = pd.Series(
...     [True, False],
...     index=pd.IntervalIndex.from_tuples([(2,4), (5,6)]),
```

(continues on next page)

(continued from previous page)

```
...     name="C",
... )
```

```
>>> piso.join(df, s)
        A  B      C
(1, 2]  4  x    NaN
(2, 3]  4  x   True
(5, 6]  3  y  False
(6, 7]  3  y    NaN
```

```
>>> piso.join(df, s, how="right")
          A    B      C
(2, 3]  4.0    x   True
(3, 4]  NaN  NaN   True
(5, 6]  3.0    y  False
```

```
>>> piso.join(df, s, how="inner")
        A  B      C
(2, 3]  4  x   True
(5, 6]  3  y  False
```

```
>>> piso.join(df, s, how="outer")
          A    B      C
(1, 2]  4.0    x    NaN
(2, 3]  4.0    x   True
(5, 6]  3.0    y  False
(6, 7]  3.0    y    NaN
(3, 4]  NaN  NaN   True
```

```
>>> piso.join(df, s, how="outer", sort=True)
          A    B      C
(1, 2]  4.0    x    NaN
(2, 3]  4.0    x   True
(3, 4]  NaN  NaN   True
(5, 6]  3.0    y  False
(6, 7]  3.0    y    NaN
```

```
>>> piso.join(df, df, suffixes=["", "2"])
        A  B  A2 B2
(1, 3]  4  x   4  x
(5, 7]  3  y   3  y
```

```
>>> df2 = pd.DataFrame(
...     {"D":[1,2]},
...     index=pd.IntervalIndex.from_tuples([(1,2), (6,7)]),
... )
```

```
>>> piso.join(df, s, df2)
        A  B      C    D
(1, 2]  4  x    NaN  1.0
```

(continues on next page)

(continued from previous page)

```
(2, 3]  4  x   True  NaN
(5, 6]  3  y  False  NaN
(6, 7]  3  y    NaN  2.0
```

```
>>> piso.join(df, s, df2, how="right")
        D  A  B    C
(1, 2]  1  4  x  NaN
(6, 7]  2  3  y  NaN
```

3.1.15 piso.adjacency_matrix

piso.**adjacency_matrix**(*interval_array*, *edges='intersect'*, *include_index=True*)

Returns a 2D array (or dataframe) of boolean values indicating edges between nodes in a graph.

The set of nodes correspond to intervals and the edges are defined by the relationship defined by the *edges* parameter.

Note that the diagonal is defined with False values by default.

Parameters

interval_array [pandas.arrays.IntervalArray or pandas.IntervalIndex] Contains the intervals.

edges [{"intersect", "disjoint"}, default "intersect"] Defines the relationship that edges between nodes represent.

include_index [bool, default True] If True then a pandas.DataFrame, indexed by the intervals, is returned. If False then a numpy.ndarray is returned.

Returns

:class:`pandas.DataFrame` or [class:*numpy.ndarray*] Boolean valued, symmetrical, with False along diagonal.

Examples

```
>>> import pandas as pd
>>> import piso
```

```
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0,4), (3,6), (5, 7), (8,9), (9,10)],
...     closed="both",
... )
```

```
>>> piso.adjacency_matrix(arr)
         [0, 4]  [3, 6]  [5, 7]  [8, 9]  [9, 10]
[0, 4]    False    True   False   False    False
[3, 6]     True   False    True   False    False
[5, 7]    False    True   False   False    False
[8, 9]    False   False   False   False     True
[9, 10]   False   False   False    True    False
```

```
>>> piso.adjacency_matrix(arr, include_index=False)
array([[False,  True, False, False, False],
       [ True, False,  True, False, False],
       [False,  True, False, False, False],
       [False, False, False, False,  True],
       [False, False, False,  True, False]])
```

```
>>> piso.adjacency_matrix(arr, edges="disjoint")
         [0, 4]  [3, 6]  [5, 7]  [8, 9]  [9, 10]
[0, 4]    False   False    True    True     True
[3, 6]    False   False   False    True     True
[5, 7]     True   False   False    True     True
[8, 9]     True    True    True   False    False
[9, 10]    True    True    True   False    False
```

3.2 Accessors

ArrayAccessor.union(*interval_arrays[, ...])	Performs a set union operation.
ArrayAccessor.intersection(*interval_arrays)	Performs a set intersection operation.
ArrayAccessor.difference(*interval_arrays[, ...])	Performs a set difference operation.
ArrayAccessor.symmetric_difference(...[, ...])	Performs a set symmetric difference operation.
ArrayAccessor.isdisjoint(*interval_arrays)	Indicates whether one, or more, sets are disjoint or not.
ArrayAccessor.issuperset(*interval_arrays[, ...])	Indicates whether a set is a superset of one, or more, other sets.
ArrayAccessor.issubset(*interval_arrays[, ...])	Indicates whether a set is a subset of one, or more, other sets.
ArrayAccessor.coverage([domain, bins, how])	Calculates the size of a domain (or possibly multiple domains) covered by a collection of intervals.
ArrayAccessor.complement([domain])	Calculates the complement of a collection of intervals (in an array) over some domain.
ArrayAccessor.contains(x[, include_index, ...])	Evaluates the intersection of a set of intervals with a set of points.
ArrayAccessor.split(x)	Given a set of intervals, and break points, splits the intervals into pieces wherever the overlap a break point.
ArrayAccessor.adjacency_matrix([edges, ...])	Returns a 2D array (or dataframe) of boolean values indicating edges between nodes in a graph.

3.2.1 piso.accessor.ArrayAccessor.union

ArrayAccessor.**union**(**interval_arrays*, *squeeze=False*, *return_type='infer'*)

Performs a set union operation.

What is considered a set is determined by the number of positional arguments used, that is, determined by the size of *interval_arrays*.

If *interval_arrays* is empty then the sets are considered to be the intervals contained in the array object the accessor belongs to (an instance of `pandas.IntervalIndex`, `pandas.arrays.IntervalArray`).

If *interval_arrays* is not empty then the sets are considered to be the elements in *interval_arrays*, in addition to the intervals in the array object the accessor belongs to. Each of these arrays is assumed to contain disjoint

intervals (and satisfy the definition of a set). Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

Parameters

***interval_arrays** [argument list of pandas.IntervalIndex or pandas.arrays.IntervalArray] May contain zero or more arguments.

squeeze [boolean, default False] If True, will try to coerce the return value to a single pandas.Interval. If supplied, must be done so as a keyword argument.

return_type [{"infer", pandas.IntervalIndex, pandas.arrays.IntervalArray}, default "infer"] If "infer" the return type will be the same as *interval_array*. If supplied, must be done so as a keyword argument.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso
>>> piso.register_accessors()
```

