Python3による
データ処理の基礎

pandas / NumPy / SciPy / matplotlib /
SQLite / SQLAlchemy

第0.8版

中村勝則

2026年2月12日

免責事項

本書の内容は参考的資料であり，掲載したプログラムリストは全て試作品である．本書の使用に伴って発生した不利益，損害の一切の責任を筆者は負わない．

目 次

1　はじめに

本書は，Python を用いてデータ処理を行うための最も基本的な事柄について解説するものである．Python にはデータ処理のための多くの優れたソフトウェアライブラリが公開されており，本書では代表的なライブラリの基本的な使用方法について解説する．特に，データ処理のための主要なライブラリとして pandas，NumPy，SciPy を取り上げる．また，データの可視化には matplotlib を取り上げる．データベースのアクセスに関しても SQLite を対象として，SQLAlchemy ライブラリによる基本的なアクセス方法について解説する．

1.1　本書を読むに当たって

本書は pandas ライブラリの基本的な扱いに関する内容に重点を置いている．また SciPy に関しては，統計処理に必要な部分である scipy.stats ライブラリの基本的な扱いに関する内容に重点を置いている．本書の内容を読み進めるためには Python 言語に関する基本的な知識が必須である．また，NumPy と matplotlib に関しても基本的な知識を持っていると本書を読み進めることが容易になる．

Python 言語の基本的な事柄に関しては他の情報源（書籍[1]，Python の公式インターネットサイトなど）を参照のこと．また，NumPy と matplotlib に関しても，より詳しい内容に関しては他の情報源[2] を参照のこと．

1.2　作業環境について

実際のデータ処理の現場では，Jupyter や IPython といった高機能な対話環境を用いることが多い．特に Jupyter の Notebook は実行の履歴管理や図表の表示において利便性が大きい．ただし，本書では簡素な形で解説するため，処理の実行例を示す場合も，その入力部分と出力部分のみを示すことが多い．従って，実際のシステムにおける実行結果の表示の詳細に関しては，適宜読者の方々に判断していただきたい．もちろん Jupyter や IPython を使用せずに本書の実行例を試すことも可能であり，OS のターミナル機能（コマンドウィンドウ）でも十分に実行可能である．

1.3　pandas の読込み

pandas は次のような形式で Python 言語処理系に読み込むことが一般的である．

例. pandas ライブラリの読込み

```
import pandas as pd
```

このように，短縮名「pd」を与えて利用することが一般的である．また，ライブラリのバージョンは次のようにして確認することができる．

例. pandas ライブラリのバージョンの確認

入力：

```
import pandas as pd
pd.__version__              ←バージョン情報
```

出力：　`'2.3.3'`　　←文字列（`str`）の形式でバージョン情報が得られる

2　データ構造

データ処理に必要となる最も基本的なものが**データ構造**である．データ構造の形式としては，要素を 1 列に並べた **1 次元のデータ構造**と，要素を縦横（行列）に並べた **2 次元のデータ構造**の 2 種類が主として扱われる．

Python は言語の仕様として各種のデータ構造を提供しており，多次元のデータを扱うためのデータ構造として最も基本的なものとして**リスト**が挙げられる．ただし，言語処理系としての Python の処理速度は決して早いとは言えず，多量のデータ[3] を扱う場合は，処理の早い各種のソフトウェアライブラリに頼ることとなる．その場合，処理の対象となるデータは，処理を担うライブラリが独自に提供するデータ構造として扱われる．そのような理由から，Python 処理系の上でデータ処理を行う場合は，使用するライブラリとそれが提供するデータ構造（データ型）を意識しなけ

[1] 拙書「Python 3 入門 - Kivy による GUI アプリケーション開発，サウンド入出力，ウェブスクレイピング」でも解説しています．

[2] 拙書「Python 3 ライブラリブック - 各種ライブラリの基本的な使用方法」でも解説しています．

[3] いわゆる**ビッグデータ**

ればならない.

Python 上でのデータ処理において重要なデータ構造を整理すると概ね表 1 のような分類となる.

表 1: Python 上でのデータ処理のためのデータ型（特に重要なもの）

使用するライブラリ	データ型	特徴	処理速度
なし（Python 本来のデータ構造）	リスト	基本的なデータ列．多次元配列として使用可	遅い（※）
	辞書	キーと値のペアを記録する構造	遅い（※）
	セット	集合論に基づく処理に適したデータ構造	遅い（※）
NumPy / SciPy	ndarray	NumPy 独自の多次元配列．主に数値を扱う	非常に高速
pandas	Series	pandas 独自の 1 次元データ列	非常に高速
	DataFrame	pandas 独自の 2 次元データ配列（データ表）	非常に高速

※ Python 処理系自体の実行速度

実際のデータ処理においては，各ライブラリが提供する機能（関数，メソッド）を使用するので，Python 処理系に複数のライブラリを同時に読み込んで使用することが多くなる．従って，必要に応じて扱うデータの型を目的のライブラリに合わせて変換する作業が求められることも多い.

pandas は NumPy の機能を応用して作られたライブラリであり，pandas のデータ構造の内部の要素は NumPy の配列データ（ndarray 型）であることが多い．従って，それらに対しては基本的に NumPy が提供する各種の関数やメソッドが使用できる.

2.1 Series（pandas のデータ構造）

pandas が提供する Series は 1 次元のデータ構造である．後に解説する DataFrame を用いる場合も，その行や列（カラム）のデータを抽出して使用する際に，この Series の形式で取り扱う局面が多い．その意味で Series は基本的かつ重要なデータ構造である.

Python の基本的データ構造であるリストと違い，Series では各要素に**インデックス**と呼ばれる識別情報が付けられる．もちろんリストを Series に変換することもできる.

例. リストを Series に変換

入力：
```
import pandas as pd                  # pandas を 'pd' という名で読込み
lst1 = [1,2,3,4,5]                    # 1〜5 の数列のリスト
lst2 = [x**2 for x in lst1]           # 上記の要素をそれぞれ 2 乗したリスト
sr = pd.Series( lst2, index=lst1 )   # Series に変換（インデックスに lst1 を与える）
sr                                    # 内容確認
```

出力：
```
1    1
2    4
3    9
4   16
5   25
dtype:  int64
```

この例ではリスト lst2 を pandas の Series である sr に変換している．この時，各要素のインデックスとして lst1 の要素を与えている．sr 自体は 1 次元のデータであるが，表示するとインデックス情報とデータの併せて 2 列の内容（左がインデックス，右がデータ列）が確認できる.

《Series の生成》

書き方： Series(データ列, index=インデックスとして与えるデータ列)

キーワード引数 'index=' を省略したり値として None を与えると，0 から始まる整数がインデックスとして自動的に与えられる.「データ列」としては，リストや NumPy の配列（ndarray）などを与えることができる.

与えるデータ列を省略する（あるいは None を与える）と空の Series オブジェクトが得られる.

▲注意▲

Series の要素としては様々な型（数値，文字列など）の値を与えることができる．ただし，各種の統計処理に Series のデータを使用するに当たって，全要素の型を揃えておいた方が処理速度の面で有利になることがある．これは，pandas の機能が NumPy の機能を応用して構築されていることに由来する．特に数値処理の対象として Series のデータを扱う際に，異なる型の値が含まれていると速度低下をもたらすことがある．

2.1.1 Series の要素のデータ型

Series は要素として**整数**，**浮動小数点数**，**文字列**をはじめとする様々な値を持つことができる．ただし，1 つの Series が保持する要素は全て同じ型に限られる．Series の要素の代表的な型を表 2 に挙げる．

表 2: Series の要素の型（一部）

カテゴリ	NumPy 系	ExtensionArray 系
整数	`'int8', 'int16', 'int32', 'int64'`	`'Int8', 'Int16', 'Int32', 'Int64'`
浮動小数点数	`'float16', 'float32', 'float64'`	`'Float32', 'Float64'`
真理値	`'bool'`	`'boolean'`
文字列	`'object'`*	`'string'`

* 'object' は数値以外のデータを幅広く扱う型

先に示した例では，Series オブジェクトを作成する際の初期値として整数のリストを与えており，作成された Series の全要素は 'int64' となっていた．本書では，データ構造作成時に型の指定を省略した形でプログラムの実行例を示すことが多いが，実際には，データ構造を作成する際に，そのデータ型を明示的に指定する方が安全である．

表 2 にあるように，データ型には **NumPy 系**と **ExtensionArray 系**の 2 つの系統がある．これらは pandas のデータ構造を支える**バックエンド**システムに由来するものであり，標準的には，計算処理に有利な NumPy 系を使用する．また，後に解説する**欠損値**を柔軟に扱う場合は ExtensionArray 系を使用[4] する．

例. 型を指定して先の例と同様の Series を作成する（先の例の続き）

入力：
```
sr = pd.Series( lst2, index=lst1, dtype='int64' )
```

参考）NumPy 系の型指定には「`np.` **型名**」という形式を用いることもできる．

2.1.2 インデックスに基づくアクセス

Series の要素には**インデックス**を指定してアクセスする（下記参照）ことができる．

《**at，loc**》
書き方 1：　Series オブジェクト.at[インデックス]
　指定した「インデックス」の要素にアクセスする．
書き方 2：　Series オブジェクト.loc[インデックス 1 : インデックス 2]
　「インデックス 1」から「インデックス 2」までの部分にアクセスする．

▲注意▲

ここで言う「インデックス」は Series オブジェクト独自のもの（生成時に 'index=' の引数で与えられたもの）であり，Python 組込みの**リスト**や**タプル**における格納位置としてのインデックスのことではないことに常に注意すること．特にこのことは，リストに対するスライスの指定と比較すると明確に理解できる．

例. at による要素の参照（先の例の続き）

入力：
```
print( sr.at[2] )          # 指定したインデックス位置の要素の取り出し
print( type(sr.at[2]) )    # データ型の調査
```

