In Python, cost-effective looping feat. itertools and pandas

Soomin Lee

Contents

- Pythonic Way (short & easy to understand)
- Optimization in a fixed loop
- Repeating inside repetition
- Possible, without a loop

Pythonic Way (for loop)

- There's no need to access anything using indexes.

```
1  chars = ['a', 'b', 'c', 'd']
2
3  for idx in range(len(chars)):
4      print(chars[idx], end='')
5
6  print()
7  for char in chars:
8      print(char, end='')
9
10 print()
11 for char in 'a,b,c,d'.split(','):
12     print(char, end='')
13
14 print()
15 for char in 'abcd':
16     print(char, end='')
```

```
abcd
abcd
abcd
abcd
```

Pythonic Way (for loop)

- Iterable ?
  - All the data types you have encountered so far that are ***collection*** or ***container types*** are iterable. These include the ***string, list, tuple, dict, set, and frozenset types***.

Python >>>

```
>>> iter('foobar')                          # String
<str_iterator object at 0x036E2750>

>>> iter(['foo', 'bar', 'baz'])             # List
<list_iterator object at 0x036E27D0>

>>> iter(('foo', 'bar', 'baz'))             # Tuple
<tuple_iterator object at 0x036E27F0>

>>> iter({'foo', 'bar', 'baz'})             # Set
<set_iterator object at 0x036DEA08>

>>> iter({'foo': 1, 'bar': 2, 'baz': 3})    # Dict
<dict_keyiterator object at 0x036DD990>
```

Python >>>

```
>>> iter(42)                                # Integer
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    iter(42)
TypeError: 'int' object is not iterable

>>> iter(3.1)                               # Float
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    iter(3.1)
TypeError: 'float' object is not iterable

>>> iter(len)                               # Built-in function
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    iter(len)
TypeError: 'builtin_function_or_method' object is not iterable
```

Pythonic Way (for loop)

- Use `enumerate`, `zip`, etc.

```
snacks = [('bacon', 350), ('donut', 240), ('muffin', 190)]
for i in range(len(snacks)):
    item = snacks[i]
    name = item[0]
    calories = item[1]
    print(f'#{i+1}: {name} has {calories} calories')

>>>
#1: bacon has 350 calories
#2: donut has 240 calories
#3: muffin has 190 calories


for rank, (name, calories) in enumerate(snacks, 1):
    print(f'#{rank}: {name} has {calories} calories')

>>>
#1: bacon has 350 calories
#2: donut has 240 calories
#3: muffin has 190 calories
```

Pythonic Way (for loop)

- Use `enumerate`, `zip`, etc.

```python
from itertools import zip_longest
```

```python
feature_names = 'C1,C2,C3,C4'.split(',')
feature_values = [
                    [1, 2, 3, 4],
                    [.2, .1, .5, .3],
                    [0, 0, 1, 0]
                ]
```

```python
for name, val in zip(feature_names, feature_values):
    print(name, val)
```

```
C1 [1, 2, 3, 4]
C2 [0.2, 0.1, 0.5, 0.3]
C3 [0, 0, 1, 0]
```

```python
for name, val in zip_longest(feature_names, feature_values):
    print(name, val)
```

```
C1 [1, 2, 3, 4]
C2 [0.2, 0.1, 0.5, 0.3]
C3 [0, 0, 1, 0]
C4 None
```

```python
twopiece_results = []
for x, y in tqdm(zip(Xs, pred_Ys)):

    t_hats, t_vps = [], []
    subseq_X, subseq_Y = split_into(INIT_REF_POINT, x, y)
    for _x, _y in zip(subseq_X, subseq_Y):

        model=pwlf.PiecewiseLinFit(_x, _y)
        pred_time = calc_time(
            x2 = model.fit(n_segments=2)[2],
            s1 = model.calc_slopes()[1],
            beta = model.predict(_x)[-1]
        )
        t_hats.append(pred_time); t_vps.append(_x[-1]);

    twopiece_results.append([t_hats, t_vps])

