Practice

5章 総合問題

【総合問題5-1　コレスキー分解】

右の行列に対して、コレスキー分解を活用して、$Ax = b$の方程式を解いてください。

入力

```
A = np.array([[5, 1, 0, 1],
              [1, 9, -5, 7],
              [0, -5, 8, -3],
              [1, 7, -3, 10]])
b = np.array([2, 10, 5, 10])
```

【総合問題5-2　積分】

$0 \leq x \leq 1$、$0 \leq y \leq 1 - x$の三角領域で定義される以下の関数の積分値を求めてみましょう。

$$\int_0^1 \int_0^{1-x} 1/(\sqrt{(x+y)}(1+x+y)^2) dydx \qquad (式5\text{-}3\text{-}12)$$

【総合問題5-3　最適化問題】

以下の最適化問題をSicpyを使って解いてみましょう。

$$min\ f(x) = x^2 + 1$$
$$s.t. x \geq -1 \qquad (式5\text{-}3\text{-}13)$$

答えはAppendix 2

Chapter 6

Pandasを使ったデータ加工処理

6章では、2章で基礎を学んだPandasについて、さらに詳しく学んでいきます。Pandasは2章で学んだように、ある条件を満たすデータを抽出したり、操作するなど、さまざまな機能があります。さらに、特定の軸で集計したり、データ同士をつなげたり、欠けているデータを補ったり、時系列データを一括計算したり、複雑な処理も柔軟に行うことができます。Pandasは後半の章や、機械学習のモデルを適応させる前のいわゆる前処理でもよく使うことになりますので、この章はしっかりと学習してください。

Goal　Pandasを使ったデータの抽出、操作、処理方法の知識を深める

Chapter 6-1

概要と事前準備

Keyword Pandas、データ加工処理、時系列データ

この章ではPandasを使ったデータ加工処理について、もう少し詳しく学んでいきます。Pandasは2章で学んだように、ある条件を満たすデータを抽出したり、操作したりするなど、さまざまな機能があります。

たとえば、全国の小学校で同じ算数のテストを実施したケースを考えてみます。それぞれの都道府県の最高点取得者だけを抜き出したいこともあるでしょうし、それぞれの都道府県の平均点を出したいこともあるでしょう。このように、さまざまな集計軸があります。さらに、都道府県×学校×クラスの3軸で平均値を算出したい場合や、さらに男女で計算したい場合など、軸が複数になっているケースもあります。Pandasを使えば、そのような集計をすることもできます。また、他のデータ（たとえば、国語の試験結果）とつなげたいときも、キー（各学生に与えられた一意となるデータなど）があれば、結合して1つのDataFrameオブジェクトにして、まとめて処理できます。

そのほか、時系列データを扱うときもPandasは役に立ちます。たとえば、ある店舗の日時の売上推移データを取り扱うときに、1週間や1か月ごとの平均値の推移を簡単に計算することができます。これらのプログラムをいちから記述するとなると大変ですが、Pandasではこのような計算も1～2行ほどのコードを書くだけで実行できます。さらに、データに欠損値や何か異常値が入っているとき、それらを何らかの方法で一括処理したい場合にも使えます。
もちろん、これらの処理は自分で、いちからPythonのプログラムを書くことで対応できますが、実装するのに時間がかかります。それに比べてPandasの機能を使えば、簡単に操作できます。また、機械学習のモデルを構築するときは、そのアルゴリズムが使えるようにデータを前処理する必要があります。たとえば、縦に並んでいたデータのカラムを横に並べたい場面などもあり、そういった操作もPandasなら簡単にできます。

上記のようなデータ操作をする場合、SQLやエクセルのピボットテーブルなどを使っても処理できますが、Pythonのプログラムだけで一貫してコーディングしたい場合はPandasを使うと便利です。
なおPandasには、グラフの描画機能もあり、ハンドリングしたデータをグラフとしてすぐに描画できます。データのグラフ化については7章で見ていくことにします。

6-1-1 この章で使うライブラリのインポート

この章では、2章で紹介した各種ライブラリを使います。次のようにインポートしていることを前提として、以下、進めていきます。

入力

```
# 以下のライブラリを使うので、あらかじめ読み込んでおいてください
import numpy as np
import numpy.random as random
import scipy as sp
import pandas as pd
from pandas import Series, DataFrame

# 可視化ライブラリ
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
%matplotlib inline

# 小数第3位まで表示
%precision 3
```

出力

```
'%.3f'
```

Chapter 6-2

Pandasの基本的なデータ操作

Keyword 階層型インデックス、内部結合、外部結合、縦結合、データのピボット操作、重複データ、マッピング、ビン分割、groupby

まずは、Pandasの基本的なデータ操作から始めます。

6-2-1 階層型インデックス

データを複数軸で集計したいとき、設定すると便利なのが**階層型インデックス**です。

2章でPandasのインデックスについて少し扱いましたが、インデックスとは索引やラベルのようなイメージです。2章では、1つのインデックスだけを扱いましたが、この章の冒頭で説明したように、複数の軸で階層的にインデックスを設定したいこともあります。階層的にインデックスを設定することで、階層ごとに集計が可能になり、便利です。

次に示すデータセットは、インデックスを2段構造で設定した例です。インデックスを設定するには、indexパラメータにその値を指定します。この例では、1階層目のインデックスとしてaとb、2階層目のインデックスとして1と2を設定しています。

また、列の側につけるカラムとして、1階層目にOsaka、Tokyo、Osaka、2階層目にBlue、Red、Redを設定しています。

入力

```
# 3列3行のデータを作成し、インデックスとカラムを設定
hier_df= DataFrame(
    np.arange(9).reshape((3,3)),
    index = [
        ['a','a','b'],
        [1,2,2]
    ],
    columns = [
        ['Osaka','Tokyo','Osaka'],
        ['Blue','Red','Red']
    ]
)
hier_df
```

出力

		Osaka	Tokyo	Osaka
		Blue	Red	Red
a	1	0	1	2
	2	3	4	5
b	2	6	7	8

これらのインデックスやカラムには、名前をつけることもできます。

入力

```
# indexに名前を付ける
hier_df.index.names =['key1','key2']
# カラムに名前を付ける
hier_df.columns.names =['city','color']
hier_df
```

出力

city		Osaka	Tokyo	Osaka
color		Blue	Red	Red
key1	key2			
a	1	0	1	2
	2	3	4	5
b	2	6	7	8

6-2-1-1 カラムの絞り込み

ここでたとえば、カラムのcityがOsakaのデータだけを見たいとしましょう。次のようにすると、グループの絞り込みができます。

入力

```
hier_df['Osaka']
```

出力

	color	Blue	Red
key1	key2		
a	1	0	2
	2	3	5
b	2	6	8

6-2-1-2 インデックスを軸にした集計

次はインデックスを軸にした集計の例です。以下の例は、key2を軸に合計を計算する例です。

入力

```
# 階層ごとの要約統計量：行合計
hier_df.sum(level = 'key2', axis = 0)
```

出力

city	Osaka	Tokyo	Osaka
color	Blue	Red	Red
key2			
1	0	1	2
2	9	11	13

同様にして、colorを軸に合計を計算する場合は、次のようにします。列方向に合計する場合は、axisパラメータを1に設定します。

入力

```
# 列合計
hier_df.sum(level = 'color', axis = 1)
```

出力

	color	Blue	Red
key1	key2		
a	1	0	3
	2	3	9
b	2	6	15

6-2-1-3 インデックスの要素の削除

あるインデックスを削除したい場合は、dropメソッドを使います。dropメソッドを使うと、インデックスの要素を削除できます。次の例では、key1のbを削除しています。

