

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	1
2	Testing Strategy	3
3	Tests	4
3.1	pandas.to_timedelta	4
3.1.1	Black-box testing	4
3.1.2	Whitebox-box testing	6
3.2	pandas.date_range	9
3.2.1	Black-box testing	9
3.3	pandas.bdate_range	11
3.3.1	Black-box testing	11
3.4	pandas.period_range	13
3.4.1	Black-box testing	13
3.5	pandas.DataFrame.tail	14
3.5.1	Black-box testing	15
3.6	pandas.merge_ordered	16
3.6.1	Whitebox-box testing	16
3.7	pandas.to_numeric	18
3.7.1	Black-box testing	18
3.7.2	White-box testing	20
3.8	pandas.pivot	23
3.8.1	Black-box testing	23
3.8.2	White-box testing	24
3.9	pandas.concat	27
3.9.1	Black-box testing	27
3.10	pandas.is_null	28
3.10.1	Black-box testing	29
3.11	pandas.merge	29
3.11.1	Black-box testing	29
3.12	pandas.DataFrame.count	30
3.12.1	Black-box testing	30
3.12.2	White-box testing	30
3.12.3	Control flow graph	32
3.13	pandas.DataFrame.copy	32
3.13.1	Black-box testing	32
3.13.2	White-box testing	33
3.14	pandas.DataFrame.bool	33

3.14.1	Black-box testing	33
3.14.2	White-box testing	33
3.15	pandas.DataFrame.insert	33
3.15.1	Black-box testing	33
3.15.2	White-box testing	34
3.16	pandas.DataFrame.drop_duplicates	34
3.16.1	Black-box testing	34
3.16.2	White-box testing	34
3.17	pandas.notnull	35
3.17.1	Black-box testing	35
3.17.2	White-box testing	35
3.18	pandas.pandas.to_datetime	35
3.18.1	Black-box testing	35
3.18.2	White-box testing	35
3.19	pandas.unique	36
3.19.1	Black-box testing	36
3.19.2	White-box testing	37
3.20	pandas.util.hash_array	37
3.20.1	Black-box testing	37
3.20.2	White-box testing	37
3.21	pandas.eval	37
3.21.1	Black-box testing	37
3.21.2	White-box testing	37
4	Appendix	38
4.1	pandas.concat	38
4.2	pandas.to_numeric	39
4.3	pandas.is_null	40
4.4	pandas.merge	41
4.5	pandas.pivot	41
4.6	pandas.date_range	43
4.7	pandas.bdate_range	45
4.8	pandas.periode_range	47
4.9	pandas.to_timedelta	48
4.10	pandas.DataFrame.tail	51

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`
- `pandas.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

```
1 import unittest
2 import pandas as pd
3 import numpy as np
4
5 class TestToTimeDelta(unittest.TestCase):
6
7     #Test if all input is for the different units is converted
8     #Expected values are in nano seconds calculated and tested on
9     #google
10    def testUnitValues(self):
11        self.assertEqual(pd.to_timedelta(0).value, 0) #zero input
12        self.assertEqual(pd.to_timedelta(1).value, 1) #default nano
13        sec
14        self.assertEqual(pd.to_timedelta(1, unit='us').value, 1000)
15        self.assertEqual(pd.to_timedelta(1, unit='ms').value, 10 **
16        6)
17        self.assertEqual(pd.to_timedelta(1, unit='s').value, 10 **
18        9)
19        self.assertEqual(pd.to_timedelta(1, unit='m').value, 60 * 10
20        ** 9)
21        self.assertEqual(pd.to_timedelta(1, unit='h').value, 3600 *
22        10 ** 9)
23        self.assertEqual(pd.to_timedelta(1, unit='d').value, 8.64 *
24        10 ** 13)
25        self.assertEqual(pd.to_timedelta(1, unit='w').value, 6.04800
26        * 10 ** 14)
27
28    #Test if value error is thrown on deprecated unit input
29    def testErrorOnDeprecatedInput(self):
30        self.assertRaises(ValueError, pd.to_timedelta, 1, unit='y')
31        self.assertRaises(ValueError, pd.to_timedelta, 1, unit='M')
32
33    #Test if different kind of input gives the expected output
```

```

26     #Different kind of string input is converted using the expected
unit
27     def testInputValue(self):
28         self.assertEqual(pd.to_timedelta(0), pd.Timedelta('0 days
00:00:00.0'))
29         self.assertEqual(pd.to_timedelta(1000), pd.Timedelta('0 days
00:00:00.000001'))
30         self.assertEqual(pd.to_timedelta('1sec'), pd.Timedelta('0
days 00:00:01.000000'))
31         self.assertEqual(pd.to_timedelta('13:00:00'), pd.Timedelta('
0 days 13:00:00.000000'))
32         self.assertEqual(pd.to_timedelta('8 days 11:23:33.123456789'
), pd.Timedelta('8 days 11:23:33.123456789'))
33
34         self.assertEqual(pd.to_timedelta('61sec'), pd.Timedelta('0
days 00:01:01.000000'))
35         self.assertEqual(pd.to_timedelta('1.5min'), pd.Timedelta('0
days 00:01:30.000000'))
36         self.assertEqual(pd.to_timedelta('0.2min'), pd.Timedelta('0
days 00:00:12.000000'))
37         self.assertEqual(pd.to_timedelta('25hours'), pd.Timedelta('1
days 01:00:00.000000'))
38         self.assertEqual(pd.to_timedelta('32 days 5 hours'), pd.
Timedelta('32 days 05:00:00.000000'))
39
40     #Test if different units are converted correctly to the '
Timedelta' format
41     #All not deprecated units are tested
42     def testUnitConversion(self):
43         self.assertEqual(pd.to_timedelta(1), pd.Timedelta('0 days
00:00:00.000000001')) #default nano sec
44         self.assertEqual(pd.to_timedelta(1, unit='us'), pd.Timedelta(
'0 days 00:00:00.000001'))
45         self.assertEqual(pd.to_timedelta(1, unit='ms'), pd.Timedelta(
'0 days 00:00:00.001000'))
46         self.assertEqual(pd.to_timedelta(1, unit='s'), pd.Timedelta(
'0 days 00:00:01.000000'))
47         self.assertEqual(pd.to_timedelta(1, unit='m'), pd.Timedelta(
'0 days 00:01:00.000000'))
48         self.assertEqual(pd.to_timedelta(1, unit='h'), pd.Timedelta(
'0 days 01:00:00.000000'))
49         self.assertEqual(pd.to_timedelta(1, unit='d'), pd.Timedelta(
'0 days 24:00:00.000000'))
50         self.assertEqual(pd.to_timedelta(1, unit='w'), pd.Timedelta(
'7 days 00:00:00.000000'))
51
52     #Test argument as list outputs list
53     def testListInput(self):

```



```

54     x = ['1 days 06:05:01.00003', '15.5us', '4hr 7minutes']
55     self.assertEqual(pd.to_timedelta(x).tolist(),
56                      [pd.Timedelta('1 days 06:05:01.00003'), pd.
Timedelta('0 days 00:00:00.000015500'),
57                      pd.Timedelta('0 days 04:07:00')])
58
59     #Test with empty list input
60     y = []
61     self.assertEqual(pd.to_timedelta(y).tolist(), [])
62
63     #Test array and unit argument output list with right unit
64     def testNumbersToUnit(self):
65
66         self.assertEqual(pd.to_timedelta(np.arange(3), unit='sec').
tolist(),
67                          [pd.Timedelta('0 days 00:00:00'), pd.
Timedelta('0 days 00:00:01'),
68                          pd.Timedelta('0 days 00:00:02')])
69
70         self.assertEqual(pd.to_timedelta(np.arange(4), unit='day').
tolist(),
71                          [pd.Timedelta('0 days 00:00:00'), pd.
Timedelta('1 days 00:00:00'),
72                          pd.Timedelta('2 days 00:00:00'), pd.
Timedelta('3 days 00:00:00')])
73
74     #Test with empty array input
75     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.

```

1  import unittest
2  import pandas as pd
3  import numpy as np
4
5  from timedeltas import to_timedelta
6
7  class TestToTimedeltaWB(unittest.TestCase):
8
9      #Line 127-128: should return 'None' if given 'None' as argument
10     def testPassNone(self):
11         self.assertEqual(to_timedelta(None, unit='w'), None)
12

```

```

13     #Line 121-122: unit parameter 'year' and 'month' is deprecated
    and should return value error
14     def testDeprecatedUnit(self):
15         #Value error on deprecated Y, y and M (year, month)
16         self.assertRaises(ValueError,to_timedelta,1,unit='Y')
17         self.assertRaises(ValueError,to_timedelta,1,unit='M')
18
19     #144-145: value error on argument as string AND defined unit.
    Unit should only be defined on non-string argument.
20     def testStringInputWithUnit(self):
21         #Value error on string input and defined unit
22         self.assertRaises(ValueError,to_timedelta,'7 days',unit='w')
23
24     #118-119: Value error on should be thrown if 'error' defined
    different from 'raise', 'ignore' or 'coerce'.
25     def testErrorInput(self):
26         #Value error on error messages different from 'raise', '
    ignore' or 'coerce'
27         self.assertRaises(ValueError,to_timedelta,1,unit='d',errors=
    'wrong')
28
29     #129-131: Pandas series object as argument
30     def testSeriesObject(self):
31         # series object
32         d = {'a': 1, 'b': 2, 'c': 3}
33         ser = pd.Series(data=d, index=['a', 'b', 'c'])
34         self.assertEqual(to_timedelta(ser, unit='d').tolist(),
35             [pd.Timedelta('1 days 00:00:00'), pd.
    Timedelta('2 days 00:00:00'),
36             pd.Timedelta('3 days 00:00:00')])
37
38     #132-33: Pandas Index object as argument
39     def testIndexObject(self):
40         # Index object
41         ind = pd.Index([1, 2, 3])
42         self.assertEqual(to_timedelta(ind, unit='d').tolist(),
43             [pd.Timedelta('1 days 00:00:00'), pd.
    Timedelta('2 days 00:00:00'),
44             pd.Timedelta('3 days 00:00:00')])
45
46     #134-136: Numpy array with zero dimation as argument
47     def testZeroDimNpArray(self):
48         # zero dimation numpy array
49         arr = np.array(1)
50         self.assertEqual(to_timedelta(arr, unit='d'), pd.Timedelta('
    1 days 00:00:00'))
51

```