```

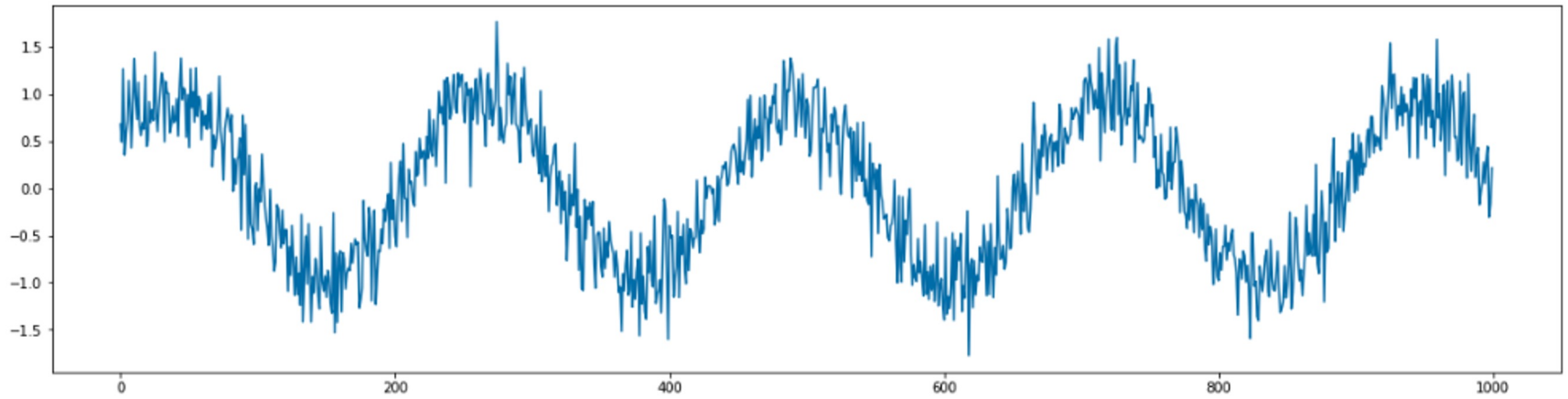
Pythonic Way (for loop)

- Example

```
In [8]:
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import pandas as pds
4 from tqdm.notebook import tqdm
```

```
In [9]:
1 fig, ax = plt.subplots(1, 1, figsize=(20, 5))
2 ax.plot(dataset[:,1])
```

```
Out[9]: [<matplotlib.lines.Line2D at 0x7f590a247400>]
```




```
In [10]:
1 dataset.shape
```

```
Out[10]: (1000, 100000)
```

Pythonic Way (for loop)

- Example

```
col_names = ['stnd', 'mean', 'median', 'quan_1', 'quan_3']
stat_df=pds.DataFrame(columns=col_names)

for idx in tqdm(range(dataset.shape[1])):
    signal = dataset[:,idx]

    stnd = np.std(signal)
    mean = np.mean(signal)
    medn = np.median(signal)
    qu_1 = np.quantile(signal, q=0.25)
    qu_3 = np.quantile(signal, q=0.75)

    row = pds.DataFrame(np.array([stnd, mean, medn, qu_1, qu_3]).reshape(1, -1) ,columns=col_names)
    pds.concat([stat_df, row])
```

Pythonic Way (for loop)

- Example

```
1  col_names = ['stnd', 'mean', 'median', 'quan_1', 'quan_3']
2  stat_df=pds.DataFrame(columns=col_names)
3
4  for idx in tqdm(range(dataset.shape[1])):
5      signal = dataset[:,idx]
6
7      stnd = np.std(signal)
8      mean = np.mean(signal)
9      medn = np.median(signal)
10     qu_1 = np.quantile(signal, q=0.25)
11     qu_3 = np.quantile(signal, q=0.75)
12
13     row = pds.DataFrame(np.array([stnd, mean, medn, qu_1, qu_3]).reshape(1, -1) ,columns=col_names)
14     pds.concat([stat_df, row])
```