[4]「2.2.2.4 整数の欠損値について」（p.27）で具体的に解説する．

出力：
```
4
<class 'numpy.int64'>
```

これは Series の要素を 1 つ参照した例であり，得られたデータの型が NumPy で扱う数値であることがわかる．これに対して lst2 にスライスを付けて内容にアクセスする例を示して比較する．

例. リストに対するスライスの指定（先の例の続き）

入力：
```
print(lst2)
lst2[2]
```

出力：
```
[1, 4, 9, 16, 25]
9
```

先の例で `sr.at[2]` とした場合と比べると，アクセスできる位置が異なることがわかる．

例. loc による指定範囲の参照（先の例の続き）

入力：
```
print( sr.loc[2:4] )        # 指定したインデックス範囲の取り出し
print( type(sr.loc[2:4]) )  # データ型の調査
```

出力：
```
2   4
3   9
4  16
dtype:  int64
<class 'pandas.core.series.Series'>
```

これは Series の指定した範囲の部分を取り出した例であり，得られたデータ型は Series であることがわかる．これに対して lst2 にスライスを付けて内容にアクセスする例を示して比較する．

例. リストに対するスライスの指定（先の例の続き）

入力：
```
print(lst2)
lst2[2:4]
```

出力：
```
[1, 4, 9, 16, 25]
[9, 16]
```

先の例で `sr.loc[2:4]` とした場合と比べると，アクセスできる位置と範囲が異なることがわかる．

at, loc で指定した部分の要素の書き換えも可能である.（次の例参照）

例. at で指定した位置の要素の書き換え（先の例の続き）

入力：
```
sr.at[2] = 999  # インデックスが 2 の要素を書き換える
sr              # 内容確認
```

出力：
```
1    1
2  999
3    9
4   16
5   25
dtype:  int64
```

例. loc で指定した範囲の要素の書き換え（先の例の続き）

入力：
```
sr.loc[2:4] = [555,666,777]    # 連続した領域を書き換える
sr                             # 内容確認
```

出力：
```
1    1
2  555
3  666
4  777
5   25
dtype:  int64
```

2.1.2.1 ドット '.' 表記によるアクセス

インデックスの要素が文字列型である場合は，Series オブジェクトに**ドット** '.' でインデックス要素をつなげることで，当該インデックスの要素にアクセスすることができる．

参考）ドット表記による要素へのアクセスの利便性はあるものの，この方法を優先的に使用することは
あまり推奨されない．

例. ドット表記による要素へのアクセス（先の例の続き）

入力：
```
words = pd.Series([' りんご',' みかん',' ぶどう'],index=['apple','orange','grape'])
words
```

出力：
```
apple    りんご
orange   みかん
grape    ぶどう
dtype:  object
```

入力：
```
print( words.apple )    # words のインデックス 'apple' の要素
print( words.grape )    # words のインデックス 'grape' の要素
```

出力：
```
りんご
ぶどう
```

ただしドット表記では，新規要素の追加はできない．(次の例参照)

例. ドット表記による新規要素追加の試み（先の例の続き）

入力：
```
words.watermelon = ' すいか'     # 新規要素の追加を試みる
words    # 内容確認
```

出力：
```
apple    りんご
orange   みかん
grape    ぶどう
dtype:  object
```

この例から，ドット表記では新規要素の追加ができないことがわかる．要素の新規追加には前述の at などを使用する．

例. 新規要素の追加の例（先の例の続き）

入力：
```
words.at['watermelon'] = ' すいか'  # 新規要素の追加
words    # 内容確認
```

出力：
```
apple        りんご
orange       みかん
grape        ぶどう
watermelon   すいか
dtype:  object
```

既存のインデックスに関しては，ドット表記で値を変更することができる．(次の例参照)

例. ドット表記による既存のインデックスの値の変更（先の例の続き）

入力：
```
words.grape = ' 葡萄'       # 既存のインデックス'grape' の値を変更
words    # 内容確認
```

出力：
```
apple        りんご
orange       みかん
grape          葡萄
watermelon   すいか
dtype:  object
```

インデックスが 'grape' の要素の値が，' ぶどう' から ' 葡萄' に変更されている．

2.1.3 格納順位に基づくアクセス

Series の要素には格納順位[5] を指定してアクセスする（下記参照）ことができる．

《**iat，iloc**》
書き方 1：　Series オブジェクト.iat[格納順位]
指定した「格納順位」の要素にアクセスする．
書き方 2：　Series オブジェクト.iloc[格納順位 1 : 格納順位 2]
「格納順位 1」以上「格納順位 2」**未満**の範囲にアクセスする．

「格納順位」とは，Series の要素の並びの順位のことであり，0 から始まる整数の番号である．これは Python のリストにアクセスする際の**スライス**と共通する付番方式である．すなわち，

```
d = ['a','b','c','d']
```

として作られたリスト d の要素にスライスを与えてアクセスすると対応する要素は次のようになる．

スライス指定	d[0]	d[1]	d[2]	d[3]
対応する要素	'a'	'b'	'c'	'd'

また，先のリスト d に対してスライスを付けて d[1:3] と範囲指定するとこれは

```
['b','c']
```

となることに注意しなければならない．すなわち，スライス $[n_1, n_2]$ の指定は n_1 以上 n_2 **未満**を意味し，結果として n_1 番目から $n_2 - 1$ 番目までの範囲となる．Series の iat や iloc に指定する格納順位もこれと同じ付番となる．実行例を次に示す．

例. iat による要素の参照（先の例の続き）

入力：
```
print(sr)
sr.iat[4]
```

出力：
```
1      1
2    555
3    666
4    777
5     25                ←格納順位（位置インデックス）の 4 番目
dtype:  int64
np.int64(25)           ←格納順位（位置インデックス）の 4 番目が得られている
```

先頭から 5 番目の要素を指し示していることに注意すること．また，実行環境によっては np.int64(25) ではなく単に 25 と表示されることがある．

例. iloc による指定範囲の参照（先の例の続き）

入力：
```
sr.iloc[1:4]
```

出力：
```
2  555
3  666
4  777
dtype:  int64
```

▲注意▲
- インデックスによる範囲指定 '$[i_1, i_2]$' では「i_1 から i_2 まで」を意味する．(loc)
- 格納順位による範囲指定 '$[n_1, n_2]$' では「n_1 から $n_2 - 1$ まで」を意味する．(iloc)

[5] **位置インデックス**（positional index）

2.1.4 スライスを用いたアクセス方法

2.1.4.1 スライスに整数値を与える場合

リストに対してするように，Series オブジェクトにスライスを付けて要素にアクセスすることができる．

例. スライスによる単純なアクセス（先の例の続き）

入力：
```
sr01 = pd.Series([2,4,6,8,10],index=[0,1,2,3,4])     # 0 から始まるインデックス
sr02 = pd.Series([2,4,6,8,10],index=[1,2,3,4,5])     # 自然数のインデックス
sr03 = pd.Series([2,4,6,8,10],index=[-2,-1,0,1,2])   # 負の値を含むインデックス
print( sr01[0:2] )          # 以降，スライス [0:2] を指定する試み
print( '--------' )
print( sr02[0:2] )
print( '--------' )
print( sr03[0:2] )
```

出力：
```
0    2
1    4
dtype:  int64
--------
1    2
2    4
dtype:  int64
--------
-2    2
-1    4
dtype:  int64
```

この例では，異なるインデックスを持つ 3 つの Series オブジェクト sr01，sr02，sr03 の 0 番目から 2 番目未満の要素を取り出している．Series オブジェクトのインデックスとは無関係に，格納順位のみに基づいて要素が取り出されていることがわかる．この場合，スライス

$[i_s : i_e]$

の指定において，i_s〜 $i_e - 1$ の範囲がアクセス対象となることに注意すること．

2.1.4.2 スライスに非整数のインデックス項目を与える場合

スライス内に整数でないインデックスの範囲を指定することができる（次の例）

例. インデックスの要素が整数値でない場合（先の例の続き）

入力：
```
sr04 = pd.Series([2,4,6,8,10],index=['0','1','2','3','4']) # 非整数インデックス (1)
sr05 = pd.Series([2,4,6,8,10],index=['a','b','c','d','e']) # 非整数インデックス (2)
print( sr04['0':'2'] )
print( '--------' )
print( sr05['a':'c'] )
```

出力：
```
0    2
1    4
2    6
dtype:  int64
--------
a    2
b    4
c    6
dtype:  int64
```

この例は，インデックスの各要素が文字列型の場合のものである．この場合，スライス

$[i_s : i_e]$

の指定において，i_s〜 i_e の範囲がアクセス対象となることに注意すること．また，スライスに非整数のインデックス項目を与えることは，誤解の元になることがあるので，あまり推奨されない．

2.1.5 要素の抽出

take メソッドを使用することで，指定した格納位置の複数の要素を一度に取り出すことができる．先の例で作成した Series オブジェクト sr を例に挙げて解説する．

例. sr の値の確認（先の例の続き）

入力：
```
sr
```

出力：
```
1     1              ←格納位置 0
2   555              ←格納位置 1
3   666              ←格納位置 2
4   777              ←格納位置 3
5    25              ←格納位置 4
dtype:  int64
```

この sr の要素の内，格納位置が 0，2，4 のものを take メソッドで抽出する例を次に示す．

例. take による要素の抽出（先の例の続き）

入力：
```
idx = [0,2,4]     # 要素の格納位置のリスト
sr.take( idx )    # 抽出
```

出力：
```
1     1
3   666
5    25
dtype:  int64
```

指定した要素が一度に得られていることがわかる．

2.1.5.1 マスクを用いた抽出

iloc にはスライスを与えることができるが，真理値の列を与えてデータを選択的に抽出することもできる．(次の例参照)

例. 真理値の列でデータを抽出（先の例の続き）

入力：
```
msk = [True,True,False,False,True]   # 抽出対象の位置に True を対応させる
sr.iloc[msk]     # iloc に与える
```

出力：
```
1     1
2   555
5    25
dtype:  int64
```

この例では msk に真理値の列を与えている．msk の要素の位置が sr の要素の位置に対応しており，抽出したい部分が True になっている．真理値の列としては，リストだけでなく Series オブジェクトでも良い．