入力

```
hier_df.drop(['b'])
```

出力

	city	Osaka	Tokyo	Osaka
	color	Blue	Red	Red
key1	key2			
a	1	0	1	2
	2	3	4	5

Practice

【練習問題 6-1】

次のデータに対して、Kyotoの列だけ抜き出してみましょう。

入力

```
hier_df1 = DataFrame(
    np.arange(12).reshape((3,4)),
    index = [['c','d','d'],[1,2,1]],
    columns = [
        ['Kyoto','Nagoya','Hokkaido','Kyoto'],
        ['Yellow','Yellow','Red','Blue']
    ]
)

hier_df1.index.names = ['key1','key2']
hier_df1.columns.names = ['city','color']
hier_df1
```

出力

	city	Kyoto	Nagoya	Hokkaido	Kyoto
	color	Yellow	Yellow	Red	Blue
key1	key2				
c	1	0	1	2	3
	2	4	5	6	7
d	1	8	9	10	11

【練習問題 6-2】

練習問題 6-1のデータに対して、cityをまとめて列同士の平均値を出してください。

【練習問題 6-3】

練習問題 6-1のデータに対して、key2ごとに行の合計値を算出してみましょう。

答えはAppendix 2

6-2-2 データの結合

データの結合については2章で少し学びました。データを結合したいケースは多々あり、データをつなげることで集計がしやすくなったり、新しい軸における値がわかったりします。ぜひ、マスターしてください。

ただし、結合と言っても、さまざまなパターンがあります。以下でそれらを紹介していきます。

まずは、この節でサンプルとして使う結合の対象となるデータを準備します。ここでは次に提示するdata1（以下、データ1）とdata2（以下、データ2）の2つのデータを使います。

入力

```
# データ1の準備
data1 = {
    'id': ['100', '101', '102', '103', '104', '106', '108', '110', '111',' 113'],
    'city': ['Tokyo', 'Osaka', 'Kyoto', 'Hokkaido', 'Tokyo', 'Tokyo', 'Osaka', 'Kyoto', 'Hokkaido',
'Tokyo'],
    'birth_year': [1990, 1989, 1992, 1997, 1982, 1991, 1988, 1990, 1995, 1981],
    'name': ['Hiroshi', 'Akiko', 'Yuki', 'Satoru', 'Steeve', 'Mituru', 'Aoi', 'Tarou', 'Suguru','Mitsuo']
}
df1 = DataFrame(data1)
df1
```

出力

	id	city	birth_year	name
0	100	Tokyo	1990	Hiroshi
1	101	Osaka	1989	Akiko
2	102	Kyoto	1992	Yuki
3	103	Hokkaido	1997	Satoru
4	104	Tokyo	1982	Steeve
5	106	Tokyo	1991	Mituru
6	108	Osaka	1988	Aoi
7	110	Kyoto	1990	Tarou
8	111	Hokkaido	1995	Suguru
9	113	Tokyo	1981	Mitsuo

入力

```
# データ2の準備
data2 = {
    'id': ['100', '101', '102', '105', '107'],
    'math': [50, 43, 33, 76, 98],
    'english': [90, 30, 20, 50, 30],
    'sex': ['M','F','F','M','M'],
    'index_num': [0, 1, 2, 3, 4]
}
df2 = DataFrame(data2)
df2
```

出力

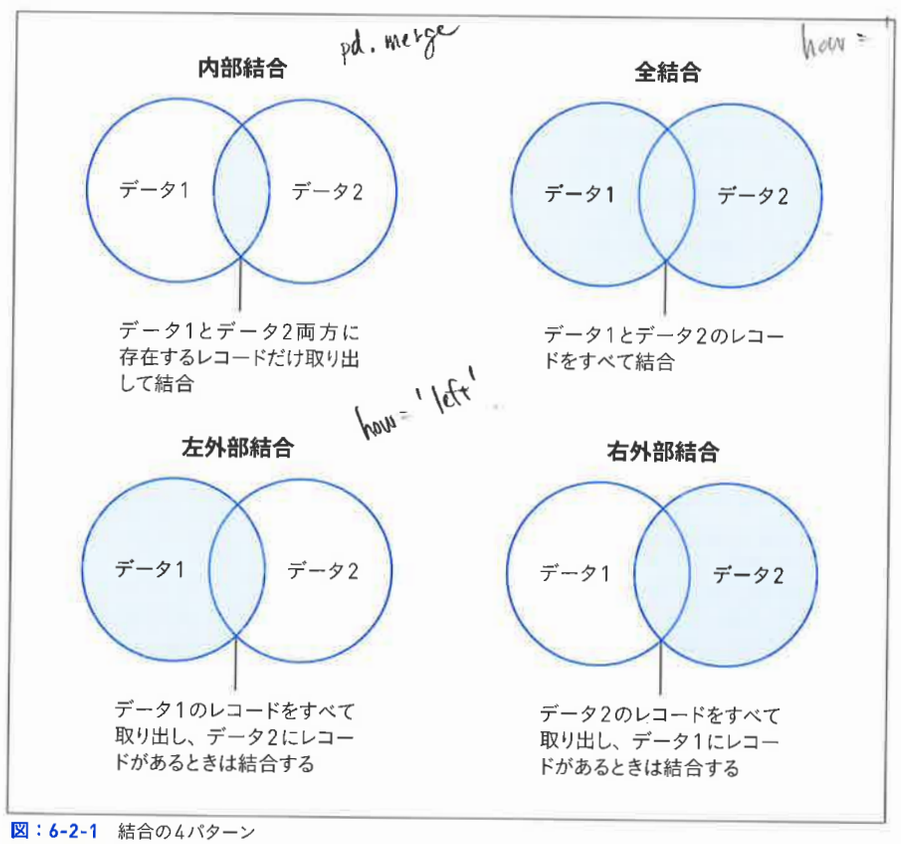
	id	math	english	sex	index_num
0	100	50	90	M	0
1	101	43	30	F	1
2	102	33	20	F	2
3	105	76	50	M	3
4	107	98	30	M	4

6-2-2-1 結合

では、この2つのデータを結合する方法を見ていきましょう。データ1とデータ2を結合する方法は、次の4パターンが考えられます。

①**内部結合（INNER JOIN）**：両方にキーがあるときに結合します。
②**全結合（FULL JOIN）**：どちらかにキーがあるときに結合します。
③**左外部結合（LEFT JOIN）**：左側にあるデータのキーがある時に結合します。
④**右外部結合（RIGHT JOIN）**：右側にあるデータのキーがある時に結合します。

以下では主に、「内部結合」と「(左) 外部結合」を使います。この2つを理解しておいてください。



図：6-2-1　結合の4パターン

6-2-2-2 内部結合

mergeメソッドの結合方法のデフォルトは内部結合です。上記のデータ2つに対して、idをキーとして内部結合すると、以下のようになります。onパラメータでキーを指定します。

入力

```
# データのマージ（内部結合。キーは自動的に認識されるが、onで明示的に指定可能）
print('・結合テーブル')
pd.merge(df1, df2, on = 'id')
```

出力

```
・結合テーブル
```

	id	city	birth_year	name	math	english	sex	index_num
0	100	Tokyo	1990	Hiroshi	50	90	M	0
1	101	Osaka	1989	Akiko	43	30	F	1
2	102	Kyoto	1992	Yuki	33	20	F	2

idの値が両方のDataframeオブジェクトに存在するものみが表示されました。

6-2-2-3 全結合

次の例は、どちらのデータにも存在するデータで結合しています。これが全結合です。全結合ではhowパラメータにouterを指定します。結合する値がない場合は、NaNになります。