Examples with *interval_arrays* empty:

```
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5), (3, 6), (7, 8), (8, 9), (10, 12)],
... )
```

```
>>> arr.piso.union()
<IntervalArray>
[(0.0, 6.0], (7.0, 9.0], (10.0, 12.0]]
Length: 3, closed: right, dtype: interval[float64]
```

```
>>> arr.set_closed("left").piso.union()
<IntervalArray>
[[0, 4), [2, 5), [3, 6), [7, 8), [8, 9), [10, 12)]
Length: 6, closed: left, dtype: interval[int64]
```

```
>>> pd.IntervalIndex(arr).piso.union()
IntervalIndex([(0.0, 6.0], (7.0, 9.0], (10.0, 12.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> arr.piso.union(return_type=pd.IntervalIndex)
IntervalIndex([(0.0, 6.0], (7.0, 9.0], (10.0, 12.0]],
              closed='right',
              dtype='interval[float64]')
```

Examples with *interval_arrays* non empty:

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (5, 6), (7, 8), (10, 12)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(3, 5), (8, 9)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(6, 8), (9, 11)],
... )
```

```
>>> arr1.piso.union(arr2)
<IntervalArray>
[(0.0, 6.0], (7.0, 9.0], (10.0, 12.0]]
Length: 3, closed: right, dtype: interval[float64]
```

```
>>> arr2.piso.union(arr3, return_type=pd.IntervalIndex)
IntervalIndex([(3.0, 5.0], (6.0, 11.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> arr1.piso.union(arr2, arr3)
<IntervalArray>
[(0.0, 12.0]]
Length: 1, closed: right, dtype: interval[float64]
```

```
>>> arr1.piso.union(arr2, arr3, squeeze=True)
Interval(0.0, 12.0, closed='right')
```

3.2.2 piso.accessor.ArrayAccessor.intersection

ArrayAccessor.**intersection**(**interval_arrays*, *min_overlaps='all'*, *squeeze=False*, *return_type='infer'*)

Performs a set intersection operation.

What is considered a set is determined by the number of positional arguments used, that is, determined by the size of *interval_arrays*.

If *interval_arrays* is empty then the sets are considered to be the intervals contained in the array object the accessor belongs to (an instance of `pandas.IntervalIndex`, `pandas.arrays.IntervalArray`).

If *interval_arrays* is not empty then the sets are considered to be the elements in *interval_arrays*, in addition to the intervals in the array object the accessor belongs to. Each of these arrays is assumed to contain disjoint intervals (and satisfy the definition of a set). Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

Parameters

***interval_arrays** [argument list of `pandas.IntervalIndex` or `pandas.arrays.IntervalArray`] May contain zero or more arguments.

min_overlaps [int or "all", default "all"] Specifies the minimum number of intervals which overlap in order to define an *intersection*. If *min_overlaps* is an int then it must be no smaller than 2. If *min_overlaps* is all then an intersection is only defined where every interval overlaps. If supplied, must be done so as a keyword argument.

squeeze [boolean, default False] If True, will try to coerce the return value to a single pandas.Interval. If supplied, must be done so as a keyword argument.

return_type [{"infer", pandas.IntervalIndex, pandas.arrays.IntervalArray}, default "infer"] If "infer" the return type will be the same as *interval_array*. If supplied, must be done so as a keyword argument.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso
>>> piso.register_accessors()
```

Examples with *interval_arrays* empty:

```
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5), (3, 6)],
... )
```

```
>>> arr.piso.intersection()
<IntervalArray>
[(3.0, 4.0]]
Length: 1, closed: right, dtype: interval[float64]
```

```
>>> pd.IntervalIndex(arr).piso.intersection()
IntervalIndex([(3.0, 4.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> arr.piso.intersection(return_type=pd.IntervalIndex)
IntervalIndex([(3.0, 4.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> arr.piso.intersection(min_overlaps=2)
<IntervalArray>
[(2.0, 5.0]]
Length: 1, closed: right, dtype: interval[float64]
```

```
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5), (3, 6), (7, 8), (8, 9), (10, 12)],
... )
```

```
>>> arr.piso.intersection()
<IntervalArray>
[]
Length: 0, closed: right, dtype: interval[int64]
```

```
>>> arr.piso.intersection(min_overlaps=2)
<IntervalArray>
[(2.0, 5.0]]
Length: 1, closed: right, dtype: interval[float64]
```

Examples with *interval_arrays* not empty:

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (5, 6), (7, 8), (10, 12)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(3, 5), (8, 9)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(6, 8), (9, 11)],
... )
```