より簡便な方法として，真理値の列を Series オブジェクトにスライスとして与える方法がある．

例. 真理値の列をスライスとして与える（先の例の続き）

入力：
```
sr[msk]     # スライスに直接与える
```

出力：
```
1     1
2   555
5    25
dtype:  int64
```

2.1.5.2 条件式から真理値の列を生成する方法

与えた条件を満たす要素の位置を調べるには，Series オブジェクトを用いた条件式を評価すると良い．

例. 条件式から真理値の列を生成する（先の例の続き）

入力：
```
sr < 100     # 条件を満たす要素の位置を調べる
```

出力：
```
1    True
2   False
3   False
4   False
5    True
dtype:  bool
```

この例では「100 未満の要素の位置」を意味する真理値列を得ている．

これの応用により，Series のスライスに直接条件式を与えて要素を抽出することができる．

例. スライスに条件式を与えて要素を抽出（先の例の続き）

入力：
```
sr[ sr < 100 ]   # スライスに条件式を記述する
```

出力：
```
1    1
5   25
dtype:  int64
```

注意）pandas 2.x の版では iat，iloc，at に条件式を与えることはできないので注意すること．（次の例）

例. iloc に条件式を与える試み（先の例の続き）

入力：
```
sr.iloc[ sr < 100 ]
```

出力：
```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
<ipython-input-19-10458c05ef3f> in <module>
----> 1 sr.iloc[ sr < 100 ]
        ⋮
      （途中省略）
        ⋮
NotImplementedError:  iLocation based boolean indexing on an integer type is not available
```

このようにエラーが発生する．ただし，pandas 3.x の版ではこれが実行できる．

次に loc に条件式を与える例を示す．

例. loc に条件式を与える試み（先の例の続き）（先の例の続き）

入力：
```
sr.loc[ sr < 100 ]
```

出力：
```
1    1
5   25
dtype:  int64
```

正しく結果が得られていることがわかる．

参考）Series オブジェクトにスライスの形式で真理値並びを与えるのは糖衣構文であり，‘.loc[真理値並び]’ を付ける方が正式な形式である．

2.1.6 重複するインデックスを持つ Series の扱い

Series には重複するインデックスを与える[6] ことができる．その場合の要素へのアクセスについて，例を挙げて説明する．

例. 重複するインデックスを持つ Series（先の例の続き）

入力：
```
ix2 = ['x','y','y','y','z']          # インデックス用のデータ列（重複あり）
sr2 = pd.Series( lst2, index=ix2 )   # Series に変換（インデックスに ix2 を与える）
sr2                                  # 内容確認
```

[6] これはあまり良いことではない．

出力：
```
x   1
y   4
y   9
y  16
z  25
dtype:  int64
```

このようにして作成した Series オブジェクト sr2 に対して at や loc を用いる例を次に示す.

例. at に重複するインデックスを指定する（先の例の続き）

入力：
```
print( sr2.at['y'] )        # 重複するインデックスを持つ要素の取り出し (1)
```

出力：
```
y   4
y   9
y  16
dtype:  int64
```

この例からわかるように，at で要素にアクセスすると重複するインデックスを持つ要素の Series が得られる．また loc を用いた場合も結果は Series の形で得られる.

例. loc に重複するインデックスを指定する（先の例の続き）

入力：
```
print( sr2.loc['y'] )        # 重複するインデックスを持つ要素の取り出し (2)
```

出力：
```
y   4
y   9
y  16
dtype:  int64
```

入力：
```
print( sr2.loc['x':'y'] )      # 重複するインデックスを持つ要素の取り出し (3)
```

出力：
```
x   1
y   4
y   9
y  16
dtype:  int64
```

2.1.7 Series から ndarray への変換

Series が保持するデータを NumPy の ndarray（NumPy の基本的な配列データ形式）に変換する方法について例を挙げて説明する.

例. Series から ndarray への変換（先の例の続き）

入力：
```
print( sr2.values )         # データを ndarray として取り出し
print( type(sr2.values) )   # データ型の調査
```

出力：
```
[ 1 4 9 16 25]
<class 'numpy.ndarray'>
```

この例のように Series オブジェクトの values プロパティとしてデータ列（ndarray 型）が得られる[7] .

注意）新しい版の pandas では Series オブジェクトの values プロパティの使用は**非推奨**となっており，Series オブジェクトに対して to_numpy メソッドを使用して ndarray を取得するべきである.（次の例）

例. Series から ndarray への変換（先の例の続き）

入力：
```
print( sr2.to_numpy() )         # データを ndarray として取り出し
print( type(sr2.to_numpy()) )   # データ型の調査
```

出力：
```
[ 1 4 9 16 25]
<class 'numpy.ndarray'>
```

[7] pandas 独自の ExtensionArray を返す場合もある.

参考）to_numpy メソッドには，戻り値の配列の型を指定する ‘dtype=’ 引数を与えることができる．

2.1.7.1 データとしてのインデックス

Series オブジェクトのインデックスをデータとして取り出すことも可能である．（次の例参照）

例. Series からインデックスを取り出す（先の例の続き）

入力：
```
print( sr2.index )          # インデックスの取り出し
print( type(sr2.index) )    # データ型の調査
```

出力：
```
Index(['x', 'y', 'y', 'y', 'z'], dtype='object')
<class 'pandas.core.indexes.base.Index'>
```

この例のように Series オブジェクトの index プロパティとしてインデックスのデータ列が得られる．得られたデータ列の型は Index である．Index オブジェクトはスライスによるアクセスができる．（次の例参照）

例. Index オブジェクトの要素へのアクセス（先の例の続き）

入力：
```
ix = sr2.index              # Index オブジェクトの取り出し
print( '0 番目:', ix[0] )
print( '1 番目以降:', ix[1:]  )
```

出力：
```
0 番目:  x
1 番目以降:  Index(['y', 'y', 'y', 'z'], dtype='object')
```

Index 型のオブジェクトを ndarray に変換するには，先と同様に values プロパティを参照する．（次の例参照）

例. インデックスを ndarray に変換する（先の例の続き）

入力：
```
print( sr2.index.values )          # インデックスを ndarray として取り出し
print( type(sr2.index.values) )    # データ型の調査
```

出力：
```
['x' 'y' 'y' 'y' 'z']
<class 'numpy.ndarray'>
```

注意）新しい版の pandas では Index オブジェクトの values プロパティの使用は**非推奨**となっており，Index オブジェクトに対して to_numpy メソッドを使用して ndarray を取得するべきである．（次の例）

例. インデックスを ndarray に変換する（先の例の続き）

入力：
```
print( sr2.index.to_numpy() )         # インデックスを ndarray として取り出し
print( type(sr2.index.to_numpy()) )   # データ型の調査
```

出力：
```
['x' 'y' 'y' 'y' 'z']
<class 'numpy.ndarray'>
```

2.1.7.2 インデックスの検索

あるインデックスを持つ要素がどの位置にあるかを調べるには Index オブジェクトに対して get_loc メソッドを使用する．

例. 指定したインデックスの格納位置を調べる（先の例の続き）

入力：
```
print( sr2.index.get_loc('z') )  # インデックスが 'z' であるデータの格納位置
print( sr2.index.get_loc('y') )  # インデックスが 'y' であるデータの格納位置
```

出力：
```
4
slice(1, 4, None)
```

この例からわかるように，重複の無いインデックス（上記の例では ’z’）のデータの位置は 1 つの整数値として得られる．重複のあるインデックス（上記の例では ’y’）のデータの位置が**スライスオブジェクト**[8] として得られている．連続していないデータが同一のインデックスを持つ場合に get_loc メソッドを使用すると，真理値を要素とする配列

[8] データの位置を示す**スライス**を明に表現するオブジェクト．詳しくは Python の文法に関する資料（書籍，公式インターネットサイト）を参照のこと．（拙書「Python3 入門 - Kivy による GUI アプリケーション開発，サウンド入出力，ウェブスクレイピング」でも解説しています）

が得られる.（次の例参照）

例. 同じインデックスが不規則な位置にある場合（先の例の続き）

入力：
```
lst22 = list( range(10) )  # 0-9 までの整数のリスト
ix22 = ['x','y','x','x','y','y','z','y','z','x']  # インデックス用のデータ列（重複あり）
sr22 = pd.Series( lst22, index=ix22 )  # Series に変換（インデックスに ix22 を与える）
sr22                                   # 内容確認
```

出力：
```
x  0
y  1
x  2
x  3
y  4
y  5
z  6
y  7
z  8
x  9
dtype:  int64
```

入力：
```
s = sr22.index.get_loc('y')     # 重複するインデックスの位置
print( s )          # 格納位置
print( type(s) )    # データ型の調査
```

出力：
```
[False True False False True True False True False False]
<class 'numpy.ndarray'>
```

この例では，インデックス 'y' の位置が不規則であり，'y' の位置を get_loc メソッドで調べた結果が真理値の配列として得られている．該当するデータの位置が True，該当しないデータの位置が False に対応していることがわかる．このような真理値の配列はデータを抽出する際（p.8「2.1.5 要素の抽出」）に利用できる．

2.1.8 ndarray から Series への変換

NumPy の ndarray（1 次元）を Series オブジェクトに変換するには，リストを Series オブジェクトに変換する方法と基本的には同じである．次に，順を追って作業の例を示す．

例. NumPy の読込みと ndarray の生成（先の例の続き）

入力：
```
import numpy as np                                  # NumPy を 'np' という名で読込み
ar = np.array( [2,4,6,8,10], dtype='float64' ) # 偶数列の配列を生成（浮動小数点数として）
print( ar )                                         # 内容確認
print( type(ar) )                                   # データ型の調査
```

出力：
```
[ 2.  4.  6.  8.  10.]
<class 'numpy.ndarray'>
```

この例では，整数のリストから ndarray を生成している．ndarray 生成時のキーワード引数 'dtype=' に浮動小数点数を意味する 'float64' を指定している.（NumPy に関して詳しくは別の情報源[9] を参照のこと）

例. ndarray を Series に変換（先の例の続き）

入力：
```
sr3 = pd.Series( ar, index=None )  # Series に変換（インデックスは自動生成）
sr3                                # 内容確認
```

出力：
```
0   2.0
1   4.0
2   6.0
3   8.0
4  10.0
dtype:  float64
```

[9]拙書「Python3 ライブラリブック - 各種ライブラリの基本的な使用方法」でも解説しています.