入力

```
# データのマージ（全結合）
pd.merge(df1, df2, how = 'outer')
```

出力

	id	city	birth_year	name	math	english	sex	index_num
0	100	Tokyo	1990.0	Hiroshi	50.0	90.0	M	0.0
1	101	Osaka	1989.0	Akiko	43.0	30.0	F	1.0
2	102	Kyoto	1992.0	Yuki	33.0	20.0	F	2.0
3	103	Hokkaido	1997.0	Satoru	NaN	NaN	NaN	NaN
4	104	Tokyo	1982.0	Steeve	NaN	NaN	NaN	NaN
5	106	Tokyo	1991.0	Mituru	NaN	NaN	NaN	NaN
6	108	Osaka	1988.0	Aoi	NaN	NaN	NaN	NaN
7	110	Kyoto	1990.0	Tarou	NaN	NaN	NaN	NaN
8	111	Hokkaido	1995.0	Suguru	NaN	NaN	NaN	NaN
9	113	Tokyo	1981.0	Mitsuo	NaN	NaN	NaN	NaN
10	105	NaN	NaN	NaN	76.0	50.0	M	3.0
11	107	NaN	NaN	NaN	98.0	30.0	M	4.0

なお、left_indexパラメータやright_onパラメータを使うと、キーをインデックスで指定して結合できます。
次の例は、左側のデータのインデックスと、右側のデータのindex_numカラムをキーとして指定するものです。

入力

```
# index によるマージ
pd.merge(df1, df2, left_index = True, right_on = 'index_num')
```

出力

	id_x	city	birth_year	name	id_y	math	english	sex	index_num
0	100	Tokyo	1990	Hiroshi	100	50	90	M	0
1	101	Osaka	1989	Akiko	101	43	30	F	1
2	102	Kyoto	1992	Yuki	102	33	20	F	2
3	103	Hokkaido	1997	Satoru	105	76	50	M	3
4	104	Tokyo	1982	Steeve	107	98	30	M	4

6-2-2-4 左外部結合

左外部結合はhowパラメータにleftを指定します。次の例は、左側のテーブル（ひとつめの引数）に合わせて、Dataframeオブジェクトのデータを結合するものです。左側に対応するデータが右（ふたつめの引数）にない場合は、NaNになります。

入力

```
# データのマージ（left）
pd.merge(df1, df2, how = 'left')
```

出力

	id	city	birth_year	name	math	english	sex	index_num
0	100	Tokyo	1990	Hiroshi	50.0	90.0	M	0.0
1	101	Osaka	1989	Akiko	43.0	30.0	F	1.0
2	102	Kyoto	1992	Yuki	33.0	20.0	F	2.0
3	103	Hokkaido	1997	Satoru	NaN	NaN	NaN	NaN
4	104	Tokyo	1982	Steeve	NaN	NaN	NaN	NaN
5	106	Tokyo	1991	Mituru	NaN	NaN	NaN	NaN
6	108	Osaka	1988	Aoi	NaN	NaN	NaN	NaN
7	110	Kyoto	1990	Tarou	NaN	NaN	NaN	NaN
8	111	Hokkaido	1995	Suguru	NaN	NaN	NaN	NaN
9	113	Tokyo	1981	Mitsuo	NaN	NaN	NaN	NaN

6-2-2-5 縦結合

これまでは、何らかのキーに紐付いてデータをマージしていましたが、concatメソッドを使うと、データを縦方向に積み上げられます。これを縦結合と言います。

入力

```
# データ3の準備
data3 = {
    'id': ['117', '118', '119', '120', '125'],
    'city': ['Chiba', 'Kanagawa', 'Tokyo', 'Fukuoka', 'Okinawa'],
    'birth_year': [1990, 1989, 1992, 1997, 1982],
    'name': ['Suguru', 'Kouichi', 'Satochi', 'Yukie', 'Akari']
}
df3 = DataFrame(data3)
df3
```

出力

	id	city	birth_year	name
0	117	Chiba	1990	Suguru
1	118	Kanagawa	1989	Kouichi
2	119	Tokyo	1992	Satochi
3	120	Fukuoka	1997	Yukie
4	125	Okinawa	1982	Akari

入力

```
# concat 縦結合
concat_data = pd.concat([df1,df3])
concat_data
```

出力

	id	city	birth_year	name
0	100	Tokyo	1990	Hiroshi
1	101	Osaka	1989	Akiko
2	102	Kyoto	1992	Yuki
3	103	Hokkaido	1997	Satoru
4	104	Tokyo	1982	Steeve
5	106	Tokyo	1991	Mituru
6	108	Osaka	1988	Aoi
7	110	Kyoto	1990	Tarou
8	111	Hokkaido	1995	Suguru
9	113	Tokyo	1981	Mitsuo
0	117	Chiba	1990	Suguru
1	118	Kanagawa	1989	Kouichi
2	119	Tokyo	1992	Satochi
3	120	Fukuoka	1997	Yukie
4	125	Okinawa	1982	Akari

Practice

【練習問題 6-4】

下記の2つのデータテーブルに対して、内部結合してみましょう。

入力

```
# データ4の準備
data4 = {
    'id': ['0', '1', '2', '3', '4', '6', '8', '11', '12', '13'],
    'city': ['Tokyo', 'Osaka', 'Kyoto', 'Hokkaido', 'Tokyo', 'Tokyo', 'Osaka', 'Kyoto',
'Hokkaido', 'Tokyo'],
    'birth_year': [1990, 1989, 1992, 1997, 1982, 1991, 1988, 1990, 1995, 1981],
    'name': ['Hiroshi', 'Akiko', 'Yuki', 'Satoru', 'Steeve', 'Mituru', 'Aoi', 'Tarou', 'Suguru',
'Mitsuo']
}
df4 = DataFrame(data4)
df4
```

出力

	id	city	birth_year	name
0	0	Tokyo	1990	Hiroshi
1	1	Osaka	1989	Akiko
2	2	Kyoto	1992	Yuki
3	3	Hokkaido	1997	Satoru
4	4	Tokyo	1982	Steeve
5	6	Tokyo	1991	Mituru
6	8	Osaka	1988	Aoi
7	11	Kyoto	1990	Tarou
8	12	Hokkaido	1995	Suguru
9	13	Tokyo	1981	Mitsuo

入力

```
# データ5の準備
data5 = {
    'id': ['0', '1', '3', '6', '8'],
    'math' : [20, 30, 50, 70, 90],
    'english': [30, 50, 50, 70, 20],
    'sex': ['M', 'F', 'F', 'M', 'M'],
    'index_num': [0, 1, 2, 3, 4]
}
df5 = DataFrame(data5)
df5
```

出力

	id	math	english	sex	index_num
0	0	20	30	M	0
1	1	30	50	F	1
2	3	50	50	F	2
3	6	70	70	M	3
4	8	90	20	M	4

【練習問題 6-5】

練習問題 6-4のデータを使って、df4をベースにdf5のテーブルを全結合してみましょう。

【練習問題 6-6】

練習問題 6-4のデータを使って、df4に対して、以下のデータを縦結合してみましょう。

入力

```
# データの準備
data6 = {
    'id': ['70', '80', '90', '120', '150'],
    'city': ['Chiba', 'Kanagawa', 'Tokyo', 'Fukuoka', 'Okinawa'],
    'birth_year': [1980, 1999, 1995, 1994, 1994],
    'name': ['Suguru', 'Kouichi', 'Satochi', 'Yukie', 'Akari']
}
df6 = DataFrame(data6)
```

