Uppsala University

Software Testing

1DL610 11015

Project Report

Team 12 members:
Martin Kjær
Morten Astrup
Murali Tejeshwar Janaswami
Somiya Khurram
Yifan Liu
Yun-Chien Chiu

January 5, 2022

Contents

1 Library choice and test choices

We choose Pandas as our project's library, and below are the test choices of all the members:
Martin Kjær:

- pandas.to_timedelta
- pandas.period_range
- pandas.date_range
- pandas.bdate_range
- pandas.DataFrame.tail
- pandas.merge_ordered

Morten Astrup:

- pandas.to_numeric
- pandas.concat
- pandas.is_null
- pandas.merge
- pandas.pivot

Murali Tejeshwar Janaswami:

- pandas.Index.is_boolean
- pandas.Index.is_integer
- pandas.Index.is_floating
- pandas.Index.is_numeric
- pandas.Index.is_mixed
- pandas.Index.is_categorical

Somiya Khurram:

- pandas.Series.mul

- pandas.Series.add
- panda.Series.sub
- pandas.Series.div
- pandas.Series.mod

Yifan Liu:

- pandas.DataFrame.count
- pandas.DataFrame.copy
- pandas.DataFrame.bool
- pandas.DataFrame.insert
- pandas.DataFrame.drop_duplicates

Yun-Chien Chiu:

- pandas.eval
- pandas.to_datetime
- pandas.util.hash_array
- pandas.unique
- pandas.notnull

2 Testing Strategy

For each member of our team, we will choose five functions from the pandas library. And we will perform black-box testing as well as white-box testing.

3 Tests

3.1 pandas.to_timedelta

The *to_timedelta* method converts given arguments into Timedelta type. It can recognise different keywords in a string input and convert it to a Timedelta with the right unit. It also takes integers as input with a defined unit or nanoseconds as default unit. Units ’year’ (’Y’) and ’month’ (’M’) is deprecated. See more in the documentation here.

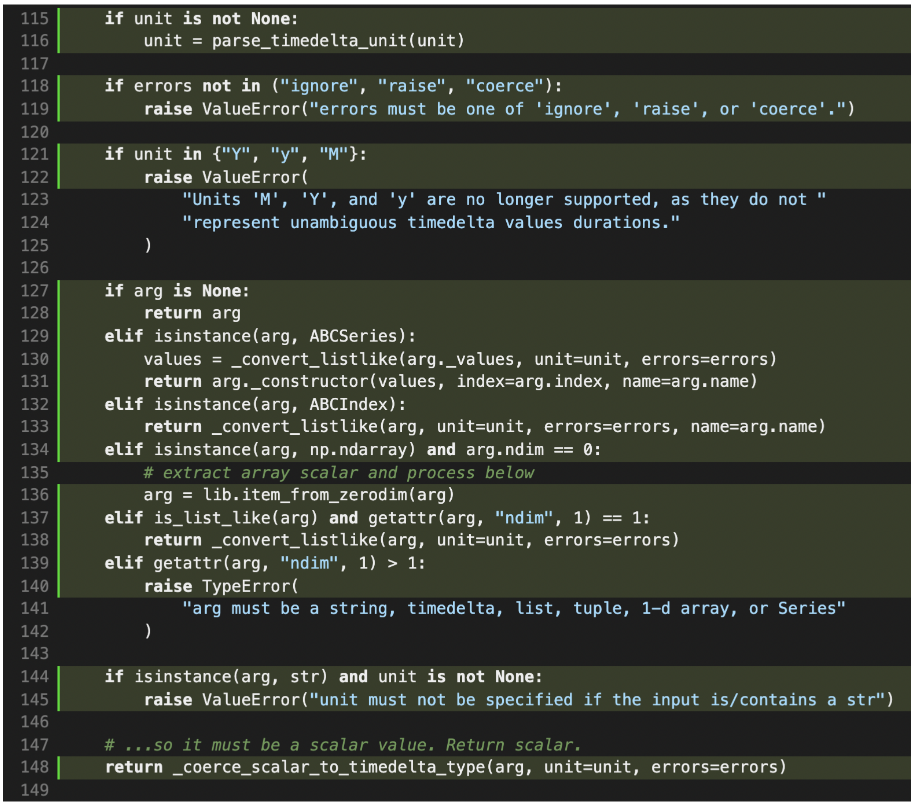
3.1.1 Black-box testing

```python
import unittest
import pandas as pd
import numpy as np

class TestToTimeDelta(unittest.TestCase):

    #Test if all input is for the different units is converted correctly to value.
    #Expected values are in nano secounds calculated and tested on google
    def testUnitValues(self):
        self.assertEqual(pd.to_timedelta(0).value, 0) #zero input
        self.assertEqual(pd.to_timedelta(1).value, 1) #default nano sec
        self.assertEqual(pd.to_timedelta(1, unit='us').value, 1000)
        self.assertEqual(pd.to_timedelta(1, unit='ms').value, 10 ** 6)
        self.assertEqual(pd.to_timedelta(1, unit='s').value, 10 ** 9)
        self.assertEqual(pd.to_timedelta(1, unit='m').value, 60 * 10 ** 9)
        self.assertEqual(pd.to_timedelta(1, unit='h').value, 3600 * 10 ** 9)
        self.assertEqual(pd.to_timedelta(1, unit='d').value, 8.64 * 10 ** 13)
        self.assertEqual(pd.to_timedelta(1, unit='w').value, 6.04800 * 10 ** 14)

    #Test if value error is thrown on deprecated unit input
    def testErrorOnDeprecatedInput(self):
        self.assertRaises(ValueError,pd.to_timedelta,1,unit='y')
        self.assertRaises(ValueError,pd.to_timedelta,1,unit='M')

    #Test if different kind of input gives the expected output
    #Different kind of string input is converted using the expected unit
    def testInputValue(self):
        self.assertEqual(pd.to_timedelta(0), pd.Timedelta('0 days 00:00:00.0'))
        self.assertEqual(pd.to_timedelta(1000), pd.Timedelta('0 days 00:00:00.000001'))
        self.assertEqual(pd.to_timedelta('1sec'), pd.Timedelta('0 days 00:00:01.000000'))
        self.assertEqual(pd.to_timedelta('13:00:00'), pd.Timedelta('0 days 13:00:00.000000'))
        self.assertEqual(pd.to_timedelta('8 days 11:23:33.123456789'), pd.Timedelta('8 days 11:23:33.123456789'))

        self.assertEqual(pd.to_timedelta('61sec'), pd.Timedelta('0 days 00:01:01.000000'))
        self.assertEqual(pd.to_timedelta('1.5min'), pd.Timedelta('0 days 00:01:30.000000'))
        self.assertEqual(pd.to_timedelta('0.2min'), pd.Timedelta('0 days 00:00:12.000000'))
        self.assertEqual(pd.to_timedelta('25hours'), pd.Timedelta('1 days 01:00:00.000000'))
        self.assertEqual(pd.to_timedelta('32 days 5 hours'), pd.Timedelta('32 days 05:00:00.000000'))

    #Test if different units are converted correctly to the 'Timedelta' format
    #All not deprecated units are tested
    def testUnitConversion(self):
        self.assertEqual(pd.to_timedelta(1), pd.Timedelta('0 days 00:00:00.000000001')) #default nano sec
        self.assertEqual(pd.to_timedelta(1, unit='us'), pd.Timedelta('0 days 00:00:00.000001'))
        self.assertEqual(pd.to_timedelta(1, unit='ms'), pd.Timedelta('0 days 00:00:00.001000'))
        self.assertEqual(pd.to_timedelta(1, unit='s'), pd.Timedelta('0 days 00:00:01.000000'))
        self.assertEqual(pd.to_timedelta(1, unit='m'), pd.Timedelta('0 days 00:01:00.000000'))
        self.assertEqual(pd.to_timedelta(1, unit='h'), pd.Timedelta('0 days 01:00:00.000000'))
        self.assertEqual(pd.to_timedelta(1, unit='d'), pd.Timedelta('0 days 24:00:00.000000'))
        self.assertEqual(pd.to_timedelta(1, unit='w'), pd.Timedelta('7 days 00:00:00.000000'))

    #Test argument as list outputs list
    def testListInput(self):
        x = ['1 days 06:05:01.00003', '15.5us', '4hr 7minutes']
        self.assertEqual(pd.to_timedelta(x).tolist(),
                         [pd.Timedelta('1 days 06:05:01.00003'), pd.Timedelta('0 days 00:00:00.000015500'),
                          pd.Timedelta('0 days 04:07:00')])

        #Test with empty list input
        y = []
        self.assertEqual(pd.to_timedelta(y).tolist(),[])

    #Test array and unit argument output list with right unit
    def testNumbersToUnit(self):

        self.assertEqual(pd.to_timedelta(np.arange(3), unit='sec').tolist(),
                         [pd.Timedelta('0 days 00:00:00'), pd.Timedelta('0 days 00:00:01'),
                          pd.Timedelta('0 days 00:00:02')])

        self.assertEqual(pd.to_timedelta(np.arange(4), unit='day').tolist(),
                         [pd.Timedelta('0 days 00:00:00'), pd.Timedelta('1 days 00:00:00'),
                          pd.Timedelta('2 days 00:00:00'),pd.Timedelta('3 days 00:00:00')])

        #Test with empty array input
        self.assertEqual(pd.to_timedelta(np.arange(0), unit='sec').tolist(),[])
```

3.1.2 Whitebox-box testing

This test was done by analysing the *sourcecode* of the function and writing testcases that will cover all the cases. The tool *coverage.py* were used to check coverage of the test on the function.

```
import unittest
import pandas as pd
import numpy as np

from timedeltas import to_timedelta

class TestToTimedeltaWB(unittest.TestCase):

    #Line 127-128: should return 'None' if given 'None' as argument
    def testPassNone(self):
        self.assertEqual(to_timedelta(None, unit='w'), None)

    #Line 121-122: unit parameter 'year' and 'month' is deprecated and should return value error
    def testDeprecatedUnit(self):
        #Value error on depricated Y, y and M (year, month)
        self.assertRaises(ValueError,to_timedelta,1,unit='Y')
        self.assertRaises(ValueError,to_timedelta,1,unit='M')

    #144-145: value error on argument as string AND defined unit. Unit should only be defined on non-string argument.
    def testStringInputWithUnit(self):
        #Value error on string input and defined unit
        self.assertRaises(ValueError,to_timedelta,'7 days',unit='w')

    #118-119: Value error on should be thrown if 'error' defined different from 'raise', 'ignore' or 'coerce'.
    def testErrorInput(self):
        #Value error on error messages different from 'raise', 'ignore' or 'coerce'
        self.assertRaises(ValueError,to_timedelta,1,unit='d',errors='wrong')

    #129-131: Pandas series object as argument
    def testSeriesObject(self):
        # series object
        d = {'a': 1, 'b': 2, 'c': 3}
        ser = pd.Series(data=d, index=['a', 'b', 'c'])
        self.assertEqual(to_timedelta(ser, unit='d').tolist(),
                         [pd.Timedelta('1 days 00:00:00'), pd.Timedelta('2 days 00:00:00'),
                          pd.Timedelta('3 days 00:00:00')])

    #132-33: Pandas Index object as argument
    def testIndexObject(self):
        # Index object
        ind = pd.Index([1, 2, 3])
        self.assertEqual(to_timedelta(ind, unit='d').tolist(),
                         [pd.Timedelta('1 days 00:00:00'), pd.Timedelta('2 days 00:00:00'),
                          pd.Timedelta('3 days 00:00:00')])

    #134-136: Numphy array with zero dimention as argument
    def testZeroDimNpArray(self):
        # zero dimention numpy array
        arr = np.array(1)
        self.assertEqual(to_timedelta(arr, unit='d'), pd.Timedelta('1 days 00:00:00'))

    #139-140: Array with multiple dimentions as argument output type error
    def testMultiDimArray(self):
        #Type error on 3 dimentional array
        arr2 = np.zeros((2, 3, 4))
        self.assertRaises(TypeError, to_timedelta, arr2)

    #137-1138: Array with one dimention as input returns list of output
    def testListInput(self):
        x = ['1 days 06:05:01.00003', '15.5us', '4hr 7minutes']
        self.assertEqual(to_timedelta(x).tolist(),
                            [pd.Timedelta('1 days 06:05:01.00003'), pd.Timedelta('0 days 00:00:00.000015500'),
                             pd.Timedelta('0 days 04:07:00')])


if __name__ == '__main__':
    unittest.main()
```



```
115     if unit is not None:
116         unit = parse_timedelta_unit(unit)
117
118     if errors not in ("ignore", "raise", "coerce"):
119         raise ValueError("errors must be one of 'ignore', 'raise', or 'coerce'.")
120
121     if unit in {"Y", "y", "M"}:
122         raise ValueError(
123             "Units 'M', 'Y', and 'y' are no longer supported, as they do not "
124             "represent unambiguous timedelta values durations."
125         )
126
127     if arg is None:
128         return arg
129     elif isinstance(arg, ABCSeries):
130         values = _convert_listlike(arg._values, unit=unit, errors=errors)
131         return arg._constructor(values, index=arg.index, name=arg.name)
132     elif isinstance(arg, ABCIndex):
133         return _convert_listlike(arg, unit=unit, errors=errors, name=arg.name)
134     elif isinstance(arg, np.ndarray) and arg.ndim == 0:
135         # extract array scalar and process below
136         arg = lib.item_from_zerodim(arg)
137     elif is_list_like(arg) and getattr(arg, "ndim", 1) == 1:
138         return _convert_listlike(arg, unit=unit, errors=errors)
139     elif getattr(arg, "ndim", 1) > 1:
140         raise TypeError(
141             "arg must be a string, timedelta, list, tuple, 1-d array, or Series"
142         )
143
144     if isinstance(arg, str) and unit is not None:
145         raise ValueError("unit must not be specified if the input is/contains a str")
146
147     # ...so it must be a scalar value. Return scalar.
148     return _coerce_scalar_to_timedelta_type(arg, unit=unit, errors=errors)
149
```

Figure 1: Code coverage to_timedelta

3.2 pandas.date_range

The *date_range* function is given a range or period with a frequency and will return a DatetimeIndex, with equal spaced time point equalling to the specified frequency. Of the four parameters *start*, *end*, *period* and *frequency*, three must be defined. See more in the documentation here.