2.1.9 整列（ソート）

Series のデータを昇順に整列（ソート）するには sort_values メソッドを使用する．

例. Series の整列（先の例の続き）

入力：
```
sr4 = pd.Series( [25,1,16,4,9], index=None )  # 整列されていない Series
sr4                                           # 内容確認
```

出力：
```
0  25
1   1
2  16
3   4
4   9
dtype:  int64
```

入力：
```
sr41 = sr4.sort_values()     # 要素を昇順に整列
print( sr41 )                # 整列結果の確認
```

出力：
```
1   1
3   4
4   9
2  16
0  25
dtype:  int64
```

これは，整列されていない Series オブジェクト sr4 を整列して sr41 を生成する例である．sort_values による整列処理では，元のデータ sr4 は変更されず[10]，整列結果のデータが sr41 として新たに生成されている．（次の例参照）

例. 元のデータの内容確認（先の例の続き）

入力：
```
sr4    # 内容確認（再度）
```

出力：
```
0  25
1   1
2  16
3   4
4   9
dtype:  int64
```

元のデータ sr4 は変更されていないことが確認できる．

2.1.9.1 整列順序の指定

sort_values メソッドによる整列処理は昇順となるが，実行時にキーワード引数 ‘ascending=’ を与えることで整列順序を指定することができる．‘ascending=True’ とすると昇順（暗黙設定），‘ascending=False’ とすると降順となる．（次の例参照）

例. 降順に整列（先の例の続き）

入力：
```
sr42 = sr4.sort_values( ascending=False )  # 要素を降順に整列
print( sr42 )                              # 整列結果の確認
```

出力：
```
0  25
2  16
4   9
3   4
1   1
dtype:  int64
```

2.1.9.2 インデックスに沿った整列

Series の内容をインデックスの順序に整列するには sort_index メソッドを使用する．

[10] sort_values 実行時ににキーワード引数 ‘inplace=True’ を与えると，整列対象の Series オブジェクト自体を変更（整列）する．

例. インデックスの順序で整列する（先の例の続き）

入力：
```
sr42.sort_index()              # インデックスの順序で整列
```

出力：
```
0   25
1    1
2   16
3    4
4    9
dtype:  int64
```

sort_index メソッドは元の Series には変更を加えず[11]，整列済みの別の Series を返す．sort_index メソッドでもキーワード引数 ‘ascending=’ による整列順序の指定ができる．

2.1.10 要素の削除

2.1.10.1 drop メソッド

Series の中の指定した要素を削除するには drop メソッドを使用する．これに関して順を追って例示する．まずサンプルの Series オブジェクトを生成する．

例. サンプルの作成（先の例の続き）

入力：
```
# サンプルの作成
sr5 = pd.Series( [111,222,333], index=['d1','d2','d3'] )
sr5    # 内容確認
```

出力：
```
d1  111
d2  222
d3  333
dtype:  int64
```

この sr5 のインデックス ’d2’ のデータを drop メソッドで削除する．（次の例参照）

例. drop による要素の削除（先の例の続き）

入力：
```
sr51 = sr5.drop('d2')     # インデックス 'd2' の要素を削除
sr51    # 内容確認
```

出力：
```
d1  111
d3  333
dtype:  int64
```

インデックスが ’d2’ の位置の要素を削除したものが sr51 として得られている．drop メソッドに与える引数はキーワード引数の形「index='d2'」として与えても良い，
drop メソッドによる要素の削除では元のデータを変更しない．（次の例参照）

例. 元のデータの内容確認（先の例の続き）

入力：
```
sr5    # 元のデータの内容確認
```

出力：
```
d1  111
d2  222
d3  333
dtype:  int64
```

sr5 は元のままである．
複数の要素を同時に削除する場合は，削除対象のインデックスをリストにして drop メソッドに与える．（次の例参照）

例. 複数の要素を同時に削除（先の例の続き）

入力：
```
sr52 = sr5.drop(['d1','d3'])     # 複数の要素を削除
sr52    # 内容確認
```

[11] sort_index 実行時ににキーワード引数 ‘inplace=True’ を与えると，整列対象の Series オブジェクト自体を変更（整列）する．

出力：
```
d2   222
dtype:  int64
```

■ 重複するインデックスを持つ要素に対する drop メソッドの処理

重複するインデックスを持つ要素に対して drop メソッドを実行すると，該当する要素が全て削除される．

例. 重複するインデックスをまとめて削除する（先の例の続き）

入力：
```
sr53 = pd.Series(range(4),index=['d1','d2','d1','d3'])  # 重複するインデックス 'd1'
sr53
```

出力：
```
d1     0
d2     1
d1     2
d3     3
dtype:  int64
```

入力：
```
sr54 = sr53.drop('d1')     # 重複するインデックスに対して drop
sr54
```

出力：
```
d2     1
d3     3
dtype:  int64
```

これは，重複するインデックス 'd1' を複数持つ Series オブジェクト sr53 に対して drop メソッドを実行した例である．当該インデックスを持つ要素が全て削除されたものが sr54 に得られていることがわかる．

次に，重複するインデックスが複数の種類存在している場合の例を示す．

例. 重複する複数のインデックスをまとめて削除する（先の例の続き）

入力：
```
# 重複するインデックス 'd1','d4' を持つ Series
sr55 = pd.Series(range(8),index=['d1','d2','d1','d3','d4','d5','d4','d6'])
sr55
```

出力：
```
d1     0
d2     1
d1     2
d3     3
d4     4
d5     5
d4     6
d6     7
dtype:  int64
```

入力：
```
sr56 = sr55.drop(['d1','d4'])  # 複数の種類の削除対象を指定する
sr56
```

出力：
```
d2     1
d3     3
d5     5
d6     7
dtype:  int64
```

これは，重複するインデックス項目が 'd1', 'd4' と 2 種類ある場合に drop を実行した例である．対象の要素が全て削除されていることがわかる．

■ 存在しないインデクスの要素に対する drop

存在しないインデクスの要素に対して drop を試みるとエラーとなる．(次の例)

例. 存在しないインデックスを指定した drop（先の例の続き）

入力：
```
sr57 = sr56.drop('d9')     # インデックス d9 は存在しない
```

出力：
```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[54], line 2
----> 1 sr57 = sr56.drop('d9')    # インデックス d9 は存在しない
      :
   （途中省略）
      :
KeyError:  "['d9'] not found in axis"
```

このように KeyError となる．存在しないインデックスに対する drop において，drop の引数に errors='ignore'」を与えるとエラーを抑制できる．

2.1.10.2 del 文による削除

drop メソッドによる削除処理では元のデータは変更されない[12] が，del 文を使用すると指定したインデックス位置のデータを元のデータから削除（元のデータを直接的に変更）する．（次の例参照）

例. del 文による直接削除

入力：
```
print( sr5 )        # 元のデータ
print( '--------' )
del sr5['d2']       # 元のデータを直接変更する
print( sr5 )        # 内容確認
```

出力：
```
 d1  111
 d2  222
 d3  333
dtype:  int64
--------
 d1  111
 d3  333
dtype:  int64
```

del 文を用いる際は注意すること．また，存在しないデータの削除を試みるとエラー（KeyError）が発生する．

2.1.11 Series オブジェクトの変更と複製について

Series オブジェクトに対して copy メソッドを実行することで，別のオブジェクトとして複製を作ることができる．copy メソッドに関する実行例を次に示す．

例. サンプルデータの作成（先の例の続き）

入力：
```
sr7 = pd.Series([11,12,13],index=['d1','d2','d3'])    # サンプルデータ
sr7
```

出力：
```
 d1    11
 d2    12
 d3    13
dtype:  int64
```

この例は，サンプルの Series オブジェクト sr7 を作成するもので，これの複製を copy メソッドによって作成する例を次に示す．

例. 複製の作成（先の例の続き）

入力：
```
sr72 = sr7.copy()    # 複製を作る
sr72['d2'] = 999     # 複製されたものを変更
sr72                 # 内容確認
```

出力：
```
 d1     11
 d2    999
 d3     13
dtype:  int64
```

[12] キーワード引数 'inplace=True' の指定によって対象の Series オブジェクトの要素を直接削除することもできる．

この例では，元のオブジェクト sr7 の複製を sr72 として作成している．また，sr72 の内容を変更しているが，これは元の sr7 には影響がない．(次の例)

例. 元のデータの確認（先の例の続き）

入力：
```
sr7     # 元のデータの内容確認
```

出力：
```
d1     11
d2     12
d3     13
dtype:  int64
```

元のデータ sr7 とは別のものとして複製 sr72 が作成されていることがわかる．

参考)

copy メソッドは**ミュータブル**なオブジェクト[13] を含む Series オブジェクトに対しても完全な複製（**深いコピー**）を作成する．しかし，要素のデータ型によっては完全な複製が作成されないこともあるので注意すること．

copy メソッドにキーワード引数 'deep=False' を与えると**浅いコピー**を作成する．

2.1.12 Series オブジェクトの連結

複数の Series オブジェクトを連結して 1 つにするには concat を用いる．

書き方： concat([Series オブジェクト 1, Series オブジェクト 2, …])

2 つの Series オブジェクトを連結する例を示す．

例. Series オブジェクトの連結

入力：
```
srA = pd.Series( [11,12,13] )
srB = pd.Series( [21,22,23] )
srAB = pd.concat( [srA,srB] )     # 上記 2 つの Series を連結
srAB
```

出力：
```
0   11
1   12
2   13
0   21
1   22
2   23
dtype:  int64
```

この例では Series オブジェクト srA, srB を連結した結果を srAB に得ている．インデックスも含めてそのまま連結されていることがわかる．連結した後でインデックスを付け直すには，次に示す reset_index メソッドを使用する．

2.1.13 インデックスの再設定

2.1.13.1 reset_index メソッドによる方法

先の例で得られた Series オブジェクトのインデックスを reset_index メソッドによって付け直す処理の例を次に示す．

例. インデックスの付け直し（先の例の続き）

入力：
```
srABr = srAB.reset_index( drop=True )
srABr
```

出力：
```
0   11
1   12
2   13
3   21
4   22
5   23
dtype:  int64
```

reset_index メソッドの使用において注意すべきこととして，キーワード引数 'drop=True' の付与がある．これを省い

[13] リストをはじめとする**変更可能**なデータ構造のこと．

て同様の処理を行った例を次に示す.

