# Li_Shuying_hw2

October 13, 2022

## 1 STA 141B Assignment 2

### 1.0.1 Part I: Basic Numpy and Pandas

**Exercise 2.1 (3 points).**

1. Apply a `.stack()` function to the data frame `df1` below. Explain how `.stack()` works. Save stacked data frame `df1` as `df2`.
2. Apply `.unstack()`, `.unstack(0)` and `.unstack(1)` function to `df2`. Explain how each of unstacking methods is different.

[1]:
```
import numpy as np
import pandas as pd

tuples = list(
    zip(
        *[
            ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
            ["one", "two", "one", "two", "one", "two", "one", "two"],
        ]
    )
)

index = pd.MultiIndex.from_tuples(tuples, names=["first", "second"])

df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=["A", "B"])

df1 = df[:4]

df1
```

[1]:
```
                     A         B
first second
bar   one    -0.235900  0.005337
      two    -1.765434  1.731782
baz   one    -0.473665 -0.167856
      two     0.635485 -0.507431
```

### 1.0.2 Exercise 2.1.1

[2]:
```
# Save the stacked dataframe df1 as df2
df2 = df1.stack()
df2
```

[2]:
```
first  second
bar    one     A   -0.235900
               B    0.005337
       two     A   -1.765434
               B    1.731782
baz    one     A   -0.473665
               B   -0.167856
       two     A    0.635485
               B   -0.507431
dtype: float64
```

[3]:
```
df1.index
```

[3]:
```
MultiIndex([('bar', 'one'),
            ('bar', 'two'),
            ('baz', 'one'),
            ('baz', 'two')],
           names=['first', 'second'])
```

[4]:
```
df2.index
```

[4]:
```
MultiIndex([('bar', 'one', 'A'),
            ('bar', 'one', 'B'),
            ('bar', 'two', 'A'),
            ('bar', 'two', 'B'),
            ('baz', 'one', 'A'),
            ('baz', 'one', 'B'),
            ('baz', 'two', 'A'),
            ('baz', 'two', 'B')],
           names=['first', 'second', None])
```

The `.stack(level=- 1, dropna=True)` returns a reshaped DataFrame or series by stacking the prescribed level(s) from columns to index. In other words, it stacks the labels from column to row or subcategorizes them to the next inner levels . This function is used for hierarchical indexing by adding a new level.

In this example,we can see that comparing to `df1`, the index of `df2` changes from 2 to 3.

### 1.0.3 Exercise 2.1.2

[5]:
```
df2.unstack()
```

[5]:
```
                     A         B
first second
bar   one     -0.235900  0.005337
      two     -1.765434  1.731782
baz   one     -0.473665 -0.167856
      two      0.635485 -0.507431
```

[6]:
```
df2.unstack().index
```

[6]:
```
MultiIndex([('bar', 'one'),
            ('bar', 'two'),
            ('baz', 'one'),
            ('baz', 'two')],
           names=['first', 'second'])
```

[7]:
```
df2.unstack(0)
```

[7]:
```
first          bar       baz
second
one    A -0.235900 -0.473665
       B  0.005337 -0.167856
two    A -1.765434  0.635485
       B  1.731782 -0.507431
```

[8]:
```
df2.unstack(0).index
```

[8]:
```
MultiIndex([('one', 'A'),
            ('one', 'B'),
            ('two', 'A'),
            ('two', 'B')],
           names=['second', None])
```

[9]:
```
df2.unstack(1)
```

[9]:
```
second        one       two
first
bar   A -0.235900 -1.765434
      B  0.005337  1.731782
baz   A -0.473665  0.635485
      B -0.167856 -0.507431
```

[10]:
```
df2.unstack(1).index
```

[10]:
```
MultiIndex([('bar', 'A'),
            ('bar', 'B'),
            ('baz', 'A'),
            ('baz', 'B')],
           names=['first', None])
```

Reversely, the `.unstack(level=- 1, fill_value=None)` converts the dataframe or series into unstacked format. In other words, this function pivots the indexed column.

The default `level = -1` pivots the latest inner level.

- The initial index of `df2` is: `('bar', 'one', 'A')`. After using `.unstack()`, the index changes to `('bar', 'one')`.

`.unstack(1)` pivots the `level = 1` inner level.

- After using `.unstack(1)`, the index changes to `('one', 'A')`.

`.unstack(0)` pivots the `level = 0` inner level.

- After using `.unstack(-)`, the index changes to `('bar', 'one')`.

**Exercise 2.2 (2 points).**

1. Give three examples of indexing a data frame with `[ ]`, `.loc[ ]`, and `.iloc[ ]`, respectively. Explain how each of these indexing methods is different.
2. What do negative indexes (as in `x[-1]` and `x[-2]`) do in Python? Explain what you think negative indexes do.