3.2.1 Black-box testing

```python
import unittest
import pandas as pd
from pandas.core.indexes.datetimes import date_range as dr

class TestDateRange(unittest.TestCase):

    #Test input of range between two defined dates
    def testStartEnd(self):
        x = pd.DatetimeIndex(['2021-11-1', '2021-11-2', '2021-11-3'], dtype='datetime64[ns]', freq='D')
        self.assertSequenceEqual(dr('2021-11-1','2021-11-3').tolist(), x.tolist())

        #Test right output on leap year
        y = pd.DatetimeIndex(['2024-02-28', '2024-02-29', '2024-03-1'], dtype='datetime64[ns]', freq='D')
        self.assertSequenceEqual(dr('2024-02-28','2024-03-1').tolist(), y.tolist())

        #Test hour frequency change date after midnight
        z = pd.DatetimeIndex(['2024-02-28  23:00:00', '2024-02-29 00:00:00', '2024-02-29 01:00:00'], dtype='datetime64[ns]', freq='H')
        self.assertSequenceEqual(dr('2024-02-28 23:00:00','2024-02-29 01:00:00',freq='H').tolist(), z.tolist())

    #Test defined start of interval and period
    def testStart(self):
        #Test 3 period with default freqency, day
        x = pd.DatetimeIndex(['2021-11-1', '2021-11-2', '2021-11-3'], dtype='datetime64[ns]', freq='D')
        self.assertSequenceEqual(pd.date_range('2021-11-1', periods=3).tolist(), x.tolist())

        #Test with month start frequency
        y = pd.DatetimeIndex(['2021-11-01', '2021-12-01', '2022-01-01'], dtype='datetime64[ns]', freq='MS')
```

```
     self.assertSequenceEqual(pd.date_range(start='2021-10-06', freq='MS', periods=3).tolist(), y.tolist())
 #Test defined end of interval and period
 def testEnd(self):
     #Test 3 periods and default frequency, day
     x = pd.DatetimeIndex(['2021-11-29', '2021-11-30', '2021-12-1'], dtype='datetime64[ns]', freq='D')
     self.assertSequenceEqual(pd.date_range(end='2021-12-1', periods=3).tolist(), x.tolist())

     #Test 3 periods with 3 month frequency
     m = pd.DatetimeIndex(['2020-01-31', '2020-04-30', '2020-07-31'], dtype='datetime64[ns]', freq='3M')
     self.assertSequenceEqual(pd.date_range(end='2020-07-31', freq='3M', periods=3).tolist(), m.tolist())

 #Test with defined timezone
 def testTimeZone(self):
     y = pd.DatetimeIndex(['2024-02-28 01:00:00','2024-02-28 02:00:00','2024-02-28 03:00:00'], freq='H',tz='Asia/Hong_Kong')
     x = pd.DatetimeIndex(['2024-02-28 01:00:00','2024-02-28 02:00:00','2024-02-28 03:00:00'], freq='H',tz='America/Los_Angeles')

     self.assertSequenceEqual(pd.date_range('2024-02-28 01:00:00', '2024-02-28 03:00:00', freq='H',tz='Asia/Hong_Kong').tolist(),y.tolist())
     self.assertSequenceEqual(pd.date_range('2024-02-28 01:00:00', '2024-02-28 03:00:00', freq='H',tz='America/Los_Angeles').tolist(),x.tolist())

     self.assertNotEqual(x.tolist(),y.tolist())
     self.assertNotEqual(x.tz,y.tz)

 #Test 'normalize' true, reset time to midnight
 def testNormalize(self):
     x = pd.DatetimeIndex(['2024-02-28  00:00:00', '2024-02-29 00:00:00', '2024-03-01 00:00:00'], dtype='datetime64[ns]', freq='D')
     self.assertSequenceEqual(pd.date_range('2024-02-28 23:00:00','2024-03-01 01:00:00',freq='D',normalize=True).tolist(), x.tolist())
     self.assertEqual(pd.date_range('2024-02-28 23:00:00','2024-03-01 01:00:00',freq='D',normalize=True)[0].hour,0)

 #Test 'closed' to right and left will discard last and first value
 def testClosed(self):
```

```python
        r = pd.DatetimeIndex([’2024-02-29’, ’2024-03-01’], dtype=’datetime64[ns]’, freq=’D’)
        l = pd.DatetimeIndex([’2024-02-28’, ’2024-02-29’], dtype=’datetime64[ns]’, freq=’D’)

        self.assertSequenceEqual(pd.date_range(’2024-02-28’,’2024-03-01’,freq=’D’,closed=’right’).tolist(), r.tolist())
        self.assertSequenceEqual(pd.date_range(’2024-02-28’,’2024-03-01’,freq=’D’,closed=’left’).tolist(), l.tolist())
```

3.3 pandas.bdate_range

The *bdate_range* function takes a range or period with a frequency and will return a DatetimeIndex of business days as default value. Of the four parameters *start*, *end*, *period* and *frequency*, three must be defined. See more in the documentation here.

3.3.1 Black-box testing

```python
import unittest
import pandas as pd

class TestBdateRange(unittest.TestCase):

    #Test input of range between two defined dates
    def testStartEnd(self):
        x = pd.DatetimeIndex([’2021-11-22’, ’2021-11-23’, ’2021-11-24’, ’2021-11-25’, ’2021-11-26’], dtype=’datetime64[ns]’, freq=’B’)
        self.assertSequenceEqual(pd.bdate_range(’2021-11-20’,’2021-11-28’).tolist(),x.tolist())

        #Test with leap year
        y = pd.DatetimeIndex([’2024-02-28’, ’2024-02-29’, ’2024-03-1’], dtype=’datetime64[ns]’, freq=’D’)
        self.assertSequenceEqual(pd.bdate_range(’2024-02-28’,’2024-03-1’).tolist(), y.tolist())

        #Test with 5 hour interval
        z = pd.DatetimeIndex([’2021-03-29 00:00:00’, ’2021-03-29 05:00:00’, ’2021-03-29 10:00:00’, ’2021-03-29 15:00:00’, ’2021-03-29 20:00:00’], dtype=’datetime64[ns]’, freq=’5H’)
        self.assertSequenceEqual(pd.bdate_range(’2021-03-29 23:00:00’,’2021-03-30 09:00:00’,freq=’5H’, normalize=True).tolist(), z.tolist())

    #Defined start date, no end date but with periode and freqence
    def testStart(self):
        #start weekend and periode of 3 days
        x = pd.DatetimeIndex(['2021-11-1', '2021-11-2', '2021-11-3'], dtype='datetime64[ns]', freq='B')
        self.assertSequenceEqual(pd.bdate_range(start='2021-10-30', periods=3, freq='B').tolist(), x.tolist())

        #Beginning of month frequence in 3 periods
        y = pd.DatetimeIndex(['2021-11-01', '2021-12-01', '2022-01-01'], dtype='datetime64[ns]', freq='MS')
        self.assertSequenceEqual(pd.bdate_range(start='2021-11-01', periods=3, freq='MS').tolist(), y.tolist())

    #Defined end date, no start date but with periode and freqence
    def testEnd(self):
        #end midt week and get 4 days before
        x = pd.DatetimeIndex(['2021-11-26', '2021-11-29', '2021-11-30', '2021-12-1'], dtype='datetime64[ns]', freq='B')
        self.assertSequenceEqual(pd.bdate_range(end='2021-12-1', periods=4).tolist(), x.tolist())

        #get 3 periods with 3 business days interval
        m = pd.DatetimeIndex(['2020-07-23', '2020-07-28', '2020-07-31'], dtype='datetime64[ns]', freq='3B')
        self.assertSequenceEqual(pd.bdate_range(end='2020-07-31', freq='3B', periods=3).tolist(), m.tolist())
    #Test with timezone
    def testTimeZone(self):
        y = pd.DatetimeIndex(['2024-02-28 00:00:00','2024-02-29 00:00:00'], freq='B',tz='Asia/Hong_Kong')
        x = pd.DatetimeIndex(['2024-02-28 00:00:00','2024-02-29 00:00:00'], freq='B',tz='America/Los_Angeles')

        #Test time zone is added
        self.assertSequenceEqual(pd.bdate_range('2024-02-28 01:00:00', '2024-02-29 03:00:00', freq='B',tz='Asia/Hong_Kong').tolist(), y.tolist())
        self.assertSequenceEqual(pd.bdate_range('2024-02-28 01:00:00', '2024-02-29 03:00:00', freq='B',tz='America/Los_Angeles').tolist(),x.tolist())
        #Output with two different timezones are not equal
        self.assertNotEqual(x.tolist(),y.tolist())
        self.assertNotEqual(x.tz,y.tz)

    #Test normalization of input will set the time to midtnight
    def testNormalize(self):
        x = pd.DatetimeIndex(['2024-02-28  00:00:00', '2024-02-29 00:00:00', '2024-03-01 00:00:00'], dtype='datetime64[ns]', freq='B')
```

```python
        self.assertSequenceEqual(pd.bdate_range('2024-02-28 23:00:00','2024-03-01 01:00:00',freq='B',normalize=True).tolist(), x.tolist())
        self.assertEqual(pd.bdate_range('2024-02-28 23:00:00','2024-03-01 01:00:00',freq='B',normalize=True)[0].hour,0)

    #Test closed from left and right will omit ends
    def testClosed(self):

        r = pd.DatetimeIndex(['2021-11-29', '2021-11-30', '2021-12-1'], dtype='datetime64[ns]', freq='B')
        l = pd.DatetimeIndex(['2021-11-26', '2021-11-29', '2021-11-30'], dtype='datetime64[ns]', freq='B')

        self.assertSequenceEqual(pd.bdate_range('2021-11-26','2021-12-01',freq='B',closed='right').tolist(), r.tolist())
        self.assertSequenceEqual(pd.bdate_range('2021-11-26','2021-12-01',freq='B',closed='left').tolist(), l.tolist())
```

3.4 pandas.period_range

The *period_range* function takes a start and end date with a frequency and will return a PeriodIndex of fixed frequency. Of the three parameters *start*, *end* and *period*, two must be defined. The default frequency is 'day'. See more in the documentation here.

3.4.1 Black-box testing

```python
import unittest
import pandas as pd

class TestPeriodRange(unittest.TestCase):

    #Test input with defined start and end
    def testStartEnd(self):
        x = pd.PeriodIndex(['2021-11-1', '2021-11-2', '2021-11-3'], dtype='period[D]')
        self.assertSequenceEqual(pd.period_range('2021-11-1','2021-11-3').tolist(), x.tolist())

        #test leap year
        y = pd.PeriodIndex(['2024-02-28', '2024-02-29', '2024-03-1'], dtype='period[D]')
        self.assertSequenceEqual(pd.period_range('2024-02-28','2024-03-1').tolist(), y.tolist())

        #test day shift after hour pass 24
        z = pd.PeriodIndex(['2024-02-28  23:00:00', '2024-02-29 00:00:00', '2024-02-29 01:00:00'], dtype='period[H]')
        self.assertSequenceEqual(pd.period_range('2024-02-28 23:00:00','2024-02-29 01:00:00',freq='H').tolist(), z.tolist())

    #Test defined start with defined period
    def testStart(self):
        #start with 3 day period
        x = pd.PeriodIndex(['2021-11-1', '2021-11-2', '2021-11-3'], dtype='period[D]')
        self.assertSequenceEqual(pd.period_range('2021-11-1',periods=3).tolist(), x.tolist())
        #start with 3 month period
        y = pd.PeriodIndex(['2021-11', '2021-12', '2022-01'], dtype='period[M]')
        self.assertSequenceEqual(pd.period_range('2021-11-01', freq='M', periods=3).tolist(), y.tolist())

    #Test definded end with defined period
    def testEnd(self):
        #end with 3 day period
        x = pd.PeriodIndex(['2021-11-29', '2021-11-30', '2021-12-1'], dtype='period[D]')
        self.assertSequenceEqual(pd.period_range(end='2021-12-1', periods=3).tolist(), x.tolist())
        #end with 3 year period
        y = pd.PeriodIndex(['2020', '2021', '2022'], dtype='period[Y]')
        self.assertSequenceEqual(pd.period_range(end='2022-01-01', freq='Y', periods=3).tolist(), y.tolist())

    #Test defined start, end and frequency
    def testPeriodInput(self):
        start = pd.Period('2021-01', freq='M')
        end = pd.Period('2021-04', freq='M')
        x = pd.PeriodIndex(['2021-01', '2021-02', '2021-03', '2021-04'], dtype='period[M]')
        self.assertSequenceEqual(pd.period_range(start,end,freq='M').tolist(), x.tolist())
```

3.5 pandas.Dataframe.tail

The *Dataframe.tail* function can be called on Dataframes type. The function takes an integer n as argument and will return the n last elements of the Dataframe. For negative integer input, the function will return the first n elements. See more in the documentation here.