例. キーワード引数無しで reset_index を実行（先の例の続き）

入力：
```
srABr = srAB.reset_index()  # drop を指定せずに実行
srABr
```

出力：
```
   index  0
0      0  11
1      1  12
2      2  13
3      0  21
4      1  22
5      2  23
--------
<class 'pandas.core.frame.DataFrame'>
```

このように，連結結果が DataFrame オブジェクトとして得られており，元の Series オブジェクトが持っていたインデックスは新たな**カラム**となる．DataFrame に関しては「2.2 DataFrame（pandas のデータ構造）」（p.22）で解説する．

2.1.13.2 与えた項目列でインデックスで置き換える方法

Series オブジェクトの index プロパティにリストなどを与えることで直接的にインデックスを再設定する方法がある．先の例で作成した Series オブジェクト srAB にインデックスを直接与える例を次に示す.

例. インデックスを直接的に書き換える（先の例の続き）

入力：
```
srAB.index = ['a','b','c','d','e','f']    # srAB 自体に変更を加える
srAB
```

出力：
```
a    11
b    12
c    13
d    21
e    22
f    23
dtype:  int64
```

srAB のインデックスが 'a'〜'f' に置き換えられていることがわかる.

2.1.14 マルチインデックス

Series オブジェクトは**マルチインデックス**と呼ばれる階層的なインデックスを保持することができる．これについて例を示して説明する.

マルチインデックスは MultiIndex オブジェクトを作成し，それを Series オブジェクトを生成するときに与える.

例. MultiIndex オブジェクトの作成（先の例の続き）

入力：
```
midx = pd.MultiIndex.from_tuples(
    [('A','a',1),('A','a',2),('A','b',1),('A','b',2),
      ('B','a',1),('B','a',2),('B','b',1),('B','b',2)])
```

この例では from_tuples メソッドを使用してマルチインデックスである MultiIndex オブジェクトを midx に作成している．この例では，1 つのインデックス項目を ('A','a',1) のようなタプルとしている.

上の例で作成した midx を与えて Series オブジェクトを作成する例を次に示す.

例. Series オブジェクトの作成（先の例の続き）

入力：
```
sr = pd.Series(list(range(8)),index=midx)    # 0〜7 の値を持つ Series オブジェクト
sr
```

出力：
```
A  a  1  0
         2  1
      b  1  2
         2  3
   B  a  1  4
         2  5
      b  1  6
         2  7
dtype:  int64
```

sr に作成された Series オブジェクトが 4 列の形式で表示されており，左 3 列がインデックスである．

マルチインデックスでは，左のインデックスが右のインデックスを階層的に含んでいる形式（第 0 レベル，第 1 レベル，…）であり，at プロパティなどに指定する際に左のインデックス項目を優先する形で記述することができる．（次の例）

例. 様々なレベルによるインデックスの指定（先の例の続き）

入力：
```
sr.at[('A')]      # 第 0 レベルのインデックスが 'A' であるもの
```
出力：
```
 a  1  0
    2  1
 b  1  2
    2  3
dtype:  int64
```

入力：
```
sr.at[('A','b')]  # 更に第 1 レベルのインデックスが 'b' であるもの
```
出力：
```
 1  2
 2  3
dtype:  int64
```

入力：
```
sr.at[('A','b',2)]  # 更に第 2 レベルのインデックスが 2 であるもの
```
出力：
```
np.int64(3)
```

この例の最後の部分の出力が ‘`np.int64(3)`’ となっているが，これは NumPy のスカラー値であることを意味する．処理環境によっては単に 3 と表示されるので，適宜読み替えること．

2.1.14.1 loc による安全なアクセス

先に示した例では，マルチインデックスによるアクセスに at を使用していたが，loc による方法がより安全であり推奨される．

例. loc によるインデックスの指定（先の例の続き）

入力：
```
sr.loc[('A')]      # 第 0 レベルのインデックスが 'A' であるもの
```
出力：
```
 a  1  0
    2  1
 b  1  2
    2  3
dtype:  int64
```

入力：
```
sr.loc[('A','b')]  # 更に第 1 レベルのインデックスが 'b' であるもの
```
出力：
```
 1  2
 2  3
dtype:  int64
```

入力：
```
sr.loc[('A','b',2)]  # 更に第 2 レベルのインデックスが 2 であるもの
```
出力：
```
np.int64(3)
```

先と同じ結果が得られていることがわかる．

2.1.14.2　インデックスレベルを指定した整列

sort_index による整列の際に，マルチインデックスの階層を指定することができる．(次の例)

例. インデックスのレベルを指定した整列（先の例の続き）

入力：
```
sr.sort_index(level=1)     # インデックスの第 1 レベルで整列
```

出力：
```
A  a  1  0
      2  1
B  a  1  4
      2  5
A  b  1  2
      2  3
B  b  1  6
      2  7
dtype:  int64
```

例. インデックスのレベルを指定した整列（先の例の続き）

入力：
```
sr.sort_index(level=2)     # インデックスの第 2 レベルで整列
```

出力：
```
A  a  1  0
   b  1  2
B  a  1  4
   b  1  6
A  a  2  1
   b  2  3
B  a  2  5
   b  2  7
dtype:  int64
```

この例のように sort_index の引数に ‘level=**インデックスレベル**’ を指定する．デフォルトは ‘level=0’ である．

2.1.14.3　インデックスレベルへの名前の付与

マルチインデックスの各階層には名前を与えることができる．(次の例)

例. MultiIndex の各階層に名前を与える（先の例の続き）

入力：
```
midx = pd.MultiIndex.from_tuples(
    [('A','a',1),('A','a',2),('A','b',1),('A','b',2),
     ('B','a',1),('B','a',2),('B','b',1),('B','b',2)],
     names=['1st','2nd','3rd'])
```

この例でも from_tuples メソッドを使用して MultiIndex オブジェクトを作成しているが，その際にキーワード引数 ‘names=’ を与えている．この例ではインデックスの第 0〜第 2 階層に対して ’1st’, ’2nd’, ’3rd’ という名前を与えている．得られた midx をインデックスとする Series オブジェクトを作成する例を次に示す．

例. Series オブジェクトの作成（先の例の続き）

入力：
```
sr = pd.Series(list(range(8)),index=midx)     # 0〜7 の値を持つ Series オブジェクト
sr
```

出力：
```
1st  2nd  3rd
 A    a    1   0
               2   1
          b    1   2
               2   3
 B    a    1   4
               2   5
          b    1   6
               2   7
dtype:  int64
```

インデックスの各階層に名前が表示されていることがわかる．

マルチインデックスの各階層に付けられた名前は， sort_index の引数に ‘level=**名前**’ などとして与えることができる．

2.1.15 その他

2.1.15.1 開始部分，終了部分の取り出し

Series オブジェクトに対して head メソッド，tail メソッドを使用すると，先頭から，あるいは末尾から指定した個数の要素の列を取り出すことができる．（次の例参照）

例. 開始部分の取り出し（先の例の続き）

入力：
```
sr6 = pd.Series( [x**2 for x in range(1000)] )    # 長い Series
sr6.head(3)    # 先頭 3 個の取り出し
```

出力：
```
0  0
1  1
2  4
dtype:  int64
```

この例では長い（1,000 個）Series オブジェクト sr6 を生成し，先頭の要素 3 個分 Series オブジェクトを得ている．同様に，終了部分の指定した個数の Series オブジェクトを得る例を次に示す．

例. 終了部分の取り出し（先の例の続き）

入力：
```
sr6.tail(3)    # 末尾 3 個の取り出し
```

出力：
```
997  994009
998  996004
999  998001
dtype:  int64
```

2.2 DataFrame（pandas のデータ構造）

pandas が提供する DataFrame は 2 次元のデータ構造であり，pandas で表形式のデータを扱う際の標準的なデータ構造である．Series と同様に DataFrame にも**インデックス**があり，DataFrame の各行はインデックスによって識別される．DataFrame において「列」は**カラム**（column）と呼ばれ，**カラム名**によって識別される．（表 3 参照）

表 3: DataFrame の構造の概略

	カラム 1	カラム 2	カラム 3	···
インデックス 1				···
インデックス 2				···
インデックス 3				···
⋮	⋮	⋮	⋮	

2.2.1 DataFrame の生成

DataFrame を生成する方法について説明する．

《DataFrame の生成》
書き方：　DataFrame(配列データ, columns=カラムとして与えるデータ並び,
index=インデックスとして与えるデータ並び)
※引数は省略可能である．

配列データとしては，入れ子のリストや ndarray の 2 次元の形式のデータを与えることができ，基本的な形式は

[**行のデータ並び 1, 行のデータ並び 2, 行のデータ並び 3,** …]

である．DataFrame コンストラクタにキーワード引数 ‘dtype=’ を与え，全ての要素に同一の型（整数，浮動小数点数など）を NumPy の規則に従って指定することが可能である[14] が，これは省略可能である．
「配列データ」を省略する[15] と「空」の DataFrame が生成される．

2.2.1.1 リストから DataFrame を生成する方法

例. リストから DataFrame を生成する

入力：
```
import pandas as pd                  # pandas を 'pd' という名で読込み
lst = [[1,1,1],[2,4,8],[3,9,27]]     # 元になる 2 次元のデータ
# DataFrame の生成
df = pd.DataFrame( lst, columns=['linear','square','cubic'],
                        index=['d1','d2','d3'] )
df       # 内容確認
```

出力：

	linear	square	cubic
d1	1	1	1
d2	2	4	8
d3	3	9	27

DataFrame 生成時にキーワード引数 ‘columns=’，‘index=’ を省略すると，自動的に整数（0 から始まる）が割り当てられる．（次の例参照）

例. インデックスとカラムを省略して DataFrame を生成（先の例の続き）

入力：
```
df = pd.DataFrame( lst )     # カラム，インデックスの省略
df                           # 内容確認
```

出力：

	0	1	2
0	1	1	1
1	2	4	8
2	3	9	27

[14] この方法はあまり推奨されない．

[15] コンストラクタの第 1 引数に None を与えることもできるが，これは推奨されない．

2.2.1.2 NumPy の配列（ndarray）から DataFrame を生成する方法

NumPy の ndarray から DataFrame を生成する例を次に示す.