```
>>> arr1.piso.intersection(arr2)
<IntervalArray>
[(3.0, 4.0]]
Length: 1, closed: right, dtype: interval[float64]
```

```
>>> arr1.piso.intersection(arr2, squeeze=True)
Interval(3.0, 4.0, closed='right')
```

```
>>> arr1.piso.intersection(arr2, arr3)
<IntervalArray>
[]
Length: 0, closed: right, dtype: interval[float64]
```

```
>>> arr1.piso.intersection(arr2, arr3, min_overlaps=2)
<IntervalArray>
[(3.0, 4.0], (10.0, 11.0]]
Length: 2, closed: right, dtype: interval[float64]
```

3.2.3 piso.accessor.ArrayAccessor.difference

ArrayAccessor.**difference**(**interval_arrays*, *squeeze=False*, *return_type='infer'*)

Performs a set difference operation.

The array elements of *interval_arrays*, and the interval array object the accessor belongs to (an instance of `pandas.IntervalIndex`, `pandas.arrays.IntervalArray`) are considered to be the sets over which the operation is performed. Each of these arrays is assumed to contain disjoint intervals (and satisfy the definition of a set). Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

The list *interval_arrays* must contain at least one element. If *interval_arrays* contains a single element then the result is the set difference between the interval array the accessor belongs to, and this single element. If *interval_arrays* contains multiple elements then the result is the set difference between the interval array the accessor belongs to and the union of the sets in *interval_arrays*. This is equivalent to iteratively applying a set difference operation with each array in *interval_arrays* as the second operand.

Parameters

***interval_arrays** [argument list of pandas.IntervalIndex or pandas.arrays.IntervalArray] Must contain at least one argument.

squeeze [boolean, default False] If True, will try to coerce the return value to a single pandas.Interval. If supplied, must be done so as a keyword argument.

return_type [{"infer", pandas.IntervalIndex, pandas.arrays.IntervalArray}, default "infer"] If "infer" the return type will be the same as *interval_array*. If supplied, must be done so as a keyword argument.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso
>>> piso.register_accessors()
```

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5), (3, 6), (7, 8), (8, 9), (10, 12)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(4, 7), (8, 11)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(2, 5), (7, 13)],
... )
```

```
>>> arr1.piso.difference(arr2)
<IntervalArray>
[(0.0, 4.0], (7.0, 8.0], (11.0, 12.0]]
Length: 3, closed: right, dtype: interval[float64]
```

```
>>> arr1.set_closed("left").piso.difference(arr2.set_closed("left"))
<IntervalArray>
[[0.0, 4.0), [7.0, 8.0), [11.0, 12.0)]
Length: 3, closed: left, dtype: interval[float64]
```

```
>>> arr1.piso.difference(arr2, return_type=pd.IntervalIndex)
IntervalIndex([(0.0, 4.0], (7.0, 8.0], (11.0, 12.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> arr1.piso.difference(arr2, arr3)
<IntervalArray>
[(0.0, 2.0]]
Length: 1, closed: right, dtype: interval[float64]
```

```
>>> arr1.piso.difference(arr2, arr3, squeeze=True)
Interval(0.0, 2.0, closed='right')
```

3.2.4 piso.accessor.ArrayAccessor.symmetric_difference

ArrayAccessor.**symmetric_difference**(**interval_arrays*, *min_overlaps=2*, *squeeze=False*, *return_type='infer'*)

Performs a set symmetric difference operation.

What is considered a set is determined by the number of positional arguments used, that is, determined by the size of *interval_arrays*.

If *interval_arrays* is empty then the sets are considered to be the intervals contained in the array object the accessor belongs to (an instance of `pandas.IntervalIndex`, `pandas.arrays.IntervalArray`).

If *interval_arrays* is not empty then the sets are considered to be the elements in *interval_arrays*, in addition to the intervals in the array object the accessor belongs to. Each of these arrays is assumed to contain disjoint intervals (and satisfy the definition of a set). Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

The symmetric difference can be defined as the set difference, of the union and the intersection. The parameter *min_overlaps* in *piso.intersection()*, which defines the minimum number of intervals in an overlap required to constitute an intersection, follows through to symmetric difference under this definition.

Parameters

***interval_arrays** [argument list of `pandas.IntervalIndex` or `pandas.arrays.IntervalArray`] May contain zero or more arguments.

min_overlaps [int or "all", default "all"] Specifies the minimum number of intervals which overlap in order to define an *intersection*. If *min_overlaps* is an int then it must be no smaller than 2. If *min_overlaps* is all then an intersection is only defined where every interval overlaps. If supplied, must be done so as a keyword argument.

squeeze [boolean, default False] If True, will try to coerce the return value to a single pandas.Interval. If supplied, must be done so as a keyword argument.

return_type [{"infer", `pandas.IntervalIndex`, `pandas.arrays.IntervalArray`}, default "infer"] If "infer" the return type will be the same as *interval_array*. If supplied, must be done so as a keyword argument.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso
>>> piso.register_accessors()
```

Examples with *interval_arrays* empty:

```
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5), (3, 6), (7, 8), (8, 9), (10, 12)],
... )
```

```
>>> arr.piso.symmetric_difference()
<IntervalArray>
[(0.0, 2.0], (5.0, 6.0], (7.0, 9.0], (10.0, 12.0]]
Length: 4, closed: right, dtype: interval[float64]
```

```
>>> pd.IntervalIndex(arr).piso.symmetric_difference()
IntervalIndex([(0.0, 2.0], (5.0, 6.0], (7.0, 9.0], (10.0, 12.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> arr.piso.symmetric_difference(return_type=pd.IntervalIndex)
IntervalIndex([(0.0, 2.0], (5.0, 6.0], (7.0, 9.0], (10.0, 12.0]],
              closed='right',
              dtype='interval[float64]')
```

```
>>> arr.piso.symmetric_difference(min_overlaps=3)
<IntervalArray>
[(0.0, 3.0], (4.0, 6.0], (7.0, 9.0], (10.0, 12.0]]
Length: 4, closed: right, dtype: interval[float64]
```

```
>>> arr.piso.symmetric_difference(min_overlaps="all")
<IntervalArray>
[(0.0, 6.0], (7.0, 9.0], (10.0, 12.0]]
Length: 3, closed: right, dtype: interval[float64]
```