3.5.1 Black-box testing

```python
import unittest
import pandas as pd
from pandas._testing import assert_frame_equal


class TestDataframeTail(unittest.TestCase):
    df = pd.DataFrame({'fruit': ['apple', 'pear', 'banana', 'melon', 'lemon', 'mango', 'lime']})
    expected = pd.DataFrame({'fruit': ['apple', 'pear', 'banana', 'melon', 'lemon', 'mango', 'lime']})

    #Test output values are the last five elements as default
    def testReturnDefault(self):
        expected = self.expected.drop([0, 1])
        assert_frame_equal(self.df.tail(),expected)

    #Test defined number of elements to return
    def testNvalueReturn(self):
        expected = self.expected.drop([0,1,2,3,4,5])
        assert_frame_equal(self.df.tail(1),expected)

        expected = self.expected.drop([0,1,2,3])
        assert_frame_equal(self.df.tail(3), expected)

        #all
        expected = self.expected
        assert_frame_equal(self.df.tail(7), expected)

    #Test if returns empty list on zero as argument
    def testZeroReturn(self):
        expected = self.expected.drop([0,1,2,3,4,5,6])
        assert_frame_equal(self.df.tail(0), expected)

    #Test with negative argument
    def testNegativeInput(self):
        expected = self.expected.drop([0,1,2,3])
        assert_frame_equal(self.df.tail(-4), expected)

        expected = self.expected.drop([0,1,2,3,4,5,6])
        assert_frame_equal(self.df.tail(-7), expected)

    #Test with positive and negative input out of range
    def testOutOffBoundInput(self):
        expected = self.expected.drop([0,1,2,3,4,5,6])
        assert_frame_equal(self.df.tail(-7000), expected)

        expected = self.expected
        assert_frame_equal(self.df.tail(7000), expected)
```

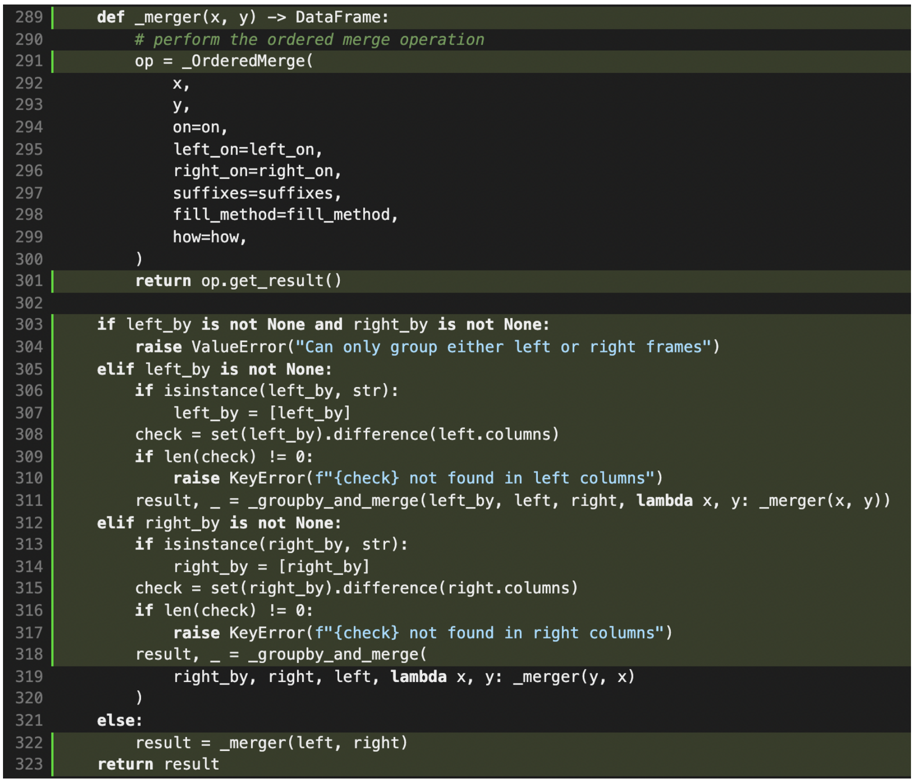
3.6 pandas.merge_ordered

The *merge_ordered* function takes two Dataframes (left and right) as argument and outputs a merged dataframe. Optional ’field’ or ’group’ to join by can be defined for either left or right dataframe. See more in the documentation here.

3.6.1 Whitebox-box testing

This test was done by analysing the *sourcecode* of the function and writing testcases that will cover all the cases. The tool *coverage.py* were used to check coverage of the test on the function.

```
import unittest
import pandas as pd

from merge import merge_ordered
from pandas._testing import assert_frame_equal


class TestMergeOrderedWB(unittest.TestCase):
    df1 = pd.DataFrame({"key": ["a", "b", "c"], "lvalue": [1, 2,
   3]})
    df2 = pd.DataFrame({"key": ["a", "b", "c"], "rvalue": [4, 5,
   6]})

    def setUp(self):
        df1 = pd.DataFrame({"key": ["a", "b", "c"],"lvalue": [1, 2,
   3]})
        df2 = pd.DataFrame({"key": ["a", "b", "c"], "rvalue": [4, 5,
    6]})

    #322: Default merge returns right
    def testNormalCase(self):
        exp = pd.DataFrame(
            {
                "key": ["a", "b", "c"],
                "lvalue": [1, 2, 3],
                "rvalue": [4, 5, 6]
            }
        )
        assert_frame_equal(merge_ordered(self.df1, self.df2), exp)

    #303-304: Throw value error when ’right-by’ AND ’left-by’ are
   defined in argument
    def testLeftByRightBy(self):
        self.assertRaises(ValueError, merge_ordered,self.df1,self.
   df2,left_by='lvalue', right_by='lvalue' )

    #305-3011: Merge order left-by
    def testLeftBy(self):
        exp = pd.DataFrame(
            {
                "key": ["a", "b", "c","a", "b", "c","a", "b", "c"],
                "lvalue": [1,1,1,2,2,2,3,3,3],
                "rvalue": [4,5,6,4,5,6,4,5,6]
            }
        )
        assert_frame_equal(merge_ordered(self.df1, self.df2, left_by
   ='lvalue'), exp)

    #309-310: Throw key error if defined 'left-by' is not found
    def testLeftByNotFound(self):
        #left by value not found
        self.assertRaises(KeyError, merge_ordered, self.df1, self.
   df2, left_by='fake')

    #312-318: Merge order right by
    def testRightBy(self):
        exp = pd.DataFrame(
            {
                "key": ["a", "b", "c","a", "b", "c","a", "b", "c"],
                "lvalue": [1,2,3,1,2,3,1,2,3],
                "rvalue": [4,4,4,5,5,5,6,6,6]
            }
        )
        assert_frame_equal(merge_ordered(self.df1, self.df2,
   right_by='rvalue'), exp)

    #316-317: Throw key error if defined 'left-by' is not found
    def testRightByNotFound(self):
        #right by value not found
        self.assertRaises(KeyError, merge_ordered, self.df1, self.
   df2, right_by='fake')

if __name__ == '__main__':
    unittest.main()
```



```
    def _merger(x, y) -> DataFrame:
        # perform the ordered merge operation
        op = _OrderedMerge(
            x,
            y,
            on=on,
            left_on=left_on,
            right_on=right_on,
            suffixes=suffixes,
            fill_method=fill_method,
            how=how,
        )
        return op.get_result()

    if left_by is not None and right_by is not None:
        raise ValueError("Can only group either left or right frames")
    elif left_by is not None:
        if isinstance(left_by, str):
            left_by = [left_by]
        check = set(left_by).difference(left.columns)
        if len(check) != 0:
            raise KeyError(f"{check} not found in left columns")
        result, _ = _groupby_and_merge(left_by, left, right, lambda x, y: _merger(x, y))
    elif right_by is not None:
        if isinstance(right_by, str):
            right_by = [right_by]
        check = set(right_by).difference(right.columns)
        if len(check) != 0:
            raise KeyError(f"{check} not found in right columns")
        result, _ = _groupby_and_merge(
            right_by, right, left, lambda x, y: _merger(y, x)
        )
    else:
        result = _merger(left, right)
    return result
```

Figure 2: Code coverage of merge_ordered

3.7 pandas.to_numeric

pandas.to_numeric(arg, errors='raise', downcast=None)

This function will convert its input to a numeric type. For example, a Series of numbers represented as strings (fx '6', not 'six'), will be converted to *float64* or *int64*.

3.7.1 Black-box testing

```
import unittest
import numpy as np
import pandas as pd
from numeric import to_numeric

# scalar, list, tuple, 1-d array, or Series
class TestPandasToNumeric(unittest.TestCase):
    def setUp(self):
        self.df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/binary.dta')

    # Testing different types of arguments such as Scalar / List.
    def testArgTypes(self):
        self.scalar = self.df['gpa']
        self.scalar.squeeze()
        self.list = list(self.df['gpa'])

        # Scalar (numpy.float32 -> numpy.float32)
        self.assertTrue(isinstance(type(to_numeric(self.scalar)[0]), type(np.float32)))
        # List (float32 -> numpy.float32)
        self.assertTrue(isinstance(type(to_numeric(self.list)[0]), type(np.float32)))

    # Testing a Series where all the strings are numbers
    def testFromString(self):
        series = pd.Series(['7', '9', '13'])
        result = to_numeric(series)

        self.assertEqual(result[0], pd.Series([7, 9, 13])[0])
        self.assertEqual(result[1], pd.Series([7, 9, 13])[1])
        self.assertEqual(result[2], pd.Series([7, 9, 13])[2])

    # Testing a combination of numbers and string-numbers
    def testFromMixedNumbers(self):
        series = pd.Series(['7', '9', 13])
        result = to_numeric(series)

        self.assertEqual(result[0], pd.Series([7, 9, 13])[0])
        self.assertEqual(result[1], pd.Series([7, 9, 13])[1])
        self.assertEqual(result[2], pd.Series([7, 9, 13])[2])

    # Testing a Series where every element is an integer
    def testFromInt(self):
        series = pd.Series([7, 9, 13])
        result = to_numeric(series)

        self.assertEqual(result[0], pd.Series([7, 9, 13])[0])
        self.assertEqual(result[1], pd.Series([7, 9, 13])[1])
        self.assertEqual(result[2], pd.Series([7, 9, 13])[2])

    # Should throw a ValueError in case of inputting strings that are not in number format
    def testWithWords(self):
        self.assertRaises(ValueError, to_numeric, pd.Series(['7', '9', 13, 'forteen']))
```

```

unittest.main(argv=[''], verbosity=2, exit=False)
```