Pythonic Way (for loop)

- Example

```
1  dataset.shape
```

```
(1000, 100000)
```

```
1  for block in dataset:
2      print(block.shape)
3      break
```

```
(100000,)
```

```
1  for block in np.transpose(dataset):
2      print(block.shape)
3      break
```

```
(1000,)
```

```
1  col_names = ['stnd', 'mean', 'median', 'quan_1', 'quan_3']
2  stat_df=pds.DataFrame(columns=col_names)
3
4  for signal in tqdm(np.transpose(dataset)):
5
6      stnd = np.std(signal)
7      mean = np.mean(signal)
8      medn = np.median(signal)
9      qu_1 = np.quantile(signal, q=0.25)
10     qu_3 = np.quantile(signal, q=0.75)
11
12     row = pds.DataFrame(np.array([stnd, mean, medn, qu_1, qu_3]).reshape(1, -1) ,columns=col_names)
13     stat_df = pds.concat([stat_df, row])
```

```
35%  35064/100000 [00:32<01:05, 995.57it/s]
```

Optimization in a fixed loop

Optimization in a fixed loop

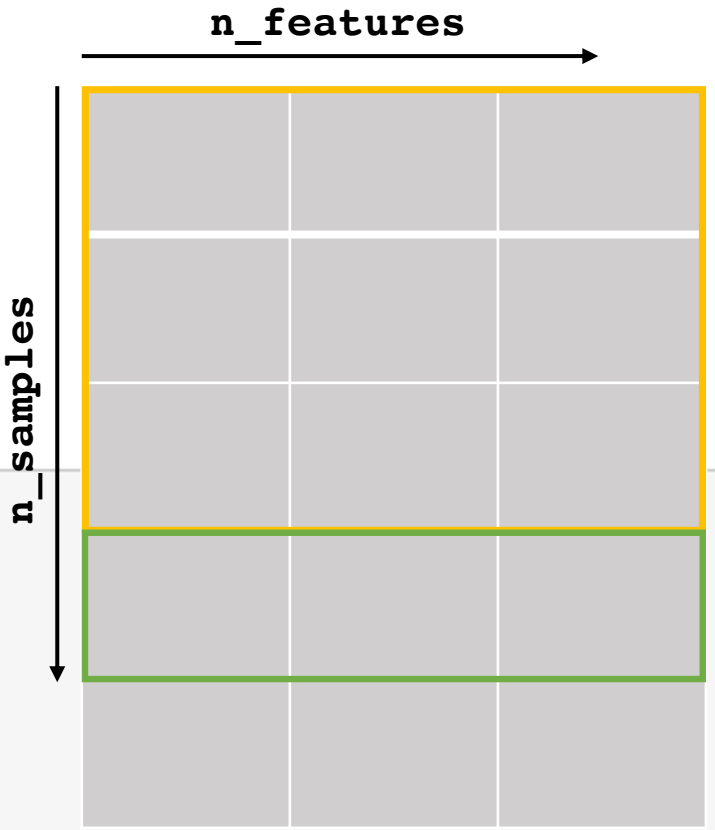
- insert row vs. column

```python
col_names = ['stnd', 'mean', 'median', 'quan_1', 'quan_3']
stat_df=pds.DataFrame(columns=col_names)

for signal in tqdm(np.transpose(dataset)):

    stnd = np.std(signal)
    mean = np.mean(signal)
    medn = np.median(signal)
    qu_1 = np.quantile(signal, q=0.25)
    qu_3 = np.quantile(signal, q=0.75)

    row = pds.DataFrame(np.array([stnd, mean, medn, qu_1, qu_3]).reshape(1, -1) ,columns=col_names)
    stat_df = pds.concat([stat_df, row])
```

```
35% 35064/100000 [00:32<01:05, 995.57it/s]
61% 61197/100000 [01:04<00:55, 695.63it/s]
82% 82479/100000 [01:38<00:30, 581.22it/s]
```