Examples with *interval_arrays* non-empty:

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (5, 6), (7, 8), (10, 12)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(3, 5), (8, 9)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(6, 8), (9, 11)],
... )
```

```
>>> arr1.piso.symmetric_difference(arr2)
<IntervalArray>
[(0.0, 3.0], (4.0, 6.0], (7.0, 9.0], (10.0, 12.0]]
Length: 4, closed: right, dtype: interval[float64]
```

```
>>> arr1.piso.symmetric_difference(arr2, arr3)
<IntervalArray>
[(0.0, 3.0], (4.0, 7.0], (8.0, 10.0], (11.0, 12.0]]
Length: 4, closed: right, dtype: interval[float64]
```

```
>>> arr1.piso.symmetric_difference(arr2, arr3, min_overlaps="all")
<IntervalArray>
[(0.0, 12.0]]
Length: 1, closed: right, dtype: interval[float64]
```

3.2.5 piso.accessor.ArrayAccessor.isdisjoint

ArrayAccessor.**isdisjoint**(**interval_arrays*)

Indicates whether one, or more, sets are disjoint or not.

interval_array must be left-closed or right-closed if *interval_arrays is non-empty. If no arguments are provided then this restriction does not apply.

What is considered a set is determined by the number of positional arguments used, that is, determined by the size of *interval_arrays*.

If *interval_arrays* is empty then the sets are considered to be the intervals contained in the array object the accessor belongs to (an instance of `pandas.IntervalIndex`, `pandas.arrays.IntervalArray`).

If *interval_arrays* is not empty then the sets are considered to be the elements in *interval_arrays*, in addition to the intervals in the array object the accessor belongs to. Each of these arrays is assumed to contain disjoint intervals (and satisfy the definition of a set). Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

Parameters

***interval_arrays** [argument list of `pandas.IntervalIndex` or `pandas.arrays.IntervalArray`] May contain zero or more arguments.

Returns

boolean

Examples

```
>>> import pandas as pd
>>> import piso
>>> piso.register_accessors()
```

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 3), (2, 4)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(4, 7), (8, 11)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(2, 4), (7, 8)],
... )
```

```
>>> arr1.piso.isdisjoint()
False
```

```
>>> arr2.piso.isdisjoint()
True
```

```
>>> arr1.piso.isdisjoint(arr2)
True
```

```
>>> arr1.piso.isdisjoint(arr3)
False
```

3.2.6 piso.accessor.ArrayAccessor.issuperset

ArrayAccessor.**issuperset**(**interval_arrays*, *squeeze=False*)

Indicates whether a set is a superset of one, or more, other sets.

The array elements of *interval_arrays*, and the interval array object the accessor belongs to (an instance of pandas.IntervalIndex, pandas.arrays.IntervalArray) are considered to be the sets over which the operation is performed. Each of these arrays is assumed to contain disjoint intervals (and satisfy the definition of a set). Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

The list *interval_arrays* must contain at least one element. The superset comparison is iteratively applied between the interval array the accessor belongs to, and each array in *interval_arrays*. When *interval_arrays* contains multiple interval arrays, the return type will be a numpy array. If it contains one interval array then the result can be coerced to a single boolean using the *squeeze* parameter.

Parameters

***interval_arrays** [argument list of pandas.IntervalIndex or pandas.arrays.IntervalArray] Must contain at least one argument.

squeeze [boolean, default True] If True, will try to coerce the return value to a single pandas.Interval. If supplied, must be done so as a keyword argument.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso
>>> piso.register_accessors()
```

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (3, 6), (7, 8), (10, 12)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(2, 5), (7, 8)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(3, 4), (10, 11)],
... )
```

```
>>> arr1.piso.issuperset(arr2)
True
```

```
>>> arr1.piso.issuperset(arr2, squeeze=False)
array([ True])
```

```
>>> arr1.piso.issuperset(arr2, arr3)
array([ True, True])
```

```
>>> arr2.piso.issuperset(arr3)
False
```

3.2.7 piso.accessor.ArrayAccessor.issubset

ArrayAccessor.**issubset**(**interval_arrays*, *squeeze=False*)

Indicates whether a set is a subset of one, or more, other sets.

The array elements of *interval_arrays*, and the interval array object the accessor belongs to (an instance of `pandas.IntervalIndex`, `pandas.arrays.IntervalArray`) are considered to be the sets over which the operation is performed. Each of these arrays is assumed to contain disjoint intervals (and satisfy the definition of a set). Any array containing overlaps between intervals will be mapped to one with disjoint intervals via a union operation.

The list *interval_arrays* must contain at least one element. The subset comparison is iteratively applied between the interval array the accessor belongs to, and each array in *interval_arrays*. When *interval_arrays* contains multiple interval arrays, the return type will be a numpy array. If it contains one interval array then the result can be coerced to a single boolean using the *squeeze* parameter.