3.7.2 White-box testing

```
import unittest
import numpy as np
import pandas as pd
from numeric import to_numeric

# scalar, list, tuple, 1-d array, or Series
class TestPandasToNumeric(unittest.TestCase):
    def setUp(self):
        self.df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/binary.dta')
        self.stringArray = pd.Series(['6', '10', '19', '25'])
        self.intArray = pd.Series([6, 10, 19, 25])
        self.floatArray = pd.Series([6., 10., 19., 25.])
        self.dateArray = pd.DatetimeIndex(['2017-12-31 16:00:00-08:00', '2017-12-31 17:00:00-08:00',
               '2017-12-31 18:00:00-08:00'],
              dtype='datetime64[ns, Europe/Stockholm]', freq='H')

    # Testing that downcasting a stringarray gives a Valueerror depending on the arguments given for
    # 'downcast' and 'errors'. Each combination is a different branch and the error is handled differently.
    def testArgumentError(self):
        self.assertRaises(ValueError, to_numeric, self.stringArray, downcast="double")
        self.assertRaises(ValueError, to_numeric, self.stringArray, downcast="int")

        self.assertRaises(ValueError, to_numeric, self.stringArray, errors="ignor")
        self.assertRaises(ValueError, to_numeric, self.stringArray, errors="")

        self.assertRaises(ValueError, to_numeric, pd.Series(['7', '9', 13, 'forteen']))

        self.assertRaises(TypeError, to_numeric, self.df)

    # Testing many different types of arguments
    def testArgTypes(self):
        # Testing Series
        result = to_numeric(self.stringArray)

        for i,v in self.intArray.iteritems():
            self.assertEqual(v, result.loc[i])

        # Testing Index
        indexArray = pd.Index([’6’, ’10’, ’19’, ’25’])
        result = to_numeric(indexArray)
        for i,v in self.intArray.iteritems():
            self.assertEqual(v, result[i])

        # Testing for List
        result = to_numeric(self.stringArray.tolist())
        for i,v in self.intArray.iteritems():
            self.assertEqual(v, result[i])

        # Testing for scalar int & decimal
        index_ = [’index_name’]

        scalarArray = pd.Series([248])
        scalarArray.index = index_
        scalar = scalarArray.item()
        result = to_numeric(scalar)
        self.assertEqual(248, result)

        scalarArrayDecimal = pd.Series([float(24.8)])
        scalarArrayDecimal.index = index_
        scalarDecimal = scalarArrayDecimal.item()
        resultDecimal = to_numeric(scalarDecimal)
        self.assertEqual(24.8, resultDecimal)

        scalarArrayString = pd.Series([’24’])
        scalarArrayString.index = index_
        scalarString = scalarArrayString.item()
        resultString = to_numeric(scalarString)
        self.assertEqual(24, resultString)

        scalarPDArray = pd.array([10.10], dtype=’float64’)
        scalarString = scalarPDArray[0]
        resultString = to_numeric(scalarString)
        self.assertEqual(10.10, resultString)

    # Testing that downcast works as intended for every downcast
    def testDowncast(self):
        integer = to_numeric(self.stringArray, downcast="integer")
        signed = to_numeric(self.stringArray, downcast="signed")
        unsigned = to_numeric(self.stringArray, downcast="unsigned")
        float = to_numeric(self.stringArray, downcast="float")

        self.assertTrue(isinstance(integer[0], type(np.int8(0))))
        self.assertTrue(isinstance(signed[0], type(np.int8(0))))
        self.assertTrue(isinstance(unsigned[0], type(np.uint8(0))))
        self.assertTrue(isinstance(float[0], type(np.float32(0))))

    # Testing that masking branch works
    def testMask(self):
        self.assertEqual(self.intArray[0], to_numeric([6, 10, 19, 25])[0])
        self.assertEqual(self.floatArray[0], to_numeric(self.floatArray)[0])
        self.assertEqual(1514764800000000000, to_numeric(self.dateArray)[0])

    # Testing that the function can handle Numpy arrays
    def testNumpy(self):
        numpyArray = np.array([7, 9, 13])
        self.assertEqual(7, to_numeric(numpyArray)[0])

    # Testing for numeric arrays
    def testNumericArray(self):
        integerArray = pd.array([1, 6, 10], dtype='Int32')
        result = to_numeric(integerArray)
        self.assertEqual(1, result[0])
        self.assertEqual(6, result[1])
        self.assertEqual(10, result[2])

        floatArray = pd.array([1.1, 6.6, 10.10], dtype='float64')
        result = to_numeric(floatArray)
        self.assertEqual(1.1, result[0])
        self.assertEqual(6.6, result[1])
        self.assertEqual(10.10, result[2])

unittest.main(argv=[''], verbosity=2, exit=False)
```

3.8 pandas.pivot

pandas.pivot(data, index=None, columns=None, values=None)

Return reshaped DataFrame organized by given index / column values. Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified index / columns to form axes of the resulting DataFrame.

3.8.1 Black-box testing

```python
import unittest
import pandas as pd
import numpy as np

class TestPandasPivot(unittest.TestCase):
    def setUp(self):
        self.df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two', 'two'],
                   'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'baz': [1, 2, 3, 4, 5, 6],
                   'zoo': ['x', 'y', 'z', 'q', 'w', 't']})

    def testBasicPivot(self):
        result = self.df.pivot(index='foo', columns='bar', values='baz')
        # Pivot should look like this:
        # bar  A   B   C
        # foo
        # one  1   2   3
        # two  4   5   6

        # Check that numbers have been pivoted correctly by retrieving individual numbers from DF
        self.assertEqual(result.iloc[0][0], 1)
        self.assertEqual(result.iloc[0][1], 2)
        self.assertEqual(result.iloc[0][2], 3)
        self.assertEqual(result.iloc[1][0], 4)
        self.assertEqual(result.iloc[1][1], 5)
        self.assertEqual(result.iloc[1][2], 6)

        # Check that columns are correct
        self.assertTrue(len(result.columns.values) == 3)
        self.assertEqual(result.columns.values[0], 'A')
        self.assertEqual(result.columns.values[1], 'B')
        self.assertEqual(result.columns.values[2], 'C')

    def testNaNValues(self):
        result = self.df.pivot(index='bar', columns='baz', values='foo')
        # Pivot should look like this:
        # baz    1    2    3    4    5    6
        # bar
        # A    one  NaN  NaN  two  NaN  NaN
        # B    NaN  one  NaN  NaN  two  NaN
        # C    NaN  NaN  one  NaN  NaN  two

        # Checking for first 3 values of A row has correct values & NaN values
        self.assertEqual(result.iloc[0].get(1), 'one')
        self.assertTrue(pd.isna(result.iloc[0].get(2)))
        self.assertTrue(pd.isna(result.iloc[0].get(3)))

    def testMultipleValues(self):
        result = self.df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])

        # Pivot should contain two columns with their own identical subcolumns
        columnNames = list(result.columns.levels[0])
        self.assertEqual(columnNames[0], 'baz')
        self.assertEqual(columnNames[1], 'zoo')

        subColumnNames = list(result.columns.levels[1])
        self.assertEqual(subColumnNames[0], 'A')
        self.assertEqual(subColumnNames[1], 'B')
        self.assertEqual(subColumnNames[2], 'C')

    def testIdenticalValuesError(self):
        # When index contains duplicate it will throw ValueError

        df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/binary.dta')
        with self.assertRaises(ValueError):
            pd.pivot(df, index='admit', columns='gpa', values='rank')



unittest.main(argv=[''], verbosity=2, exit=False)
```

3.8.2 White-box testing

```
import unittest
import pandas as pd
```

```
from pivot import pivot
import numpy as np

class TestPandasPivot(unittest.TestCase):
    def setUp(self):
        self.df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two', 'two'],
                   'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'baz': [1, 2, 3, 4, 5, 6],
                   'zoo': ['x', 'y', 'z', 'q', 'w', 't']})

    def testBasicPivot(self):
        result = pivot(self.df, index='foo', columns='bar', values='baz')
        # Pivot should look like this:
        # bar  A   B   C
        # foo
        # one  1   2   3
        # two  4   5   6

        # Check that numbers have been pivoted correctly by retrieving individual numbers from DF
        self.assertEqual(result.iloc[0][0], 1)
        self.assertEqual(result.iloc[0][1], 2)
        self.assertEqual(result.iloc[0][2], 3)
        self.assertEqual(result.iloc[1][0], 4)
        self.assertEqual(result.iloc[1][1], 5)
        self.assertEqual(result.iloc[1][2], 6)

        # Check that columns are correct
        self.assertTrue(len(result.columns.values) == 3)
        self.assertEqual(result.columns.values[0], 'A')
        self.assertEqual(result.columns.values[1], 'B')
        self.assertEqual(result.columns.values[2], 'C')

    def testNaNValues(self):
        result = pivot(self.df, index='bar', columns='baz', values='foo')
        # Pivot should look like this:
        # baz    1    2    3    4    5    6
        # bar
        # A    one  NaN  NaN  two  NaN  NaN
        # B    NaN  one  NaN  NaN  two  NaN
        # C    NaN  NaN  one  NaN  NaN  two

        # Checking for first 3 values of A row has correct values & NaN values
        self.assertEqual(result.iloc[0].get(1), 'one')
        self.assertTrue(pd.isna(result.iloc[0].get(2)))
        self.assertTrue(pd.isna(result.iloc[0].get(3)))

    def testMultipleValues(self):
        result = pivot(self.df, index=’foo’, columns=’bar’, values=[’baz’, ’zoo’])

        # Pivot should contain two columns with their own identical subcolumns
        columnNames = list(result.columns.levels[0])
        self.assertEqual(columnNames[0], ’baz’)
        self.assertEqual(columnNames[1], ’zoo’)

        subColumnNames = list(result.columns.levels[1])
        self.assertEqual(subColumnNames[0], ’A’)
        self.assertEqual(subColumnNames[1], ’B’)
        self.assertEqual(subColumnNames[2], ’C’)

    def testIdenticalValuesError(self):
        # When index contains duplicate it will throw ValueError

        df = pd.io.stata.read_stata(’https://stats.idre.ucla.edu/stat/stata/dae/binary.dta’)
        with self.assertRaises(ValueError):
            pivot(df, index=’admit’, columns=’gpa’, values=’rank’)

    def testMissingColumnArg(self):
        with self.assertRaises(TypeError):
            pivot(self.df, index=’foo’, values=’bar’)

    def testNullArguments(self):
        # Testing with different missing arguments, allowing Pandas to handle these different situations. The following 3 cases are individual
        # pieces of code that will be tested

        # Missing values: Use all remaining values
        pivotOne = pivot(self.df, index=’foo’, columns=’bar’)

        # Missing values & index: use remaining values AND create automatic index 1..n
        pivotTwo = pivot(self.df, columns=’bar’)

        # Missing Index: Use all values but create automatic index 1..n
        pivotThree = pivot(self.df, columns=’bar’, values=’bar’)

        # Pivot no values
        # Pivot should contain two columns with their own identical subcolumns
        self.assertEqual(pivotOne.columns.levels[0][0], 'baz')
        self.assertEqual(pivotOne.columns.levels[0][1], 'zoo')

        # Pivot no values + index
        self.assertEqual(pivotTwo.columns.levels[0][0], 'foo')
        self.assertEqual(pivotTwo.columns.levels[0][1], 'baz')

        self.assertEqual(pivotTwo.index[0], 0)
        self.assertEqual(pivotTwo.index[1], 1)
        self.assertEqual(pivotTwo.index[2], 2)
        self.assertEqual(pivotTwo.index[3], 3)
        self.assertEqual(pivotTwo.index[4], 4)

        # Pivot no index
        self.assertEqual(pivotThree.index[0], 0)
        self.assertEqual(pivotThree.index[1], 1)
        self.assertEqual(pivotThree.index[2], 2)
        self.assertEqual(pivotThree.index[3], 3)
        self.assertEqual(pivotThree.index[4], 4)


unittest.main(argv=[''], verbosity=2, exit=False)
```

3.9 pandas.concat

pandas.concat(objs, axis=0, join='outer', ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, sort=False, copy=True)

Concatenate pandas objects along a particular axis with optional set logic along the other axes. Can also add a layer of hierarchical indexing on the concatenation axis, which may be useful if the labels are the same (or overlapping) on the passed axis number.

3.9.1 Black-box testing

```
import unittest
import pandas as pd
import numpy as np


class TestPandasConcat(unittest.TestCase):
    def setUp(self):
        self.df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/binary.dta')

        # Includes start index, excludes end index
        self.df1 = self.df.iloc[0:3]
        self.df2 = self.df.iloc[3:6]

    # Testing whether the amount of rows is correct
    def test1(self):
        expected = 6
        df_concat = pd.concat([self.df1, self.df2])

        self.assertEqual(len(df_concat.index), expected)

    # Testing whether defining the axis on column instead of row functions as intended
    # Concattenated dataframe should be put input as 4 new columns
    def test2(self):
        expected = ['admit', 'gre', 'gpa', 'rank', 'admit', 'gre', 'gpa', 'rank']
        df_concat = pd.concat([self.df1, self.df2], axis=1)
        self.assertEqual(list(df_concat.columns), expected)
        self.assertEqual(len(df_concat.index), 6)

    # Test that ignoring the index will re-index from 0-n whilst true will keep original index.
    def test3(self):
        df_concat_true = pd.concat([self.df2, self.df1], ignore_index=True)
        df_concat_false = pd.concat([self.df2, self.df1], ignore_index=False)
        # Ignore Index (true):          [0, 1, 2, 3, 4, 5]
        # Don't ignore index (false):   [3, 4, 5, 0, 1, 2]

        self.assertFalse(list(df_concat_true.index) == list(df_concat_false.index))
        self.assertEqual(list(df_concat_true.index), [0, 1, 2, 3, 4, 5])

unittest.main(argv=[''], verbosity=2, exit=False)
```

3.10 pandas.is_null

pandas.isnull(obj)

Detect missing values for an array-like object. This function takes a scalar or array-like object and indicates whether values are missing (NaN in numeric arrays, None or NaN in object arrays, NaT in datetimelike).

3.10.1 Black-box testing

```python
import unittest
import pandas as pd
import numpy as np
from numpy.core.numeric import NaN
from pandas._libs.missing import NA

class TestPandasIsNull(unittest.TestCase):
    def setUp(self):
        self.df = pd.io.stata.read_stata(’https://stats.idre.ucla.edu/stat/stata/dae/binary.dta’)

    def testShouldDetectNullValueTypes(self):
        numericArray = pd.Series([7, 9, 13, NaN, NA])
        objectArray = pd.Series([None, NaN, {}])

        self.assertFalse(pd.isnull(numericArray[0]))
        self.assertFalse(pd.isnull(numericArray[1]))
        self.assertFalse(pd.isnull(numericArray[2]))
        self.assertTrue(pd.isnull(numericArray[3]))
        self.assertTrue(pd.isnull(numericArray[4]))

        self.assertTrue(pd.isnull(objectArray[0]))
        self.assertTrue(pd.isnull(objectArray[1]))
        self.assertFalse(pd.isnull(objectArray[2]))

    def testShouldDetectNullValuesBoolarray(self):
        series = pd.Series([7, 9, 13, NaN, NA])
        expected_result = pd.Series([False, False, False, True, True])

        for i in range(5):
            self.assertEqual(pd.isnull(series[i]), expected_result[i])

unittest.main(argv=[’’], verbosity=2, exit=False)
```

3.11 pandas.merge

Merge DataFrame or named Series objects with a database-style join. A named Series object is treated as a DataFrame with a single named column.