```

52     #139-140: Array with multiple dimentions as argument output type
        error
53     def testMultiDimArray(self):
54         #Type error on 3 dimensional array
55         arr2 = np.zeros((2, 3, 4))
56         self.assertRaises(TypeError, to_timedelta, arr2)
57
58     #137-1138: Array with one dimention as input returns list of
        output
59     def testListInput(self):
60         x = ['1 days 06:05:01.00003', '15.5us', '4hr 7minutes']
61         self.assertEqual(to_timedelta(x).tolist(),
62                          [pd.Timedelta('1 days 06:05:01.00003'), pd.
Timedelta('0 days 00:00:00.000015500'),
63                          pd.Timedelta('0 days 04:07:00')])
64
65
66 if __name__ == '__main__':
67     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

```
1 import unittest
2 import pandas as pd
3 from pandas.core.indexes.datetimes import date_range as dr
4
5 class TestDateRange(unittest.TestCase):
6
7     #Test input of range between two defined dates
8     def testStartEnd(self):
9         x = pd.DatetimeIndex(['2021-11-1', '2021-11-2', '2021-11-3'
10 ], dtype='datetime64[ns]', freq='D')
11         self.assertEqual(dr('2021-11-1', '2021-11-3').tolist(), x.tolist())
12
13     #Test right output on leap year
14     y = pd.DatetimeIndex(['2024-02-28', '2024-02-29', '2024-03-1'], dtype='datetime64[ns]', freq='D')
15     self.assertEqual(dr('2024-02-28', '2024-03-1').tolist(), y.tolist())
16
17     #Test hour frequency change date after midnight
18     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')
19     self.assertEqual(dr('2024-02-28 23:00:00', '2024-02-29 01:00:00', freq='H').tolist(), z.tolist())
20
21     #Test defined start of interval and period
22     def testStart(self):
23         #Test 3 period with default frequency, day
24         x = pd.DatetimeIndex(['2021-11-1', '2021-11-2', '2021-11-3'], dtype='datetime64[ns]', freq='D')
25         self.assertEqual(pd.date_range('2021-11-1', periods=3).tolist(), x.tolist())
26
27     #Test with month start frequency
28     y = pd.DatetimeIndex(['2021-11-01', '2021-12-01', '2022-01-01'], dtype='datetime64[ns]', freq='MS')
```

```

28         self.assertSequenceEqual(pd.date_range(start='2021-10-06',
29         freq='MS', periods=3).tolist(), y.tolist())
30         #Test defined end of interval and period
31         def testEnd(self):
32             #Test 3 periods and default frequency, day
33             x = pd.DatetimeIndex(['2021-11-29', '2021-11-30', '2021-12-1
34             '], dtype='datetime64[ns]', freq='D')
35             self.assertSequenceEqual(pd.date_range(end='2021-12-1',
36             periods=3).tolist(), x.tolist())
37
38             #Test 3 periods with 3 month frequency
39             m = pd.DatetimeIndex(['2020-01-31', '2020-04-30', '
40             2020-07-31'], dtype='datetime64[ns]', freq='3M')
41             self.assertSequenceEqual(pd.date_range(end='2020-07-31',
42             freq='3M', periods=3).tolist(), m.tolist())
43
44             #Test with defined timezone
45             def testTimeZone(self):
46                 y = pd.DatetimeIndex(['2024-02-28 01:00:00', '2024-02-28
47                 02:00:00', '2024-02-28 03:00:00'], freq='H', tz='Asia/Hong_Kong')
48                 x = pd.DatetimeIndex(['2024-02-28 01:00:00', '2024-02-28
49                 02:00:00', '2024-02-28 03:00:00'], freq='H', tz='America/
50                 Los_Angeles')
51
52                 self.assertSequenceEqual(pd.date_range('2024-02-28 01:00:00'
53                 , '2024-02-28 03:00:00', freq='H', tz='Asia/Hong_Kong').tolist(), y
54                 .tolist())
55                 self.assertSequenceEqual(pd.date_range('2024-02-28 01:00:00'
56                 , '2024-02-28 03:00:00', freq='H', tz='America/Los_Angeles').
57                 tolist(), x.tolist())
58
59                 self.assertNotEqual(x.tolist(), y.tolist())
60                 self.assertNotEqual(x.tz, y.tz)
61
62             #Test 'normalize' true, reset time to midnight
63             def testNormalize(self):
64                 x = pd.DatetimeIndex(['2024-02-28 00:00:00', '2024-02-29
65                 00:00:00', '2024-03-01 00:00:00'], dtype='datetime64[ns]', freq='
66                 D')
67                 self.assertSequenceEqual(pd.date_range('2024-02-28 23:00:00'
68                 , '2024-03-01 01:00:00', freq='D', normalize=True).tolist(), x.
69                 tolist())
70                 self.assertEqual(pd.date_range('2024-02-28 23:00:00', '
71                 2024-03-01 01:00:00', freq='D', normalize=True)[0].hour, 0)
72
73             #Test 'closed' to right and left will discard last and first
74             value
75             def testClosed(self):

```

```

58     r = pd.DatetimeIndex(['2024-02-29', '2024-03-01'], dtype='
datetime64[ns]', freq='D')
59     l = pd.DatetimeIndex(['2024-02-28', '2024-02-29'], dtype='
datetime64[ns]', freq='D')
60
61     self.assertSequenceEqual(pd.date_range('2024-02-28', '
2024-03-01', freq='D', closed='right').tolist(), r.tolist())
62     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

```

1  import unittest
2  import pandas as pd
3
4  class TestBdateRange(unittest.TestCase):
5
6      #Test input of range between two defined dates
7      def testStartEnd(self):
8          x = pd.DatetimeIndex(['2021-11-22', '2021-11-23', '
2021-11-24', '2021-11-25', '2021-11-26'], dtype='datetime64[ns]',
freq='B')
9          self.assertSequenceEqual(pd.bdate_range('2021-11-20', '
2021-11-28').tolist(), x.tolist())
10
11         #Test with leap year
12         y = pd.DatetimeIndex(['2024-02-28', '2024-02-29', '2024-03-1
'], dtype='datetime64[ns]', freq='D')
13         self.assertSequenceEqual(pd.bdate_range('2024-02-28', '
2024-03-1').tolist(), y.tolist())
14
15         #Test with 5 hour interval
16         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')
17         self.assertSequenceEqual(pd.bdate_range('2021-03-29 23:00:00
', '2021-03-30 09:00:00', freq='5H', normalize=True).tolist(), z.
tolist())
18
19         #Defined start date, no end date but with periode and frequence
20         def testStart(self):

```

```

21         #start weekend and periode of 3 days
22         x = pd.DatetimeIndex(['2021-11-1', '2021-11-2', '2021-11-3'
23                                ], dtype='datetime64[ns]', freq='B')
24         self.assertSequenceEqual(pd.bdate_range(start='2021-10-30',
25                                                    periods=3, freq='B').tolist(), x.tolist())
26
27         #Beginning of month frequency in 3 periods
28         y = pd.DatetimeIndex(['2021-11-01', '2021-12-01', '
29                                2022-01-01'], dtype='datetime64[ns]', freq='MS')
30         self.assertSequenceEqual(pd.bdate_range(start='2021-11-01',
31                                                    periods=3, freq='MS').tolist(), y.tolist())
32
33         #Defined end date, no start date but with periode and frequency
34         def testEnd(self):
35             #end midt week and get 4 days before
36             x = pd.DatetimeIndex(['2021-11-26', '2021-11-29', '
37                                    2021-11-30', '2021-12-1'], dtype='datetime64[ns]', freq='B')
38             self.assertSequenceEqual(pd.bdate_range(end='2021-12-1',
39                                                       periods=4).tolist(), x.tolist())
40
41         #get 3 periods with 3 business days interval
42         m = pd.DatetimeIndex(['2020-07-23', '2020-07-28', '
43                                2020-07-31'], dtype='datetime64[ns]', freq='3B')
44         self.assertSequenceEqual(pd.bdate_range(end='2020-07-31',
45                                                    freq='3B', periods=3).tolist(), m.tolist())
46
47         #Test with timezone
48         def testTimeZone(self):
49             y = pd.DatetimeIndex(['2024-02-28 00:00:00', '2024-02-29
50                                    00:00:00'], freq='B', tz='Asia/Hong_Kong')
51             x = pd.DatetimeIndex(['2024-02-28 00:00:00', '2024-02-29
52                                    00:00:00'], freq='B', tz='America/Los_Angeles')
53
54         #Test time zone is added
55         self.assertSequenceEqual(pd.bdate_range('2024-02-28 01:00:00
56            ', '2024-02-29 03:00:00', freq='B', tz='Asia/Hong_Kong').tolist(),
57            y.tolist())
58         self.assertSequenceEqual(pd.bdate_range('2024-02-28 01:00:00
59            ', '2024-02-29 03:00:00', freq='B', tz='America/Los_Angeles').
60            tolist(), x.tolist())
61
62         #Output with two different timezones are not equal
63         self.assertNotEqual(x.tolist(), y.tolist())
64         self.assertNotEqual(x.tz, y.tz)
65
66         #Test normalization of input will set the time to midnight
67         def testNormalize(self):
68             x = pd.DatetimeIndex(['2024-02-28 00:00:00', '2024-02-29
69                                    00:00:00', '2024-03-01 00:00:00'], dtype='datetime64[ns]', freq='
70                                    B')

```

```

53     self.assertSequenceEqual(pd.bdate_range('2024-02-28 23:00:00',
54         '2024-03-01 01:00:00', freq='B', normalize=True).tolist(), x.
55         tolist())
56     self.assertEqual(pd.bdate_range('2024-02-28 23:00:00', '
57         2024-03-01 01:00:00', freq='B', normalize=True)[0].hour, 0)
58
59     #Test closed from left and right will omit ends
60     def testClosed(self):
61
62         r = pd.DatetimeIndex(['2021-11-29', '2021-11-30', '2021-12-1
63             '], dtype='datetime64[ns]', freq='B')
64         l = pd.DatetimeIndex(['2021-11-26', '2021-11-29', '
65             2021-11-30'], dtype='datetime64[ns]', freq='B')
66
67         self.assertSequenceEqual(pd.bdate_range('2021-11-26', '
68             2021-12-01', freq='B', closed='right').tolist(), r.tolist())
69         self.assertSequenceEqual(pd.bdate_range('2021-11-26', '
70             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

```

1  import unittest
2  import pandas as pd
3
4  class TestPeriodRange(unittest.TestCase):
5
6      #Test input with defined start and end
7      def testStartEnd(self):
8          x = pd.PeriodIndex(['2021-11-1', '2021-11-2', '2021-11-3'],
9              dtype='period[D]')
10         self.assertSequenceEqual(pd.period_range('2021-11-1', '
11             2021-11-3').tolist(), x.tolist())
12
13         #test leap year
14         y = pd.PeriodIndex(['2024-02-28', '2024-02-29', '2024-03-1'
15             ], dtype='period[D]')
16         self.assertSequenceEqual(pd.period_range('2024-02-28', '
17             2024-03-1').tolist(), y.tolist())
18
19         #test day shift after hour pass 24