Optimization in a fixed loop

- insert row vs. column



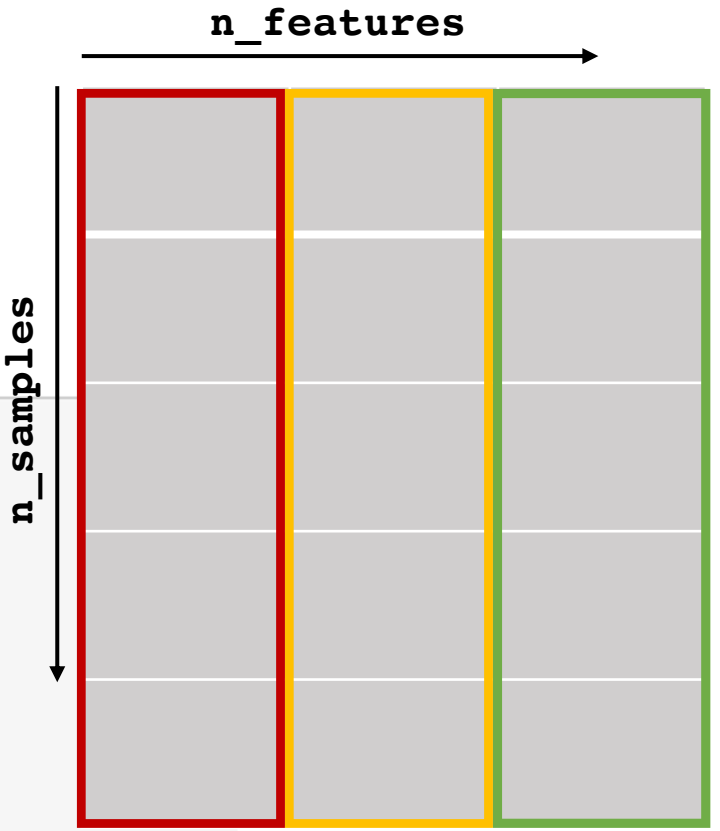

```
col_names = ['stnd', 'mean', 'median', 'quan_1', 'quan_3']
stat_df=pds.DataFrame(columns=col_names)

for signal in tqdm(np.transpose(dataset)):

    stnd = np.std(signal)
    mean = np.mean(signal)
    medn = np.median(signal)
    qu_1 = np.quantile(signal, q=0.25)
    qu_3 = np.quantile(signal, q=0.75)

    row = pds.DataFrame(np.array([stnd, mean, medn, qu_1, qu_3]).reshape(1, -1) ,columns=col_names)
    stat_df = pds.concat([stat_df, row])
```

Optimization in a fixed loop

- insert row vs. column

```
col_names = ['stnd', 'mean', 'median', 'quan_1', 'quan_3']
stat_df=pds.DataFrame()
stnds, means, medns, qu_1s, qu_3s = [], [], [], [], []
for signal in tqdm(np.transpose(dataset)):

    stnds.append(np.std(signal))
    means.append(np.mean(signal))
    medns.append(np.median(signal))
    qu_1s.append(np.quantile(signal, q=0.25))
    qu_3s.append(np.quantile(signal, q=0.75))

stat_df['stnd'] = stnds
stat_df['mean'] = means
stat_df['median'] = medn
stat_df['quan_1'] = qu_1
stat_df['quan_3'] = qu_3
```

```
100% 100000/100000 [00:31<00:00, 3187.44it/s]
```