3.11.1 Black-box testing

```python
import unittest
import pandas as pd
import numpy as np

class TestPandasMerge(unittest.TestCase):
    def setUp(self):
        self.df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/binary.dta')

        # Includes start index, excludes end index
        self.df1 = self.df.iloc[0:3]
        self.df2 = self.df.iloc[3:6]

    def testBasicMerge(self):
        # Merge the two arrays on 'admit' column. This column is shared in output
        result = pd.merge(left=self.df1, right=self.df2, left_on='admit', right_on='admit')
        self.assertTrue(len(result.columns) == 7)

    def testReturnsDataframe(self):
        result = pd.merge(left=self.df1, right=self.df2, left_on='admit', right_on='admit')
        self.assertTrue(type(result) == pd.DataFrame)

unittest.main(argv=[''], verbosity=2, exit=False)
```

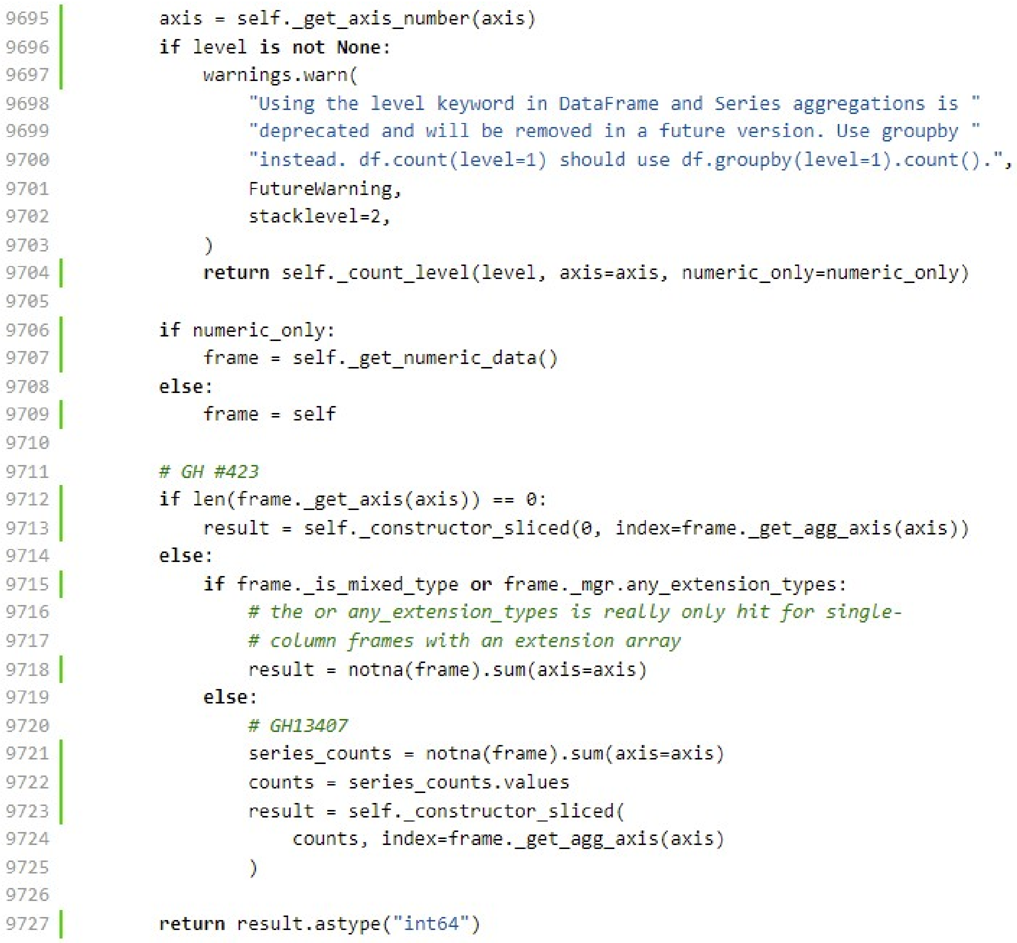
3.12 pandas.DataFrame.count

3.12.1 Black-box testing

Please see this google colab link: Here. All the documentation are included.

3.12.2 White-box testing

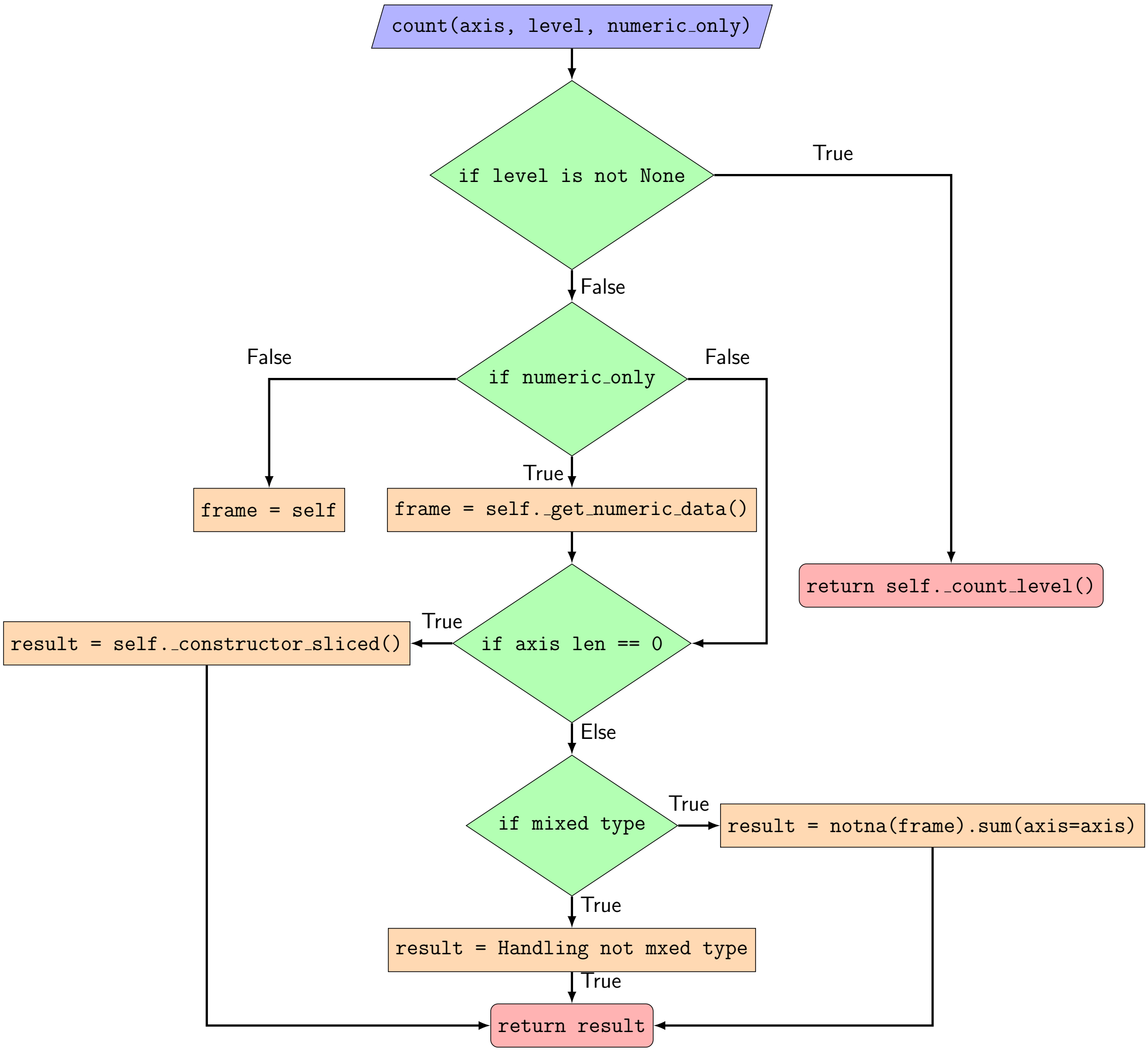
For detailed information please see attached testing code: appendix.??. Here is the coverage report of selected function. For source code, please see: here line 9625 to line 9727.



```
        axis = self._get_axis_number(axis)
        if level is not None:
            warnings.warn(
                "Using the level keyword in DataFrame and Series aggregations is "
                "deprecated and will be removed in a future version. Use groupby "
                "instead. df.count(level=1) should use df.groupby(level=1).count().",
                FutureWarning,
                stacklevel=2,
            )
            return self._count_level(level, axis=axis, numeric_only=numeric_only)

        if numeric_only:
            frame = self._get_numeric_data()
        else:
            frame = self

        # GH #423
        if len(frame._get_axis(axis)) == 0:
            result = self._constructor_sliced(0, index=frame._get_agg_axis(axis))
        else:
            if frame._is_mixed_type or frame._mgr.any_extension_types:
                # the or any_extension_types is really only hit for single-
                # column frames with an extension array
                result = notna(frame).sum(axis=axis)
            else:
                # GH13407
                series_counts = notna(frame).sum(axis=axis)
                counts = series_counts.values
                result = self._constructor_sliced(
                    counts, index=frame._get_agg_axis(axis)
                )

        return result.astype("int64")
```

Figure 3: Code coverage

3.12.3 Control flow graph


count(axis, level, numeric_only)
if level is not None
True
False
return self._count_level()
if numeric_only
False
False
True
frame = self
frame = self._get_numeric_data()
if axis len == 0
True
result = self._constructor_sliced()
Else
if mixed type
True
result = notna(frame).sum(axis=axis)
True
result = Handling not mxed type
True
return result

As the flow graph is shown above, to cover all the branches, we need to have test cases according to the branches, to make sure we have full branches coverage, as you can see from the test code.

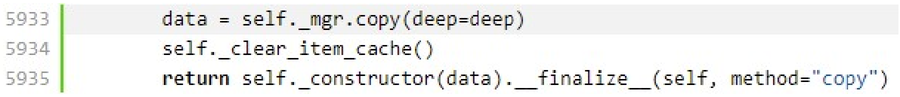
3.13 pandas.DataFrame.copy

3.13.1 Black-box testing

Please see this google colab link: Here. All the documentation are included.

3.13.2 White-box testing

For detailed information please see attached testing code: appendix.??. Here is the coverage report of selected function. For source code, please see: here line 5827 to line 5935.



```
5933        data = self._mgr.copy(deep=deep)
5934        self._clear_item_cache()
5935        return self._constructor(data).__finalize__(self, method="copy")
```

Figure 4: Code coverage

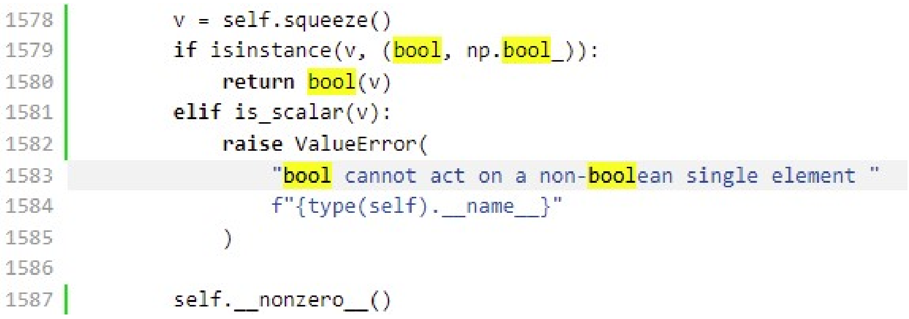
3.14 pandas.DataFrame.bool

3.14.1 Black-box testing

Please see this google colab link: Here. All the documentation are included.

3.14.2 White-box testing

For detailed information please see attached testing code: appendix.??. Here is the coverage report of selected function. For source code, please see: here line 5827 to line 5935.



```
        v = self.squeeze()
        if isinstance(v, (bool, np.bool_)):
            return bool(v)
        elif is_scalar(v):
            raise ValueError(
                "bool cannot act on a non-boolean single element "
                f"{type(self).__name__}"
            )

        self.__nonzero__()
```

Figure 5: Code coverage

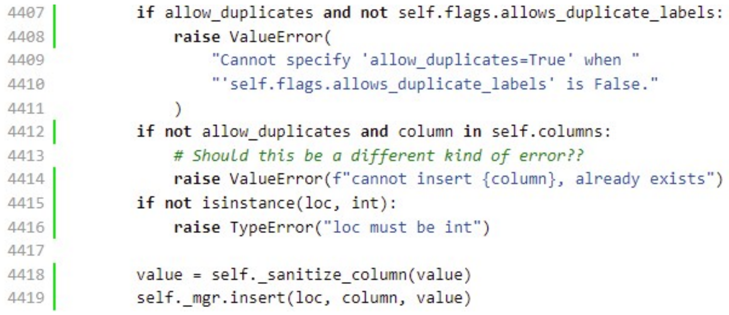
3.15 pandas.DataFrame.insert

3.15.1 Black-box testing

Please see this google colab link: Here. All the documentation are included.