答えはAppendix 2

6-2-3 データの操作と変換

次に、データの操作と変換（ピボット操作、データの重複があった場合の処理、マッピング、ビン分割など）について扱っていきましょう。

6-2-3-1 ピボット操作

まずは、データのピボット操作について学びます。ピボット操作とは、行を列に、列を行にする操作です。もう一度、これまで使ってきた階層テーブルhier_dfを例に考えます。

入力

```
# hier_dfを用意
hier_df= DataFrame(
    np.arange(9).reshape((3, 3)),
    index = [
        ['a', 'a', 'b'],
        [1, 2, 2]
    ],
    columns = [
        ['Osaka', 'Tokyo', 'Osaka'],
        ['Blue','Red','Red']
    ]
)
hier_df
```

出力

		Osaka	Tokyo	Osaka
		Blue	Red	Red
a	1	0	1	2
	2	3	4	5
b	2	6	7	8

次のようにstackメソッドを実行すると、行と列が入れ替わったDataFrameオブジェクトを再構成できます。

入力

```
# ピボット操作で「Blue、Red」の列を行に変更
hier_df.stack()
```

出力

			Osaka	Tokyo
a	1	Blue	0	NaN
		Red	2	1.0
	2	Blue	3	NaN
		Red	5	4.0
b	2	Blue	6	NaN
		Red	8	7.0

unstackメソッドを使うと、逆の操作が可能です。

入力

```
# unstackメソッドで、「Blue、Red」の行を列に変更
hier_df.stack().unstack()
```

出力

		Osaka		Tokyo	
		Blue	Red	Blue	Red
a	1	0	2	NaN	1.0
	2	3	5	NaN	4.0
b	2	6	8	NaN	7.0

上記のデータ操作では、列にあったものを行に持ってきたり、行であったものを列に持ってきたりしています。
これらのテクニックは、データのモデリング前の処理として使うことも多く便利ですので、ぜひ理解して使えるようしてください。

6-2-3-2 重複データの除去

次は、重複があるデータの処理です。データ分析をしていると、データに重複があることもありますし、自分で実際に集計等していて重複が混じることもあり、そのチェックをするという意味で重要です。
まず例として、重複があるデータを準備します。

入力

```
# 重複があるデータ
dupli_data = DataFrame({
        'col1': [1, 1, 2, 3, 4, 4, 6, 6],
        'col2': ['a', 'b', 'b', 'b', 'c', 'c', 'b', 'b']
})
print('・元のデータ')
dupli_data
```

出力

・元のデータ

	col1	col2
0	1	a
1	1	b
2	2	b
3	3	b
4	4	c
5	4	c
6	6	b
7	6	b

重複の判定にはduplicatedメソッドを使います。それぞれの行が確認され、重複があるときは、Trueとなります。ただし、重複のあるデータでも1回目ではFalseとなり、2回目からTrueになります。

入力

```
# 重複判定
dupli_data.duplicated()
```

出力

```
0    False
1    False
2    False
3    False
4    False
5     True
6    False
7     True
dtype: bool
```

drop_duplicatesメソッドを使うと、重複したデータを削除した結果のデータが返されます。

入力

```
# 重複削除
dupli_data.drop_duplicates()
```

出力

	col1	col2
0	1	a
1	1	b
2	2	b
3	3	b
4	4	c
6	6	b

6-2-3-3 マッピング処理

次に、マッピング処理を説明します。これは、Excelのvlookup関数のような処理です。共通のキーとなるデータに対して、一方の(参照)テーブルからそのキーに対応するデータを引っ張ってくる機能です。以下は、都道府県名と地域名を対応付けた参照データです。

・Tokyo(東京)→Kanto(関東)
・Hokkaido(北海道)→Hokkaido(北海道)
・Osaka(大阪)→Kansai(関西)
・Kyoto(京都)→Kansai(関西)

まず次のように参照データを作ります。

入力

```
# 参照データ
city_map ={
    'Tokyo': 'Kanto',
    'Hokkaido': 'Hokkaido',
    'Osaka': 'Kansai',
    'Kyoto':'Kansai'
}
city_map
```

出力

```
{'Tokyo': 'Kanto',
'Hokkaido': 'Hokkaido',
'Osaka': 'Kansai',
'Kyoto': 'Kansai'}
```

次の例は、df1のcityカラムをベースとして、上の参照データcity_mapから対応する地域名データを持ってきて、新しく一番右にregionというカラムとして追加するものです。

入力

```
# 参照データを結合
# もし対応するデータがなかったら、NaNになる。
df1['region'] = df1['city'].map(city_map)
df1
```

出力

	id	city	birth_year	name	region
0	100	Tokyo	1990	Hiroshi	Kanto
1	101	Osaka	1989	Akiko	Kansai
2	102	Kyoto	1992	Yuki	Kansai
3	103	Hokkaido	1997	Satoru	Hokkaido
4	104	Tokyo	1982	Steeve	Kanto
5	106	Tokyo	1991	Mituru	Kanto
6	108	Osaka	1988	Aoi	Kansai
7	110	Kyoto	1990	Tarou	Kansai
8	111	Hokkaido	1995	Suguru	Hokkaido
9	113	Tokyo	1981	Mitsuo	Kanto

6-2-3-4 無名関数とmapを組み合わせる

次は、1章で学んだ無名関数とmapを使って、カラムの中の一部のデータを取り出す処理をする例です。具体的には、birth_yearの上3桁を取得します。関数適応やループなどを使って要素を1つ1つ取り出して処理するより便利なので、まとめて処理したい場合は、このようなやり方を検討することをおすすめします。

入力

```
# birth_year の上3つの数字・文字を取り出す
df1['up_two_num'] = df1['birth_year'].map(lambda x: str(x)[0:3])
df1
```

出力

	id	city	birth_year	name	region	up_two_num
0	100	Tokyo	1990	Hiroshi	Kanto	199
1	101	Osaka	1989	Akiko	Kansai	198
2	102	Kyoto	1992	Yuki	Kansai	199
3	103	Hokkaido	1997	Satoru	Hokkaido	199
4	104	Tokyo	1982	Steeve	Kanto	198
5	106	Tokyo	1991	Mituru	Kanto	199
6	108	Osaka	1988	Aoi	Kansai	198
7	110	Kyoto	1990	Tarou	Kansai	199
8	111	Hokkaido	1995	Suguru	Hokkaido	199
9	113	Tokyo	1981	Mitsuo	Kanto	198

6-2-3-5 ビン分割

最後にビン分割について説明します。これは、ある離散的な範囲にデータを分割して集計したい場合に、便利な機能です。具体的には、上のデータのbirth_yearに対して、5年区切りで集計をしたい場合など、ある特定の分割をして計算をしたいときに使います。

たとえば以下のように、1980、1985、1990、1995、2000のように5年単位でビン分割するためのリストを用意し、Pandasのcut関数を使うと、そのように分割できます。cut関数では、1つ目の引数に分割するデータ、2つ目の引数に分割する境界値を、それぞれ指定します。

入力

```
# 分割の粒度
birth_year_bins = [1980, 1985, 1990, 1995, 2000]

# ビン分割の実施
birth_year_cut_data = pd.cut(df1.birth_year, birth_year_bins)
birth_year_cut_data
```

出力

```
0    (1985, 1990]
1    (1985, 1990]
2    (1990, 1995]
3    (1995, 2000]
4    (1980, 1985]
5    (1990, 1995]
6    (1985, 1990]
7    (1985, 1990]
8    (1990, 1995]
9    (1980, 1985]
Name: birth_year, dtype: category
Categories (4, interval[int64]): [(1980, 1985] < (1985, 1990] < (1990, 1995] < (1995, 2000]]
```

なお、上記のプログラムでは、「1980～1985」の区切りの中には1980は含まれませんが、1985は含まれています。つまり、指定した基準は、「～より後で、～以前」という区切り方として使われます。この動作は、cut関数にleftオプションやrightオプションを指定することで変更できます。

上記の結果を使って、それぞれの数を集計したい場合は、value_counts関数を使います。

入力

```
# 集計結果
pd.value_counts(birth_year_cut_data)
```

出力

```
(1985, 1990]    4
(1990, 1995]    3
(1980, 1985]    2
(1995, 2000]    1
Name: birth_year, dtype: int64
```

labelsパラメータを指定することで、それぞれのビンに名前をつけることもできます。

入力

```
# 名前をつける
group_names = ['early1980s', 'late1980s', 'early1990s', 'late1990s']
birth_year_cut_data = pd.cut(df1.birth_year, birth_year_bins, labels = group_names)
pd.value_counts(birth_year_cut_data)
```

出力

```
late1980s     4
early1990s    3
early1980s    2
late1990s     1
Name: birth_year, dtype: int64
```

上記では、ビン分割のリストを用意しましたが、あらかじめ分割数を指定したい場合は、以下のように設定できます。なお、データによってはきれいに割り切れず小数点以下がでてくるので注意しましょう。

入力

```
# 数字で分割数指定可能。ここでは2つに分割
pd.cut(df1.birth_year, 2)
```

出力

```
0      (1989.0, 1997.0]
1    (1980.984, 1989.0]
2      (1989.0, 1997.0]
3      (1989.0, 1997.0]
4    (1980.984, 1989.0]
5      (1989.0, 1997.0]
6    (1980.984, 1989.0]
7      (1989.0, 1997.0]
8      (1989.0, 1997.0]
9    (1980.984, 1989.0]
Name: birth_year, dtype: category
Categories (2, interval[float64]): [(1980.984, 1989.0] < (1989.0, 1997.0]]
```

またqcut関数を使うと、分位点での分割もできます。qcut関数を使うことで、ほぼ同じサイズのビンを作成することができます。

入力

```
pd.value_counts(pd.qcut(df1.birth_year, 2))
```

出力

```
(1980.999, 1990.0]    6
(1990.0, 1997.0]      4
Name: birth_year, dtype: int64
```

ここでは対象としたデータが、1981、1982、1988、1989、1990、1990、1991、1992、1995、1997と、ちょうど中央値にあたる値が2つあるため、6つと4つで分割されました。

このビン分割、はじめ何に使うのかイメージがわきにくいかもしれませんが、具体的には、顧客の購買金額合計を分けて、それぞれの顧客層（優良顧客など）を分析したい場合など、マーケティング分析にも使えます。次の7章の総合問題演習で扱っていくことにしましょう。

Practice

【練習問題 6-7】

3章で使用した数学の成績を示すデータである「student-mat.csv」を読み込み、年齢（age）を2倍にしたカラムを末尾に追加してみましょう。

【練習問題 6-8】

練習問題 6-7と同じデータで、「absences」のカラムについて、以下の3つのビンに分けてそれぞれの人数を数えてみましょう。なお、cutのデフォルトの挙動は右側が開区間です。今回は、cut関数に対してright=Falseのオプションを指定して、右側を開区間としてください。

(0,5]
開 閉

入力

```
# 分割の粒度
absences_bins = [0,1,5,100]
```

【練習問題 6-9】

上記と同じデータで、「absences」のカラムについて、qcut関数を用いて3つのビンに分けてみましょう。

答えはAppendix 2

6-2-4 データの集約とグループ演算

ここでは、あるカラムを軸にして集計する処理を学びます。

2章で少し扱いましたが、groupbyメソッドを使うことで、ある変数を軸として、その単位で集計処理をします。以前使ったdf1データを対象に、集約やグループ演算をしたいと思います。

入力

```
# データを用意（確認）、ただし、region付き
df1
```

出力

	id	city	birth_year	name	region	up_two_num
0	100	Tokyo	1990	Hiroshi	Kanto	199
1	101	Osaka	1989	Akiko	Kansai	198
2	102	Kyoto	1992	Yuki	Kansai	199
3	103	Hokkaido	1997	Satoru	Hokkaido	199
4	104	Tokyo	1982	Steeve	Kanto	198
5	106	Tokyo	1991	Mituru	Kanto	199

```
6   108      Osaka       1988       Aoi     Kansai       198
7   110      Kyoto       1990     Tarou     Kansai       199
8   111   Hokkaido       1995    Suguru  Hokkaido       199
9   113      Tokyo       1981    Mitsuo     Kanto       198
```

以下のようにgroupbyメソッドでグループ化してからsizeメソッドを使うと、それぞれのcityの値がいくつかあるのかを計算できます。

入力

```
# サイズ情報
df1.groupby('city').size()
```

出力

```
city
Hokkaido    2
Kyoto       2
Osaka       2
Tokyo       4
dtype: int64
```

次は、cityを軸として、birth_yearの平均値を算出する例です。

入力

```
# cityを軸に、birth_yearの平均値を求める
df1.groupby('city')['birth_year'].mean()
```

出力

```
city
Hokkaido    1996.0
Kyoto       1991.0
Osaka       1988.5
Tokyo       1986.0
Name: birth_year, dtype: float64
```

軸は複数設定することもできます。たとえば、region、cityを2軸として、birth_yearの平均値を求めると、次のようになります。

入力

```
df1.groupby(['region', 'city'])['birth_year'].mean()
```

出力

```
region    city
Hokkaido  Hokkaido    1996.0
Kansai    Kyoto       1991.0
          Osaka       1988.5
Kanto     Tokyo       1986.0
Name: birth_year, dtype: float64
```

なお、groupbyメソッドにas_index = Falseパラメータを設定すると、インデックスが設定されなくなります。そのままテーブルとして扱いたいときに便利です。

入力

```
df1.groupby(['region', 'city'], as_index = False)['birth_year'].mean()
```

出力

```
     region      city  birth_year
0  Hokkaido  Hokkaido      1996.0
1    Kansai     Kyoto      1991.0
2    Kansai     Osaka      1988.5
3    Kanto      Tokyo      1986.0
```

他にもgroupbyメソッドには、イテレータという、反復的に値を取り出す機能があり、次のように、結果の要素をPythonのforなどでループ処理できて便利です。

以下の例は、groupはregionの名前取り出し、subdfはそのregionのみの行をすべて抽出するというものです。

入力

```
for group, subdf in df1.groupby('region'):
    print('=========================================================')
    print('Region Name:{0}'.format(group))
    print(subdf)
```

出力

```
=========================================================
Region Name:Hokkaido
    id      city  birth_year    name    region up_two_num
3  103  Hokkaido        1997  Satoru  Hokkaido        199
8  111  Hokkaido        1995  Suguru  Hokkaido        199
=========================================================
Region Name:Kansai
    id    city  birth_year   name  region up_two_num
1  101   Osaka        1989  Akiko  Kansai        198
2  102   Kyoto        1992   Yuki  Kansai        199
6  108   Osaka        1988    Aoi  Kansai        198
7  110   Kyoto        1990  Tarou  Kansai        199
=========================================================
Region Name:Kanto
     id    city  birth_year     name region up_two_num
0   100   Tokyo        1990  Hiroshi  Kanto        199
4   104   Tokyo        1982   Steeve  Kanto        198
5   106   Tokyo        1991   Mituru  Kanto        199
9   113   Tokyo        1981   Mitsuo  Kanto        198
```

データに対して、複数の計算をまとめて行いたいときには、aggメソッドを使うと便利です。aggメソッドの引数には、実行したい関数名のリストを渡します。

以下は、カウント、平均、最大、最小を計算する例です。

なお、以下の例では、対象データとして3章で扱ったstudent-mat.csvを使って計算しています。このデータがあるファイルディレクトリに移動して、データを読み込んで実行してください。

入力

```
# 3章で用意したデータがあるpathに移動してください。例）cd ＜3章のデータがあるpath＞
# 以下を実行
student_data_math = pd.read_csv('student-mat.csv', sep = ';')

# 列に複数の関数を適応
functions = ['count','mean','max','min']
grouped_student_math_data1 = student_data_math.groupby(['sex','address'])
grouped_student_math_data1['age','G1'].agg(functions)
```

出力

		age				G1			
		count	mean	max	min	count	mean	max	min
sex	address								
F	R	44	16.977273	19	15	44	10.295455	19	6
	U	164	16.664634	20	15	164	10.707317	18	4
M	R	44	17.113636	21	15	44	10.659091	18	3
	U	143	16.517483	22	15	143	11.405594	19	5

Practice

【練習問題 6-10】

練習問題 6-7で使用した「student-mat.csv」を使って、Pandasの集計処理をしてみましょう。まずは、学校（school）を軸にして、G1の平均点をそれぞれ求めてみましょう。

【練習問題 6-11】

練習問題 6-7で使用した「student-mat.csv」を使って、学校（school）と性別（sex）を軸にして、G1、G2、G3の平均点をそれぞれ求めてみましょう。

【練習問題 6-12】

練習問題 6-7で使用した「student-mat.csv」を使って、学校（school）と性別（sex）を軸にして、G1、G2、G3の最大値、最小値をまとめて算出してみましょう。

答えはAppendix 2

Chapter 6-3

欠損データと異常値の取り扱いの基礎

Keyword リストワイズ削除、ペアワイズ削除、平均値代入法、異常値、箱ひげ図、パーセンタイル、VaR（Value At Risk）

データを扱っていると必ずといっていいほど、欠損しているデータや異常値データの存在があります。この節では、基礎の基礎レベルで欠損データや異常データについての判定や扱い方について学ぶことにします。もっと深く学びたい方は、ぜひ参考文献「A-12」を読んでください。

6-3-1 欠損データの扱い方

まずは、欠損データの取り扱いについてです。データの欠損は、入力忘れ、無回答、システム上の問題などさまざまな要因があります。「ない」データについては、無視をするのがいいのか、除外をするのがいいのか、もっともらしい値を入れるのがいいのか、それが問題です。アプローチによっては、大きなバイアスのある結果を与え、誤った意思決定につながり、大きな損失につながる可能性もあります。慎重に扱っていきましょう。

この節では、次のようなデータをサンプルとして扱います。値をNaN（NA）にした部分が欠損データであるとして、以下、説明を続けます。

入力

```
# データの準備
import numpy as np
from numpy import nan as NA
import pandas as pd

df = pd.DataFrame(np.random.rand(10, 4))

# NAにする
df.iloc[1,0] = NA
df.iloc[2:3,2] = NA
df.iloc[5:,3] = NA
```

入力

```
df
```

出力

	0	1	2	3
0	0.485775	0.042397	0.539116	0.926647
1	NaN	0.470748	0.241323	0.103007
2	0.618467	0.910260	NaN	0.090963
3	0.319467	0.553239	0.057040	0.206173
4	0.888791	0.291158	0.775008	0.779764
5	0.034683	0.458730	0.632387	NaN
6	0.358828	0.230845	0.016502	NaN
7	0.461881	0.963180	0.937040	NaN
8	0.874005	0.825269	0.115018	NaN
9	0.271005	0.462655	0.799126	NaN

※以下、ランダム生成のため、紙面と実際は異なります

以下では、この擬似的な欠損データに対して、削除や0や直前の数字、平均値等で穴埋めをしていきます。本書では、これらの単純な方法のみ紹介しますが、他の方法として、最尤推定法で推定したり、回帰代入やScipyで実施したスプライン補間などもあります。注意が必要なのは、これらの方法がバイアスを生む可能性があることです。ここで紹介する方法がベストであるとはいえません。深く学びたい方はぜひ参考文献「A-12」などを読んで、欠損データを埋める方法への理解を深めてください。

6-3-1-1 リストワイズ削除

NaNがある行をすべて取り除くには、dropnaメソッドを使います。これを**リストワイズ削除**といいます。以下は、先ほどのデータにおいて、dropnaメソッドを適用し、すべてのカラムにデータがある行だけを抽出したものです。NaNがある行は除外されます。

入力

```
df.dropna()
```

出力

	0	1	2	3
0	0.485775	0.042397	0.539116	0.926647
3	0.319467	0.553239	0.057040	0.206173
4	0.888791	0.291158	0.775008	0.779764

6-3-1-2 ペアワイズ削除

この結果からわかるように、リストワイズ削除では元々10行あったデータが極端に少なくなって、データが全く使えないという状況が考えられます。このとき、欠損している列のデータを無視して、利用可能なデータのみ（例：列の0番目と1番目のみ存在）を使う方法があります。これを**ペアワイズ削除**といいます。ペアワイズ削除では、使いたい列を取り出してからdropnaメソッドを適用します。

入力

```
df[[0,1]].dropna()
```

出力

	0	1
0	0.485775	0.042397
2	0.618467	0.910260
3	0.319467	0.553239
4	0.888791	0.291158
5	0.034683	0.458730
6	0.358828	0.230845
7	0.461881	0.963180
8	0.874005	0.825269
9	0.271005	0.462655

6-3-1-3 fillnaで埋める

他の処理として、fillna（値）で、NaNになっている箇所をある値で埋める方法もあります。たとえばNaNを0として扱うケースです。次のようにfillna(0)とすると、NaNが0に置き変わります。

入力

```
df.fillna(0)
```

出力

	0	1	2	3
0	0.485775	0.042397	0.539116	0.926647
1	0.000000	0.470748	0.241323	0.103007
2	0.618467	0.910260	0.000000	0.090963
3	0.319467	0.553239	0.057040	0.206173
4	0.888791	0.291158	0.775008	0.779764
5	0.034683	0.458730	0.632387	0.000000
6	0.358828	0.230845	0.016502	0.000000
7	0.461881	0.963180	0.937040	0.000000
8	0.874005	0.825269	0.115018	0.000000
9	0.271005	0.462655	0.799126	0.000000

※以下、実際の出力には囲みはありません

6-3-1-4 前の値で埋める

ffillメソッドを適用すると、直前の行の値で埋めることができます。具体的には、2行1列目（インデックス「1」/カラム「0」の値）は先ほど

```
df.iloc[1,0] = NA
```

でNAにしましたが、直前の1行1列目の値は、0.485775でしたので、この値で埋めることができます。この処理は金融の時系列データの処理などで使うことができ、便利です。

入力

```
df.fillna(method = 'ffill')
```

出力

	0	1	2	3
0	0.485775	0.042397	0.539116	0.926647
1	0.485775	0.470748	0.241323	0.103007
2	0.618467	0.910260	0.241323	0.090963
3	0.319467	0.553239	0.057040	0.206173
4	0.888791	0.291158	0.775008	0.779764
5	0.034683	0.458730	0.632387	0.779764
6	0.358828	0.230845	0.016502	0.779764
7	0.461881	0.963180	0.937040	0.779764
8	0.874005	0.825269	0.115018	0.779764
9	0.271005	0.462655	0.799126	0.779764

6-3-1-5 平均値で埋める

他に、平均値で穴埋めする方法もあります。これを**平均値代入法**といい、meanメソッドを使います。なお、注意点として、時系列データを扱う際、この方法は未来情報を含むことがある（過去に欠損したデータを、未来のデータを使った平均値で埋める）ので、気を付けましょう。

入力

```
# 各カラムの平均値（確認用）
df.mean()
```

出力

```
0    0.479211
1    0.520848
2    0.456951
3    0.421311
dtype: float64
```

入力

```
# 平均値で埋める
df.fillna(df.mean())
```

出力

	0	1	2	3
0	0.485775	0.042397	0.539116	0.926647
1	0.479211	0.470748	0.241323	0.103007
2	0.618467	0.910260	0.456951	0.090963
3	0.319467	0.553239	0.057040	0.206173
4	0.888791	0.291158	0.775008	0.779764
5	0.034683	0.458730	0.632387	0.421311
6	0.358828	0.230845	0.016502	0.421311
7	0.461881	0.963180	0.937040	0.421311
8	0.874005	0.825269	0.115018	0.421311
9	0.271005	0.462655	0.799126	0.421311

他にも色々とオプションがあるので、?df.fillna等で調べてみてください。

欠損データについて、ここではサンプルデータにおいて、一定の値を機械的に置換しました。ただし、これらの方法はいつも使えるというわけではありません。データの状況、背景等を考え、適切に対処することが重要です。

Practice

【練習問題 6-13】

以下のデータに対して、1列でもNaNがある場合は削除し、その結果を表示してください。

入力

```
# データの準備
import numpy as np
from numpy import nan as NA
import pandas as pd

df2 = pd.DataFrame(np.random.rand(15,6))

# NAにする
df2.iloc[2,0] = NA
df2.iloc[5:8,2] = NA
df2.iloc[7:9,3] = NA
df2.iloc[10,5] = NA

df2
```

出力

	0	1	2	3	4	5
0	0.415247	0.550350	0.557778	0.383570	0.482254	0.142117
1	0.066697	0.908009	0.197264	0.227380	0.291084	0.305750
2	NaN	0.481305	0.963701	0.289538	0.662069	0.883058
3	0.469084	0.717253	0.467172	0.661786	0.539626	0.862264
4	0.314643	0.129364	0.291149	0.210694	0.891432	0.583443
5	0.672456	0.111327	NaN	0.197844	0.361385	0.703919
6	0.943599	0.047140	NaN	0.222312	0.270678	0.985113
7	0.172857	0.359706	NaN	NaN	0.559918	0.181495
8	0.650042	0.845300	NaN	NaN	0.706246	0.634860
9	0.696152	0.353721	0.999253	NaN	0.616951	0.278251
10	0.126199	0.791196	0.856410	0.959452	0.826969	NaN
11	0.700689	0.894851	0.918055	0.108752	0.502343	0.749123
12	0.393294	0.468172	0.711183	0.725584	0.355825	0.562409
13	0.403318	0.076329	0.642033	0.344418	0.453335	0.916017
14	0.898894	0.926813	0.620625	0.089307	0.362026	0.497475

※ランダム生成のため、紙面と実際は異なります

【練習問題 6-14】

練習問題6-13で準備したデータに対して、NaNを0で埋めてください。

【練習問題 6-15】

練習問題6-13で準備したデータに対して、NaNをそれぞれの列の平均値で埋めてください。

答えはAppendix 2

6-3-2 異常データの扱い方

次は、異常値（外れ値）についてです。異常値データの扱いは、そのままにして何もしないのか、異常値を除去するか、もっともらしい値に入れかえて使うかが問題になります。

そもそも異常値とは一体何でしょうか。実は、統一的な見解というものはなく、そのデータを扱うアナリストや意思決定者が判断することもあります。ビジネスの現場では、不正アクセスのパターン（セキュリティ分野）や機械の故障、金融リスク管理（VaR）など、さまざまな分野で使われており、それぞれ色々な方法でアプローチされています。

異常値検出のアプローチには、単純に箱ひげ図などを書いて、あるパーセンタイル以上のデータを異常値としてみなす方法、正規分布を利用する方法、データの空間的な近さに基づく方法などがあります。他には以降の章で学ぶ機械学習（教師なし学習も含む）を用いた方法もあります。

ここでは特に練習問題はありませんが、興味のある方はぜひ巻末の参考文献「A-13」や参考URL「B-15」などで学んでください。

また、異常値の分野に関連して、極端な値を研究する極値統計学という分野もあります。データの中で大きな値をとる極値データの挙動について、さまざまな研究がなされており、稀ではありますがそれが起きれば非常に大きな影響を及ぼす現象（自然現象、災害など）を研究します。気象学だけではなく、ファイナンスや情報通信の分野でも応用されているので、興味のある方は参考文献「A-14」などを参照してみてください。

以上で、欠損値と異常値の扱いについてはこれで終わりになります。データ分析において、データの前処理が8割だとよく言われ、欠損データや異常値データには、たびたび遭遇します。また、世の中には実にさまざまな形式のデータが存在し、それらを整えるだけでも大変な作業です。ここで紹介したテクニックも重要ですが、それらに対してどのように対処していくのか戦略を立てることも重要です。参考文献「A-15」にも、ぜひ目を通してみてください。

Chapter 6-4

時系列データの取り扱いの基礎

Keyword リサンプリング、シフト、移動平均

最後にPandasを使った時系列データの取り扱いについて学びます。ここでは、サンプルとして為替の時系列データを扱います。あらかじめAppendixを参考にpandas-datareaderというライブラリをダウンロードしてインストールしてから進めてください。
インストールしたら、次のようにインポートしてください。

入力

```
import pandas_datareader.data as pdr
```

6-4-1 時系列データの処理と変換

ここでは、サンプルデータに含まれる2001/1/2から2016/12/30までのドル円の為替レートデータ（DEXJPUS）を使います。日ごとのレートデータで、欠損している日（休日など）もあります。

入力

```
start_date = '2001/1/2'
end_date = '2016/12/30'

fx_jpusdata = pdr.DataReader('DEXJPUS', 'fred', start_date, end_date)
```

headメソッドを使って、読み込んだfx_jpusdataの先頭5行を読み出します。

入力

```
fx_jpusdata.head()
```

出力

DATE	DEXJPUS
2001-01-02	114.73
2001-01-03	114.26
2001-01-04	115.47
2001-01-05	116.19
2001-01-08	115.97

サンプルには、15年分のデータがありますが、これをどう分析するかはそのビジネスニーズ次第です。たとえば、最後の2016年の4月のデータだけ欲しいこともありますし、月末のレートだけを見たいこともあります。さらに、上記では、2001/1/6はデータとしてありませんが、それを前日の値で埋めたいこともありますし、前の日と比べてどれだけレートが上がったのか調べたい場合もあるでしょう。これらのことはすべてPandasで簡単に計算することができます。

6-4-1-1 特定の年月のデータを参照する

まずは、特定の年月のデータを参照する方法です。2016年の4月のデータだけ見たい場合は、以下のように年月を指定します。

入力

```
fx_jpusdata['2016-04']
```

出力

DATE	DEXJPUS
2016-04-01	112.06
2016-04-04	111.18
2016-04-05	110.26
2016-04-06	109.63
2016-04-07	107.98
2016-04-08	108.36
2016-04-11	107.96
2016-04-12	108.54
2016-04-13	109.21
2016-04-14	109.20
2016-04-15	108.76
2016-04-18	108.85
2016-04-19	109.16
2016-04-20	109.51
2016-04-21	109.41
2016-04-22	111.50
2016-04-25	111.08
2016-04-26	111.23
2016-04-27	111.26
2016-04-28	108.55
2016-04-29	106.90

そのほか、特定の年や日にちにだけ抽出することもできます。次に、月末レートだけ取り出してみましょう。resampleメソッドの引数にMを指定することで、月ごとのデータを取り出し、lastメソッドで末尾のデータを取り出しています。具体的には、以下の結果をみるとわかる通り、1月、2月、3月…の月末のレートを取り出せます。

入力

```
fx_jpusdata.resample('M').last().head()
```

出力

DATE	DEXJPUS
2001-01-31	116.39
2001-02-28	117.28
2001-03-31	125.54
2001-04-30	123.57
2001-05-31	118.88

日付を取り出したい場合は「D」、年を取り出したい場合は「Y」を、それぞれ引数に指定します。このように、ある頻度のデータを、別の頻度のデータで取り出し直す処理をリサンプリングといいます。また、最後のデータではなく、その平均を計算したい場合はmeanメソッドを使うことで計算できます。他にもいろいろとパラメータを設定できるので、必要な処理があるときに、調べてみてください。

6-4-1-2 欠損がある場合の操作

次に、時系列データに欠損がある場合の処理をみていきます。欠損処理については、前の節でも扱った通り、さまざまな方法があります。先ほどのレートでは、2001/1/6がまずレコードとして存在していませんでしたが、日ごとにデータを用意したいときは、先ほどのリサンプリングを行います。具体的には、以下のようにします。

入力

```
fx_jpusdata.resample('D').last().head()
```

出力

DATE	DEXJPUS
2001-01-02	114.73
2001-01-03	114.26
2001-01-04	115.47
2001-01-05	116.19
2001-01-06	NaN

上記より、2001/1/6は空のままなので、前の日の値で埋める処理をします。ここでは、次に示すようにffillメソッドを使います。

入力

```
fx_jpusdata.resample('D').ffill().head()
```

出力

DATE	DEXJPUS
2001-01-02	114.73
2001-01-03	114.26
2001-01-04	115.47
2001-01-05	116.19
2001-01-06	116.19

6-4-1-3 データをズラして比率を計算する

次に、前日とのレート比較をしたい場合を考えます。上の例でいうと、2001-01-02のレートは114.73で、2001-01-03のレートは114.26になり、その比率を計算することもできますが、それをすべての日付について適応させる処理をします。shiftメソッドを使うことで、インデックスは固定したまま、データだけをずらすことができます。以下はデータを1つあとにずらしており、2001-01-02のレートは114.73でしたが、2001-01-03のレートとして扱われるようになります。

入力

```
fx_jpusdata.shift(1).head()
```

出力

DATE	DEXJPUS
2001-01-02	NaN
2001-01-03	114.73
2001-01-04	114.26
2001-01-05	115.47
2001-01-08	116.19

このように加工すると、前日のレートと当日のレートの比率を一気に算出することができます。これがPandasを使うメリットです。なお、以下で2001-01-02がNaNになっているのは、その前日のデータがもともとないためです。

入力

```
fx_jpusdata_ratio = fx_jpusdata / fx_jpusdata.shift(1)
fx_jpusdata_ratio.head()
```

出力

DATE	DEXJPUS
2001-01-02	NaN
2001-01-03	0.995903
2001-01-04	1.010590
2001-01-05	1.006235
2001-01-08	0.998107

なお、差分や比率を取る方法については、diffやpct_changeなどもありますので、興味がある方は調べてみてください。

Let's Try

diffやpct_changeについて、それらの機能を調べて、使ってみましょう。

Practice

【練習問題 6-16】

「6-4-1」で読み込んだfx_jpusdataを使って、年ごとの平均値の推移データを作成してください。

答えはAppendix 2

6-4-2 移動平均

次に、時系列のデータ処理でよく使われる移動平均の処理方法をみていきます。さきほど扱ったfx_jpusdataのデータについて、3日間の移動平均線を作成することを考えます。まず先頭から5行のデータを取り出してみます。

入力

```
fx_jpusdata.head()
```

出力

DATE	DEXJPUS
2001-01-02	114.73
2001-01-03	114.26
2001-01-04	115.47
2001-01-05	116.19
2001-01-08	115.97

結果を見るとわかるように、2001-01-04までのデータは、2001-01-02が114.73、2001-01-03が114.26、2001-01-04が115.47ですから、その平均を計算すると114.82です。

同様にして、2001-01-05、2001-01-06と続けて計算をしていきます。それにはPandasのrollingメソッドを使うと、簡単に計算できます。以下は、その3日間の移動平均を計算した結果です。rollingメソッドを実行した後に、meanメソッドを使って平均を計算しています。

入力

```
fx_jpusdata.rolling(3).mean().head()
```

出力

DATE	DEXJPUS
2001-01-02	NaN
2001-01-03	NaN
2001-01-04	114.820000
2001-01-05	115.306667
2001-01-08	115.876667

移動平均ではなく標準偏差の推移を算出したいのなら、meanメソッドの代わりにstdメソッドを使います。
以下は3日間の標準偏差の推移です。

入力

```
fx_jpusdata.rolling(3).std().head()
```

出力

DATE	DEXJPUS
2001-01-02	NaN
2001-01-03	NaN
2001-01-04	0.610000
2001-01-05	0.975312
2001-01-08	0.368963

rollingメソッドには、パラメータが他にもいろいろとありますので、必要に応じて調べて実行してみてください。

以上で、Pandasの章は終了です。一部、なかなかイメージを掴みにくい箇所もあったかもしれません。しかし、実際に「こんな感じでデータ加工や変換したいのになあ」と思ったときに、ここを参考にしてプログラミングをしてみてください。データ加工処理のニーズが出てきて、実際に使うことで一層理解が進む箇所かもしれません。ここで紹介したテクニックはほんの一部です。この他にも、さまざまなデータ処理・加工方法があるので、参考文献「A-10」などを読んで、手を動かして実行してみてください。

Let's Try

ここで扱った集計軸以外にも、対象データに対していろいろな軸で処理をしてみましょう。

Practice

【練習問題 6-17】

練習問題 6-16で使用したfx_jpusdataを使って、20日間の移動平均データを作成してください。ただしNaNは削除してください。なお、レコードとして存在しないデータであれば、特に補填する必要はありません。

答えはAppendix 2

Practice

6章 総合問題

【総合問題6-1　データ操作】

3章で使用した、数学の成績を示すデータである「student-mat.csv」を使って、以下の問いに答えてください。

1. 上記のデータに対して、年齢（age）×性別（sex）でG1の平均点を算出し、縦軸が年齢（age）、横軸が性別（sex）となるような表（テーブル）を作成しましょう。

2. 1.で表示した結果テーブルについて、NaNになっている行（レコード）をすべて削除した結果を表示しましょう。

答えはAppendix 2

Chapter 7

Matplotlibを使ったデータ可視化

この章では、2章で基礎を学んだMatplotlibについて、さらに深く学びます。2章では折れ線グラフやヒストグラムを扱いましたが、ここでは棒グラフや円グラフ、バブルチャートの作成方法について学びます。
そして、この章の最後に今までの総合問題として、時系列データの分析とマーケティングの分析の問題を用意しています。これまで学んだ手法を試せる機会ですので、ぜひチャレンジしてみてください。

Goal　Matplotlibを使って、さまざまなデータを可視化することができる。この章の総合問題が解ける