```



```

16     z = pd.PeriodIndex(['2024-02-28 23:00:00', '2024-02-29
17     00:00:00', '2024-02-29 01:00:00'], dtype='period[H]')
18     self.assertSequenceEqual(pd.period_range('2024-02-28
19     23:00:00', '2024-02-29 01:00:00', freq='H').tolist(), z.tolist())
20
21     #Test defined start with defined period
22     def testStart(self):
23         #start with 3 day period
24         x = pd.PeriodIndex(['2021-11-1', '2021-11-2', '2021-11-3'],
25         dtype='period[D]')
26         self.assertSequenceEqual(pd.period_range('2021-11-1', periods
27         =3).tolist(), x.tolist())
28         #start with 3 month period
29         y = pd.PeriodIndex(['2021-11', '2021-12', '2022-01'], dtype=
30         'period[M]')
31         self.assertSequenceEqual(pd.period_range('2021-11-01', freq=
32         'M', periods=3).tolist(), y.tolist())
33
34         #Test defined end with defined period
35         def testEnd(self):
36             #end with 3 day period
37             x = pd.PeriodIndex(['2021-11-29', '2021-11-30', '2021-12-1'
38             ], dtype='period[D]')
39             self.assertSequenceEqual(pd.period_range(end='2021-12-1',
40             periods=3).tolist(), x.tolist())
41             #end with 3 year period
42             y = pd.PeriodIndex(['2020', '2021', '2022'], dtype='period[Y
43             ]')
44             self.assertSequenceEqual(pd.period_range(end='2022-01-01',
45             freq='Y', periods=3).tolist(), y.tolist())
46
47         #Test defined start, end and frequency
48         def testPeriodInput(self):
49             start = pd.Period('2021-01', freq='M')
50             end = pd.Period('2021-04', freq='M')
51             x = pd.PeriodIndex(['2021-01', '2021-02', '2021-03', '
52             2021-04'], dtype='period[M]')
53             self.assertSequenceEqual(pd.period_range(start, end, freq='M')
54             .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

```
1 import unittest
2 import pandas as pd
3 from pandas._testing import assert_frame_equal
4
5
6 class TestDataframeTail(unittest.TestCase):
7     df = pd.DataFrame({'fruit': ['apple', 'pear', 'banana', 'melon',
8     'lemon', 'mango', 'lime']})
9     expected = pd.DataFrame({'fruit': ['apple', 'pear', 'banana', '
10     melon', 'lemon', 'mango', 'lime']})
11
12     #Test output values are the last five elements as default
13     def testReturnDefault(self):
14         expected = self.expected.drop([0, 1])
15         assert_frame_equal(self.df.tail(), expected)
16
17     #Test defined number of elements to return
18     def testNvalueReturn(self):
19         expected = self.expected.drop([0,1,2,3,4,5])
20         assert_frame_equal(self.df.tail(1), expected)
21
22         expected = self.expected.drop([0,1,2,3])
23         assert_frame_equal(self.df.tail(3), expected)
24
25         #all
26         expected = self.expected
27         assert_frame_equal(self.df.tail(7), expected)
28
29     #Test if returns empty list on zero as argument
30     def testZeroReturn(self):
31         expected = self.expected.drop([0,1,2,3,4,5,6])
32         assert_frame_equal(self.df.tail(0), expected)
33
34     #Test with negative argument
35     def testNegativeInput(self):
36         expected = self.expected.drop([0,1,2,3])
37         assert_frame_equal(self.df.tail(-4), expected)
38
39         expected = self.expected.drop([0,1,2,3,4,5,6])
40         assert_frame_equal(self.df.tail(-7), expected)
41
42     #Test with positive and negative input out of range
43     def testOutOffBoundInput(self):
44         expected = self.expected.drop([0,1,2,3,4,5,6])
45         assert_frame_equal(self.df.tail(-7000), expected)
```

```

45         expected = self.expected
46         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.

```

1  import unittest
2  import pandas as pd
3
4  from merge import merge_ordered
5  from pandas._testing import assert_frame_equal
6
7
8  class TestMergeOrderedWB(unittest.TestCase):
9      df1 = pd.DataFrame({"key": ["a", "b", "c"], "lvalue": [1, 2,
10         3]})
11      df2 = pd.DataFrame({"key": ["a", "b", "c"], "rvalue": [4, 5,
12         6]})
13
14      def setUp(self):
15          df1 = pd.DataFrame({"key": ["a", "b", "c"], "lvalue": [1, 2,
16             3]})
17          df2 = pd.DataFrame({"key": ["a", "b", "c"], "rvalue": [4, 5,
18             6]})
19
20      #322: Default merge returns right
21      def testNormalCase(self):
22          exp = pd.DataFrame(
23              {
24                  "key": ["a", "b", "c"],
25                  "lvalue": [1, 2, 3],
26                  "rvalue": [4, 5, 6]
27              }
28          )
29          assert_frame_equal(merge_ordered(self.df1, self.df2), exp)
30
31      #303-304: Throw value error when 'right-by' AND 'left-by' are
32      #defined in argument
33      def testLeftByRightBy(self):

```

```

29         self.assertRaises(ValueError, merge_ordered, self.df1, self.
df2, left_by='lvalue', right_by='lvalue' )
30
31     #305-3011: Merge order left-by
32     def testLeftBy(self):
33         exp = pd.DataFrame(
34             {
35                 "key": ["a", "b", "c", "a", "b", "c", "a", "b", "c"],
36                 "lvalue": [1, 1, 1, 2, 2, 2, 3, 3, 3],
37                 "rvalue": [4, 5, 6, 4, 5, 6, 4, 5, 6]
38             }
39         )
40         assert_frame_equal(merge_ordered(self.df1, self.df2, left_by
='lvalue'), exp)
41
42     #309-310: Throw key error if defined 'left-by' is not found
43     def testLeftByNotFound(self):
44         #left by value not found
45         self.assertRaises(KeyError, merge_ordered, self.df1, self.
df2, left_by='fake')
46
47     #312-318: Merge order right by
48     def testRightBy(self):
49         exp = pd.DataFrame(
50             {
51                 "key": ["a", "b", "c", "a", "b", "c", "a", "b", "c"],
52                 "lvalue": [1, 2, 3, 1, 2, 3, 1, 2, 3],
53                 "rvalue": [4, 4, 4, 5, 5, 5, 6, 6, 6]
54             }
55         )
56         assert_frame_equal(merge_ordered(self.df1, self.df2,
right_by='rvalue'), exp)
57
58     #316-317: Throw key error if defined 'left-by' is not found
59     def testRightByNotFound(self):
60         #right by value not found
61         self.assertRaises(KeyError, merge_ordered, self.df1, self.
df2, right_by='fake')
62
63 if __name__ == '__main__':
64     unittest.main()

```

```

289 | def _merger(x, y) -> DataFrame:
290 |     # perform the ordered merge operation
291 |     op = _OrderedMerge(
292 |         x,
293 |         y,
294 |         on=on,
295 |         left_on=left_on,
296 |         right_on=right_on,
297 |         suffixes=suffixes,
298 |         fill_method=fill_method,
299 |         how=how,
300 |     )
301 |     return op.get_result()
302 |
303 |     if left_by is not None and right_by is not None:
304 |         raise ValueError("Can only group either left or right frames")
305 |     elif left_by is not None:
306 |         if isinstance(left_by, str):
307 |             left_by = [left_by]
308 |         check = set(left_by).difference(left.columns)
309 |         if len(check) != 0:
310 |             raise KeyError(f"{check} not found in left columns")
311 |         result, _ = _groupby_and_merge(left_by, left, right, lambda x, y: _merger(x, y))
312 |     elif right_by is not None:
313 |         if isinstance(right_by, str):
314 |             right_by = [right_by]
315 |         check = set(right_by).difference(right.columns)
316 |         if len(check) != 0:
317 |             raise KeyError(f"{check} not found in right columns")
318 |         result, _ = _groupby_and_merge(
319 |             right_by, right, left, lambda x, y: _merger(y, x)
320 |         )
321 |     else:
322 |         result = _merger(left, right)
323 |     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

```

1 | import unittest
2 | import numpy as np
3 | import pandas as pd
4 | from numeric import to_numeric
5 |
6 | # scalar, list, tuple, 1-d array, or Series
7 | class TestPandasToNumeric(unittest.TestCase):
8 |     def setUp(self):

```

```

9         self.df = pd.io.stata.read_stata('https://stats.idre.ucla.
edu/stat/stata/dae/binary.dta')
10
11     # Testing different types of arguments such as Scalar / List.
12     def testArgTypes(self):
13         self.scalar = self.df['gpa']
14         self.scalar.squeeze()
15         self.list = list(self.df['gpa'])
16
17         # Scalar (numpy.float32 -> numpy.float32)
18         self.assertTrue(isinstance(type(to_numeric(self.scalar)[0]),
type(np.float32)))
19         # List (float32 -> numpy.float32)
20         self.assertTrue(isinstance(type(to_numeric(self.list)[0]),
type(np.float32)))
21
22     # Testing a Series where all the strings are numbers
23     def testFromString(self):
24         series = pd.Series(['7', '9', '13'])
25         result = to_numeric(series)
26
27         self.assertEqual(result[0], pd.Series([7, 9, 13])[0])
28         self.assertEqual(result[1], pd.Series([7, 9, 13])[1])
29         self.assertEqual(result[2], pd.Series([7, 9, 13])[2])
30
31     # Testing a combination of numbers and string-numbers
32     def testFromMixedNumbers(self):
33         series = pd.Series(['7', '9', 13])
34         result = to_numeric(series)
35
36         self.assertEqual(result[0], pd.Series([7, 9, 13])[0])
37         self.assertEqual(result[1], pd.Series([7, 9, 13])[1])
38         self.assertEqual(result[2], pd.Series([7, 9, 13])[2])
39
40     # Testing a Series where every element is an integer
41     def testFromInt(self):
42         series = pd.Series([7, 9, 13])
43         result = to_numeric(series)
44
45         self.assertEqual(result[0], pd.Series([7, 9, 13])[0])
46         self.assertEqual(result[1], pd.Series([7, 9, 13])[1])
47         self.assertEqual(result[2], pd.Series([7, 9, 13])[2])
48
49     # Should throw a ValueError in case of inputting strings that
are not in number format
50     def testWithWords(self):
51         self.assertRaises(ValueError, to_numeric, pd.Series(['7', '9
', 13, 'forteen']))

```

```

52
53 unittest.main(argv=[''], verbosity=2, exit=False)

```

3.7.2 White-box testing