3.15.2 White-box testing

For detailed information please see attached testing code: appendix.??. Here is the coverage report of selected function. For source code, please see: line 4361 to line 4419.



```
4407        if allow_duplicates and not self.flags.allows_duplicate_labels:
4408            raise ValueError(
4409                "Cannot specify 'allow_duplicates=True' when "
4410                "'self.flags.allows_duplicate_labels' is False."
4411            )
4412        if not allow_duplicates and column in self.columns:
4413            # Should this be a different kind of error??
4414            raise ValueError(f"cannot insert {column}, already exists")
4415        if not isinstance(loc, int):
4416            raise TypeError("loc must be int")
4417
4418        value = self._sanitize_column(value)
4419        self._mgr.insert(loc, column, value)
```

Figure 6: Code coverage

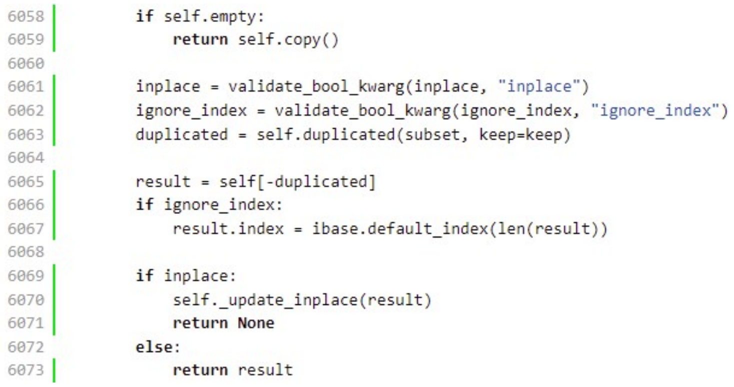
3.16 pandas.DataFrame.drop_duplicates

3.16.1 Black-box testing

Please see this google colab link: Here. All the documentation are included.

3.16.2 White-box testing

For detailed information please see attached testing code: appendix.??. Here is the coverage report of selected function. For source code, please see: line 5977 to line 6073.



```
6058        if self.empty:
6059            return self.copy()
6060
6061        inplace = validate_bool_kwarg(inplace, "inplace")
6062        ignore_index = validate_bool_kwarg(ignore_index, "ignore_index")
6063        duplicated = self.duplicated(subset, keep=keep)
6064
6065        result = self[-duplicated]
6066        if ignore_index:
6067            result.index = ibase.default_index(len(result))
6068
6069        if inplace:
6070            self._update_inplace(result)
6071            return None
6072        else:
6073            return result
```

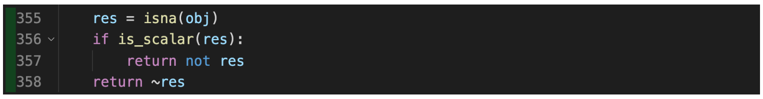
Figure 7: Code coverage

3.17 pandas.notnull

3.17.1 Black-box testing

Please see this google colab link: Here. All the documentation are included.

3.17.2 White-box testing



```python
res = isna(obj)
if is_scalar(res):
    return not res
return ~res
```

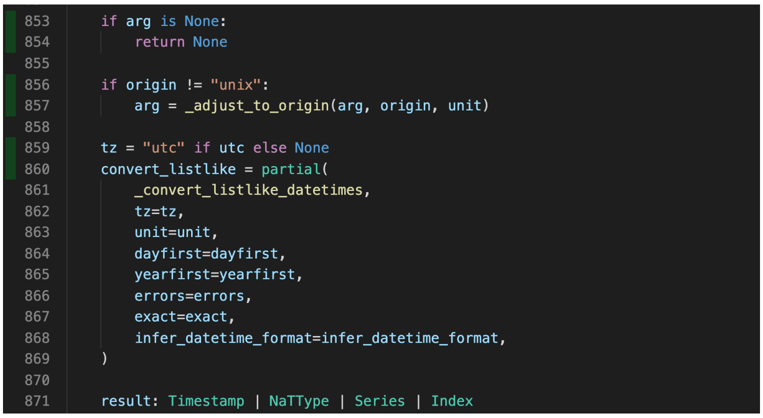
Figure 8: Code coverage

3.18 pandas.pandas.to_datetime

3.18.1 Black-box testing

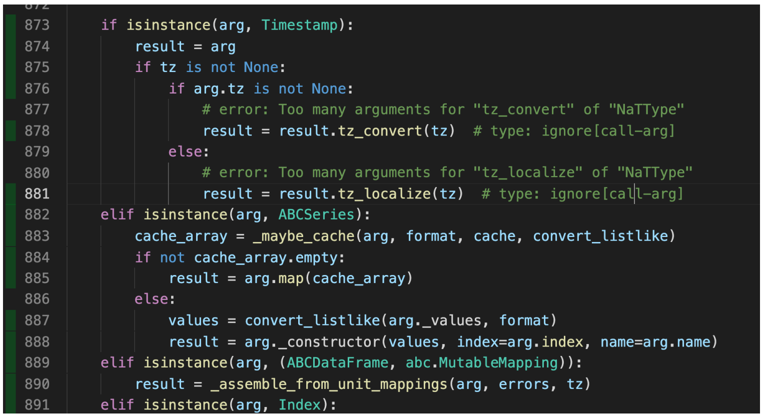
Please see this google colab link: Here. All the documentation are included.

3.18.2 White-box testing



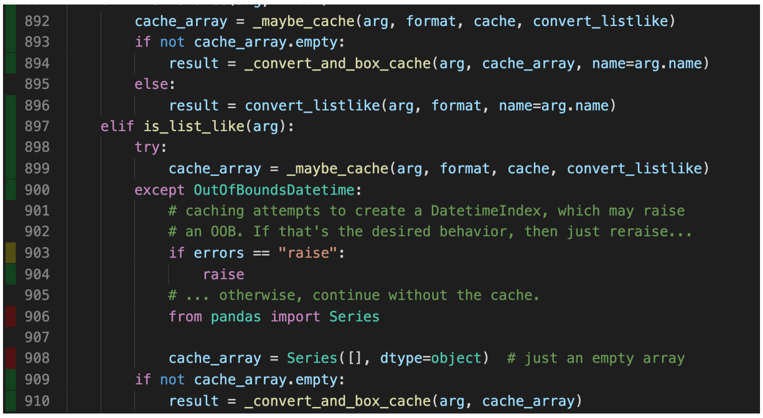
```python
if arg is None:
    return None

if origin != "unix":
    arg = _adjust_to_origin(arg, origin, unit)

tz = "utc" if utc else None
convert_listlike = partial(
    _convert_listlike_datetimes,
    tz=tz,
    unit=unit,
    dayfirst=dayfirst,
    yearfirst=yearfirst,
    errors=errors,
    exact=exact,
    infer_datetime_format=infer_datetime_format,
)

result: Timestamp | NaTType | Series | Index
```

Figure 9: Code coverage1



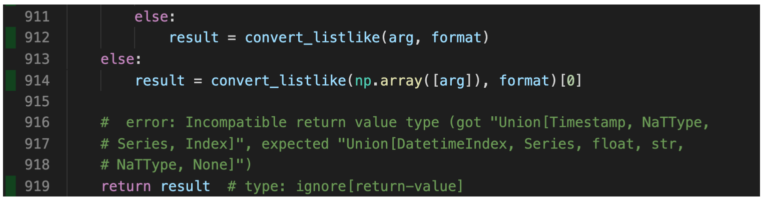
```
    if isinstance(arg, Timestamp):
        result = arg
        if tz is not None:
            if arg.tz is not None:
                # error: Too many arguments for "tz_convert" of "NaTType"
                result = result.tz_convert(tz)  # type: ignore[call-arg]
            else:
                # error: Too many arguments for "tz_localize" of "NaTType"
                result = result.tz_localize(tz)  # type: ignore[call-arg]
    elif isinstance(arg, ABCSeries):
        cache_array = _maybe_cache(arg, format, cache, convert_listlike)
        if not cache_array.empty:
            result = arg.map(cache_array)
        else:
            values = convert_listlike(arg._values, format)
            result = arg._constructor(values, index=arg.index, name=arg.name)
    elif isinstance(arg, (ABCDataFrame, abc.MutableMapping)):
        result = _assemble_from_unit_mappings(arg, errors, tz)
    elif isinstance(arg, Index):
```

Figure 10: Code coverage2



```
        cache_array = _maybe_cache(arg, format, cache, convert_listlike)
        if not cache_array.empty:
            result = _convert_and_box_cache(arg, cache_array, name=arg.name)
        else:
            result = convert_listlike(arg, format, name=arg.name)
    elif is_list_like(arg):
        try:
            cache_array = _maybe_cache(arg, format, cache, convert_listlike)
        except OutOfBoundsDatetime:
            # caching attempts to create a DatetimeIndex, which may raise
            # an OOB. If that's the desired behavior, then just reraise...
            if errors == "raise":
                raise
            # ... otherwise, continue without the cache.
            from pandas import Series

            cache_array = Series([], dtype=object)  # just an empty array
        if not cache_array.empty:
            result = _convert_and_box_cache(arg, cache_array)
```

Figure 11: Code coverage3



```
        else:
            result = convert_listlike(arg, format)
    else:
        result = convert_listlike(np.array([arg]), format)[0]

    #  error: Incompatible return value type (got "Union[Timestamp, NaTType,
    # Series, Index]", expected "Union[DatetimeIndex, Series, float, str,
    # NaTType, None]")
    return result  # type: ignore[return-value]
```

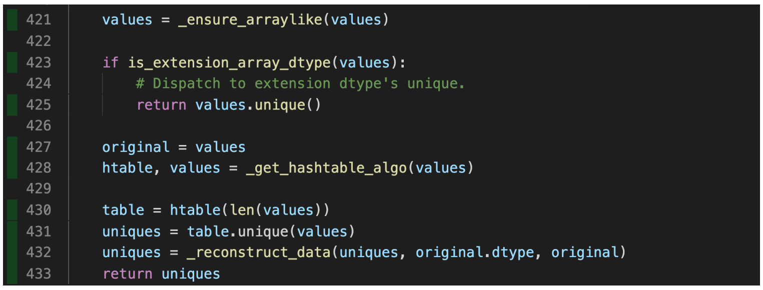
Figure 12: Code coverage4

3.19 pandas.unique

3.19.1 Black-box testing

Please see this google colab link: Here. All the documentation are included.

3.19.2 White-box testing



```
    values = _ensure_arraylike(values)

    if is_extension_array_dtype(values):
        # Dispatch to extension dtype's unique.
        return values.unique()

    original = values
    htable, values = _get_hashtable_algo(values)

    table = htable(len(values))
    uniques = table.unique(values)
    uniques = _reconstruct_data(uniques, original.dtype, original)
    return uniques
```

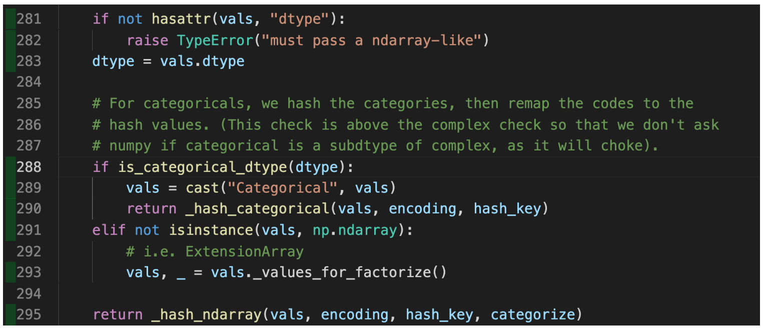
Figure 13: Code coverage

3.20 pandas.util.hash_array

3.20.1 Black-box testing

Please see this google colab link: Here. All the documentation are included.

3.20.2 White-box testing



```
    if not hasattr(vals, "dtype"):
        raise TypeError("must pass a ndarray-like")
    dtype = vals.dtype

    # For categoricals, we hash the categories, then remap the codes to the
    # hash values. (This check is above the complex check so that we don't ask
    # numpy if categorical is a subdtype of complex, as it will choke).
    if is_categorical_dtype(dtype):
        vals = cast("Categorical", vals)
        return _hash_categorical(vals, encoding, hash_key)
    elif not isinstance(vals, np.ndarray):
        # i.e. ExtensionArray
        vals, _ = vals._values_for_factorize()

    return _hash_ndarray(vals, encoding, hash_key, categorize)
```

Figure 14: Code coverage

3.21 pandas.eval

3.21.1 Black-box testing

Please see this google colab link: Here. All the documentation are included.