Parameters

***interval_arrays** [argument list of `pandas.IntervalIndex` or `pandas.arrays.IntervalArray`] Must contain at least one argument.

squeeze [boolean, default True] If True, will try to coerce the return value to a single pandas.Interval. If supplied, must be done so as a keyword argument.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso
>>> piso.register_accessors()
```

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(2, 5), (7, 8)],
... )
>>> arr2 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (3, 6), (7, 8), (10, 12)],
... )
>>> arr3 = pd.arrays.IntervalArray.from_tuples(
...     [(3, 4), (10, 11)],
... )
```

```
>>> arr1.piso.issubset(arr2)
True
```

```
>>> arr1.piso.issubset(arr2, squeeze=False)
array([ True])
```

```
>>> arr1.piso.issubset(arr2, arr3)
array([ True, False])
```

```
>>> arr1.piso.issubset(arr3)
False
```

3.2.8 piso.accessor.ArrayAccessor.coverage

ArrayAccessor.**coverage**(*domain=None*, *bins=False*, *how='fraction'*)

Calculates the size of a domain (or possibly multiple domains) covered by a collection of intervals.

The intervals are contained in the array object the accessor belongs to. The (possibly overlapping) intervals may not, or partially, or wholly cover the domain.

Calculation over multiple domains is only possible when *bins* = True.

Parameters

domain [tuple, pandas.Interval, pandas.IntervalIndex or pandas.arrays.IntervalArray, optional] Specifies the domain over which to calculate the "coverage". If *domain* is *None*, then the domain is considered to be the extremities of the intervals contained in the interval array the accessor belongs to. If *domain* is a tuple then it should specify lower and upper bounds, and be equivalent to a pandas.Interval. If *domain* is a pandas.IntervalIndex or pandas.arrays.IntervalArray then the intervals it contains define a possibly disconnected domain. If *bins* = True then *domain* must be pandas.IntervalIndex or pandas.arrays.IntervalArray with disjoint intervals.

bins [boolean, default False] If False, then the *domain* is interpreted as a single domain and returns one value. If True, then *domain* is interpreted as disjoint bins over which coverage is calculated for each.

how [{"fraction", "sum"}, default "fraction"] If *how* = "fraction" then the result is a fraction of the size of the domain. If *how* = "sum" then the result is the length of the domain covered.

New in version 0.8.0.

Returns

float or [class:*pandas.Series*]

Examples

```
>>> import pandas as pd
>>> import piso
>>> piso.register_accessors()
```

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (3, 5), (7, 8)],
... )
```

```
>>> arr1.piso.coverage()
0.75
```

```
>>> arr1.piso.coverage((0, 10))
0.6
```

```
>>> arr1.piso.coverage(pd.Interval(-10, 10))
0.3
```

```
>>> arr1.piso.coverage(pd.Interval(-10, 10), how="sum")
6
```

```
>>> domain = pd.arrays.IntervalArray.from_tuples(
...     [(4,6), (7, 10)],
... )
>>> arr1.piso.coverage(domain)
0.4
```

```
>>> arr1.piso.coverage(domain, bins=True)
(4, 6]     0.500000
(7, 10]    0.333333
dtype: float64
```

```
>>> arr1.piso.coverage(domain, bins=True, how="sum")
(4, 6]     1.0
(7, 10]    1.0
dtype: float64
```

3.2.9 piso.accessor.ArrayAccessor.complement

ArrayAccessor.**complement**(*domain=None*)

Calculates the complement of a collection of intervals (in an array) over some domain.

Equivalent to the set difference of the domain and the intervals in the array that the accessor belongs to.

Parameters

domain [tuple, pandas.Interval, pandas.IntervalIndex or pandas.arrays.IntervalArray, optional] Specifies the domain over which to calculate the "complement". If *domain* is *None*, then the domain is considered to be the extremities of the intervals contained in the interval array that the accessor belongs to. If *domain* is a tuple then it should specify lower and upper bounds, and be equivalent to a pandas.Interval. If *domain* is a pandas.IntervalIndex or pandas.arrays.IntervalArray then the intervals it contains define a possibly disconnected domain.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*] The return type will be the same as the interval array object the accessor belongs to.

Examples

```
>>> import pandas as pd
>>> import piso
```

```
>>> arr1 = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (3, 5), (7, 8)],
... )
```

```
>>> arr1.piso.complement()
<IntervalArray>
[(5, 7]]
Length: 1, closed: right, dtype: interval[int64]
```

```
>>> arr1.piso.complement((-5, 10))
<IntervalArray>
[(-5, 0], (5, 7], (8, 10]]
Length: 3, closed: right, dtype: interval[int64]
```

```
>>> arr1.piso.complement(pd.Interval(-5, 6))
<IntervalArray>
[(-5, 0], (5, 6]]
Length: 2, closed: right, dtype: interval[int64]
```

```
>>> domain = pd.arrays.IntervalArray.from_tuples(
...     [(-5,-2), (7,10)],
... )
```

```
>>> arr1.piso.complement(domain)
<IntervalArray>
[(-5, -2], (8, 10]]
Length: 2, closed: right, dtype: interval[int64]
```

3.2.10 piso.accessor.ArrayAccessor.contains

ArrayAccessor.**contains**(*x*, *include_index=True*, *result='cartesian'*, *how='any'*)

Evaluates the intersection of a set of intervals with a set of points.

The format of the result is dependent on the *result* parameter. If *result = "cartesian"* then the the function returns a 2-dimensional boolean mask *M* of shape *(m,n)* where *m* is the number of intervals, and *n* is the number of points. The element in the i-th row and j-th column is True if the i-th interval contains the j-th point.

If *result = "points"* then the result is a 1-dimensional boolean mask of length *n*. If *result = "intervals"* then the result is a 1-dimensional boolean mask of length *m*.

Parameters

interval_array [pandas.IntervalIndex or pandas.arrays.IntervalArray] Contains the intervals. May be left-closed, right-closed, both, or neither.

x [scalar, or array-like of scalars] Values in *x* should belong to the same domain as the intervals contained by the object the accessor belongs to.

include_index [boolean, default True] Indicates whether to return a numpy.ndarray or pandas.DataFrame indexed by *interval_array* and column names equal to *x*

result [{"cartesian", "points", "intervals"}, default "cartesian"] If *result* = "cartesian" then the result will be two dimensional, otherwise it will be one dimensional.

how [{"any", "all"}, default "any"] Only relevant if *result* is not "cartesian". This parameter indicates either: - a True value means any or all points are contained within an interval, or - a True value means any or all intervals contained a point. Which of these interpretations is dependent on the *result* parameter.

Returns

:class:`numpy.ndarray`, [class:*pandas.DataFrame* or pandas.Series] One, or two, dimensional and boolean valued. Return type dependent on *include_index* and *result*.

Examples