```

1 import unittest
2 import numpy as np
3 import pandas as pd
4 from numeric import to_numeric
5
6 # scalar, list, tuple, 1-d array, or Series
7 class TestPandasToNumeric(unittest.TestCase):
8     def setUp(self):
9         self.df = pd.io.stata.read_stata('https://stats.idre.ucla.
10         edu/stat/stata/dae/binary.dta')
11         self.stringArray = pd.Series(['6', '10', '19', '25'])
12         self.intArray = pd.Series([6, 10, 19, 25])
13         self.floatArray = pd.Series([6., 10., 19., 25.])
14         self.dateArray = pd.DatetimeIndex(['2017-12-31
15         16:00:00-08:00', '2017-12-31 17:00:00-08:00',
16         '2017-12-31 18:00:00-08:00'],
17         dtype='datetime64[ns, Europe/Stockholm]', freq='H')
18
19     # Testing that downcasting a stringarray gives a ValueError
20     # depending on the arguments given for
21     # 'downcast' and 'errors'. Each combination is a different
22     # branch and the error is handled differently.
23     def testArgumentError(self):
24         self.assertRaises(ValueError, to_numeric, self.stringArray,
25         downcast="double")
26         self.assertRaises(ValueError, to_numeric, self.stringArray,
27         downcast="int")
28
29         self.assertRaises(ValueError, to_numeric, self.stringArray,
30         errors="ignor")
31         self.assertRaises(ValueError, to_numeric, self.stringArray,
32         errors="")
33
34         self.assertRaises(ValueError, to_numeric, pd.Series(['7', '9
35         ', 13, 'forteen']))
36
37         self.assertRaises(TypeError, to_numeric, self.df)
38
39     # Testing many different types of arguments
40     def testArgTypes(self):
41         # Testing Series
42         result = to_numeric(self.stringArray)

```

```

35         for i,v in self.intArray.iteritems():
36             self.assertEqual(v, result.loc[i])
37
38         # Testing Index
39         indexArray = pd.Index(['6', '10', '19', '25'])
40         result = to_numeric(indexArray)
41         for i,v in self.intArray.iteritems():
42             self.assertEqual(v, result[i])
43
44         # Testing for List
45         result = to_numeric(self.stringArray.tolist())
46         for i,v in self.intArray.iteritems():
47             self.assertEqual(v, result[i])
48
49         # Testing for scalar int & decimal
50         index_ = ['index_name']
51
52         scalarArray = pd.Series([248])
53         scalarArray.index = index_
54         scalar = scalarArray.item()
55         result = to_numeric(scalar)
56         self.assertEqual(248, result)
57
58         scalarArrayDecimal = pd.Series([float(24.8)])
59         scalarArrayDecimal.index = index_
60         scalarDecimal = scalarArrayDecimal.item()
61         resultDecimal = to_numeric(scalarDecimal)
62         self.assertEqual(24.8, resultDecimal)
63
64         scalarArrayString = pd.Series(['24'])
65         scalarArrayString.index = index_
66         scalarString = scalarArrayString.item()
67         resultString = to_numeric(scalarString)
68         self.assertEqual(24, resultString)
69
70         scalarPDArray = pd.array([10.10], dtype='float64')
71         scalarString = scalarPDArray[0]
72         resultString = to_numeric(scalarString)
73         self.assertEqual(10.10, resultString)
74
75         # Testing that downcast works as intended for every downcast
76         def testDowncast(self):
77             integer = to_numeric(self.stringArray, downcast="integer")
78             signed = to_numeric(self.stringArray, downcast="signed")
79             unsigned = to_numeric(self.stringArray, downcast="unsigned")
80             float = to_numeric(self.stringArray, downcast="float")
81
82             self.assertTrue(isinstance(integer[0], type(np.int8(0))))

```



```

83         self.assertTrue(isinstance(signed[0], type(np.int8(0))))
84         self.assertTrue(isinstance(unsigned[0], type(np.uint8(0))))
85         self.assertTrue(isinstance(float[0], type(np.float32(0))))
86
87     # Testing that masking branch works
88     def testMask(self):
89         self.assertEqual(self.intArray[0], to_numeric([6, 10, 19,
90 25])[0])
91         self.assertEqual(self.floatArray[0], to_numeric(self.
floatArray)[0])
92         self.assertEqual(1514764800000000000, to_numeric(self.
dateArray)[0])
93
94     # Testing that the function can handle Numpy arrays
95     def testNumpy(self):
96         numpyArray = np.array([7, 9, 13])
97         self.assertEqual(7, to_numeric(numpyArray)[0])
98
99     # Testing for numeric arrays
100    def testNumericArray(self):
101        integerArray = pd.array([1, 6, 10], dtype='Int32')
102        result = to_numeric(integerArray)
103        self.assertEqual(1, result[0])
104        self.assertEqual(6, result[1])
105        self.assertEqual(10, result[2])
106
107        floatArray = pd.array([1.1, 6.6, 10.10], dtype='float64')
108        result = to_numeric(floatArray)
109        self.assertEqual(1.1, result[0])
110        self.assertEqual(6.6, result[1])
111        self.assertEqual(10.10, result[2])
112    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

```
1 import unittest
2 import pandas as pd
3 import numpy as np
4
5 class TestPandasPivot(unittest.TestCase):
6     def setUp(self):
7         self.df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two',
8             'two', 'two'],
9                                'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
10                                'baz': [1, 2, 3, 4, 5, 6],
11                                'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
12
13     def testBasicPivot(self):
14         result = self.df.pivot(index='foo', columns='bar', values='
15         baz')
16         # Pivot should look like this:
17         # bar  A    B    C
18         # foo
19         # one  1    2    3
20         # two  4    5    6
21
22         # Check that numbers have been pivoted correctly by
23         # retrieving individual numbers from DF
24         self.assertEqual(result.iloc[0][0], 1)
25         self.assertEqual(result.iloc[0][1], 2)
26         self.assertEqual(result.iloc[0][2], 3)
27         self.assertEqual(result.iloc[1][0], 4)
28         self.assertEqual(result.iloc[1][1], 5)
29         self.assertEqual(result.iloc[1][2], 6)
30
31         # Check that columns are correct
32         self.assertTrue(len(result.columns.values) == 3)
33         self.assertEqual(result.columns.values[0], 'A')
34         self.assertEqual(result.columns.values[1], 'B')
35         self.assertEqual(result.columns.values[2], 'C')
```

```

34     def testNaNValues(self):
35         result = self.df.pivot(index='bar', columns='baz', values='
foo')
36         # Pivot should look like this:
37         # baz      1      2      3      4      5      6
38         # bar
39         # A      one  NaN  NaN  two  NaN  NaN
40         # B      NaN  one  NaN  NaN  two  NaN
41         # C      NaN  NaN  one  NaN  NaN  two
42
43         # Checking for first 3 values of A row has correct values &
NaN values
44         self.assertEqual(result.iloc[0].get(1), 'one')
45         self.assertTrue(pd.isna(result.iloc[0].get(2)))
46         self.assertTrue(pd.isna(result.iloc[0].get(3)))
47
48     def testMultipleValues(self):
49         result = self.df.pivot(index='foo', columns='bar', values=['
baz', 'zoo'])
50
51         # Pivot should contain two columns with their own identical
subcolumns
52         columnNames = list(result.columns.levels[0])
53         self.assertEqual(columnNames[0], 'baz')
54         self.assertEqual(columnNames[1], 'zoo')
55
56         subColumnNames = list(result.columns.levels[1])
57         self.assertEqual(subColumnNames[0], 'A')
58         self.assertEqual(subColumnNames[1], 'B')
59         self.assertEqual(subColumnNames[2], 'C')
60
61     def testIdenticalValuesError(self):
62         # When index contains duplicate it will throw ValueError
63
64         df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/
stat/stata/dae/binary.dta')
65         with self.assertRaises(ValueError):
66             pd.pivot(df, index='admit', columns='gpa', values='rank'
)
67
68
69
70 unittest.main(argv=[''], verbosity=2, exit=False)

```

3.8.2 White-box testing

```

1 import unittest
2 import pandas as pd

```

```

3 from pivot import pivot
4 import numpy as np
5
6 class TestPandasPivot(unittest.TestCase):
7     def setUp(self):
8         self.df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two',
9                                         'two', 'two'],
10                                'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
11                                'baz': [1, 2, 3, 4, 5, 6],
12                                'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
13
14     def testBasicPivot(self):
15         result = pivot(self.df, index='foo', columns='bar', values='
16         baz')
17         # Pivot should look like this:
18         # bar  A    B    C
19         # foo
20         # one  1    2    3
21         # two  4    5    6
22
23         # Check that numbers have been pivoted correctly by
24         # retrieving individual numbers from DF
25         self.assertEqual(result.iloc[0][0], 1)
26         self.assertEqual(result.iloc[0][1], 2)
27         self.assertEqual(result.iloc[0][2], 3)
28         self.assertEqual(result.iloc[1][0], 4)
29         self.assertEqual(result.iloc[1][1], 5)
30         self.assertEqual(result.iloc[1][2], 6)
31
32         # Check that columns are correct
33         self.assertTrue(len(result.columns.values) == 3)
34         self.assertEqual(result.columns.values[0], 'A')
35         self.assertEqual(result.columns.values[1], 'B')
36         self.assertEqual(result.columns.values[2], 'C')
37
38     def testNaNValues(self):
39         result = pivot(self.df, index='bar', columns='baz', values='
40         foo')
41         # Pivot should look like this:
42         # baz    1    2    3    4    5    6
43         # bar
44         # A      one  NaN  NaN  two  NaN  NaN
45         # B      NaN  one  NaN  NaN  two  NaN
46         # C      NaN  NaN  one  NaN  NaN  two
47
48         # Checking for first 3 values of A row has correct values &
49         # NaN values
50         self.assertEqual(result.iloc[0].get(1), 'one')

```

```

46         self.assertTrue(pd.isna(result.iloc[0].get(2)))
47         self.assertTrue(pd.isna(result.iloc[0].get(3)))
48
49     def testMultipleValues(self):
50         result = pivot(self.df, index='foo', columns='bar', values=[
51             'baz', 'zoo'])
52
53         # Pivot should contain two columns with their own identical
54         subcolumns
55         columnNames = list(result.columns.levels[0])
56         self.assertEqual(columnNames[0], 'baz')
57         self.assertEqual(columnNames[1], 'zoo')
58
59         subColumnNames = list(result.columns.levels[1])
60         self.assertEqual(subColumnNames[0], 'A')
61         self.assertEqual(subColumnNames[1], 'B')
62         self.assertEqual(subColumnNames[2], 'C')
63
64     def testIdenticalValuesError(self):
65         # When index contains duplicate it will throw ValueError
66
67         df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/
68             stat/stata/dae/binary.dta')
69         with self.assertRaises(ValueError):
70             pivot(df, index='admit', columns='gpa', values='rank')
71
72     def testMissingColumnArg(self):
73         with self.assertRaises(TypeError):
74             pivot(self.df, index='foo', values='bar')
75
76     def testNullArguments(self):
77         # Testing with different missing arguments, allowing Pandas
78         # to handle these different situations. The following 3 cases are
79         # individual
80         # pieces of code that will be tested
81
82         # Missing values: Use all remaining values
83         pivotOne = pivot(self.df, index='foo', columns='bar')
84
85         # Missing values & index: use remaining values AND create
86         # automatic index 1..n
87         pivotTwo = pivot(self.df, columns='bar')
88
89         # Missing Index: Use all values but create automatic index
90         # 1..n
91         pivotThree = pivot(self.df, columns='bar', values='bar')
92
93         # Pivot no values