3.21.2 White-box testing

4 Appendix

4.1 pandas.concat

```python
import unittest
import pandas as pd
import numpy as np


class TestPandasConcat(unittest.TestCase):
    def setUp(self):
        self.df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/binary.dta')

        # Includes start index, excludes end index
        self.df1 = self.df.iloc[0:3]
        self.df2 = self.df.iloc[3:6]

    # Testing whether the amount of rows is correct
    def test1(self):
        expected = 6
        df_concat = pd.concat([self.df1, self.df2])

        self.assertEqual(len(df_concat.index), expected)

    # Testing whether defining the axis on column instead of row functions as intended
    # Concattenated dataframe should be put input as 4 new columns
    def test2(self):
        expected = ['admit', 'gre', 'gpa', 'rank', 'admit', 'gre', 'gpa', 'rank']
        df_concat = pd.concat([self.df1, self.df2], axis=1)
        self.assertEqual(list(df_concat.columns), expected)
        self.assertEqual(len(df_concat.index), 6)

    # Test that ignoring the index will re-index from 0-n whilst true will keep original index.
    def test3(self):
        df_concat_true = pd.concat([self.df2, self.df1], ignore_index=True)
        df_concat_false = pd.concat([self.df2, self.df1], ignore_index=False)
        # Ignore Index (true):          [0, 1, 2, 3, 4, 5]
        # Don't ignore index (false):   [3, 4, 5, 0, 1, 2]

        self.assertFalse(list(df_concat_true.index) == list(df_concat_false.index))
        self.assertEqual(list(df_concat_true.index), [0, 1, 2, 3, 4, 5])

unittest.main(argv=[’’], verbosity=2, exit=False)
```

4.2 pandas.to_numeric

```
import unittest
import numpy as np
import pandas as pd
from numeric import to_numeric

# scalar, list, tuple, 1-d array, or Series
class TestPandasToNumeric(unittest.TestCase):
    def setUp(self):
        self.df = pd.io.stata.read_stata(’https://stats.idre.ucla.edu/stat/stata/dae/binary.dta’)

    # Testing different types of arguments such as Scalar / List.
    def testArgTypes(self):
        self.scalar = self.df[’gpa’]
        self.scalar.squeeze()
        self.list = list(self.df[’gpa’])

        # Scalar (numpy.float32 -> numpy.float32)
        self.assertTrue(isinstance(type(to_numeric(self.scalar)[0]), type(np.float32)))
        # List (float32 -> numpy.float32)
        self.assertTrue(isinstance(type(to_numeric(self.list)[0]), type(np.float32)))

    # Testing a Series where all the strings are numbers
    def testFromString(self):
        series = pd.Series([’7’, ’9’, ’13’])
        result = to_numeric(series)

        self.assertEqual(result[0], pd.Series([7, 9, 13])[0])
        self.assertEqual(result[1], pd.Series([7, 9, 13])[1])
        self.assertEqual(result[2], pd.Series([7, 9, 13])[2])

    # Testing a combination of numbers and string-numbers
    def testFromMixedNumbers(self):
        series = pd.Series([’7’, ’9’, 13])
        result = to_numeric(series)

        self.assertEqual(result[0], pd.Series([7, 9, 13])[0])
        self.assertEqual(result[1], pd.Series([7, 9, 13])[1])
        self.assertEqual(result[2], pd.Series([7, 9, 13])[2])
```

```python

    # Testing a Series where every element is an integer
    def testFromInt(self):
        series = pd.Series([7, 9, 13])
        result = to_numeric(series)

        self.assertEqual(result[0], pd.Series([7, 9, 13])[0])
        self.assertEqual(result[1], pd.Series([7, 9, 13])[1])
        self.assertEqual(result[2], pd.Series([7, 9, 13])[2])

    # Should throw a ValueError in case of inputting strings that are not in number format
    def testWithWords(self):
        self.assertRaises(ValueError, to_numeric, pd.Series(['7', '9', 13, 'forteen']))

unittest.main(argv=[''], verbosity=2, exit=False)
```

4.3 pandas.is_null

```python
import unittest
import pandas as pd
import numpy as np
from numpy.core.numeric import NaN
from pandas._libs.missing import NA

class TestPandasIsNull(unittest.TestCase):
    def setUp(self):
        self.df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/binary.dta')

    def testShouldDetectNullValueTypes(self):
        numericArray = pd.Series([7, 9, 13, NaN, NA])
        objectArray = pd.Series([None, NaN, {}])

        self.assertFalse(pd.isnull(numericArray[0]))
        self.assertFalse(pd.isnull(numericArray[1]))
        self.assertFalse(pd.isnull(numericArray[2]))
        self.assertTrue(pd.isnull(numericArray[3]))
        self.assertTrue(pd.isnull(numericArray[4]))

        self.assertTrue(pd.isnull(objectArray[0]))
        self.assertTrue(pd.isnull(objectArray[1]))
        self.assertFalse(pd.isnull(objectArray[2]))

    def testShouldDetectNullValuesBoolarray(self):
        series = pd.Series([7, 9, 13, NaN, NA])
        expected_result = pd.Series([False, False, False, True, True
])

        for i in range(5):
            self.assertEqual(pd.isnull(series[i]), expected_result[i])

unittest.main(argv=[''], verbosity=2, exit=False)
```

4.4 pandas.merge

```python
import unittest
import pandas as pd
import numpy as np

class TestPandasMerge(unittest.TestCase):
    def setUp(self):
        self.df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/binary.dta')

        # Includes start index, excludes end index
        self.df1 = self.df.iloc[0:3]
        self.df2 = self.df.iloc[3:6]

    def testBasicMerge(self):
        # Merge the two arrays on 'admit' column. This column is shared in output
        result = pd.merge(left=self.df1, right=self.df2, left_on='admit', right_on='admit')
        self.assertTrue(len(result.columns) == 7)

    def testReturnsDataframe(self):
        result = pd.merge(left=self.df1, right=self.df2, left_on='admit', right_on='admit')
        self.assertTrue(type(result) == pd.DataFrame)

unittest.main(argv=[''], verbosity=2, exit=False)
```

4.5 pandas.pivot

```python
import unittest
import pandas as pd
import numpy as np

class TestPandasPivot(unittest.TestCase):
    def setUp(self):
        self.df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two', 'two'],
                        'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'baz': [1, 2, 3, 4, 5, 6],
                   'zoo': ['x', 'y', 'z', 'q', 'w', 't']})

    def testBasicPivot(self):
        result = self.df.pivot(index='foo', columns='bar', values='baz')
        # Pivot should look like this:
        # bar  A   B   C
        # foo
        # one  1   2   3
        # two  4   5   6

        # Check that numbers have been pivoted correctly by retrieving individual numbers from DF
        self.assertEqual(result.iloc[0][0], 1)
        self.assertEqual(result.iloc[0][1], 2)
        self.assertEqual(result.iloc[0][2], 3)
        self.assertEqual(result.iloc[1][0], 4)
        self.assertEqual(result.iloc[1][1], 5)
        self.assertEqual(result.iloc[1][2], 6)

        # Check that columns are correct
        self.assertTrue(len(result.columns.values) == 3)
        self.assertEqual(result.columns.values[0], 'A')
        self.assertEqual(result.columns.values[1], 'B')
        self.assertEqual(result.columns.values[2], 'C')

    def testNaNValues(self):
        result = self.df.pivot(index='bar', columns='baz', values='foo')
        # Pivot should look like this:
        # baz    1    2    3    4    5    6
        # bar
        # A    one  NaN  NaN  two  NaN  NaN
        # B    NaN  one  NaN  NaN  two  NaN
        # C    NaN  NaN  one  NaN  NaN  two

        # Checking for first 3 values of A row has correct values & NaN values
        self.assertEqual(result.iloc[0].get(1), 'one')
        self.assertTrue(pd.isna(result.iloc[0].get(2)))
        self.assertTrue(pd.isna(result.iloc[0].get(3)))

    def testMultipleValues(self):
        result = self.df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])

        # Pivot should contain two columns with their own identical
```

```
subcolumns
        columnNames = list(result.columns.levels[0])
        self.assertEqual(columnNames[0], 'baz')
        self.assertEqual(columnNames[1], 'zoo')

        subColumnNames = list(result.columns.levels[1])
        self.assertEqual(subColumnNames[0], 'A')
        self.assertEqual(subColumnNames[1], 'B')
        self.assertEqual(subColumnNames[2], 'C')

    def testIdenticalValuesError(self):
        # When index contains duplicate it will throw ValueError

        df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/binary.dta')
        with self.assertRaises(ValueError):
            pd.pivot(df, index='admit', columns='gpa', values='rank')



unittest.main(argv=[''], verbosity=2, exit=False)
```

4.6 pandas.date_range

```
import unittest
import pandas as pd
from pandas.core.indexes.datetimes import date_range as dr

class TestDateRange(unittest.TestCase):

    #Test input of range between two defined dates
    def testStartEnd(self):
        x = pd.DatetimeIndex(['2021-11-1', '2021-11-2', '2021-11-3'], dtype='datetime64[ns]', freq='D')
        self.assertSequenceEqual(dr('2021-11-1','2021-11-3').tolist(), x.tolist())

        #Test right output on leap year
        y = pd.DatetimeIndex(['2024-02-28', '2024-02-29', '2024-03-1'], dtype='datetime64[ns]', freq='D')
        self.assertSequenceEqual(dr('2024-02-28','2024-03-1').tolist(), y.tolist())

        #Test hour frequency change date after midnight
        z = pd.DatetimeIndex(['2024-02-28  23:00:00', '2024-02-29 00:00:00', '2024-02-29 01:00:00'], dtype='datetime64[ns]', freq='H')
        self.assertSequenceEqual(dr('2024-02-28 23:00:00','2024-02-29 01:00:00',freq='H').tolist(), z.tolist())

    #Test defined start of interval and period
    def testStart(self):
        #Test 3 period with default freqency, day
        x = pd.DatetimeIndex(['2021-11-1', '2021-11-2', '2021-11-3'], dtype='datetime64[ns]', freq='D')
        self.assertSequenceEqual(pd.date_range('2021-11-1', periods=3).tolist(), x.tolist())

        #Test with month start frequency
        y = pd.DatetimeIndex(['2021-11-01', '2021-12-01', '2022-01-01'], dtype='datetime64[ns]', freq='MS')
        self.assertSequenceEqual(pd.date_range(start='2021-10-06', freq='MS', periods=3).tolist(), y.tolist())
    #Test defined end of interval and period
    def testEnd(self):
        #Test 3 periods and default frequency, day
        x = pd.DatetimeIndex(['2021-11-29', '2021-11-30', '2021-12-1'], dtype='datetime64[ns]', freq='D')
        self.assertSequenceEqual(pd.date_range(end='2021-12-1', periods=3).tolist(), x.tolist())

        #Test 3 periods with 3 month frequency
        m = pd.DatetimeIndex(['2020-01-31', '2020-04-30', '2020-07-31'], dtype='datetime64[ns]', freq='3M')
        self.assertSequenceEqual(pd.date_range(end='2020-07-31', freq='3M', periods=3).tolist(), m.tolist())

    #Test with defined timezone
    def testTimeZone(self):
        y = pd.DatetimeIndex(['2024-02-28 01:00:00','2024-02-28 02:00:00','2024-02-28 03:00:00'], freq='H',tz='Asia/Hong_Kong')
        x = pd.DatetimeIndex(['2024-02-28 01:00:00','2024-02-28 02:00:00','2024-02-28 03:00:00'], freq='H',tz='America/Los_Angeles')

        self.assertSequenceEqual(pd.date_range('2024-02-28 01:00:00', '2024-02-28 03:00:00', freq='H',tz='Asia/Hong_Kong').tolist(),y.tolist())
        self.assertSequenceEqual(pd.date_range('2024-02-28 01:00:00', '2024-02-28 03:00:00', freq='H',tz='America/Los_Angeles').tolist(),x.tolist())

        self.assertNotEqual(x.tolist(),y.tolist())
        self.assertNotEqual(x.tz,y.tz)

```

```python
    #Test 'normalize' true, reset time to midnight
    def testNormalize(self):
        x = pd.DatetimeIndex(['2024-02-28  00:00:00', '2024-02-29 00:00:00', '2024-03-01 00:00:00'], dtype='datetime64[ns]', freq='D')
        self.assertSequenceEqual(pd.date_range('2024-02-28 23:00:00','2024-03-01 01:00:00',freq='D',normalize=True).tolist(), x.tolist())
        self.assertEqual(pd.date_range('2024-02-28 23:00:00','2024-03-01 01:00:00',freq='D',normalize=True)[0].hour,0)

    #Test 'closed' to right and left will discard last and first value
    def testClosed(self):
        r = pd.DatetimeIndex(['2024-02-29', '2024-03-01'], dtype='datetime64[ns]', freq='D')
        l = pd.DatetimeIndex(['2024-02-28', '2024-02-29'], dtype='datetime64[ns]', freq='D')

        self.assertSequenceEqual(pd.date_range('2024-02-28','2024-03-01',freq='D',closed='right').tolist(), r.tolist())
        self.assertSequenceEqual(pd.date_range('2024-02-28','2024-03-01',freq='D',closed='left').tolist(), l.tolist())
```