Optimization in a fixed loop

- insert row vs. column

```
col_names = ['stnd', 'mean', 'median', 'quan_1', 'quan_3']
stat_df=pds.DataFrame(columns=col_names)
rows = []
for signal in tqdm(np.transpose(dataset)):

    stnd = np.std(signal)
    mean = np.mean(signal)
    medn = np.median(signal)
    qu_1 = np.quantile(signal, q=0.25)
    qu_3 = np.quantile(signal, q=0.75)

    rows.append(pds.DataFrame(np.array([stnd, mean, medn, qu_1, qu_3]).reshape(1, -1) ,columns=col_names))

```


```
%%time
df=pds.concat(rows).reset_index(drop=True)
```

```
CPU times: user 8.63 s, sys: 35.3 ms, total: 8.67 s
Wall time: 8.67 s
```

Repeating inside repetition

Repeating inside repetition

```
1  x = [i for i in range(1_000_000)]
2  y = [j for j in 'A.B.C.D.E.F.G'.split('.')]
3  z = [True, False]
```

- The rule of thumb is to avoid using more than two control subexpressions in a comprehension.

```
1  %%time
2
3  tmp = list()
4  for i in x:
5      for j in y:
6          for k in z:
7              tmp.append((i, j, k))
```

```
CPU times: user 3.69 s, sys: 625 ms, total: 4.32 s
Wall time: 4.32 s
```

```
1  %%time
2
3  tmp = list()
4  for i in z:
5      for j in y:
6          for k in x:
7              tmp.append((k, j, i))
```

```
CPU times: user 3.36 s, sys: 687 ms, total: 4.05 s
Wall time: 4.04 s
```

Repeating inside repetition

```
1 x = [i for i in range(1_000_000)]
2 y = [j for j in 'A.B.C.D.E.F.G'.split('.')]
3 z = [True, False]
```

- The rule of thumb is to avoid using more than two control subexpressions in a comprehension.

```
1 %%time
2 tmp=[(k, j, i) for i in z for j in y for k in x]
```

```
CPU times: user 1.85 s, sys: 555 ms, total: 2.41 s
Wall time: 2.41 s
```

```
1 from itertools import product
```

```
1 %%time
2 tmp=list(product(x, y, z))
```

```
CPU times: user 1.53 s, sys: 602 ms, total: 2.13 s
Wall time: 2.12 s
```

```
1 tmp
```

```
[(0, 'A', True),
 (0, 'A', False),
 (0, 'B', True),
 (0, 'B', False),
 (0, 'C', True),
 (0, 'C', False),
 (0, 'D', True),
 (0, 'D', False),
 (0, 'E', True),
 (0, 'E', False),
 (0, 'F', True),
 (0, 'F', False),
 (0, 'G', True),
 (0, 'G', False),
 (1, 'A', True),
 (1, 'A', False),
 (1, 'B', True)
```

Repeating inside repetition

- Example

```
save_dirs = os.path.join(paths.weights, 'cmhs/STL')

for idx, (model, label) in enumerate(product(*[models, 'CE,SE'.split(',')])):
    print(f'{idx}')
    i = idx//2

    for n_fold in range(10):
        DL = cmhs(dataset_path,n_fold,scaling_factor=True)

        if i < 4:
            save_name='LSTM_bid_{}_res_{}'.format(
                rnn_params[i]['bidirectional'],
                rnn_params[i]['residual']
            )
        elif i == 4:
            save_name='TCN'

        checkpoint_path = os.path.join(
            save_dirs,label,f'Fold_{n_fold}',save_name,'cp.ckpt'
        )
        checkpoint_dir = os.path.dirname(checkpoint_path)

        model.load_weights(checkpoint_path)
```