```

```

87     # Pivot should contain two columns with their own identical
    subcolumns
88     self.assertEqual(pivotOne.columns.levels[0][0], 'baz')
89     self.assertEqual(pivotOne.columns.levels[0][1], 'zoo')
90
91     # Pivot no values + index
92     self.assertEqual(pivotTwo.columns.levels[0][0], 'foo')
93     self.assertEqual(pivotTwo.columns.levels[0][1], 'baz')
94
95     self.assertEqual(pivotTwo.index[0], 0)
96     self.assertEqual(pivotTwo.index[1], 1)
97     self.assertEqual(pivotTwo.index[2], 2)
98     self.assertEqual(pivotTwo.index[3], 3)
99     self.assertEqual(pivotTwo.index[4], 4)
100
101     # Pivot no index
102     self.assertEqual(pivotThree.index[0], 0)
103     self.assertEqual(pivotThree.index[1], 1)
104     self.assertEqual(pivotThree.index[2], 2)
105     self.assertEqual(pivotThree.index[3], 3)
106     self.assertEqual(pivotThree.index[4], 4)
107
108
109 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

```

1 import unittest
2 import pandas as pd
3 import numpy as np
4
5
6 class TestPandasConcat(unittest.TestCase):
7     def setUp(self):
8         self.df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/binary.dta')

```

```

9
10     # Includes start index, excludes end index
11     self.df1 = self.df.iloc[0:3]
12     self.df2 = self.df.iloc[3:6]
13
14     # Testing whether the amount of rows is correct
15     def test1(self):
16         expected = 6
17         df_concat = pd.concat([self.df1, self.df2])
18
19         self.assertEqual(len(df_concat.index), expected)
20
21     # Testing whether defining the axis on column instead of row
22     # functions as intended
23     # Concattennated dataframe should be put input as 4 new columns
24     def test2(self):
25         expected = ['admit', 'gre', 'gpa', 'rank', 'admit', 'gre', 'gpa', 'rank']
26         df_concat = pd.concat([self.df1, self.df2], axis=1)
27         self.assertEqual(list(df_concat.columns), expected)
28         self.assertEqual(len(df_concat.index), 6)
29
30     # Test that ignoring the index will re-index from 0-n whilst
31     # true will keep original index.
32     def test3(self):
33         df_concat_true = pd.concat([self.df2, self.df1],
34                                     ignore_index=True)
35         df_concat_false = pd.concat([self.df2, self.df1],
36                                     ignore_index=False)
37         # Ignore Index (true):          [0, 1, 2, 3, 4, 5]
38         # Don't ignore index (false):    [3, 4, 5, 0, 1, 2]
39
40         self.assertFalse(list(df_concat_true.index) == list(
41             df_concat_false.index))
42         self.assertEqual(list(df_concat_true.index), [0, 1, 2, 3, 4,
43             5])
44
45 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

```
1 import unittest
2 import pandas as pd
3 import numpy as np
4 from numpy.core.numeric import NaN
5 from pandas._libs.missing import NA
6
7 class TestPandasIsNull(unittest.TestCase):
8     def setUp(self):
9         self.df = pd.io.stata.read_stata('https://stats.idre.ucla.
10         edu/stat/stata/dae/binary.dta')
11
12     def testShouldDetectNullValueTypes(self):
13         numericArray = pd.Series([7, 9, 13, NaN, NA])
14         objectArray = pd.Series([None, NaN, {}])
15
16         self.assertFalse(pd.isnull(numericArray[0]))
17         self.assertFalse(pd.isnull(numericArray[1]))
18         self.assertFalse(pd.isnull(numericArray[2]))
19         self.assertTrue(pd.isnull(numericArray[3]))
20         self.assertTrue(pd.isnull(numericArray[4]))
21
22         self.assertTrue(pd.isnull(objectArray[0]))
23         self.assertTrue(pd.isnull(objectArray[1]))
24         self.assertFalse(pd.isnull(objectArray[2]))
25
26     def testShouldDetectNullValuesBoolarray(self):
27         series = pd.Series([7, 9, 13, NaN, NA])
28         expected_result = pd.Series([False, False, False, True, True])
29
30     for i in range(5):
31         self.assertEqual(pd.isnull(series[i]), expected_result[i])
32
33 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

```
1 import unittest
2 import pandas as pd
```



```

3 import numpy as np
4
5 class TestPandasMerge(unittest.TestCase):
6     def setUp(self):
7         self.df = pd.io.stata.read_stata('https://stats.idre.ucla.
            edu/stat/stata/dae/binary.dta')
8
9         # Includes start index, excludes end index
10        self.df1 = self.df.iloc[0:3]
11        self.df2 = self.df.iloc[3:6]
12
13    def testBasicMerge(self):
14        # Merge the two arrays on 'admit' column. This column is
            shared in output
15        result = pd.merge(left=self.df1, right=self.df2, left_on='
            admit', right_on='admit')
16        self.assertTrue(len(result.columns) == 7)
17
18    def testReturnsDataframe(self):
19        result = pd.merge(left=self.df1, right=self.df2, left_on='
            admit', right_on='admit')
20        self.assertTrue(type(result) == pd.DataFrame)
21
22 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.

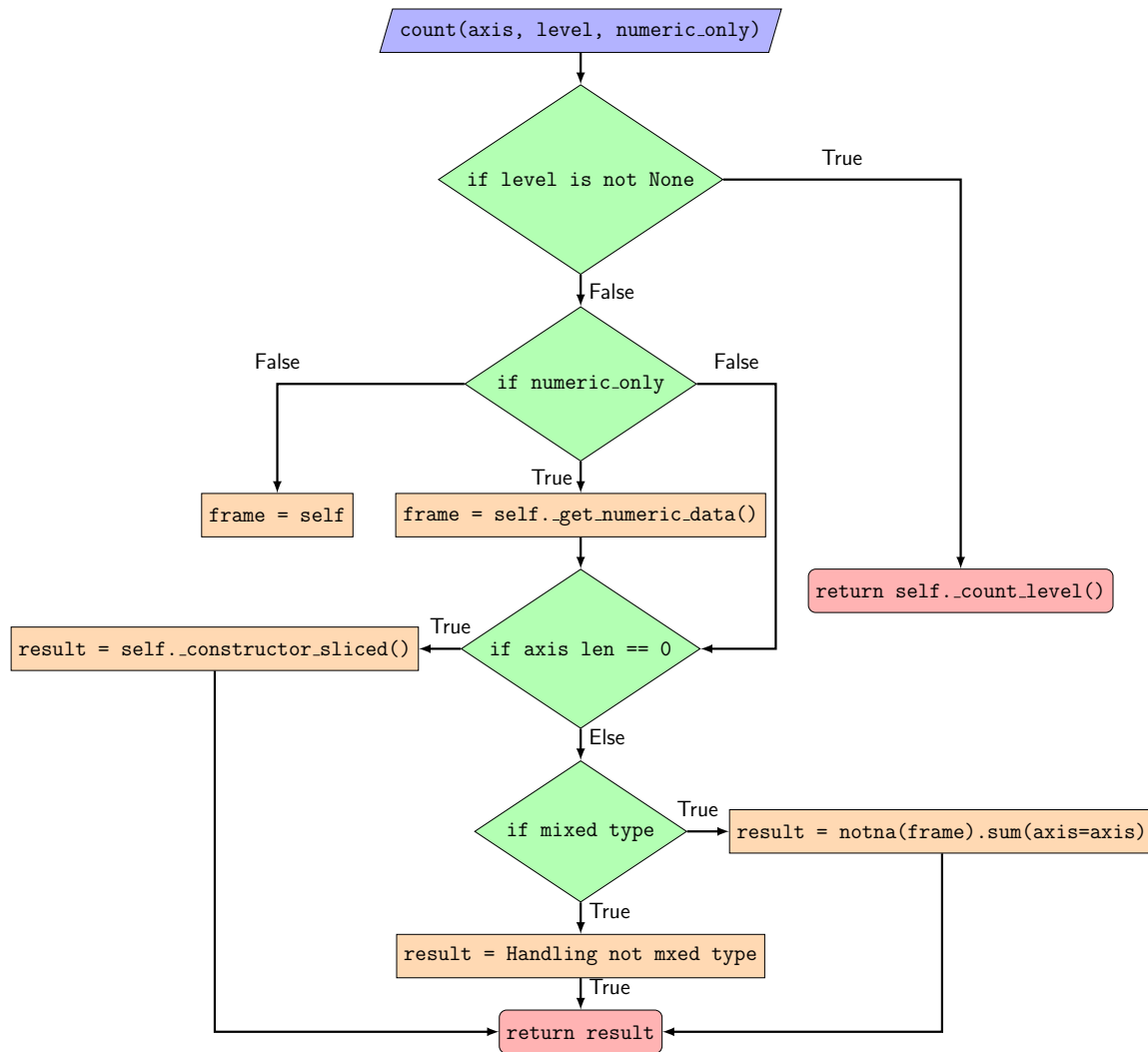
```

9695 | axis = self._get_axis_number(axis)
9696 | if level is not None:
9697 |     warnings.warn(
9698 |         "Using the level keyword in DataFrame and Series aggregations is "
9699 |         "deprecated and will be removed in a future version. Use groupby "
9700 |         "instead. df.count(level=1) should use df.groupby(level=1).count().",
9701 |         FutureWarning,
9702 |         stacklevel=2,
9703 |     )
9704 |     return self._count_level(level, axis=axis, numeric_only=numeric_only)
9705 |
9706 | if numeric_only:
9707 |     frame = self._get_numeric_data()
9708 | else:
9709 |     frame = self
9710 |
9711 | # GH #423
9712 | if len(frame._get_axis(axis)) == 0:
9713 |     result = self._constructor_sliced(0, index=frame._get_agg_axis(axis))
9714 | else:
9715 |     if frame._is_mixed_type or frame._mgr.any_extension_types:
9716 |         # the or any_extension_types is really only hit for single-
9717 |         # column frames with an extension array
9718 |         result = notna(frame).sum(axis=axis)
9719 |     else:
9720 |         # GH13407
9721 |         series_counts = notna(frame).sum(axis=axis)
9722 |         counts = series_counts.values
9723 |         result = self._constructor_sliced(
9724 |             counts, index=frame._get_agg_axis(axis)
9725 |         )
9726 |
9727 | return result.astype("int64")

```

Figure 3: Code coverage

3.12.3 Control flow graph



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.

```
1578 | v = self.squeeze()
1579 | if isinstance(v, (bool, np.bool_)):
1580 |     return bool(v)
1581 | elif is_scalar(v):
1582 |     raise ValueError(
1583 |         "bool cannot act on a non-boolean single element "
1584 |         f"{type(self).__name__}"
1585 |     )
1586 |
1587 | 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