```
>>> import pandas as pd
>>> import piso
>>> piso.register_accessors()
```

```
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5)],
... )
```

```
>>> arr.piso.contains(1)
           1
(0, 4]   True
(2, 5]  False
```

```
>>> arr.piso.contains([0, 1, 3, 4])
            0      1     3     4
(0, 4]  False   True  True  True
(2, 5]  False  False  True  True
```

```
>>> arr.piso.contains([0, 1, 3, 4], include_index=False)
array([[False,  True,  True,  True],
       [False, False,  True,  True]])
```

```
>>> arr.piso.contains([0, 1, 3, 4], result="points")
0    False
1     True
3     True
4     True
dtype: bool
```

```
>>> arr.piso.contains([0, 1, 3, 4], result="points", how="all")
0    False
1    False
3     True
4     True
dtype: bool
```

```
>>> arr.piso.contains([0, 1, 3, 4], result="intervals")
(0, 4]    True
(2, 5]    True
dtype: bool
```

```
>>> pd.IntervalIndex.from_tuples([(0,2)]).piso.contains(1, include_index=False)
array([[ True]])
```

3.2.11 piso.accessor.ArrayAccessor.split

ArrayAccessor.**split**(*x*)

Given a set of intervals, and break points, splits the intervals into pieces wherever the overlap a break point.

The intervals are contained in the object the accessor belongs to. They may be left-closed, right-closed, both, or neither, and contain overlapping intervals.

Parameters

x [scalar, or array-like of scalars] Values in *x* should belong to the same domain as the intervals in *interval_array*. May contain duplicates and be unsorted.

Returns

:class:`pandas.IntervalIndex` or [class:*pandas.arrays.IntervalArray*] Return type will be the same type as the object the accessor belongs to.

Examples

```
>>> import pandas as pd
>>> import piso
>>> piso.register_accessors()
```

```
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0, 4), (2, 5)],
... )
```

```
>>> arr.piso.split(3)
<IntervalArray>
[(0, 3], (3, 4], (2, 3], (3, 5]]
Length: 4, closed: right, dtype: interval[int64]
```

```
>>> arr.piso.split([3,3,3,3])
<IntervalArray>
[(0, 3], (3, 4], (2, 3], (3, 5]]
Length: 4, closed: right, dtype: interval[int64]
```

```
>>> arr = pd.IntervalIndex.from_tuples(
...     [(0, 4), (2, 5)], closed="neither",
... )
```

```
>>> arr.piso.split([1, 6, 4])
IntervalIndex([(0.0, 1.0), (1.0, 4.0), (2.0, 4.0), (4.0, 5.0)],
              closed='neither',
              dtype='interval[float64]')
```

3.2.12 piso.accessor.ArrayAccessor.adjacency_matrix

ArrayAccessor.**adjacency_matrix**(*edges='intersect'*, *include_index=True*)

Returns a 2D array (or dataframe) of boolean values indicating edges between nodes in a graph.

The set of nodes correspond to intervals and the edges are defined by the relationship defined by the *edges* parameter.

Note that the diagonal is defined with False values by default.

Parameters

edges [{"intersect", "disjoint"}, default "intersect"] Defines the relationship that edges between nodes represent.

include_index [bool, default True] If True then a `pandas.DataFrame`, indexed by the intervals, is returned. If False then a `numpy.ndarray` is returned.

Returns

:class:`pandas.DataFrame` or [class:*numpy.ndarray*] Boolean valued, symmetrical, with False along diagonal.

Examples

```
>>> import pandas as pd
>>> import piso
>>> piso.register_accessors()
```

```
>>> arr = pd.arrays.IntervalArray.from_tuples(
...     [(0,4), (3,6), (5, 7), (8,9), (9,10)],
...     closed="both",
... )
```

```
>>> arr.piso.adjacency_matrix()
         [0, 4]  [3, 6]  [5, 7]  [8, 9]  [9, 10]
[0, 4]    False    True   False   False    False
[3, 6]     True   False    True   False    False
[5, 7]    False    True   False   False    False
[8, 9]    False   False   False   False     True
[9, 10]   False   False   False    True    False
```

```
>>> arr.piso.adjacency_matrix(arr, include_index=False)
array([[False,  True, False, False, False],
       [ True, False,  True, False, False],
       [False,  True, False, False, False],
       [False, False, False, False,  True],
       [False, False, False,  True, False]])
```

```
>>> arr.piso.adjacency_matrix(arr, edges="disjoint")
         [0, 4]  [3, 6]  [5, 7]  [8, 9]  [9, 10]
[0, 4]    False   False    True    True     True
[3, 6]    False   False   False    True     True
[5, 7]     True   False   False    True     True
```

(continues on next page)

(continued from previous page)