Possible, without a loop

Possible, without a loop

- Specifically, in Pandas

```
1 import numpy as np
2 import pandas as pds
3 from tqdm.notebook import tqdm
```

```
1 tbl = pds.read_csv('sample.csv')
2 lbl = pds.read_csv('sample_lbl.csv')
```

```
1 tbl.shape, lbl.shape
```

```
((500000, 198), (458913, 2))
```

customer_ID	S_2	P_2	D_39	B_1	B_2	R_1	S_3	D_41	B_3	...	D_136	D_137	D_138	D_139	D_140	D_141
e9d7c79be5f...	2017-03-09	0.938469	0.001733	0.008724	1.006838	0.009228	0.124035	0.008771	0.004709	...	NaN	NaN	NaN	0.002427	0.003706	0.003818
e9d7c79be5f...	2017-04-07	0.936665	0.005775	0.004923	1.000653	0.006151	0.126750	0.000798	0.002714	...	NaN	NaN	NaN	0.003954	0.003167	0.005032
e9d7c79be5f...	2017-05-28	0.954180	0.091505	0.021655	1.009672	0.006815	0.123977	0.007598	0.009423	...	NaN	NaN	NaN	0.003269	0.007329	0.000427
e9d7c79be5f...	2017-06-13	0.960384	0.002455	0.013683	1.002700	0.001373	0.117169	0.000685	0.005531	...	NaN	NaN	NaN	0.006117	0.004516	0.003200
e9d7c79be5f...	2017-07-16	0.947248	0.002483	0.015193	1.000727	0.007605	0.117325	0.004653	0.009312	...	NaN	NaN	NaN	0.003671	0.004946	0.008889
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
6a71b4e727...	2017-03-23	0.465217	0.005068	0.113282	1.007081	0.505163	0.608732	0.153365	0.002805	...	NaN	NaN	NaN	0.003300	0.004040	0.008041
6a71b4e727...	2017-04-09	0.525224	0.390019	0.065326	0.816233	0.000675	0.620330	0.142519	0.010256	...	NaN	NaN	NaN	0.003356	0.007763	0.000864
6a71b4e727...	2017-05-09	0.519320	0.392221	0.051505	1.005239	0.007719	0.685221	0.284388	0.010108	...	NaN	NaN	NaN	0.003430	0.005983	0.001171
6a71b4e727...	2017-06-01	0.503259	1.061541	0.063314	0.858517	0.003517	0.804081	0.441097	0.039857	...	NaN	NaN	NaN	0.003135	0.004013	0.006450
6a71b4e727...	2017-07-15	0.519973	0.564801	0.094378	0.349181	0.502475	0.731669	0.283724	0.138456	...	NaN	NaN	NaN	0.007290	0.001797	0.008835

Possible, without a loop

- Case 1: counting the unique values

```
1  id2count = dict()
2  IDs = tbl['customer_ID']
3  for _id in tqdm(IDs.unique()):
4      count=len(np.where(IDs==_id)[0])
5      id2count[_id]=count
```

```
1% |                                   533/41444 [00:10<12:57,
```

```
1  %%time
2  tbl.customer_ID.value_counts()
```

```
CPU times: user 49.2 ms, sys: 44 µs, total: 49.2 ms
Wall time: 48 ms

01a584d42ff4326faa6bb96d6cfa615123007c6a3137a7ffec4d5ba6a04c7f4d    13
0cf5ab151f851460247ba79afa4a0050d2906df7e75b5d73ebbdef902afd0bf8    13
07b5c9ec11fc05524a097eef1fa5cff8451a2314a0f752283e9daffbbb831689    13
0701f445ca8d516bac1a9b852a8ab55b9ed775020d65dfc3aeaf89f28111a044    13
062e9cfa78e4d79372203cf72b48a2ce6004a05966ad5060201db62e056ab46c    13
                                                                    ..
0e18f993864db470defb583189c78547e97d7bad67b8ceb63af0fbfdbfe83d33     1
09c65abfe5a58a97ea269b87b6c4878a38b28230dd3ca9997b91021458cefcfa     1
0ef5a71f3efccc83b92c70c21c640cd04da3bb73095130d62cfe5e48f8ea7ee4     1
0d029f8a7fdfda26bd7cdbf26c91add85d946f498a10b4ce8140439d283434b2     1
11702a473f5ad50d82535113d1dd325564cd340171bb44ae331cf492fd416dac     1
Name: customer_ID, Length: 41444, dtype: int64
```