```
355     res = isna(obj)
356     if is_scalar(res):
357         return not res
358     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

```
853     if arg is None:
854         return None
855
856     if origin != "unix":
857         arg = _adjust_to_origin(arg, origin, unit)
858
859     tz = "utc" if utc else None
860     convert_listlike = partial(
861         _convert_listlike_datetimes,
862         tz=tz,
863         unit=unit,
864         dayfirst=dayfirst,
865         yearfirst=yearfirst,
866         errors=errors,
867         exact=exact,
868         infer_datetime_format=infer_datetime_format,
869     )
870
871     result: Timestamp | NaTType | Series | Index
```

Figure 9: Code coverage1

```

872
873     if isinstance(arg, Timestamp):
874         result = arg
875         if tz is not None:
876             if arg.tz is not None:
877                 # error: Too many arguments for "tz_convert" of "NaTType"
878                 result = result.tz_convert(tz) # type: ignore[call-arg]
879             else:
880                 # error: Too many arguments for "tz_localize" of "NaTType"
881                 result = result.tz_localize(tz) # type: ignore[call-arg]
882     elif isinstance(arg, ABCSeries):
883         cache_array = _maybe_cache(arg, format, cache, convert_listlike)
884         if not cache_array.empty:
885             result = arg.map(cache_array)
886         else:
887             values = convert_listlike(arg._values, format)
888             result = arg._constructor(values, index=arg.index, name=arg.name)
889     elif isinstance(arg, (ABCDDataFrame, abc.MutableMapping)):
890         result = _assemble_from_unit_mappings(arg, errors, tz)
891     elif isinstance(arg, Index):

```

Figure 10: Code coverage2

```

892         cache_array = _maybe_cache(arg, format, cache, convert_listlike)
893         if not cache_array.empty:
894             result = _convert_and_box_cache(arg, cache_array, name=arg.name)
895         else:
896             result = convert_listlike(arg, format, name=arg.name)
897     elif is_list_like(arg):
898         try:
899             cache_array = _maybe_cache(arg, format, cache, convert_listlike)
900         except OutOfBoundsDatetime:
901             # caching attempts to create a DatetimeIndex, which may raise
902             # an OOB. If that's the desired behavior, then just reraise...
903             if errors == "raise":
904                 raise
905             # ... otherwise, continue without the cache.
906             from pandas import Series
907
908             cache_array = Series([], dtype=object) # just an empty array
909         if not cache_array.empty:
910             result = _convert_and_box_cache(arg, cache_array)

```

Figure 11: Code coverage3

```

911         else:
912             result = convert_listlike(arg, format)
913     else:
914         result = convert_listlike(np.array([arg]), format)[0]
915
916     # error: Incompatible return value type (got "Union[Timestamp, NaTType,
917     # Series, Index]", expected "Union[DatetimeIndex, Series, float, str,
918     # NaTType, None]")
919     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

```
421     values = _ensure_arraylike(values)
422
423     if is_extension_array_dtype(values):
424         # Dispatch to extension dtype's unique.
425         return values.unique()
426
427     original = values
428     htable, values = _get_hashtable_algo(values)
429
430     table = htable(len(values))
431     uniques = table.unique(values)
432     uniques = _reconstruct_data(uniques, original.dtype, original)
433     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

```
281     if not hasattr(vals, "dtype"):
282         raise TypeError("must pass a ndarray-like")
283     dtype = vals.dtype
284
285     # For categoricals, we hash the categories, then remap the codes to the
286     # hash values. (This check is above the complex check so that we don't ask
287     # numpy if categorical is a subtype of complex, as it will choke).
288     if is_categorical_dtype(dtype):
289         vals = cast("Categorical", vals)
290         return _hash_categorical(vals, encoding, hash_key)
291     elif not isinstance(vals, np.ndarray):
292         # i.e. ExtensionArray
293         vals, _ = vals._values_for_factorize()
294
295     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

```
1 import unittest
2 import pandas as pd
3 import numpy as np
4
5
6 class TestPandasConcat(unittest.TestCase):
7     def setUp(self):
8         self.df = pd.io.stata.read_stata('https://stats.idre.ucla.
9         edu/stat/stata/dae/binary.dta')
10
11         # Includes start index, excludes end index
12         self.df1 = self.df.iloc[0:3]
13         self.df2 = self.df.iloc[3:6]
14
15         # Testing whether the amount of rows is correct
16         def test1(self):
17             expected = 6
18             df_concat = pd.concat([self.df1, self.df2])
19
20             self.assertEqual(len(df_concat.index), expected)
21
22         # Testing whether defining the axis on column instead of row
23         # functions as intended
24         # Concattented dataframe should be put input as 4 new columns
25         def test2(self):
26             expected = ['admit', 'gre', 'gpa', 'rank', 'admit', 'gre', '
27             gpa', 'rank']
28             df_concat = pd.concat([self.df1, self.df2], axis=1)
29             self.assertEqual(list(df_concat.columns), expected)
30             self.assertEqual(len(df_concat.index), 6)
31
32         # Test that ignoring the index will re-index from 0-n whilst
33         # true will keep original index.
34         def test3(self):
35             df_concat_true = pd.concat([self.df2, self.df1],
36             ignore_index=True)
37             df_concat_false = pd.concat([self.df2, self.df1],
38             ignore_index=False)
39
40             # Ignore Index (true):          [0, 1, 2, 3, 4, 5]
41             # Don't ignore index (false):    [3, 4, 5, 0, 1, 2]
42
43             self.assertFalse(list(df_concat_true.index) == list(
44             df_concat_false.index))
```

```

37         self.assertEqual(list(df_concat_true.index), [0, 1, 2, 3, 4,
38         5])
39 unittest.main(argv=[''], verbosity=2, exit=False)

```

4.2 pandas.to_numeric

```

1  import unittest
2  import numpy as np
3  import pandas as pd
4  from numeric import to_numeric
5
6  # scalar, list, tuple, 1-d array, or Series
7  class TestPandasToNumeric(unittest.TestCase):
8      def setUp(self):
9          self.df = pd.io.stata.read_stata('https://stats.idre.ucla.
10          edu/stat/stata/dae/binary.dta')
11
12      # Testing different types of arguments such as Scalar / List.
13      def testArgTypes(self):
14          self.scalar = self.df['gpa']
15          self.scalar.squeeze()
16          self.list = list(self.df['gpa'])
17
18          # Scalar (numpy.float32 -> numpy.float32)
19          self.assertTrue(isinstance(to_numeric(self.scalar)[0]),
20          type(np.float32))
21          # List (float32 -> numpy.float32)
22          self.assertTrue(isinstance(to_numeric(self.list)[0]),
23          type(np.float32))
24
25      # Testing a Series where all the strings are numbers
26      def testFromString(self):
27          series = pd.Series(['7', '9', '13'])
28          result = to_numeric(series)
29
30          self.assertEqual(result[0], pd.Series([7, 9, 13])[0])
31          self.assertEqual(result[1], pd.Series([7, 9, 13])[1])
32          self.assertEqual(result[2], pd.Series([7, 9, 13])[2])
33
34      # Testing a combination of numbers and string-numbers
35      def testFromMixedNumbers(self):
36          series = pd.Series(['7', '9', 13])
37          result = to_numeric(series)
38
39          self.assertEqual(result[0], pd.Series([7, 9, 13])[0])
40          self.assertEqual(result[1], pd.Series([7, 9, 13])[1])
41          self.assertEqual(result[2], pd.Series([7, 9, 13])[2])

```

```

39
40     # Testing a Series where every element is an integer
41     def testFromInt(self):
42         series = pd.Series([7, 9, 13])
43         result = to_numeric(series)
44
45         self.assertEqual(result[0], pd.Series([7, 9, 13])[0])
46         self.assertEqual(result[1], pd.Series([7, 9, 13])[1])
47         self.assertEqual(result[2], pd.Series([7, 9, 13])[2])
48
49     # Should throw a ValueError in case of inputting strings that
50     # are not in number format
51     def testWithWords(self):
52         self.assertRaises(ValueError, to_numeric, pd.Series(['7', '9', 13, 'forteen']))
53
54 unittest.main(argv=[''], verbosity=2, exit=False)

```

4.3 pandas.is_null

```

1  import unittest
2  import pandas as pd
3  import numpy as np
4  from numpy.core.numeric import NaN
5  from pandas._libs.missing import NA
6
7  class TestPandasIsNull(unittest.TestCase):
8      def setUp(self):
9          self.df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/binary.dta')
10
11      def testShouldDetectNullValueTypes(self):
12          numericArray = pd.Series([7, 9, 13, NaN, NA])
13          objectArray = pd.Series([None, NaN, {}])
14
15          self.assertFalse(pd.isnull(numericArray[0]))
16          self.assertFalse(pd.isnull(numericArray[1]))
17          self.assertFalse(pd.isnull(numericArray[2]))
18          self.assertTrue(pd.isnull(numericArray[3]))
19          self.assertTrue(pd.isnull(numericArray[4]))
20
21          self.assertTrue(pd.isnull(objectArray[0]))
22          self.assertTrue(pd.isnull(objectArray[1]))
23          self.assertFalse(pd.isnull(objectArray[2]))
24
25      def testShouldDetectNullValuesBoolarray(self):
26          series = pd.Series([7, 9, 13, NaN, NA])
27          expected_result = pd.Series([False, False, False, True, True])

```

```

    ])
28
29     for i in range(5):
30         self.assertEqual(pd.isnull(series[i]), expected_result[i])
31
32 unittest.main(argv=[''], verbosity=2, exit=False)

```

4.4 pandas.merge

```

1 import unittest
2 import pandas as pd
3 import numpy as np
4
5 class TestPandasMerge(unittest.TestCase):
6     def setUp(self):
7         self.df = pd.io.stata.read_stata('https://stats.idre.ucla.
            edu/stat/stata/dae/binary.dta')
8
9         # Includes start index, excludes end index
10        self.df1 = self.df.iloc[0:3]
11        self.df2 = self.df.iloc[3:6]
12
13    def testBasicMerge(self):
14        # Merge the two arrays on 'admit' column. This column is
            shared in output
15        result = pd.merge(left=self.df1, right=self.df2, left_on='
            admit', right_on='admit')
16        self.assertTrue(len(result.columns) == 7)
17
18    def testReturnsDataframe(self):
19        result = pd.merge(left=self.df1, right=self.df2, left_on='
            admit', right_on='admit')
20        self.assertTrue(type(result) == pd.DataFrame)
21
22 unittest.main(argv=[''], verbosity=2, exit=False)

```

4.5 pandas.pivot

```

1 import unittest
2 import pandas as pd
3 import numpy as np
4
5 class TestPandasPivot(unittest.TestCase):
6     def setUp(self):
7         self.df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two',
            'two', 'two'],
8                                'bar': ['A', 'B', 'C', 'A', 'B', 'C'],

```

