In [ ]:

```
bjana[bjana.month2 > 10].groupby(bjana.year2).value.max().plot()
```

12.3.3 计算爆表天数

In [ ]:

```
bjana[bjana.value > 500].groupby(bjana.year2).value.count().plot.bar()
```

In [ ]:

```
bjana.query("value > 500 and month2 > 10").\
    groupby(bjana.year2).value.count().plot.bar()
```

In [ ]:

```
bjana.query("value > 500 and month2 >= 10").\
    groupby(bjana.year2).value.count().plot.bar()
```

13 如何优化Pandas

- 除非对相应的优化手段已经非常熟悉，否则代码的可读性应当被放在首位。过早优化是万恶之源！
- pandas为了易用性，确实牺牲了一些效率，但同时也预留了相应的优化路径。因此如果要进行优化，熟悉并优先使用pandas自身提供的优化套路至关重要。
- 尽量使用pandas（或者numpy）内置的函数进行运算，一般效率都会更高。
- 在可以用几种内部函数实现相同需求时，最好进行计算效率的比较，差距可能很大。
- pandas官方提供的讨论如何进行优化的文档：https://pandas.pydata.org/pandas-docs/stable/user_guide/enhancingperf.html (https://pandas.pydata.org/pandas-docs/stable/user_guide/enhancingperf.html)

In [ ]:

```
def m_readdata(filename, startline = 2):
    return pd.read_csv(filename, header = startline,
                       usecols = [0,2,3,4,5,6,7,9,10])

bj08 = m_readdata("pm25/Beijing_2008_HourlyPM2.5_created20140325.csv")
bj09 = m_readdata("pm25/Beijing_2009_HourlyPM25_created20140709.csv")
bj10 = m_readdata("pm25/Beijing_2010_HourlyPM25_created20140709.csv")
bj11 = m_readdata("pm25/Beijing_2011_HourlyPM25_created20140709.csv")
bj12 = m_readdata("pm25/Beijing_2012_HourlyPM2.5_created20140325.csv")
bj13 = m_readdata("pm25/Beijing_2013_HourlyPM2.5_created20140325.csv")
bj14 = m_readdata("pm25/Beijing_2014_HourlyPM25_created20150203.csv")
bj15 = m_readdata("pm25/Beijing_2015_HourlyPM25_created20160201.csv",3)
bj16 = m_readdata("pm25/Beijing_2016_HourlyPM25_created20170201.csv",3)
bj17 = m_readdata("pm25/Beijing_2017_HourlyPM25_created20170803.csv",3)

bj = bj08.append([bj09,bj10,bj11,bj12,bj13,bj14,bj15,bj16,bj17])
print(bj.shape)
bj.head()
```

13.1 学会使用各种计时工具

%time和%timeit，以及%%

需要在IPython下才可以使用。

In [ ]:

```
%time bj["tmp"] = bj.Month + 10 # 运行当前代码所需时间
```

In [ ]:

```
%timeit bj["tmp"] = bj.Month + 10
```

In [ ]:

```
%%timeit # 运行整个程序段所需平均时间

bj["tmp"] = bj.Month + 10
bj["tmp"] = bj.Month + 10
bj["tmp"] = bj.Month + 10
bj["tmp"] = bj.Month + 10
```

datetime方式

In [ ]:

```
import datetime
import time

time0 = datetime.datetime.now()

bj["tmp"] = bj.Month + 10

print(str(datetime.datetime.now() - time0))
```

用line_profiler做深入分析

In [ ]:

```
from line_profiler import LineProfiler
import random

def M_test(numbers):
    s = sum(numbers)
    l = [numbers[i]/43 for i in range(len(numbers))]
    m = ['hello' + str(numbers[i]) for i in range(len(numbers))]

lp = LineProfiler() # 定义一个LineProfiler对象
lp_wrapper = lp(M_test) # 用LineProfiler对象监控需要分析的函数
```

In [ ]:

```
lp.functions
```

In [ ]:

```
nums = [random.randint(1,100) for i in range(1000)]
print(nums[:5])
lp_wrapper(nums) # 运行被监控的函数，获取各部分占用的运行时间数据
```

In [ ]:

```
lp.print_stats()
```

13.2 超大数据文件的处理

超大数据文件在使用pandas进行处理时可能需要考虑两个问题：读取速度，内存用量。

- 往往会考虑读入部分数据进行代码编写和调试，然后再对完整数据进行处理。
- 在数据读入和处理时需要加快处理速度，减少资源占用。

13.2.1 一些基本原则

当明确知道数据列的取值范围时，读取数据时可以使用dtype参数来手动指定类型，如np.uint8或者np.int16，否则默认的np.int64类型等内存开销明显非常大。

尽量少用类型模糊的object，改用更明确的category等（用astype()转换）。

对类别取值较少，但案例数极多的变量，读入后尽量转换为数值代码进行处理。

In [ ]:

```
data = pd.DataFrame({"a" : [0,1, 2, 3, 4, 5, 6, 7, 8, 9],
                     "b" : ["员工编号","员工编号","领导编号","领导编号",
                            "员工编号","员工编号","员工编号","领导编号",
                            "领导编号","员工编号"]})
print(data)
data.info()
```

In [ ]:

```
data['a'] = pd.to_numeric(data['a'], downcast = 'integer')
data['b'] = data['b'].astype('category')
print(data)
```

In [ ]:

```
data.b.head() # category格式明显更节省内存
```

In [ ]:

```
data.info()
```

In [ ]:

```
data.b.astype('str') # 必要时category也可以转换回str
```

13.2.2 对文件进行分段读取

使用chunksize参数

In [ ]:

```
filename = "pm25/Beijing_2008_HourlyPM2.5_created20140325.csv"
dftmp = pd.read_csv(filename, header = 2,
                    usecols = [0,2,3,4,5,6,7], chunksize = 5)

type(dftmp) # 注意得到的并不是一个数据框，而是TextFileReader
```

In [ ]:

```
cnt = 0
for item in dftmp: # 注意重复运行之后的效果
    print(item)
    cnt += 1
    if cnt > 2:
        break
```

使用iterator参数和get_chunk()组合

In [ ]:

```
dfiter = pd.read_csv(filename, header = 2,
                     usecols = [0,2,3,4,5,6,7], iterator = True)

type(dfiter) # 注意得到的并不是一个数据框，而是TextFileReader
```

In [ ]:

```
cnt = 0
for item in dfiter: # 注意结果是否正确
    print(item)
    cnt += 1
    if cnt > 2:
        break
```

In [ ]:

```
# TextFileReader使用完毕后已经失效，需要重新建立
dfiter.get_chunk(10) # 注意重复运行之后的效果
```

13.2.3 使用modin包

modin使用并行方式对pandas中的文件读取和常见操作进行执行。

```
windows环境下只能使用dask引擎，其余环境下还可以使用ray
目前还不支持pandas 1.00及以上版本，只支持到0.25版本
pip install modin[dask]
```

In [ ]:

```
filename = "pm25/Beijing_2008_HourlyPM2.5_created20140325.csv"
%timeit dftmp = pd.read_csv(filename, header = 2, usecols = [0,2,3,4,5,6,7,9,10])
```

In [ ]:

```
import modin.pandas as pd # 只需要在这里进行修改即可

%timeit dftmp = pd.read_csv(filename, header = 2, usecols = [0,2,3,4,5,6,7,9,10])
```

13.3 如何优化pandas的代码

13.3.1 加速！加速！再加速！

尽量不要在pandas中使用循环（行/列/单元格遍历）！！！

如果循环很难避免，尽量在循环体中使用numpy做计算。

In [ ]:

```
%%time

# 标准的行/列遍历循环方式效率最差，> 1 min

bj['new'] = 0
for i in range(bj.shape[0]):
    bj.iloc[i, 9] = bj.iloc[i, 3] + 10
```

In [ ]:

```
# apply自定义函数/外部函数效率稍差
def m_add(x):
    return x + 10

%timeit bj["new"] = bj.Month.apply(lambda x : m_add(x))
```

In [ ]:

```
# 在apply中应用内置函数方式多数情况下速度更快
%timeit bj["new"] = bj.Month.apply(lambda x : x + 10)
```

In [ ]:

```
# 直接应用原生内置函数方式速度最快（此即所谓的矢量化计算）
%timeit bj["new"] = bj.Month + 10
```

13.3.2 如何进行多列数据的计算

同时涉及多列数据的计算不仅需要考虑速度优化问题，还需要考虑代码简洁性的问题。

利用lambda函数完成

In [ ]:

```
%%timeit

def m_add(a, b, c):
    return a + b + c

bj['new'] = bj.apply(lambda x :
                     m_add(x['Month'], x['Day'], x['Hour']), axis = 1)
```

In [ ]:

```
%%timeit

# lambda多参方式可能效率反而更低

bj['new'] = bj[['Month', 'Day', 'Hour']].apply(lambda x :
                                               x[0] + x[1] + x[2],
                                               axis = 1)
```

In [ ]:

```
# 还是原生方式最香
%timeit bj['new'] = bj['Month'] + bj['Day'] + bj['Hour']
```

pd.eval()命令

pd.eval()的功能仍是给表达式估值，但是基于pandas时，就可以直接利用列名等信息用于计算。

In [ ]:

```
# eval()默认会使用numexpr而不是python计算引擎，复杂计算时效率很高
# engine参数在调用numexpr失败时会切换为python引擎，一般不用干涉
%timeit bj.eval('new = Month + Day + Hour', inplace = True)
```

In [ ]:

```
# eval()默认会使用numexpr而不是python计算引擎，复杂计算时效率很高
%timeit bj.eval('new = Month + Day + Hour', engine = 'python', inplace = True)
```

In [ ]:

```
%%timeit

bj.eval("""
    ...: n1 = Month + 1
    ...: n2 = Month + 2
    ...: n3 = Day + 3
    """, inplace = True)
# 在一条eval命令中可以同时完成多个新列的计算，但需要注意代码兼容性

bj.head()
```

In [ ]:

```
%%timeit

bj['n1'] = bj.Month + 1
bj['n2'] = bj.Month + 2
bj['n3'] = bj.Day + 3

bj.head()
```

pd.在eval()表达式中进一步使用局部变量

@varname：在表达式中使用名称为varname的Python局部变量。

- 该@符号在query()和eval()中均可使用。

In [ ]:

```
varx = 3
bj.eval('new = Month + Day + Hour + @varx').head()
```

13.4 利用各种pandas加速外挂

大部分外挂包都是基于linux环境，windows下能用的不多。

大部分外挂包还处于测试阶段，功能上并未完善。

13.4.1 numba

对编写的自定义函数进行编译，以提高运行效率。

```
A Just-In-Time Compiler for Numerical Functions in Python
具体使用的是C++编写的LLVM(Low Level Virtual Machine) compiler
实际上主要是针对numpy库进行优化编译，并不仅限于pandas
```

In [ ]:

```
import random

def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples

%timeit monte_carlo_pi(20) # 运行被监控的函数，获取各部分占用的运行时间数据
```

In [ ]:

```
from numba import jit

@jit(nopython = True) # nopython模式下性能最好
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples

%timeit monte_carlo_pi(20)
```

In [ ]:

```
from numba import jit

@jit
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples

%timeit monte_carlo_pi(20)
```

13.4.2 swifter

对apply函数进行并行操作。

是专门针对pandas进行优化的工具包

In [ ]:

```
df = pd.DataFrame({'x': [1, 2, 3, 4], 'y': [5, 6, 7, 8]})

# runs on single core
%timeit df['x2'] = df['x'].apply(lambda x: x**2)
```

In [ ]:

```
import swifter

# runs on multiple cores
%timeit df['x2'] = df['x'].swifter.apply(lambda x: x**2)
```

In [ ]:

```
%timeit df['agg'] = df.apply(lambda x: x.sum() - x.min())
```

In [ ]:

```
# use swifter apply on whole dataframe
%timeit df['agg'] = df.swifter.apply(lambda x: x.sum() - x.min())
```