### 1.0.4 Exercise 2.2.1

[11]:
```
df1['A'] # it returns column A's data.
```

[11]:
```
first  second
bar    one      -0.235900
       two      -1.765434
baz    one      -0.473665
       two       0.635485
Name: A, dtype: float64
```

[12]:
```
df1.iloc[0,0]
```

[12]:
```
-0.2358999087066614
```

[13]:
```
df1.loc['bar'] # it returns to the row bar's data.
```

[13]:
```
                A         B
second
one     -0.235900  0.005337
two     -1.765434  1.731782
```

[14]:
```
df1.loc[df1['A'] > 1.0] # condition example
```

[14]:
```
Empty DataFrame
Columns: [A, B]
Index: []
```

`[ ]` is by position, name, or condition.

`.iloc[ ]` is by position.

`.loc[ ]` is by name or condition

### 1.0.5 Exercise 2.2.2

Generally, negative index means indexing from the end of an iterable. It can be used in different data types.

Unlike R, python uses zero-based indexing.

Example of a list `[1,2,3,4,5,6]`:

```
INDEX: | 0 | 1 | 2 | 3 | 4 | 5 |
       -------------------------
VAL:   | 1 | 2 | 3 | 4 | 5 | 6 |
       -------------------------
INDEX: | -6 | -5 | -4 | -3 | -2 | -1 |
```

[15]:
```
# Example 1 - list
x = [1,2,3,4]
print('x[-1]:', x[-1])
print('x[-2]:', x[-2])
```

```
x[-1]: 4
x[-2]: 3
```

[16]:
```
# Example 2 - tuple
x = (1,2,3)
print('x[-1]:', x[-1])
print('x[-2]:', x[-2])
```

```
x[-1]: 3
x[-2]: 2
```

[17]:
```
# Example 3 - string
# It's useful in the process of data preprocessing
x = 'abc-'
print('x[-1]:', x[-1])
print('x[-2]:', x[-2])
```

```
x[-1]: -
x[-2]: c
```

[18]:
```
# Example 4 - numpy array
import numpy as np
x = np.array([[1, 2], [3, 4]])
print(x)
print('x[-1]:', x[-1])
print('x[-2]:', x[-2])
```

```
[[1 2]
 [3 4]]
x[-1]: [3 4]
x[-2]: [1 2]
```

[19]:
```
# Example 5 - Dataframe
df1.iloc[-1] #the last row of the dataframe
```

[19]:
```
A    0.635485
B   -0.507431
Name: (baz, two), dtype: float64
```

**Exercise 2.3 (3 points).**

1. Give an example and explain Pandas' data alignment (or index alignment) feature.
2. Explain the difference between the similarly-named data frame methods `.reindex()` and `.reset_index()`. Give two examples to show what each method respectively does.
3. How might these methods be useful when combined with Pandas' data alignment feature?

*Hint: Besides the Pandas documentation, `.reindex()` is explained in Python for Data Analysis 5.2, and `.reset_index()` is explained here.*

### 1.0.6 Exercise 2.3.1

[20]:
```
import pandas as pd
# The index is automatically labeled from 0 to N.
pd.Series([1, 2, 3])
```

[20]:
```
0    1
1    2
2    3
dtype: int64
```

[21]:
```
# The index is manually labeled by us.
x = pd.Series([1, 2, 3], index = ["a", "b", "c"])
y = pd.Series([1, 2, 3, 1], index = ["b", "a", "d", "c"])
print(x,'\n')
print(y)
```

```
a    1
b    2
c    3
dtype: int64

b    1
a    2
d    3
c    1
dtype: int64
```

[22]:
```
# It also automatically labels dataframe's index
d = {'col1': [1, 2], 'col2': [3, 4]}
df = pd.DataFrame(data=d)
df
```

[22]:
```
   col1  col2
0     1     3
1     2     4
```

[23]:
```
# Operation feature
x*y
```

[23]:
```
a    2.0
b    2.0
c    3.0
d    NaN
dtype: float64
```

Pandas automatically aligns labels starting from 0. Pandas supports vectorized operations, but elements are automatically aligned by index. One of the features of data alignment is that operations will be performed on values with the same row and same column label and are aligned by index not by order. In the example of `x*y`, we can see that the values only be multiplied when they having same index. That's why in the index `d`, it appears as `NaN` which is missing values.

### 1.0.7 Exercise 2.3.2

[24]:
```
# Example of .reindex()
import numpy as np
eg = pd.Series(np.random.randn(4), index = list('abcd'))
print('After using reindex: \n', eg,'\n')
eg1 = eg.reindex(['a','b','c','d','e'], fill_value = 0)
print('After using reindex and add a new index: \n', eg1,'\n')
eg2 = eg.reindex(['d','b','c','a'])
print('After using reindex: \n', eg2,'\n')
```

```
After using reindex:
 a   -0.193475
b    0.782026
c    0.196673
d   -0.660256
dtype: float64

After using reindex and add a new index:
 a   -0.193475
b    0.782026
c    0.196673
d   -0.660256
e    0.000000
dtype: float64
```

```
After using reindex:
 d   -0.660256
b    0.782026
c    0.196673
a   -0.193475
dtype: float64
```

`.reindex()` conforms Series or DataFrame to new index with optional filling logic.

[25]:
```
# Example of .reset_index(drop=True)
y.reset_index(drop = True)
```

[25]:
```
0    1
1    2
2    3
3    1
dtype: int64
```

[26]:
```
# Example of .reset_index()
x.reset_index()
```

[26]:
```
  index  0
0     a  1
1     b  2
2     c  3
```

We can use the `.reset_index()` method to reset the indexes on a series or data frame.