Possible, without a loop

- Case 1: counting the unique values

```
1  id2count = dict()
2  IDs = tbl['customer_ID']
3  for _id in tqdm(IDs.unique()):
4      count=len(np.where(IDs==_id)[0])
5      id2count[_id]=count
```

```
1%  533/41444 [00:10<12:57,
```

```
1  from collections import Counter
```

```
1  %%time
2  Counter(tbl.customer_ID.values)
```

```
CPU times: user 28 ms, sys: 0 ns, total: 28 ms
Wall time: 27.9 ms

Counter({'0000099d6bd597052cdcda90ffabf56573fe9d7c79be5fbac11a8ed792feb62a': 13,
         '00000fd6641609c6ece5454664794f0340ad84dddce9a267a310b5ae68e9d8e5': 13,
         '00001b22f846c82c51f6e3958ccd81970162bae8b007e80662ef27519fcc18c1': 13,
         '000041bdba6ecadd89a52d11886e8eaaec9325906c9723355abb5ca523658edc': 13,
         '00007889e4fcd2614b6cbe7f8f3d2e5c728eca32d9eb8ad51ca8b8c4a24cefed': 13,
         '000084e5023181993c2e1b665ac88dbb1ce9ef621ec5370150fc2f8bdca6202c': 13,
         '000098081fde4fd64bc4d503a5d6f86a0aedc425c96f5235f98b0f47c9d7d8d4': 13,
         '0000d17a1447b25a01e42e1ac56b091bb7cbb06317be4cb59b50fec59e0b6381': 13,
         '0000f99513770170a1aba690daeeb8a96da4a39f11fc27da5c30a79db61c1e85': 13,
         '00013181a0c5fc8f1ea38cd2b90fe8ad2fa8cad9d9f13e4063bdf6b0f7d51eb6': 13,
         '0001337ded4e1c2539d1a78ff44a457bd4a95caa55ba1730b2849b92ea687f9e': 3,
         '00013c6e1cec7c21bede7cb319f1e28eb994f5625257f479c53ad6e90c177f7c': 13,
         '0001812036f1558332e5c0880ecbad70b13a6f28ab04a8db6d83a26ef40aadb0': 13,
```

Possible, without a loop

- Case 2: one-hot encoding

```
1 cat_fea = ['B_30', 'B_38', 'D_63', 'D_64', 'D_66', 'D_68', 'D_114', 'D_116', \
2            'D_117', 'D_120', 'D_126']
```

```
1 tbl[cat_fea]
```

	B_30	B_38	D_63	D_64	D_66	D_68	D_114	D_116	D_117	D_120	D_126
0	0.0	2.0	CR	O	NaN	6.0	1.0	0.0	4.0	0.0	1.0
1	0.0	2.0	CR	O	NaN	6.0	1.0	0.0	4.0	0.0	1.0
2	0.0	2.0	CR	O	NaN	6.0	1.0	0.0	4.0	0.0	1.0
3	0.0	2.0	CR	O	NaN	6.0	1.0	0.0	4.0	0.0	1.0
4	0.0	2.0	CR	O	NaN	6.0	1.0	0.0	4.0	0.0	1.0
...	...	...	...	...	...	...	...	...	...	...	...
499995	0.0	2.0	CR	R	1.0	6.0	1.0	0.0	6.0	1.0	1.0
499996	0.0	2.0	CR	R	1.0	6.0	1.0	0.0	6.0	0.0	1.0
499997	0.0	2.0	CR	R	1.0	6.0	1.0	0.0	6.0	0.0	1.0
499998	0.0	3.0	CR	R	1.0	6.0	1.0	0.0	-1.0	0.0	1.0
499999	0.0	3.0	CR	R	1.0	6.0	1.0	0.0	-1.0	0.0	1.0