4.7 pandas.bdate_range

```python
import unittest
import pandas as pd

class TestBdateRange(unittest.TestCase):

    #Test input of range between two defined dates
    def testStartEnd(self):
        x = pd.DatetimeIndex(['2021-11-22', '2021-11-23', '2021-11-24', '2021-11-25', '2021-11-26'], dtype='datetime64[ns]', freq='B')
        self.assertSequenceEqual(pd.bdate_range('2021-11-20','2021-11-28').tolist(),x.tolist())

        #Test with leap year
        y = pd.DatetimeIndex(['2024-02-28', '2024-02-29', '2024-03-1'], dtype='datetime64[ns]', freq='D')
        self.assertSequenceEqual(pd.bdate_range('2024-02-28','2024-03-1').tolist(), y.tolist())

        #Test with 5 hour interval
        z = pd.DatetimeIndex(['2021-03-29 00:00:00', '2021-03-29 05:00:00', '2021-03-29 10:00:00', '2021-03-29 15:00:00', '
2021-03-29 20:00:00'], dtype='datetime64[ns]', freq='5H')
        self.assertSequenceEqual(pd.bdate_range('2021-03-29 23:00:00','2021-03-30 09:00:00',freq='5H', normalize=True).tolist(), z.tolist())

    #Defined start date, no end date but with periode and freqence
    def testStart(self):
        #start weekend and periode of 3 days
        x = pd.DatetimeIndex(['2021-11-1', '2021-11-2', '2021-11-3'], dtype='datetime64[ns]', freq='B')
        self.assertSequenceEqual(pd.bdate_range(start='2021-10-30', periods=3, freq='B').tolist(), x.tolist())

        #Beginning of month frequence in 3 periods
        y = pd.DatetimeIndex(['2021-11-01', '2021-12-01', '2022-01-01'], dtype='datetime64[ns]', freq='MS')
        self.assertSequenceEqual(pd.bdate_range(start='2021-11-01', periods=3, freq='MS').tolist(), y.tolist())

    #Defined end date, no start date but with periode and freqence
    def testEnd(self):
        #end midt week and get 4 days before
        x = pd.DatetimeIndex(['2021-11-26', '2021-11-29', '2021-11-30', '2021-12-1'], dtype='datetime64[ns]', freq='B')
        self.assertSequenceEqual(pd.bdate_range(end='2021-12-1', periods=4).tolist(), x.tolist())

        #get 3 periods with 3 business days interval
        m = pd.DatetimeIndex(['2020-07-23', '2020-07-28', '2020-07-31'], dtype='datetime64[ns]', freq='3B')
        self.assertSequenceEqual(pd.bdate_range(end='2020-07-31', freq='3B', periods=3).tolist(), m.tolist())
    #Test with timezone
    def testTimeZone(self):
        y = pd.DatetimeIndex(['2024-02-28 00:00:00','2024-02-29 00:00:00'], freq='B',tz='Asia/Hong_Kong')
        x = pd.DatetimeIndex(['2024-02-28 00:00:00','2024-02-29 00:00:00'], freq='B',tz='America/Los_Angeles')

        #Test time zone is added
        self.assertSequenceEqual(pd.bdate_range('2024-02-28 01:00:00', '2024-02-29 03:00:00', freq='B',tz='Asia/Hong_Kong').tolist(), y.tolist())
        self.assertSequenceEqual(pd.bdate_range('2024-02-28 01:00:00', '2024-02-29 03:00:00', freq='B',tz='America/Los_Angeles').tolist(),x.tolist())
        #Output with two different timezones are not equal
        self.assertNotEqual(x.tolist(),y.tolist())
        self.assertNotEqual(x.tz,y.tz)

    #Test normalization of input will set the time to midtnight
    def testNormalize(self):
        x = pd.DatetimeIndex(['2024-02-28  00:00:00', '2024-02-29 00:00:00', '2024-03-01 00:00:00'], dtype='datetime64[ns]', freq='B')
        self.assertSequenceEqual(pd.bdate_range('2024-02-28 23:00:00','2024-03-01 01:00:00',freq='B',normalize=True).tolist(), x.tolist())
        self.assertEqual(pd.bdate_range('2024-02-28 23:00:00','2024-03-01 01:00:00',freq='B',normalize=True)[0].hour,0)

    #Test closed from left and right will omit ends
    def testClosed(self):

        r = pd.DatetimeIndex(['2021-11-29', '2021-11-30', '2021-12-1'], dtype='datetime64[ns]', freq='B')
        l = pd.DatetimeIndex(['2021-11-26', '2021-11-29', '2021-11-30'], dtype='datetime64[ns]', freq='B')

        self.assertSequenceEqual(pd.bdate_range('2021-11-26','2021-12-01',freq='B',closed='right').tolist(), r.tolist())
        self.assertSequenceEqual(pd.bdate_range('2021-11-26','2021-12-01',freq='B',closed='left').tolist(), l.tolist())
```

4.8 pandas.periode_range

```
import unittest
import pandas as pd

class TestPeriodRange(unittest.TestCase):

    #Test input with defined start and end
    def testStartEnd(self):
        x = pd.PeriodIndex(['2021-11-1', '2021-11-2', '2021-11-3'], dtype='period[D]')
        self.assertSequenceEqual(pd.period_range('2021-11-1','2021-11-3').tolist(), x.tolist())

        #test leap year
        y = pd.PeriodIndex(['2024-02-28', '2024-02-29', '2024-03-1'], dtype='period[D]')
        self.assertSequenceEqual(pd.period_range('2024-02-28','2024-03-1').tolist(), y.tolist())

        #test day shift after hour pass 24
        z = pd.PeriodIndex(['2024-02-28  23:00:00', '2024-02-29
00:00:00', '2024-02-29 01:00:00'], dtype='period[H]')
        self.assertSequenceEqual(pd.period_range('2024-02-28 23:00:00','2024-02-29 01:00:00',freq='H').tolist(), z.tolist())

    #Test defined start with defined period
    def testStart(self):
        #start with 3 day period
        x = pd.PeriodIndex(['2021-11-1', '2021-11-2', '2021-11-3'], dtype='period[D]')
        self.assertSequenceEqual(pd.period_range('2021-11-1',periods=3).tolist(), x.tolist())
        #start with 3 month period
        y = pd.PeriodIndex(['2021-11', '2021-12', '2022-01'], dtype='period[M]')
        self.assertSequenceEqual(pd.period_range('2021-11-01', freq='M', periods=3).tolist(), y.tolist())

    #Test definded end with defined period
    def testEnd(self):
        #end with 3 day period
        x = pd.PeriodIndex(['2021-11-29', '2021-11-30', '2021-12-1'], dtype='period[D]')
        self.assertSequenceEqual(pd.period_range(end='2021-12-1', periods=3).tolist(), x.tolist())
        #end with 3 year period
        y = pd.PeriodIndex(['2020', '2021', '2022'], dtype='period[Y]')
        self.assertSequenceEqual(pd.period_range(end='2022-01-01', freq='Y', periods=3).tolist(), y.tolist())

    #Test defined start, end and frequency
    def testPeriodInput(self):
        start = pd.Period('2021-01', freq='M')
        end = pd.Period('2021-04', freq='M')
        x = pd.PeriodIndex(['2021-01', '2021-02', '2021-03', '2021-04'], dtype='period[M]')
        self.assertSequenceEqual(pd.period_range(start,end,freq='M').tolist(), x.tolist())
```

4.9 pandas.to_timedelta

```
import unittest
import pandas as pd
import numpy as np

class TestToTimeDelta(unittest.TestCase):

    #Test if all input is for the different units is converted
```

```
correctly to value.
 #Expected values are in nano seconds calculated and tested on google
 def testUnitValues(self):
     self.assertEqual(pd.to_timedelta(0).value, 0) #zero input
     self.assertEqual(pd.to_timedelta(1).value, 1) #default nano sec
     self.assertEqual(pd.to_timedelta(1, unit='us').value, 1000)
     self.assertEqual(pd.to_timedelta(1, unit='ms').value, 10 ** 6)
     self.assertEqual(pd.to_timedelta(1, unit='s').value, 10 ** 9)
     self.assertEqual(pd.to_timedelta(1, unit='m').value, 60 * 10 ** 9)
     self.assertEqual(pd.to_timedelta(1, unit='h').value, 3600 * 10 ** 9)
     self.assertEqual(pd.to_timedelta(1, unit='d').value, 8.64 * 10 ** 13)
     self.assertEqual(pd.to_timedelta(1, unit='w').value, 6.04800 * 10 ** 14)

 #Test if value error is thrown on deprecated unit input
 def testErrorOnDeprecatedInput(self):
     self.assertRaises(ValueError,pd.to_timedelta,1,unit='y')
     self.assertRaises(ValueError,pd.to_timedelta,1,unit='M')

 #Test if different kind of input gives the expected output
 #Different kind of string input is converted using the expected unit
 def testInputValue(self):
     self.assertEqual(pd.to_timedelta(0), pd.Timedelta('0 days 00:00:00.0'))
     self.assertEqual(pd.to_timedelta(1000), pd.Timedelta('0 days 00:00:00.000001'))
     self.assertEqual(pd.to_timedelta('1sec'), pd.Timedelta('0 days 00:00:01.000000'))
     self.assertEqual(pd.to_timedelta('13:00:00'), pd.Timedelta('0 days 13:00:00.000000'))
     self.assertEqual(pd.to_timedelta('8 days 11:23:33.123456789'), pd.Timedelta('8 days 11:23:33.123456789'))

     self.assertEqual(pd.to_timedelta('61sec'), pd.Timedelta('0 days 00:01:01.000000'))
     self.assertEqual(pd.to_timedelta('1.5min'), pd.Timedelta('0 days 00:01:30.000000'))
     self.assertEqual(pd.to_timedelta('0.2min'), pd.Timedelta('0 days 00:00:12.000000'))
     self.assertEqual(pd.to_timedelta('25hours'), pd.Timedelta('1
```

```python
 days 01:00:00.000000'))
        self.assertEqual(pd.to_timedelta('32 days 5 hours'), pd.Timedelta('32 days 05:00:00.000000'))

    #Test if different units are converted correctly to the 'Timedelta' format
    #All not deprecated units are tested
    def testUnitConversion(self):
        self.assertEqual(pd.to_timedelta(1), pd.Timedelta('0 days 00:00:00.000000001')) #default nano sec
        self.assertEqual(pd.to_timedelta(1, unit='us'), pd.Timedelta('0 days 00:00:00.000001'))
        self.assertEqual(pd.to_timedelta(1, unit='ms'), pd.Timedelta('0 days 00:00:00.001000'))
        self.assertEqual(pd.to_timedelta(1, unit='s'), pd.Timedelta('0 days 00:00:01.000000'))
        self.assertEqual(pd.to_timedelta(1, unit='m'), pd.Timedelta('0 days 00:01:00.000000'))
        self.assertEqual(pd.to_timedelta(1, unit='h'), pd.Timedelta('0 days 01:00:00.000000'))
        self.assertEqual(pd.to_timedelta(1, unit='d'), pd.Timedelta('0 days 24:00:00.000000'))
        self.assertEqual(pd.to_timedelta(1, unit='w'), pd.Timedelta('7 days 00:00:00.000000'))

    #Test argument as list outputs list
    def testListInput(self):
        x = ['1 days 06:05:01.00003', '15.5us', '4hr 7minutes']
        self.assertEqual(pd.to_timedelta(x).tolist(),
                         [pd.Timedelta('1 days 06:05:01.00003'), pd.Timedelta('0 days 00:00:00.000015500'),
                          pd.Timedelta('0 days 04:07:00')])

        #Test with empty list input
        y = []
        self.assertEqual(pd.to_timedelta(y).tolist(),[])

    #Test array and unit argument output list with right unit
    def testNumbersToUnit(self):

        self.assertEqual(pd.to_timedelta(np.arange(3), unit='sec').tolist(),
                         [pd.Timedelta('0 days 00:00:00'), pd.Timedelta('0 days 00:00:01'),
                          pd.Timedelta('0 days 00:00:02')])

        self.assertEqual(pd.to_timedelta(np.arange(4), unit='day').tolist(),
                              [pd.Timedelta(’0 days 00:00:00’), pd.Timedelta(’1 days 00:00:00’),
                               pd.Timedelta(’2 days 00:00:00’),pd.Timedelta(’3 days 00:00:00’)])

        #Test with empty array input
        self.assertEqual(pd.to_timedelta(np.arange(0), unit=’sec’).tolist(),[])
```

4.10 pandas.DataFrame.tail

```
import unittest
import pandas as pd
from pandas._testing import assert_frame_equal


class TestDataframeTail(unittest.TestCase):
    df = pd.DataFrame({’fruit’: [’apple’, ’pear’, ’banana’, ’melon’, ’lemon’, ’mango’, ’lime’]})
    expected = pd.DataFrame({’fruit’: [’apple’, ’pear’, ’banana’, ’melon’, ’lemon’, ’mango’, ’lime’]})

    #Test output values are the last five elements as default
    def testReturnDefault(self):
        expected = self.expected.drop([0, 1])
        assert_frame_equal(self.df.tail(),expected)

    #Test defined number of elements to return
    def testNvalueReturn(self):
        expected = self.expected.drop([0,1,2,3,4,5])
        assert_frame_equal(self.df.tail(1),expected)

        expected = self.expected.drop([0,1,2,3])
        assert_frame_equal(self.df.tail(3), expected)

        #all
        expected = self.expected
        assert_frame_equal(self.df.tail(7), expected)

    #Test if returns empty list on zero as argument
    def testZeroReturn(self):
        expected = self.expected.drop([0,1,2,3,4,5,6])
        assert_frame_equal(self.df.tail(0), expected)

    #Test with negative argument
    def testNegativeInput(self):
        expected = self.expected.drop([0,1,2,3])
        assert_frame_equal(self.df.tail(-4), expected)

        expected = self.expected.drop([0,1,2,3,4,5,6])
        assert_frame_equal(self.df.tail(-7), expected)

    #Test with positive and negative input out of range
    def testOutOffBoundInput(self):
        expected = self.expected.drop([0,1,2,3,4,5,6])
        assert_frame_equal(self.df.tail(-7000), expected)

        expected = self.expected
        assert_frame_equal(self.df.tail(7000), expected)
```