The difference of two functions is that:

- `.reset_index(drop=False)` or `.reset_index(drop=True)` will keep the original index as a new column or it will drop the old index by using the default index. Besides, it can use to rearrange multiindex.
- `.reindex()` can also rearrange but only the existed index. A new object is produced unless the new index is equivalent to the current one.

### 1.0.8 Exercise 2.3.3

Combinding with Pandas' data alignment feature, these methods are useful when manipulating multiple dataframes or doing operations, or merging.

The following are some simply applications:

[27]:
```
# Example 1: usage of reset_index() after dropping null values
from numpy import nan
d = {'col1': [1, 2, nan, 3, 4], 'col2': [1, 2, nan, 3, 4]}
df = pd.DataFrame(data=d)
print('Before dropping null values: \n', df, '\n')
```

```
df = df.dropna()
print('After dropping null values: \n', df, '\n')
df = df.reset_index(drop=True)
print('After resetting index: \n', df, '\n')
```

```
Before dropping null values:
    col1  col2
0   1.0   1.0
1   2.0   2.0
2   NaN   NaN
3   3.0   3.0
4   4.0   4.0

After dropping null values:
    col1  col2
0   1.0   1.0
1   2.0   2.0
3   3.0   3.0
4   4.0   4.0

After resetting index:
    col1  col2
0   1.0   1.0
1   2.0   2.0
2   3.0   3.0
3   4.0   4.0
```

From the above example, we can see that using `.reset_index()` after dropping null values makes the index more consistent.

[28]:
```
# Example 2 - row: usage of reindex()
d = {'col1': [1, 2, nan, 3, 4, nan, 5, 6]}
df = pd.DataFrame(data=d, index = list('abcdefgh'))
print('Before using reindex to drop null values: \n', df, '\n')
df = df.reindex(['a','b','d','e','g','h'])
print('Result: \n', df, '\n')
```

```
Before using reindex to drop null values:
    col1
a   1.0
b   2.0
c   NaN
d   3.0
e   4.0
f   NaN
g   5.0
h   6.0
```

```
Result:
    col1
a   1.0
b   2.0
d   3.0
e   4.0
g   5.0
h   6.0
```

These two methods are effective if we want to drop or rearrange index, which can make our data more readable and easier to do further operations. The `reset_index()` is extremely useful to reset index when combining multiple dataframes since some functions in Pandas use index as a reference. The `reindex()` is useful for adding new rows/columns with null or specific values. For example, before combining two dataframes, we can use `reindex()` to make sure they have same amount and names of index. Or we can use it to drop specific columns or rows from the dataframe.

**Exercise 2.4 (2 points).** Write a function `root` that uses the Newton-Raphson algorithm to compute one of the $p$-th roots for a constant $c$. Your function does not need to find complex roots, only real roots. Your function should have arguments

- `c`, the constant
- `p`, the power
- `x0`, the initial guess
- `N`, the maximum number of iterations

Test your function for $c = 2$, $p = 2$, $N = 200$. Try different values of $x_0$. Can you find initial guesses to get both roots? Explain what happens when the initial guess is $x_0 = 0$.

[29]:
```
from math import isclose

def root(c, p, x0, N):
    """Return a root of f(x) = 0, using Newton's method, starting from
    initial guess x0"""
    # Initilization
    xn = float(x0)
    iterCount = 0
    # Using iteration based on User's input N
    while iterCount < N + 1:
        funValue = xn**p - c # calculte the f(xn) values
        primeValue = p*xn**(p - 1) # calculate the f'(xn) values

        # Break the iteration if it the previous xn and next xn are␣
 ↪approximatly equal
        # REMARK: the rel_tol is highly precise
        if isclose(xn, (xn - funValue / primeValue), rel_tol=1e-9):
            break
        else:
```

```
            xn = xn - funValue / primeValue # Apply the Newton-Raphson
 ↪algorithm formula
        iterCount += 1
    return xn
```

[33]:
```
# Root 1
root(2, 2, 2, 200)
```

[33]:
```
1.2599210498953948
```

[36]:
```
# Root 2
root(2, 2, -2, 200)
```

[36]:
```
1.2599210498948885
```

[32]:
```
# xn = 0
root(2, 2, 0, 200)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Input In [32], in <module>
      1 # xn = 0
----> 2 root(2, 2, 0, 200)

Input In [29], in root(c, p, x0, N)
     12 primeValue = p*xn**(p - 1) # calculate the f'(xn) values
     14 # Break the iteration if it the previous xn and next xn are approximatl
 ↪equal
     15 # REMARK: the rel_tol is highly precise
---> 16 if isclose(xn, (xn - funValue / primeValue), rel_tol=1e-9):
     17     break
     18 else:

ZeroDivisionError: float division by zero
```

When the initial guess is 0, it will has the `ZeroDivisionError: float division by zero` error because it will cause the denominator be 0 when calculating the next guessed root. To solve it, we can update our code by using `try` and `except` and change the value of `x0`.