500000 rows × 11 columns

```
1 tbl[cat_fea].nunique()
```

```
B_30      3
B_38      7
D_63      6
D_64      4
D_66      2
D_68      7
D_114     2
D_116     2
D_117     7
D_120     2
D_126     3
dtype: int64
```

```
1 tbl[cat_fea].nunique().values.sum()
```

```
45
```

Possible, without a loop

- Case 2: one-hot encoding

```
1 pds.get_dummies(tbl[cat_fea])
```

	B_30	B_38	D_66	D_68	D_114	D_116	D_117	D_120	D_126	D_63_CL	D_63_CO	D_63_CR	D_63_XL	D_63_XM	D_6
0	0.0	2.0	NaN	6.0	1.0	0.0	4.0	0.0	1.0	0	0	1	0	0	
1	0.0	2.0	NaN	6.0	1.0	0.0	4.0	0.0	1.0	0	0	1	0	0	
2	0.0	2.0	NaN	6.0	1.0	0.0	4.0	0.0	1.0	0	0	1	0	0	
3	0.0	2.0	NaN	6.0	1.0	0.0	4.0	0.0	1.0	0	0	1	0	0	
4	0.0	2.0	NaN	6.0	1.0	0.0	4.0	0.0	1.0	0	0	1	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
499995	0.0	2.0	1.0	6.0	1.0	0.0	6.0	1.0	1.0	0	0	1	0	0	
499996	0.0	2.0	1.0	6.0	1.0	0.0	6.0	0.0	1.0	0	0	1	0	0	
499997	0.0	2.0	1.0	6.0	1.0	0.0	6.0	0.0	1.0	0	0	1	0	0	
499998	0.0	3.0	1.0	6.0	1.0	0.0	-1.0	0.0	1.0	0	0	1	0	0	
499999	0.0	3.0	1.0	6.0	1.0	0.0	-1.0	0.0	1.0	0	0	1	0	0	

500000 rows × 19 columns ?

Possible, without a loop

- Case 2: one-hot encoding

```
1  tbl[cat_fea].info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 500000 entries, 0 to 499999
Data columns (total 11 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   B_30    499819 non-null  float64
 1   B_38    499819 non-null  float64
 2   D_63    500000 non-null  object
 3   D_64    480339 non-null  object
 4   D_66    55942 non-null   float64
 5   D_68    480558 non-null  float64
 6   D_114   484247 non-null  float64
 7   D_116   484247 non-null  float64
 8   D_117   484247 non-null  float64
 9   D_120   484247 non-null  float64
 10  D_126   489542 non-null  float64
dtypes: float64(9), object(2)
memory usage: 42.0+ MB
```

```
1  %%time
2
3  tar_enc=tbl[cat_fea].astype(str)
```

```
CPU times: user 1.42 s, sys: 67.5 ms, total: 1.49 s
Wall time: 1.49 s
```

```
1  %%time
2
3  tar_enc=tbl[cat_fea].applymap(lambda x: str(x))
```

```
CPU times: user 1.33 s, sys: 106 ms, total: 1.44 s
Wall time: 1.44 s
```

```
1  tar_enc.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 500000 entries, 0 to 499999
Data columns (total 11 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   B_30    500000 non-null  object
 1   B_38    500000 non-null  object
 2   D_63    500000 non-null  object
 3   D_64    500000 non-null  object
 4   D_66    500000 non-null  object
 5   D_68    500000 non-null  object
 6   D_114   500000 non-null  object
 7   D_116   500000 non-null  object
 8   D_117   500000 non-null  object
 9   D_120   500000 non-null  object
 10  D_126   500000 non-null  object
dtypes: object(11)
memory usage: 42.0+ MB
```

Homework

- implement the one-hot encoding function

```
'''
df - Dataframe contained only the categorical variables

ret_df - Encoded dataframe
'''

def encoder(df, method='one-hot'):
    assert method == 'one-hot', f'{method} has not been implemented'


    return ret_df
```

Q & A