```

9             'baz': [1, 2, 3, 4, 5, 6],
10             'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
11
12     def testBasicPivot(self):
13         result = self.df.pivot(index='foo', columns='bar', values='
baz')
14         # Pivot should look like this:
15         # bar  A    B    C
16         # foo
17         # one  1    2    3
18         # two  4    5    6
19
20         # Check that numbers have been pivoted correctly by
retrieving individual numbers from DF
21         self.assertEqual(result.iloc[0][0], 1)
22         self.assertEqual(result.iloc[0][1], 2)
23         self.assertEqual(result.iloc[0][2], 3)
24         self.assertEqual(result.iloc[1][0], 4)
25         self.assertEqual(result.iloc[1][1], 5)
26         self.assertEqual(result.iloc[1][2], 6)
27
28         # Check that columns are correct
29         self.assertTrue(len(result.columns.values) == 3)
30         self.assertEqual(result.columns.values[0], 'A')
31         self.assertEqual(result.columns.values[1], 'B')
32         self.assertEqual(result.columns.values[2], 'C')
33
34     def testNaNValues(self):
35         result = self.df.pivot(index='bar', columns='baz', values='
foo')
36         # Pivot should look like this:
37         # baz    1    2    3    4    5    6
38         # bar
39         # A      one  NaN  NaN  two  NaN  NaN
40         # B      NaN  one  NaN  NaN  two  NaN
41         # C      NaN  NaN  one  NaN  NaN  two
42
43         # Checking for first 3 values of A row has correct values &
NaN values
44         self.assertEqual(result.iloc[0].get(1), 'one')
45         self.assertTrue(pd.isna(result.iloc[0].get(2)))
46         self.assertTrue(pd.isna(result.iloc[0].get(3)))
47
48     def testMultipleValues(self):
49         result = self.df.pivot(index='foo', columns='bar', values=['
baz', 'zoo'])
50
51         # Pivot should contain two columns with their own identical

```

```

52     subcolumns
53         columnNames = list(result.columns.levels[0])
54         self.assertEqual(columnNames[0], 'baz')
55         self.assertEqual(columnNames[1], 'zoo')
56
57         subColumnNames = list(result.columns.levels[1])
58         self.assertEqual(subColumnNames[0], 'A')
59         self.assertEqual(subColumnNames[1], 'B')
60         self.assertEqual(subColumnNames[2], 'C')
61
62     def testIdenticalValuesError(self):
63         # When index contains duplicate it will throw ValueError
64
65         df = pd.io.stata.read_stata('https://stats.idre.ucla.edu/
66         stat/stata/dae/binary.dta')
67         with self.assertRaises(ValueError):
68             pd.pivot(df, index='admit', columns='gpa', values='rank'
69         )
70
71 unittest.main(argv=[''], verbosity=2, exit=False)

```

4.6 pandas.date_range

```

1  import unittest
2  import pandas as pd
3  from pandas.core.indexes.datetimes import date_range as dr
4
5  class TestDateRange(unittest.TestCase):
6
7      #Test input of range between two defined dates
8      def testStartEnd(self):
9          x = pd.DatetimeIndex(['2021-11-1', '2021-11-2', '2021-11-3'
10         ], dtype='datetime64[ns]', freq='D')
11          self.assertEqual(dr('2021-11-1', '2021-11-3').tolist
12         (), x.tolist())
13
14      #Test right output on leap year
15      y = pd.DatetimeIndex(['2024-02-28', '2024-02-29', '2024-03-1
16         '], dtype='datetime64[ns]', freq='D')
17          self.assertEqual(dr('2024-02-28', '2024-03-1').tolist
18         (), y.tolist())
19
20      #Test hour frequency change date after midnight
21      z = pd.DatetimeIndex(['2024-02-28 23:00:00', '2024-02-29
22         00:00:00', '2024-02-29 01:00:00'], dtype='datetime64[ns]', freq='
23         H')

```

```

18         self.assertSequenceEqual(dr('2024-02-28 23:00:00', '
2024-02-29 01:00:00', freq='H').tolist(), z.tolist())
19
20     #Test defined start of interval and period
21     def testStart(self):
22         #Test 3 period with default frequency, day
23         x = pd.DatetimeIndex(['2021-11-1', '2021-11-2', '2021-11-3'
24 ], dtype='datetime64[ns]', freq='D')
25         self.assertSequenceEqual(pd.date_range('2021-11-1', periods
26 =3).tolist(), x.tolist())
27
28         #Test with month start frequency
29         y = pd.DatetimeIndex(['2021-11-01', '2021-12-01', '
30 2022-01-01'], dtype='datetime64[ns]', freq='MS')
31         self.assertSequenceEqual(pd.date_range(start='2021-10-06',
32 freq='MS', periods=3).tolist(), y.tolist())
33         #Test defined end of interval and period
34         def testEnd(self):
35             #Test 3 periods and default frequency, day
36             x = pd.DatetimeIndex(['2021-11-29', '2021-11-30', '2021-12-1
37 '], dtype='datetime64[ns]', freq='D')
38             self.assertSequenceEqual(pd.date_range(end='2021-12-1',
39 periods=3).tolist(), x.tolist())
40
41         #Test 3 periods with 3 month frequency
42         m = pd.DatetimeIndex(['2020-01-31', '2020-04-30', '
43 2020-07-31'], dtype='datetime64[ns]', freq='3M')
44         self.assertSequenceEqual(pd.date_range(end='2020-07-31',
45 freq='3M', periods=3).tolist(), m.tolist())
46
47         #Test with defined timezone
48         def testTimeZone(self):
49             y = pd.DatetimeIndex(['2024-02-28 01:00:00', '2024-02-28
50 02:00:00', '2024-02-28 03:00:00'], freq='H', tz='Asia/Hong_Kong')
51             x = pd.DatetimeIndex(['2024-02-28 01:00:00', '2024-02-28
52 02:00:00', '2024-02-28 03:00:00'], freq='H', tz='America/
53 Los_Angeles')
54
55             self.assertSequenceEqual(pd.date_range('2024-02-28 01:00:00'
56 , '2024-02-28 03:00:00', freq='H', tz='Asia/Hong_Kong').tolist(), y
57 .tolist())
58             self.assertSequenceEqual(pd.date_range('2024-02-28 01:00:00'
59 , '2024-02-28 03:00:00', freq='H', tz='America/Los_Angeles').
60 tolist(), x.tolist())
61
62             self.assertNotEqual(x.tolist(), y.tolist())
63             self.assertNotEqual(x.tz, y.tz)

```

```

50     #Test 'normalize' true, reset time to midnight
51     def testNormalize(self):
52         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')
53         self.assertSequenceEqual(pd.date_range('2024-02-28 23:00:00'
, '2024-03-01 01:00:00', freq='D', normalize=True).tolist(), x.
tolist())
54         self.assertEqual(pd.date_range('2024-02-28 23:00:00', '
2024-03-01 01:00:00', freq='D', normalize=True)[0].hour, 0)
55
56     #Test 'closed' to right and left will discard last and first
value
57     def testClosed(self):
58         r = pd.DatetimeIndex(['2024-02-29', '2024-03-01'], dtype='
datetime64[ns]', freq='D')
59         l = pd.DatetimeIndex(['2024-02-28', '2024-02-29'], dtype='
datetime64[ns]', freq='D')
60
61         self.assertSequenceEqual(pd.date_range('2024-02-28', '
2024-03-01', freq='D', closed='right').tolist(), r.tolist())
62         self.assertSequenceEqual(pd.date_range('2024-02-28', '
2024-03-01', freq='D', closed='left').tolist(), l.tolist())

```

4.7 pandas.bdate_range

```

1 import unittest
2 import pandas as pd
3
4 class TestBdateRange(unittest.TestCase):
5
6     #Test input of range between two defined dates
7     def testStartEnd(self):
8         x = pd.DatetimeIndex(['2021-11-22', '2021-11-23', '
2021-11-24', '2021-11-25', '2021-11-26'], dtype='datetime64[ns]',
freq='B')
9         self.assertSequenceEqual(pd.bdate_range('2021-11-20', '
2021-11-28').tolist(), x.tolist())
10
11     #Test with leap year
12     y = pd.DatetimeIndex(['2024-02-28', '2024-02-29', '2024-03-1
'], dtype='datetime64[ns]', freq='D')
13     self.assertSequenceEqual(pd.bdate_range('2024-02-28', '
2024-03-1').tolist(), y.tolist())
14
15     #Test with 5 hour interval
16     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')
17     self.assertSequenceEqual(pd.bdate_range('2021-03-29 23:00:00',
    '2021-03-30 09:00:00', freq='5H', normalize=True).tolist(), z.
    tolist())
18
19     #Defined start date, no end date but with periode and frequence
20     def testStart(self):
21         #start weekend and periode of 3 days
22         x = pd.DatetimeIndex(['2021-11-1', '2021-11-2', '2021-11-3'
    ], dtype='datetime64[ns]', freq='B')
23         self.assertSequenceEqual(pd.bdate_range(start='2021-10-30',
    periods=3, freq='B').tolist(), x.tolist())
24
25         #Beginning of month frequency in 3 periods
26         y = pd.DatetimeIndex(['2021-11-01', '2021-12-01', '
    2022-01-01'], dtype='datetime64[ns]', freq='MS')
27         self.assertSequenceEqual(pd.bdate_range(start='2021-11-01',
    periods=3, freq='MS').tolist(), y.tolist())
28
29     #Defined end date, no start date but with periode and frequence
30     def testEnd(self):
31         #end midt week and get 4 days before
32         x = pd.DatetimeIndex(['2021-11-26', '2021-11-29', '
    2021-11-30', '2021-12-1'], dtype='datetime64[ns]', freq='B')
33         self.assertSequenceEqual(pd.bdate_range(end='2021-12-1',
    periods=4).tolist(), x.tolist())
34
35         #get 3 periods with 3 business days interval
36         m = pd.DatetimeIndex(['2020-07-23', '2020-07-28', '
    2020-07-31'], dtype='datetime64[ns]', freq='3B')
37         self.assertSequenceEqual(pd.bdate_range(end='2020-07-31',
    freq='3B', periods=3).tolist(), m.tolist())
38     #Test with timezone
39     def testTimeZone(self):
40         y = pd.DatetimeIndex(['2024-02-28 00:00:00', '2024-02-29
    00:00:00'], freq='B', tz='Asia/Hong_Kong')
41         x = pd.DatetimeIndex(['2024-02-28 00:00:00', '2024-02-29
    00:00:00'], freq='B', tz='America/Los_Angeles')
42
43         #Test time zone is added
44         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())
45         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())
46         #Output with two different timezones are not equal
47         self.assertNotEqual(x.tolist(), y.tolist())

```