例. NumPy の ndarray から DataFrame を生成する（先の例の続き）

入力：
```
import numpy as np                          # NumPy を 'np' の名で読込み
ar = np.array( lst, dtype='int32' )     # NumPy の配列（整数）に変換
# DataFrame の生成
df = pd.DataFrame( ar, columns=['linear','square','cubic'] )
df                                          # 内容確認
```

出力：

	linear	square	cubic
0	1	1	1
1	2	4	8
2	3	9	27

2.2.1.3 辞書から DataFrame を生成する方法

Python の標準的なデータ構造である**辞書**から DataFrame を生成する例を次に示す.

例. 辞書から DataFrame を生成する（先の例の続き）

入力：
```
dc = {'linear':[1,2,3], 'square':[1,4,9], 'cubic':[1,8,27]}     # 辞書
df3 = pd.DataFrame( dc )     # DataFrame の生成
df3                          # 内容確認
```

出力：

	linear	square	cubic
0	1	1	1
1	2	4	8
2	3	9	27

辞書の**キー**が DataFrame のカラムに対応することがわかる.
次に，DataFrame のインデックスに対応させる形で辞書データを作成する例を示す.

例. DataFrame のインデックスに対応させる方法（先の例の続き）

入力：
```
# 辞書から DataFrame を生成（インデックスも与える）
dc = {
    'linear':{'d1':1, 'd2':2, 'd3':3},
    'square':{'d1':1, 'd2':4, 'd3':9},
    'cubic':  {'d1':1, 'd2':8, 'd3':27}
}
df3 = pd.DataFrame( dc )     # DataFrame の生成
df3                           # 内容確認
```

出力：

	linear	square	cubic
d1	1	1	1
d2	2	4	8
d3	3	9	27

辞書の要素は「キーと値のペア」で記述されるが，値の部分を更に辞書にしている．要素としての辞書のキーが DataFrame のインデックスに対応していることがわかる.

2.2.1.4 カラム単位でデータを与える方法

先に解説した方法は，DataFrame の要素が全て同じ型のデータになるケースであるが，実用的な局面では，DataFrame のカラム毎に異なる型のデータを保持することがより一般的である．またその場合，1 つのカラム内のデータの型は（全行で）統一されていることが一般的である.

次に示す例は，空の DataFrame を作成した後に，個別にカラムデータを追加してゆくものである.

例. DataFrame を順次構築する作業（先の例の続き）

入力：
```
# 3 系統のデータを用意
c1 = np.array([1,2,3],dtype=np.int64)          # NumPy の 64 ビット符号付き整数
c2 = np.array([1.0,4.0,9.0],dtype=np.float64)  # NumPy の 64 ビット浮動小数点数
c3 = ['first','second','third']                # 文字列のリスト
```

入力：
```
# DataFrame の構築
df32 = pd.DataFrame()     # 空の DataFrame を作成
df32['int']    = c1       # カラム毎に型が
df32['double'] = c2       # 異なるものを与える
df32['string'] = c3       # 同一カラム内のデータは同一の型
df32.index = ['d1','d2','d3']   # 後からインデックスを与える
df32                      # 内容確認
```

出力：

	int	double	string
d1	1	1.0	first
d2	2	4.0	second
d3	3	9.0	third

1 つのカラム内では同じ型のデータが並んでいるが，各カラムの型は異なったものになっている．

2.2.1.5 カラムに Series を与える際の注意事項

Series オブジェクトを DataFrame のカラムとして与えることができる．ただし，この場合に注意しなければならないことがある．Series オブジェクトもインデックスを持つので，DataFrame のカラムとしてそれを追加する際は，DataFrame の既存のインデックスとの対応に注意しなければならない．このことを，事例を挙げて解説する．

例. Series オブジェクトから DataFrame を作成する（先の例の続き）

入力：
```
sr1 = pd.Series([1,2],index=['d1','d2'])  # これら 2 つの
sr2 = pd.Series([3,4],index=['d3','d4'])  # Series から
df_base = pd.DataFrame({'col1':sr1, 'col2':sr2})  # DataFrame を作成
df_base  # 内容確認
```

出力：

	col1	col2
d1	1.0	NaN
d2	2.0	NaN
d3	NaN	3.0
d4	NaN	4.0

この例では，Series オブジェクト sr1，sr2 がそれぞれ異なるインデックスを持っており，それらを元に DataFrame を作成すると，それらが DataFrame 内で異なる行となる．従って実行結果が示すように DataFrame 内に欠損部分が発生し，その部分は**欠損値** NaN となる．欠損値については後の「2.2.2.2 NaN（欠損値）について」（p.25）から「2.2.2.4 整数の欠損値について」（p.27）で解説する．

注意）次に示すケースは一見して理解しにくいので注意が必要である．

例. 一見，理解し難いケース（先の例の続き）

入力：
```
sr1 = pd.Series([1,2]) # 元になる Series(1)
sr2 = pd.Series([3,4]) # 元になる Series(2)
df_base = pd.DataFrame({'col1':sr1, 'col2':sr2},
             index=['d1','d2']) # DataFrame 作成時のインデックス指定
df_base  # 内容確認
```

出力：

	col1	col2
d1	NaN	NaN
d2	NaN	NaN

この例ではインデックスを与えずに Series オブジェクト sr1，sr2 を作成し，それを元にして DataFrame オブジェクト df_base を作成しようとしている．結果として，全要素が NaN となっている．

DataFrame を作成する際には，Series オブジェクトの index と，コンストラクタに与えられた index とが照合される．この例では，sr1，sr2 のインデックス（0,1）と，DataFrame に指定されたインデックス（'d1','d2'）が一致しないため，対応するデータが存在せず，結果としてすべて欠損値となる．

2.2.2 DataFrame の要素にアクセスする方法

2.2.2.1 at によるアクセス

インデックス，カラムを指定して DataFrame の要素にアクセスする方法について説明する．DataFrame オブジェクトに対して at を用いて要素の位置を指定することができる．

書き方：　DataFrame オブジェクト.at[インデックス, カラム]

次に例を挙げて説明する．まず，空の DataFrame を用意する．

例. 空の DataFrame の作成（先の例の続き）

入力：
```
df2 = pd.DataFrame( )   # 空の DataFrame の生成
df2                     # 内容確認
```

出力：———

このようにしてできた DataFrame オブジェクト df2 に対してインデックス，カラムを指定してデータを書き込む．（次の例参照）

例. at で指定した位置にデータを書き込む（先の例の続き）

入力：
```
# インデックスとカラムを指定して直接値を設定
df2.at['d1','linear']=1; df2.at['d1','square']=1; df2.at['d1','cubic']=1
df2.at['d2','linear']=2; df2.at['d2','square']=4; df2.at['d2','cubic']=8
df2.at['d3','linear']=3; df2.at['d3','square']=9; df2.at['d3','cubic']=27
df2      # 内容確認
```

出力：

	linear	square	cubic
d1	1.0	1.0	1.0
d2	2.0	4.0	8.0
d3	3.0	9.0	27.0

DataFrame の内容が出来上がっていることが確認できる．数値を代入するとデフォルトでは NumPy の 64 ビット浮動小数点数の型となる．

at による要素の参照ももちろん可能である．

例. at による値の参照（先の例の続き）

入力：
```
print( df2.at['d2','square'] )          # 要素の値の参照
print( type(df2.at['d2','square']) )    # データ型の調査
```

出力：
```
4.0
<class 'numpy.float64'>
```

データの型が NumPy のものであることが確認できる．

2.2.2.2 NaN（欠損値）について

DataFrame の中の値が設定されていない箇所は NaN と表示[16] される．これは**欠損値**と呼ばれる．（次の例参照）

例. 欠損値を含む DataFrame（先の例の続き）

入力：
```
df3 = pd.DataFrame( )    # 空の DataFrame の生成
df3.at['d1','linear']=1; df3.at['d2','square']=4; df3.at['d3','cubic']=27
df3    # 内容確認
```

[16] "**N**ot **a** **N**umber" の略．

出力：

	linear	square	cubic
d1	1.0	NaN	NaN
d2	NaN	4.0	NaN
d3	NaN	NaN	27.0

このようにして得られた DataFrame オブジェクト df3 は欠損値（NaN）を含んでいる．これは NumPy ライブラリが提供する欠損値 nan と同じものである．

例. NaN の型を調べる（先の例の続き）

入力：
```
print( type(df3.at['d1','square']) )
df3.at['d1','square']
```

出力：
```
<class 'numpy.float64'>          ←型は NumPy の 64 ビット浮動小数点数
np.float64(nan)                  ← NaN の実体
```