```
[8, 9]     True    True    True   False    False
[9, 10]    True    True    True   False    False
```

3.3 Interval

union(interval1, interval2[, squeeze])	Performs the union of two pandas.Interval
intersection(interval1, interval2[, squeeze])	Performs the intersection of two pandas.Interval
difference(interval1, interval2[, squeeze])	Performs the set difference of two pandas.Interval
symmetric_difference(interval1, interval2[, ...])	Performs the symmetric difference of two pandas.Interval
issuperset(interval, *intervals[, squeeze])	Indicates whether one pandas.Interval is a superset of one, or more, others.
issubset(interval, *intervals[, squeeze])	Indicates whether one pandas.Interval is a subset of one, or more, others.

3.3.1 piso.interval.union

piso.interval.**union**(*interval1*, *interval2*, *squeeze=True*)

Performs the union of two pandas.Interval

Parameters

interval1 [pandas.Interval] the first operand

interval2 [pandas.Interval] the second operand

squeeze [boolean, default True] If True, will try to coerce the return value to a pandas.Interval

Returns

:class:`pandas.Interval` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso.interval
```

```
>>> piso.interval.union(
...     pd.Interval(0, 3),
...     pd.Interval(2, 4),
... )
Interval(0.0, 4.0, closed='right')
```

```
>>> piso.interval.union(
...     pd.Interval(0, 3),
...     pd.Interval(2, 4),
...     squeeze=False,
... )
```

(continues on next page)

(continued from previous page)

```
<IntervalArray>
[(0.0, 4.0]]
Length: 1, closed: right, dtype: interval[float64]
```

```
>>> piso.interval.union(
...     pd.Interval(0, 3, closed="left"),
...     pd.Interval(2, 4, closed="left"),
... )
Interval(0.0, 4.0, closed='left')
```

```
>>> piso.interval.union(
...     pd.Interval(0, 1),
...     pd.Interval(3, 4),
... )
<IntervalArray>
[(0.0, 1.0], (3.0, 4.0]]
Length: 2, closed: right, dtype: interval[float64]
```

3.3.2 piso.interval.intersection

piso.interval.**intersection**(*interval1*, *interval2*, *squeeze=True*)

Performs the intersection of two `pandas.Interval`

Parameters

interval1 [pandas.Interval] the first operand

interval2 [pandas.Interval] the second operand

squeeze [boolean, default True] If True, will try to coerce the return value to a `pandas.Interval`

Returns

:class:`pandas.Interval` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso.interval
```

```
>>> piso.interval.intersection(
...     pd.Interval(0, 3),
...     pd.Interval(2, 4),
... )
Interval(2.0, 3.0, closed='right')
```

```
>>> piso.interval.intersection(
...     pd.Interval(0, 3),
...     pd.Interval(2, 4),
...     squeeze=False,
```

(continues on next page)

(continued from previous page)

```
... )
<IntervalArray>
[(2.0, 3.0]]
Length: 1, closed: right, dtype: interval[float64]
```

```
>>> piso.interval.intersection(
...     pd.Interval(0, 3, closed="left"),
...     pd.Interval(2, 4, closed="left"),
... )
Interval(2.0, 3.0, closed='left')
```

```
>>> piso.interval.intersection(
...     pd.Interval(0, 1),
...     pd.Interval(3, 4),
... )
<IntervalArray>
[]
Length: 0, closed: right, dtype: interval[int64]
```

3.3.3 piso.interval.difference

piso.interval.**difference**(*interval1*, *interval2*, *squeeze=True*)

Performs the set difference of two pandas.Interval

Parameters

interval1 [pandas.Interval] the first operand

interval2 [pandas.Interval] the second operand

squeeze [boolean, default True] If True, will try to coerce the return value to a pandas.Interval

Returns

:class:`pandas.Interval` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso.interval
```

```
>>> piso.interval.difference(
...     pd.Interval(0, 3),
...     pd.Interval(2, 4),
... )
Interval(0.0, 2.0, closed='right')
```

```
>>> piso.interval.difference(
...     pd.Interval(0, 3),
...     pd.Interval(2, 4),
```

(continues on next page)

(continued from previous page)

```
...     squeeze=False,
... )
<IntervalArray>
[(0.0, 2.0]]
Length: 1, closed: right, dtype: interval[float64]
```

```
>>> piso.interval.difference(
...     pd.Interval(0, 4, closed="left"),
...     pd.Interval(2, 3, closed="left"),
... )
<IntervalArray>
[[0.0, 2.0), [3.0, 4.0)]
Length: 2, closed: left, dtype: interval[float64]
```

```
>>> piso.interval.difference(
...     pd.Interval(2, 3),
...     pd.Interval(0, 4),
... )
<IntervalArray>
[]
Length: 0, closed: right, dtype: interval[int64]
```

3.3.4 piso.interval.symmetric_difference

piso.interval.**symmetric_difference**(*interval1*, *interval2*, *squeeze=True*)

Performs the symmetric difference of two `pandas.Interval`

Parameters

interval1 [pandas.Interval] the first operand

interval2 [pandas.Interval] the second operand

squeeze [boolean, default True] If True, will try to coerce the return value to a `pandas.Interval`

Returns

:class:`pandas.Interval` or [class:*pandas.arrays.IntervalArray*]

Examples

```
>>> import pandas as pd
>>> import piso.interval
```

```
>>> piso.interval.symmetric_difference(
...     pd.Interval(0, 3),
...     pd.Interval(2, 4),
... )
<IntervalArray>
[(0.0, 2.0], (3.0, 4.0]]
Length: 2, closed: right, dtype: interval[float64]
```

```
>>> piso.interval.symmetric_difference(
...     pd.Interval(0, 3),
...     pd.Interval(2, 3),
... )
Interval(0.0, 2.0, closed='right')
```

```
>>> piso.interval.symmetric_difference(
...     pd.Interval(0, 3, closed="left"),
...     pd.Interval(2, 4, closed="left"),
... )
<IntervalArray>
[[0.0, 2.0), [3.0, 4.0)]
Length: 2, closed: left, dtype: interval[float64]
```

```
>>> piso.interval.symmetric_difference(
...     pd.Interval(2, 3),
...     pd.Interval(0, 4),
... )
<IntervalArray>
[(0.0, 2.0], (3.0, 4.0]]
Length: 2, closed: right, dtype: interval[float64]
```

3.3.5 piso.interval.issuperset

piso.interval.**issuperset**(*interval*, **intervals*, *squeeze=True*)

Indicates whether one `pandas.Interval` is a superset of one, or more, others.