```

48         self.assertNotEqual(x.tz,y.tz)
49
50         #Test normalization of input will set the time to midnight
51         def testNormalize(self):
52             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')
53             self.assertSequenceEqual(pd.bdate_range('2024-02-28 23:00:00
', '2024-03-01 01:00:00', freq='B', normalize=True).tolist(), x.
tolist())
54             self.assertEqual(pd.bdate_range('2024-02-28 23:00:00', '
2024-03-01 01:00:00', freq='B', normalize=True)[0].hour, 0)
55
56         #Test closed from left and right will omit ends
57         def testClosed(self):
58
59             r = pd.DatetimeIndex(['2021-11-29', '2021-11-30', '2021-12-1
'], dtype='datetime64[ns]', freq='B')
60             l = pd.DatetimeIndex(['2021-11-26', '2021-11-29', '
2021-11-30'], dtype='datetime64[ns]', freq='B')
61
62             self.assertSequenceEqual(pd.bdate_range('2021-11-26', '
2021-12-01', freq='B', closed='right').tolist(), r.tolist())
63             self.assertSequenceEqual(pd.bdate_range('2021-11-26', '
2021-12-01', freq='B', closed='left').tolist(), l.tolist())

```

4.8 pandas.period_range

```

1 import unittest
2 import pandas as pd
3
4 class TestPeriodRange(unittest.TestCase):
5
6     #Test input with defined start and end
7     def testStartEnd(self):
8         x = pd.PeriodIndex(['2021-11-1', '2021-11-2', '2021-11-3'],
dtype='period[D]')
9         self.assertSequenceEqual(pd.period_range('2021-11-1', '
2021-11-3').tolist(), x.tolist())
10
11         #test leap year
12         y = pd.PeriodIndex(['2024-02-28', '2024-02-29', '2024-03-1'
], dtype='period[D]')
13         self.assertSequenceEqual(pd.period_range('2024-02-28', '
2024-03-1').tolist(), y.tolist())
14
15         #test day shift after hour pass 24
16         z = pd.PeriodIndex(['2024-02-28 23:00:00', '2024-02-29

```

```

17         '00:00:00', '2024-02-29 01:00:00'], dtype='period[H]')
18         self.assertSequenceEqual(pd.period_range('2024-02-28
19         23:00:00', '2024-02-29 01:00:00', freq='H').tolist(), z.tolist())
20
21     #Test defined start with defined period
22     def testStart(self):
23         #start with 3 day period
24         x = pd.PeriodIndex(['2021-11-1', '2021-11-2', '2021-11-3'],
25         dtype='period[D]')
26         self.assertSequenceEqual(pd.period_range('2021-11-1', periods
27         =3).tolist(), x.tolist())
28         #start with 3 month period
29         y = pd.PeriodIndex(['2021-11', '2021-12', '2022-01'], dtype=
30         'period[M]')
31         self.assertSequenceEqual(pd.period_range('2021-11-01', freq=
32         'M', periods=3).tolist(), y.tolist())
33
34     #Test defined end with defined period
35     def testEnd(self):
36         #end with 3 day period
37         x = pd.PeriodIndex(['2021-11-29', '2021-11-30', '2021-12-1'
38         ], dtype='period[D]')
39         self.assertSequenceEqual(pd.period_range(end='2021-12-1',
40         periods=3).tolist(), x.tolist())
41         #end with 3 year period
42         y = pd.PeriodIndex(['2020', '2021', '2022'], dtype='period[Y
43         ]')
44         self.assertSequenceEqual(pd.period_range(end='2022-01-01',
45         freq='Y', periods=3).tolist(), y.tolist())
46
47     #Test defined start, end and frequency
48     def testPeriodInput(self):
49         start = pd.Period('2021-01', freq='M')
50         end = pd.Period('2021-04', freq='M')
51         x = pd.PeriodIndex(['2021-01', '2021-02', '2021-03', '
52         2021-04'], dtype='period[M]')
53         self.assertSequenceEqual(pd.period_range(start, end, freq='M')
54         .tolist(), x.tolist())

```

4.9 pandas.to_timedelta

```

1 import unittest
2 import pandas as pd
3 import numpy as np
4
5 class TestToTimeDelta(unittest.TestCase):
6
7     #Test if all input is for the different units is converted

```

```

correctly to value.
8     #Expected values are in nano seconds calculated and tested on
google
9     def testUnitValues(self):
10         self.assertEqual(pd.to_timedelta(0).value, 0) #zero input
11         self.assertEqual(pd.to_timedelta(1).value, 1) #default nano
sec
12         self.assertEqual(pd.to_timedelta(1, unit='us').value, 1000)
13         self.assertEqual(pd.to_timedelta(1, unit='ms').value, 10 **
6)
14         self.assertEqual(pd.to_timedelta(1, unit='s').value, 10 **
9)
15         self.assertEqual(pd.to_timedelta(1, unit='m').value, 60 * 10
** 9)
16         self.assertEqual(pd.to_timedelta(1, unit='h').value, 3600 *
10 ** 9)
17         self.assertEqual(pd.to_timedelta(1, unit='d').value, 8.64 *
10 ** 13)
18         self.assertEqual(pd.to_timedelta(1, unit='w').value, 6.04800
* 10 ** 14)
19
20     #Test if value error is thrown on deprecated unit input
21     def testErrorOnDeprecatedInput(self):
22         self.assertRaises(ValueError, pd.to_timedelta, 1, unit='y')
23         self.assertRaises(ValueError, pd.to_timedelta, 1, unit='M')
24
25     #Test if different kind of input gives the expected output
26     #Different kind of string input is converted using the expected
unit
27     def testInputValue(self):
28         self.assertEqual(pd.to_timedelta(0), pd.Timedelta('0 days
00:00:00.0'))
29         self.assertEqual(pd.to_timedelta(1000), pd.Timedelta('0 days
00:00:00.000001'))
30         self.assertEqual(pd.to_timedelta('1sec'), pd.Timedelta('0
days 00:00:01.000000'))
31         self.assertEqual(pd.to_timedelta('13:00:00'), pd.Timedelta('
0 days 13:00:00.000000'))
32         self.assertEqual(pd.to_timedelta('8 days 11:23:33.123456789'
), pd.Timedelta('8 days 11:23:33.123456789'))
33
34         self.assertEqual(pd.to_timedelta('61sec'), pd.Timedelta('0
days 00:01:01.000000'))
35         self.assertEqual(pd.to_timedelta('1.5min'), pd.Timedelta('0
days 00:01:30.000000'))
36         self.assertEqual(pd.to_timedelta('0.2min'), pd.Timedelta('0
days 00:00:12.000000'))
37         self.assertEqual(pd.to_timedelta('25hours'), pd.Timedelta('1

```

```

    days 01:00:00.000000'))
38     self.assertEqual(pd.to_timedelta('32 days 5 hours'), pd.
Timedelta('32 days 05:00:00.000000'))
39
40     #Test if different units are converted correctly to the '
Timedelta' format
41     #All not deprecated units are tested
42     def testUnitConversion(self):
43         self.assertEqual(pd.to_timedelta(1), pd.Timedelta('0 days
00:00:00.000000001')) #default nano sec
44         self.assertEqual(pd.to_timedelta(1, unit='us'), pd.Timedelta(
'0 days 00:00:00.000001'))
45         self.assertEqual(pd.to_timedelta(1, unit='ms'), pd.Timedelta(
'0 days 00:00:00.001000'))
46         self.assertEqual(pd.to_timedelta(1, unit='s'), pd.Timedelta(
'0 days 00:00:01.000000'))
47         self.assertEqual(pd.to_timedelta(1, unit='m'), pd.Timedelta(
'0 days 00:01:00.000000'))
48         self.assertEqual(pd.to_timedelta(1, unit='h'), pd.Timedelta(
'0 days 01:00:00.000000'))
49         self.assertEqual(pd.to_timedelta(1, unit='d'), pd.Timedelta(
'0 days 24:00:00.000000'))
50         self.assertEqual(pd.to_timedelta(1, unit='w'), pd.Timedelta(
'7 days 00:00:00.000000'))
51
52     #Test argument as list outputs list
53     def testListInput(self):
54         x = ['1 days 06:05:01.00003', '15.5us', '4hr 7minutes']
55         self.assertEqual(pd.to_timedelta(x).tolist(),
56                          [pd.Timedelta('1 days 06:05:01.00003'), pd.
Timedelta('0 days 00:00:00.000015500'),
57                          pd.Timedelta('0 days 04:07:00')])
58
59     #Test with empty list input
60     y = []
61     self.assertEqual(pd.to_timedelta(y).tolist(),[])
62
63     #Test array and unit argument output list with right unit
64     def testNumbersToUnit(self):
65
66         self.assertEqual(pd.to_timedelta(np.arange(3), unit='sec').
tolist(),
67                          [pd.Timedelta('0 days 00:00:00'), pd.
Timedelta('0 days 00:00:01'),
68                          pd.Timedelta('0 days 00:00:02')])
69
70         self.assertEqual(pd.to_timedelta(np.arange(4), unit='day').
tolist(),

```

```

71         [pd.Timedelta('0 days 00:00:00'), pd.
Timedelta('1 days 00:00:00'),
72         pd.Timedelta('2 days 00:00:00'),pd.
Timedelta('3 days 00:00:00')])
73
74     #Test with empty array input
75     self.assertEqual(pd.to_timedelta(np.arange(0), unit='sec').
tolist(),[])

```

4.10 pandas.DataFrame.tail

```

1  import unittest
2  import pandas as pd
3  from pandas._testing import assert_frame_equal
4
5
6  class TestDataframeTail(unittest.TestCase):
7      df = pd.DataFrame({'fruit': ['apple', 'pear', 'banana', 'melon',
'lemon', 'mango', 'lime']})
8      expected = pd.DataFrame({'fruit': ['apple', 'pear', 'banana', '
melon', 'lemon', 'mango', 'lime']})
9
10     #Test output values are the last five elements as default
11     def testReturnDefault(self):
12         expected = self.expected.drop([0, 1])
13         assert_frame_equal(self.df.tail(),expected)
14
15     #Test defined number of elements to return
16     def testNvalueReturn(self):
17         expected = self.expected.drop([0,1,2,3,4,5])
18         assert_frame_equal(self.df.tail(1),expected)
19
20         expected = self.expected.drop([0,1,2,3])
21         assert_frame_equal(self.df.tail(3), expected)
22
23     #all
24     expected = self.expected
25     assert_frame_equal(self.df.tail(7), expected)
26
27     #Test if returns empty list on zero as argument
28     def testZeroReturn(self):
29         expected = self.expected.drop([0,1,2,3,4,5,6])
30         assert_frame_equal(self.df.tail(0), expected)
31
32     #Test with negative argument
33     def testNegativeInput(self):
34         expected = self.expected.drop([0,1,2,3])
35         assert_frame_equal(self.df.tail(-4), expected)

```

```
36         expected = self.expected.drop([0,1,2,3,4,5,6])
37         assert_frame_equal(self.df.tail(-7), expected)
38
39
40     #Test with positive and negative input out of range
41     def testOutOfBoundsInput(self):
42         expected = self.expected.drop([0,1,2,3,4,5,6])
43         assert_frame_equal(self.df.tail(-7000), expected)
44
45         expected = self.expected
46         assert_frame_equal(self.df.tail(7000), expected)
```