2.2.2.3 pandas 独自の欠損値〈NA〉

NaN は，数値型のデータとしての欠損値である．数値でないデータとしての欠損値は，pandas が独自に定義する NA である．（次の例）

例. NA を含む DataFrame（先の例の続き）

入力：
```
df32 = pd.DataFrame( [[None,None,None],[None,None,None],[None,None,None]],
                columns=['c1','c2','c3'], index=['d1','d2','d3'], dtype='string')
df32.at['d1','c1']='first'; df32.at['d2','c2']='second'; df32.at['d3','c3']='third'
df32      # 内容確認
```

出力：

	c1	c2	c3
d1	first	<NA>	<NA>
d2	<NA>	second	<NA>
d3	<NA>	<NA>	third

この例では，DataFrame のコンストラクタに「dtype='string'」与えているが，この型は NumPy が提供するものではなく，pandas 独自の型である．df32 の各所に <NA> が見られ，これが pandas 独自の欠損値であり，型は NAType である．

例. <NA>の型（先の例の続き）

入力：
```
print( type(df32.at['d1','c2']) )
df32.at['d1','c2']
```

出力：
```
<class 'pandas._libs.missing.NAType'>
<NA>
```

<NA> のリテラル表記は pd.NA である．

欠損値を別の値で置き換えるには，対象の DataFrame に対して fillna メソッドを使用する．（次の例参照）

例. 欠損値を 0 で置き換える（先の例の続き）

入力：
```
df3.fillna( 0 )    # NaN を 0 で置き換える
```

出力：

	linear	square	cubic
d1	1.0	0.0	0.0
d2	0.0	4.0	0.0
d3	0.0	0.0	27.0

fillna メソッドは元の DataFrame を変更せず，置き換え処理を施した別の DataFrame を返す．ただし，fillna メソッドにキーワード引数 'inplace=True' を与えて実行すると，対象の DataFrame 自体を変更する．

fillna メソッドは <NA> を置き換えることもできる．（次の例）

例. <NA>を空文字で置き換える（先の例の続き）

入力：
```
df32.fillna( '' )
```

出力：

	c1	c2	c3
d1	first		
d2		second	
d3			third

2.2.2.4 整数の欠損値について

DataFrame の整数値のカラム（あるいは，整数値の Series）に欠損値を与える場合は注意が必要である．特に通常の状況では，pandas のデータ構造のバックエンドは NumPy の配列を採用しており，NumPy の整数配列が**非数**（np.nan）を保持できないことに起因する問題が発生することがある．このことについて，例を挙げて解説する．

まず次のようにして，DataFrame のサンプルを作成する．

例. 整数と浮動小数点数のカラムを持つ DataFrame（先の例の続き）

入力：
```
df_num = pd.DataFrame( 'int'    :  [  1,   2,   3 ],
                       'float'  :  [ 1.2, 3.4, 5.6 ] ,
                     index=['d1','d2','d3'])
df_num   # 内容確認
```

出力：

	int	float
d1	1	1.2
d2	2	3.4
d3	3	5.6

これは，整数のカラムと浮動小数点数のカラムを持つ DataFrame オブジェクト df_num を作成する例である．このオブジェクトの型に関する情報は dtypes 属性で確認できる．（次の例）

例. データ型の確認（先の例の続き）

入力：
```
df_num.dtypes
```

出力：
```
int         int64          ← NumPy の 64 ビット整数
float     float64          ← NumPy の 64 ビット浮動小数点数
dtype:  object
```

この DataFrame オブジェクトの一部に欠損値を与えると予期せぬことが起こる．（次の例）

例. 欠損値を設定することで起こること（先の例の続き）

入力：
```
df_num.loc['d2','int']    = np.nan   # 部分的に NumPy の非数（nan）
df_num.loc['d2','float'] = np.nan   # を与える
df_num   # 内容確認
```

出力：

	int	float
d1	1.0	1.2
d2	NaN	NaN
d3	3.0	5.6

整数のカラム 'int' の数値に小数点が表示されている．この df_num の型を調べる．（次の例）

例. 欠損値設定後のデータ型の確認（先の例の続き）

入力：
```
df_num.dtypes
```

出力：
```
int       float64          ← NumPy の 64 ビット浮動小数点数となった
float     float64          ←こちらの型は元のまま
dtype:  object
```

'int' のカラムが整数型から浮動小数点数の型に変更されている．これは，NumPy の整数型の値として**非数**（nan）が定義されていないことに起因する．すなわち，整数カラムの一部に後から np.nan を代入することで，そのカラムが浮動小数点数の型（非数が定義されている型）に変更された．

このような問題を回避し，欠損値を保持できる整数値のカラムを実現するには，そのカラムの型を ExtensionArray

の型にすると良い．このことを次の例で示す．

例. ExtensionArray の型で DataFrame を作る（先の例の続き）

入力：
```
sr1 = pd.Series( [   1,   2,   3 ], index=['d1','d2','d3'], dtype='Int64' )
sr2 = pd.Series( [ 1.2, 3.4, 5.6 ], index=['d1','d2','d3'], dtype='Float64' )
df_num2 = pd.DataFrame( 'int':sr1, 'float':sr2 )
df_num2  # 内容確認
```

出力：

	int	float
d1	1	1.2
d2	2	3.4
d3	3	5.6

この例では，DataFrame オブジェクト df_num2 のカラムの元になるデータを予め Series オブジェクトとして作成しているが，この際に，データ型として ExtensionArray のもの[17] を指定している．この df_num2 の型を確認する例を次に示す．

例. df_num2 の型の確認（先の例の続き）

入力：
```
df_num2.dtypes
```

出力：
```
int         Int64          ← ExtensionArray の整数型である
float     Float64          ← ExtensionArray の浮動小数点数の型である
dtype:  object
```

この df_num2 に部分的に欠損値を与える試みを次に示す．ExtensionArray での欠損値は<NA> である．

例. 要素として欠損値を与える（先の例の続き）

入力：
```
df_num2.loc['d2','int']   = pd.NA # ExtensionArray の欠損値
df_num2.loc['d2','float'] = pd.NA # である<NA>を与える
df_num2  # 内容確認
```

出力：

	int	float
d1	1	1.2
d2	<NA>	<NA>
d3	3	5.6

'int' のカラムの値に小数点表示はない．次に df_num2 の型について調べる．(次の例)

例. df_num2 の型の確認（先の例の続き）

入力：
```
df_num2.dtypes
```

出力：
```
int         Int64          ← ExtensionArray の整数型のままである
float     Float64          ←これも変わらず
dtype:  object
```

以上のことから，ExtensionArray の整数型は，欠損値を保持できることがわかる．

注意）Series や DataFrame のバックエンドとしては，NumPy の型を採用するほうが，各種の処理における速度の面で有利であり，ExtensionArray をバックエンドにするのは，柔軟にデータを保持する場合に向いている．

2.2.2.5 iat によるアクセス

iat による要素へのアクセスでは格納位置（整数）を指定する．(次の例参照)

例. iat による値の読み出し（先の例の続き）

入力：
```
print( df2.iat[1,1] )     # 格納位置でアクセス
```

出力：
```
4.0
```

格納位置の開始（先頭行のインデックス，左端のカラムのインデックス）は 0 である．

当然のことではあるが iat では要素が全く存在しない領域にアクセスすることができない．このことを，先に作成

[17]「2.1.1 Series の要素のデータ型」(p.3) で解説している．表 2 を参照のこと．

した df2 を用いて確かめる.（次の例）

例. 要素の存在しない領域を参照する試み（先の例の続き）

入力：
```
df2.iat[3,1]            # 要素が全く存在しない領域の参照
```

出力：
```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-84-60e47d8f88df> in <module>
----> 1 df2.iat[3,1]
      ⋮
   （途中省略）
      ⋮
IndexError:  index 3 is out of bounds for axis 0 with size 3
```

この例ようにエラーが発生して処理が中断する．また iat では，要素が全く存在しない領域に新たな要素を追加することもできないという点に注意しなければならない.（次の例）

例. 要素の存在しない領域に値を設定する試み（先の例の続き）

入力：
```
df2.iat[3,1] = 16
```

出力：
```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-86-908120e41727> in <module>
----> 1 df2.iat[3,1] = 16
      ⋮
   （途中省略）
      ⋮
IndexError:  index 3 is out of bounds for axis 0 with size 3
```

エラーが発生して処理が中断していることがわかる.

2.2.2.6 loc によるアクセス

loc を使用すると DataFrame の中の指定した連続範囲のデータ群にアクセスすることができる.

書き方：　DataFrame オブジェクト.loc[インデックスの範囲, カラムの範囲]

loc におけるインデックスとカラムの範囲は ‘[開始:終了]’ で記述し，「終了」の位置のデータを含む.（次の例参照）

例. loc による範囲指定（先の例の続き）

入力：
```
# DataFrame の部分抽出（インデックスとカラムで範囲指定）
df22 = df2.loc['d1':'d2','linear':'square']
display( df22 )        # 整形表示
print( '--------' )
print( type(df22) )    # データ型の調査
```

出力：

	linear	square
d1	1.0	1.0
d2	2.0	4.0

```
--------
<class 'pandas.core.frame.DataFrame'>
```

取り出された部分 df22 もまた DataFrame であることがわかる.