Parameters

interval [`pandas.Interval`] An interval, against which all other intervals belonging to *intervals* are compared.

***intervals** [argument list of `pandas.Interval`] Must contain at least one argument.

squeeze [boolean, default True] If True, will try to coerce the return value to a single boolean

Returns

boolean, or [class:*numpy.ndarray* of booleans]

Examples

```
>>> import pandas as pd
>>> import piso.interval
```

```
>>> piso.interval.issuperset(
...     pd.Interval(1, 4),
...     pd.Interval(2, 4),
... )
True
```

```
>>> piso.interval.issuperset(
...     pd.Interval(1, 4),
...     pd.Interval(0, 3),
... )
False
```

```
>>> piso.interval.issuperset(
...     pd.Interval(1, 4),
...     pd.Interval(2, 4),
...     pd.Interval(0, 3),
... )
array([ True,  False])
```

```
>>> piso.interval.issuperset(
...     pd.Interval(0, 3),
...     pd.Interval(0, 3),
...     squeeze=False
... )
array([ True])
```

3.3.6 piso.interval.issubset

piso.interval.**issubset**(*interval*, **intervals*, *squeeze=True*)

Indicates whether one `pandas.Interval` is a subset of one, or more, others.

Parameters

interval [`pandas.Interval`] An interval, against which all other intervals belonging to *intervals* are compared.

***intervals** [argument list of `pandas.Interval`] Must contain at least one argument.

squeeze [boolean, default True] If True, will try to coerce the return value to a single boolean

Returns

boolean, or [class:*numpy.ndarray* of booleans]

Examples

```
>>> import pandas as pd
>>> import piso.interval
```

```
>>> piso.interval.issubset(
...     pd.Interval(2, 4),
...     pd.Interval(1, 4),
... )
True
```

```
>>> piso.interval.issubset(
...     pd.Interval(2, 4),
...     pd.Interval(0, 3),
```

(continues on next page)

(continued from previous page)

```
... )
False
```

```
>>> piso.interval.issubset(
...     pd.Interval(2, 4),
...     pd.Interval(1, 4),
...     pd.Interval(0, 3),
... )
array([ True,  False])
```

```
>>> piso.interval.issubset(
...     pd.Interval(1, 4),
...     pd.Interval(1, 4),
...     squeeze=False
... )
array([ True])
```

CHAPTER

FOUR

RELEASE NOTES

v0.8.0 2022-01-29

- Added *bins* parameter to *piso.coverage()* and *ArrayAccessor.coverage()*
- Added *how* parameter to *piso.coverage()* and *ArrayAccessor.coverage()*
- Added *result* parameter to *piso.contains()* and *ArrayAccessor.contains()*
- Added *how* parameter to *piso.contains()* and *ArrayAccessor.contains()*

v0.7.0 2021-11-20

Added the following methods

- *piso.split()*
- *piso.adjacency_matrix()*
- *ArrayAccessor.split()*
- *ArrayAccessor.adjacency_matrix()*

Removed the following methods

- removed `piso.get_indexer()` in favour of `pandas.IntervalIndex.get_indexer()`

v0.6.0 2021-11-05

The following methods were extended to accommodate intervals with *closed* = *"both"* or *"neither"*

- *piso.contains()* (and *ArrayAccessor.contains()*)
- `piso.get_indexer()` (and `ArrayAccessor.get_indexer()`)
- *piso.lookup()*
- *piso.isdisjoint()* (and *ArrayAccessor.isdisjoint()*)

v0.5.0 2021-11-02

Added the following methods

- *piso.join()* for *join operations* with interval indexes
- *piso.contains()*
- *ArrayAccessor.contains()*

Performance improvements for

- *piso.lookup()*
- `piso.get_indexer()`

v0.4.0 2021-10-30

Added the following methods

- *piso.lookup()*
- piso.get_indexer()
- ArrayAccessor.get_indexer()

v0.3.0 2021-10-23

Added the following methods

- *piso.coverage()*
- *piso.complement()*
- *ArrayAccessor.coverage()*
- *ArrayAccessor.complement()*

v0.2.0 2021-10-15

Added the following methods

- *piso.isdisjoint()*
- *piso.issuperset()*
- *piso.issubset()*
- *ArrayAccessor.isdisjoint()*
- *ArrayAccessor.issuperset()*
- *ArrayAccessor.issubset()*
- *piso.interval.issuperset()*
- *piso.interval.issubset()*

v0.1.0 2021-10-10

The following methods are included in the initial release of *piso*

- *piso.register_accessors()*
- *piso.union()*
- *piso.intersection()*
- *piso.difference()*
- *piso.symmetric_difference()*
- *ArrayAccessor.union()*
- *ArrayAccessor.intersection()*
- *ArrayAccessor.difference()*
- *ArrayAccessor.symmetric_difference()*
- *piso.interval.union()*
- *piso.interval.intersection()*
- *piso.interval.difference()*
- *piso.interval.symmetric_difference()*

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX