

Python pandas入門

pandasとは

- ・ Pythonのデータ解析支援ライブラリ
- ・ Rのようなデータフレームというデータ形式

DataFrameとSeries

- DataFrame 2次元表
- Series 1次元表

indexとcolumn

- index 行の名前
- column 列の名前

Seriesはindexのみ。
indexは行番号（配列インデクス）、
columnはテーブルのカラム名のように使える。
ただし、index、columnとも重複を許す点に注意。

DataFrameの作成1

```
>>> import pandas as pd
>>> df = pd.DataFrame([{'s': 'x', 'n': 1}, {'s': 'y', 'n': 2}])
>>> df
   n  s
0  1  x
1  2  y
>>>
>>> df = pd.DataFrame([[1, 'x'], [2, 'y'], [3, 'z']])
>>> df
   0  1
0  1  x
1  2  y
2  3  z
>>>
>>> df.columns = ['s', 'n'] # カラム設定
>>> df
   n  s
0  1  x
1  2  y
2  3  z
```

DataFrameの作成2

```
>>> import pandas as pd
>>> df = pd.DataFrame([list('abcd'), list('1234'), list('+-* /')])
>>> df
   0  1  2  3
0  a  b  c  d
1  1  2  3  4
2  +  -  *  /
>>>
>>> df = df.T # 行列反転
>>> df
   0  1  2
0  a  1  +
1  b  2  -
2  c  3  *
3  d  4  /
>>>
>>> df.columns = ['arph', 'num', 'ope'] # カラム設定
>>> df
   arph  num ope
0    a    1  +
1    b    2  -
2    c    3  *
3    d    4  /
```

CSVの入出力

```
>>> df.columns = list('あいう')
>>> df.to_csv('a.csv', index=False, encoding='CP932') # CSV出力
>>>
>>> pd.read_csv('a.csv', index_col=None, encoding='CP932') # CSV入力
   あ   い   う
0  a   1   +
1  b   2   -
2  c   3   *
3  d   4   /
```

pandas.DataFrame.from_csv()は後方互換性のために、残されている関数。オプション引数の既定値が異なる。pandas.read_csv()を使うことが推奨されており、オプション引数の数も多い。

行・列の抽出

```
>>> df.head(10) # 先頭から10行
>>> df.iloc[:10] # 先頭から10行
>>> df.ix[:10] # インデクス10までの行を抽出
>>> df.ix[:, 1:3] # 列順による抽出(2、3列目)
>>> df[['city', 'pop2010']] # 列名による抽出。列順変更にも使える。
```

DataFrame.ixについて

- ・ `df.ix[<行の指定>, <列の指定>]`
- ・ 「:」は全部
- ・ インデクスが非intの場合は行指定は行番号指定と判定される

条件抽出

```
>>> smr[smr['pop2050'] > smr['pop2010']*0.8]      # where pop2050 > pop2010 * 0.8
>>> smr[(10 < smr['ken']) & (smr['ken'] < 30)]    # where 10 < ken and ken < 30
>>> smr[(smr['ken'] < 5) | (40 < smr['ken'])]      # where ken < 5 or 40 < ken

>>> smr[smr['name'].str.contains('.*京.*')]        # where name like '%京%'
>>> smr[~smr['name'].str.contains('.*京.*')]      # where name not like '%京%'

>>> smr.query("ken < 5 or 40 < ken")              # query()を使った or 条件検索
```

データ加工

```
>>> df['ken'] = df['city'].apply(lambda s: int(s[:len(s) - 3])) # 既存の列を編集して新しい列を作成
>>> df['pop2010'] = df['pop2010'].apply(lambda s: int(s))      # 数値型に変換

>>> df = pd.merge(df, prefs, on='ken')                          # データフレームを結合 where t1.ken = t2.ken
>>> df.groupby(['ken', 'name'])['pop2010'].sum()               # 集約 sum(pop2010) ... groupby ken, name
>>> df.groupby(['ken', 'name']).size()                         # 集約 count() ... groupby ken, name

>>> kens = df.drop_duplicates(['ken', 'name'])                 # ユニーク化 select distinct ken, name
>>> kens.sort_values('ken')                                    # ソート order by ken

>>> pd.concat([df1, df2])   # データフレームを縦結合
```

集約結果を使ってデータ加工を継続するには

集約結果はSeriesで返される。データ加工を続ける場合はreset_index()でDataframeに変換するとよい。

```
>>> smr = df.groupby(['ken', 'name']).size().reset_index(name='count')
```

.(ドット)によるカラムアクセスについて

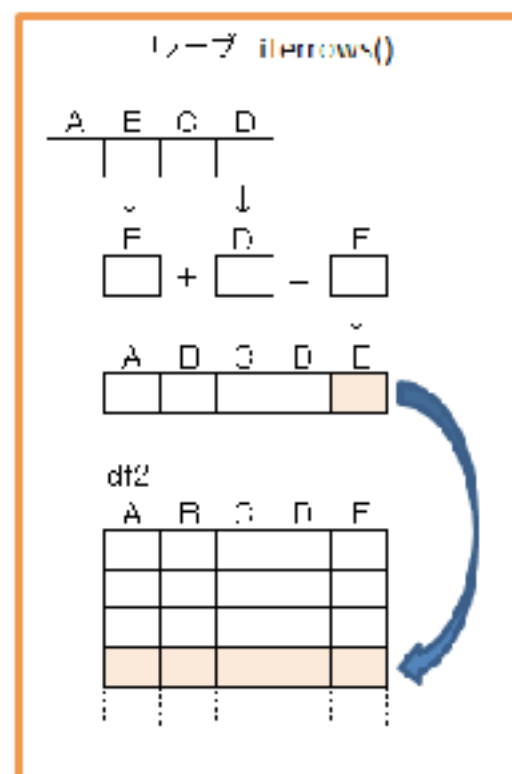
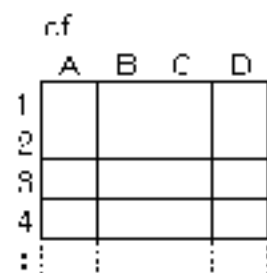
Dataframeのカラム参照 (`df['col']`) は`df.col`と記述可能。
ただし、カラム追加時は不可。

- ・カラム追加時には使えない。
- ・カラム追加時に`hoge.col1 = ~`とやるとデータフレーム外に要素が追加されて混乱する。
- ・データフレームのカラムとデータフレーム外の要素が両方ある状態で`hoge.c`とするとデータフレーム外の要素が参照される。
- ・データフレーム外の要素を`del hoge.c`として消すと`hoge.c`でデータフレームのカラムを参照するようになる。
- ・カラム削除の際も`del hoge.c`ではなく`del hoge['c']`とする必要あり。

データ加工のコツ（アンチパターン）

```
df2 = pd.DataFrame()
for i, seri in df.iterrows():
    datetime_str0 = seri.date0 + " " + seri.time0
    dt0 = datetime.strptime(datetime_str0, "%Y-%m-%d %H:%M:%S")
    dt = dt0 + timedelta(hours=9)
    seri['date_time'] = dt.strftime("%Y-%m-%d %H:%M:%S")
    df2 = df2.append(seri)
```

date0 time0による日時
を9時間進めてdate time
を作成。

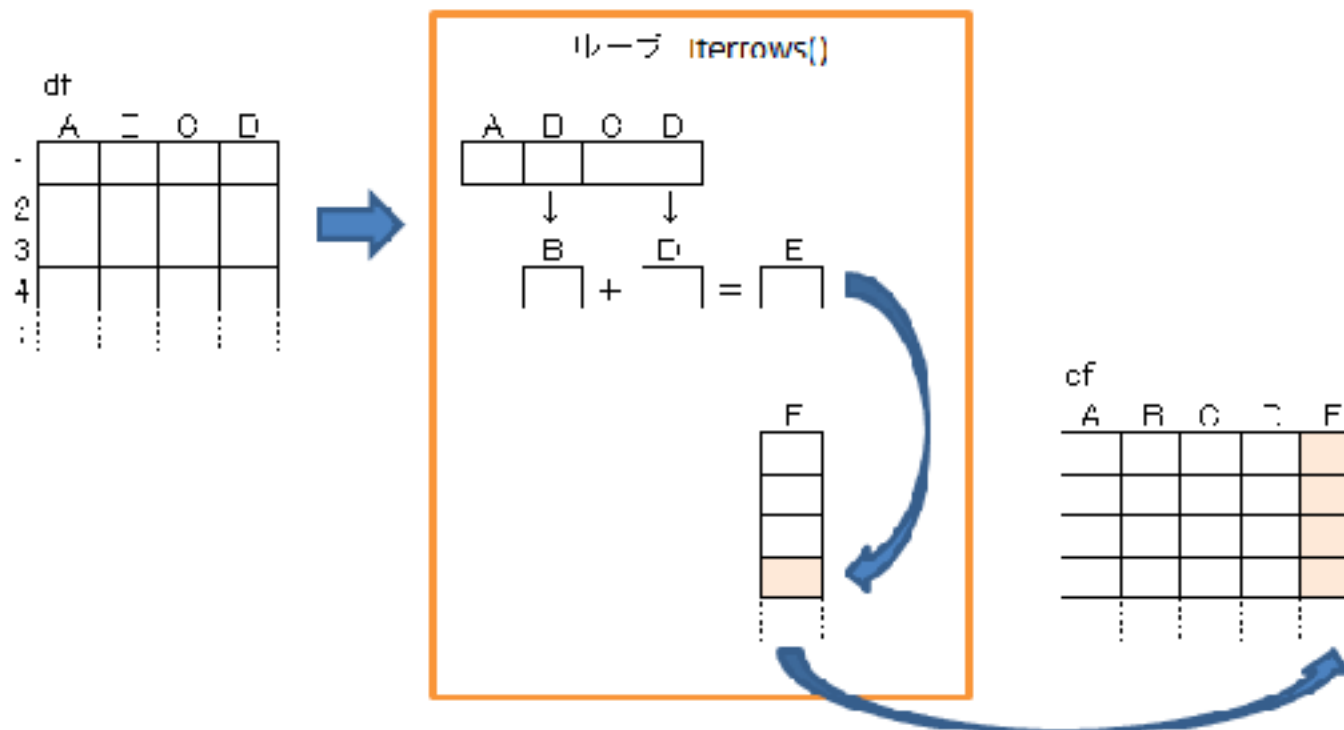


ループによる処理

約2万件で約5分かかった。

データ加工のコツ（ちょっと改善）

```
date_time_ary = []
for i, seri in df.iterrows():
    datetime_str0 = seri.date0 + " " + seri.time0
    dt0 = datetime.strptime(datetime_str0, "%Y-%m-%d %H:%M:%S")
    dt = dt0 + timedelta(hours=9)
    date_time_ary.append(dt.strftime("%Y-%m-%d %H:%M:%S"))
df['date_time'] = date_time_ary
```

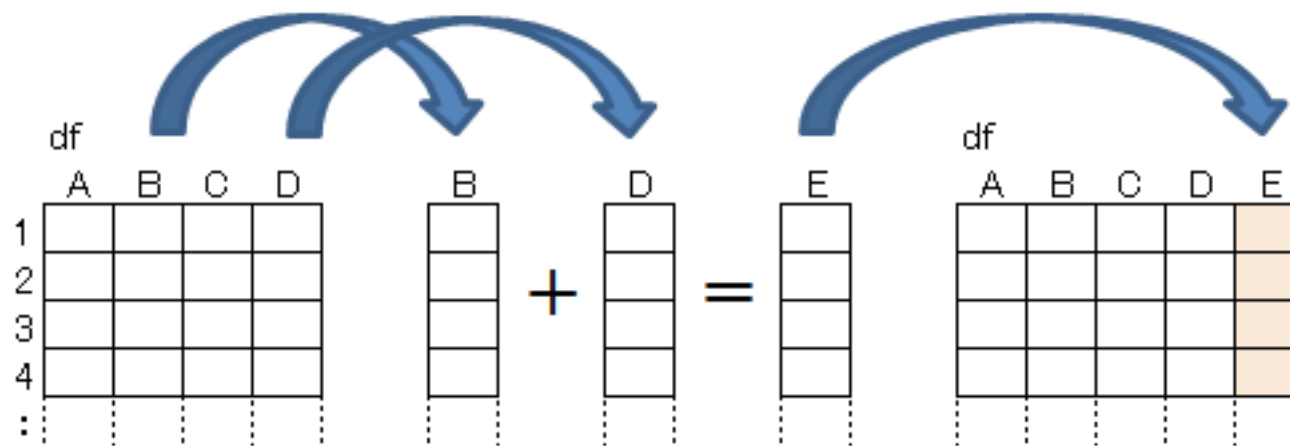


ループ処理ちょっと改良

約2万件で2.4秒。

データ加工のコツ（列ごとに処理する）

```
df['datetime_str0'] = df.date0 + " " + df.time0
df['datetime0'] = df.datetime_str0.apply(lambda dts0: datetime.strptime(dts0, "%Y-%m-%d %H:%M:%S"))
df['datetime1'] = df.datetime0.apply(lambda dt0: dt0 + timedelta(hours=9))
df['date_time'] = df.datetime1.apply(lambda dt: dt.strftime("%Y-%m-%d %H:%M:%S"))
df = df.drop(['datetime_str0', 'datetime0', 'datetime1'], axis=1)
```



列ごとに処理

約2万件で1.4秒。

Appendix

```
>>> df.as_matrix()          # データフレームの配列化
>>> sr.values.flatten()     # シリーズの配列化。こっちのほうが短い→sr.tolist()
>>> df.reset_index(drop=True)    # インデクス番号の振り直し
>>> pd.read_csv(path, nrows=100)  # 先頭から100行だけCSVを読み込み
>>> df.dtypes                 # カラムの型を確認

>>> pd.merge(df, df2, on='key')    # 内部結合
>>> pd.merge(df, df2, on='key', how='left')  # 左側外部結合
>>> pd.merge(df, df2, on='key', how='outer') # 完全外部結合

>>> df.apply(lambda row: row.col1 + row.col2, axis=1) # 複数カラムを使った処理。(axis=1を忘れがち！)

>>> df.astype(str).groupby('b').sum() # NaNのレコードを集計対象に(NaNは'nan'に変換される)
>>> df[df.col1 == df.col1]           # NaNの比較。(col1がNaNでない行を抽出。np.nan == np.nanはFalse)
>>> math.isnan(x)                   # NaNの比較。列丸ごと比較でなければ普通はこう書く。

>>> df.rename(columns={'before': 'after'})    # カラム改名

>>> df.drop(['col1', 'col2'], axis=1) # カラム削除。1列ならこれも→del df['col1']

>>> df.col.str.match('^(\\d*):.*').str[0] # 正規表現 (NaNでエラーにならない)
>>> df.col.str[:2]                       # 部分文字列 (NaNでエラーにならない)

>>> df.columns.values    # カラム名リストを配列化
>>> df.dtypes           # カラム属性を確認

>>> df.sample(sample_count, random_state=seed) # ランダムサンプリング
```


Appendix 2

```
>>> pd.options.display.width = 150          # ipythonで表示幅変更
>>> pd.merge(df, pd.concat([df2]*len(df)))    # 1行のデータフレームをコピーしてマージ
>>> df.fillna(0)                             # NaNを0で埋める
>>> list(df.columns)                         # カラム名リスト
```

\$ ipython --no-autoindent iPython起動時に左記オプションでインデント込みの複数行を貼り付けられるようになる。

エンコードエラーを無視してCSVを読み込む

```
with codecs.open("file/to/path", "r", "Shift-JIS", "ignore") as file:
    df = pd.read_table(file, delimiter=",")
    print(df)
```