参考）上の例の中で使用している display は，IPython が提供する関数であり，Python 標準の print 関数よりも DataFrame を綺麗に表示する．display は Jupyter のノートブックといった IPython 環境で使用できる.

loc に与える範囲の記述としてコロンのみを与えると全範囲の指定となる.（次の例）

例. loc にコロンのみを与える例（先の例の続き）

入力：
```
df2.loc['d1':'d2', :  ]     # 全てのカラムが対象
```

出力：

	linear	square	cubic
d1	1.0	1.0	1.0
d2	2.0	4.0	8.0

入力：
```
df2.loc[ :  , 'linear':'square']     # 全ての行が対象
```

出力：

	linear	square
d1	1.0	1.0
d2	2.0	4.0
d3	3.0	9.0

飛び飛びの部分を抽出するには loc の範囲指定を次のように記述する．

書き方：　DataFrame オブジェクト.loc[[インデックスのリスト], [カラムのリスト]]

次に例を示す．

例. 飛び飛びの部分の抽出（先の例の続き）

入力：
```
# 飛び飛びの部分の取り出し（インデックスとカラムで対象部分を指定）
df23 = df2.loc[['d1','d3'],['linear','cubic']]
display( df23 )        # 整形表示
print( '--------' )
print( type(df23) )    # データ型の調査
```

出力：

	linear	cubic
d1	1.0	1.0
d3	3.0	27.0

```
--------
<class 'pandas.core.frame.DataFrame'>
```

2.2.2.7　iloc によるアクセス

iloc によるアクセスでは，格納位置（整数）で対象部分を指定する．（開始番号は 0）次に例を示す．

例. iloc による部分抽出（先の例の続き）

入力：
```
# DataFrame の部分抽出（格納順位で範囲指定）
df2.iloc[0:2,0:2]     # インデックス，カラム共に 0 以上 2 未満の範囲
```

出力：

	linear	square
d1	1.0	1.0
d2	2.0	4.0

格納位置による範囲指定に関する注意事項は「2.1.3 格納順位に基づくアクセス」（p.6）で説明した通りである．

iloc でもコロンのみを与えると全範囲の指定となる．また，iloc でも飛び飛びの部分を抽出することができる．（次の例参照）

例. 飛び飛びの部分の抽出（先の例の続き）

入力：
```
# 飛び飛びの部分の取り出し（格納順位で対象部分を指定）
df2.iloc[[0,2],[0,2]]
```

出力：

	linear	cubic
d1	1.0	1.0
d3	3.0	27.0

iloc で指定した範囲に値を設定することもできる．まず次のようなサンプルの DataFrame を用意する．

例. サンプルデータの用意（先の例の続き）

入力：
```
df24 = pd.DataFrame([[x+y for x in range(4)] for y in range(4)],
                    index=['i1','i2','i3','i4'], columns=['c1','c2','c3','c4'])
df24
```

出力：

	c1	c2	c3	c4
i1	0	1	2	3
i2	1	2	3	4
i3	2	3	4	5
i4	3	4	5	6

このようにして作成した df24 の指定した部分の値を一括して変更する例を次に示す.

例. iloc で指定した範囲を変更（先の例の続き）

入力：
```
df24.iloc[1:3,1:3] = [[66,77],[88,99]]
df24
```

出力：

	c1	c2	c3	c4
i1	0	1	2	3
i2	1	66	77	4
i3	2	88	99	5
i4	3	4	5	6

この例では，iloc で指定した範囲にリストで記述した値を新たに設定している.

2.2.2.8 列（カラム）の取出しと追加

DataFrame の指定した列（カラム）にアクセスするにはスライス [ ] を用いて

DataFrame オブジェクト [カラム名]

とする．これに関する例を示す.

例. 特定の列（カラム）の取出し（先の例の続き）

入力：
```
print( df2['linear'] )          # 特定の列にアクセス
print( '--------' )
print( type(df2['linear']) )    # データ型の調査
```

出力：
```
 d1  1.0
 d2  2.0
 d3  3.0
Name:  linear, dtype:  float64
--------
<class 'pandas.core.series.Series'>
```

指定したカラム 'linear' が Series オブジェクトとして得られることがわかる.
Series オブジェクトを新たな列として DataFrame に追加することもできる.（次の例参照）

例. カラムの追加（先の例の続き）

入力：
```
# 追加するカラムを Series として生成
cl = pd.Series( ['Taro','Jiro','Hanako'], index=['d1','d2','d3'])
df2['name'] = cl    # カラム名 'name' として追加
display( df2 )      # 内容確認
```

出力：

	linear	square	cubic	name
d1	1.0	1.0	1.0	Taro
d2	2.0	4.0	8.0	Jiro
d3	3.0	9.0	27.0	Hanako

作成した Series オブジェクト cl を DataFrame df2 のカラム 'name' として追加する例である．cl のインデックスが df2 のインデックスに対応している様子がわかる.

DataFrame から，指定した複数のカラムを取り出すには次のように記述する.

DataFrame オブジェクト [[カラム名 1, カラム名 2, … , カラム名 n]]

これに関する例を示す.

例. 複数のカラムの取り出し（先の例の続き）

入力：
```
display( df2[['linear','cubic']] )  # 複数のカラムを指定して抽出
```

出力：

	linear	cubic
d1	1.0	1.0
d2	2.0	8.0
d3	3.0	27.0

結果は別の DataFrame として得られる.

[ ] には数値のカラム名を与えることもできる.（次の例）

例. カラム名が数値の場合（先の例の続き）

入力：
```
df3 = pd.DataFrame([[1,2],[3,4]])    # カラム名を整数値にする
df3
```

出力：

	0	1
0	1	2
1	3	4

入力：
```
df3[0]    # 数値のカラム名をスライスに与える
```

出力：
```
0    1
1    3
Name:  0, dtype:  int64
```

注意）DataFrame オブジェクトにスライスを付ける場合に

DataFrame オブジェクト [$s_1 : s_2$]

のようにコロン ':' を用いる場合はカラムの範囲指定とはならないことに注意しなければならない.（次の例）

例. DataFrame に付けるスライスにコロンを記述する試み（先の例の続き）

入力：
```
df2['linear':'cubic']      # カラムの範囲を指定する試み…
```

出力：

	linear	square	cubic	name

これは，カラムの範囲を指定しようとして失敗した例である.

DataFrame オブジェクトに付けるスライスにコロンを記述すると，行の範囲指定となる．従って，上の例では存在しないインデックス範囲を指定したことにより，データが全く選択されなかったことを示す結果となっている.

DataFrame オブジェクトに付けるスライスにコロンを記述する場合の正しい使用例（行範囲の指定の例）を次に示す.

例. 行範囲の指定（先の例の続き）

入力：
```
df2['d1':'d2']      # 行範囲の指定
```

出力：

	linear	square	cubic	name
d1	1.0	1.0	1.0	Taro
d2	2.0	4.0	8.0	Jiro

行位置を意味する整数値をスライスに与えることもできる.（次の例）

例. 行範囲の指定（先の例の続き）

入力：
```
df2[1:3]    # 整数で行範囲を指定
```

出力：

	linear	square	cubic	name
d2	2.0	4.0	8.0	Jiro
d3	3.0	9.0	27.0	Hanako

この場合の `df2[1:3]` は，インデックス 1 から3 未満の行を取り出すことになるという点に注意すること.

2.2.2.9　ドット ‘.’ 表記による列（カラム）へのアクセス

DataFrame オブジェクトに**ドット ‘.’** でカラム名をつなげる[18] ことで，当該カラムにアクセスすることができる．

例. ドット表記によるカラムへのアクセス（先の例の続き）

入力：
```
# DataFrame の作成
df4 = pd.DataFrame([[’中村’,’大阪府’,’53’],[’田中’,’東京’,’31’]],
                   columns=[’name’,’address’,’age’])
df4      # 内容確認
```

出力：

	name	address	age
0	中村	大阪府	53
1	田中	東京	31

入力：
```
df4.name    # ’name’ のカラム
```

出力：
```
0  中村
1  田中
Name:  name, dtype:  object
```

入力：
```
df4.name[1]    # ’name’ のカラムのインデックス [1]
```

出力：
```
’田中’
```

DataFrame オブジェクト df4 のカラム ’name’ にアクセスできていることがわかる．ただしこの方法では，新規カラムを追加することはできない[19] .（次の例参照）

例. ドット表記による新規カラム追加の試み（先の例の続き）

入力：
```
df4.gender = [’男’,’女’]       # 新規カラムの追加を試みる
df4      # 内容確認
```

出力：　（警告メッセージが表示されることがある）

	name	address	age
0	中村	大阪府	53
1	田中	東京	31

←カラムは追加されていない

新規カラムの追加には先に述べた方法を取る.（次の例参照）

例. 新規カラムの追加（先の例の続き）

入力：
```
df4[’gender’] = [’男’,’女’]      # 新規カラムの追加
display( df4 )                   # 内容確認
print( ’df4.gender[0] =’, df4.gender[0] )    # ドット表記によるアクセス
```

出力：

	name	address	age	gender
0	中村	大阪府	53	男
1	田中	東京	31	女

```
df4.gender[0] = 男
```

2.2.2.10　行の取出しと追加

DataFrame の指定した行にアクセスするには loc もしくは iloc を用いる．ただし書き方は，

DataFrame オブジェクト.loc[インデックス名]
DataFrame オブジェクト.iloc[格納位置]

とする．これは，DataFrame の loc，iloc プロパティに対するアクセスにおいて，スライス ‘[ ]’ 内のカラム指定を省略した形式である．これに関する例を示す．

[18] カラム名が整数値の場合はこの方法は使えないので注意すること．

[19] pandas 2.x の版での事情．新しい版では仕様が変更される可能性あり．

例. 特定の行の取出し（先の例の続き）

入力：
```
print( df2.loc['d2'] )          # 特定の行へのアクセス
print( '--------' )
print( type(df2.loc['d2']) )    # データ型の調査
```

出力：
```
linear     2
square     4
cubic      8
name    Jiro
Name:  d2, dtype:  object
--------
<class 'pandas.core.series.Series'>
```

指定した行 'd2' が Series オブジェクトとして得られることがわかる．
同様の処理を iloc で行ったものが次の例である．

例. 特定の行の取出し（先の例の続き）

入力：
```
print( df2.iloc[1] )    # 特定の行へのアクセス（格納位置指定）
```

出力：
```
linear     2
square     4
cubic      8
name    Jiro
Name:  d2, dtype:  object
```

Series オブジェクトを新たな行として DataFrame に追加することもできる．（次の例参照）

例. 行の追加（先の例の続き）

入力：
```
# 追加する行を Series として生成
lin = pd.Series( [4,16,64,'Junko'], index=['linear','square','cubic','name'] )
df2.loc['d4'] = lin    # インデックス 'd4' の行として追加
df2                    # 内容確認
```

出力：

	linear	square	cubic	name
d1	1.0	1.0	1.0	Taro
d2	2.0	4.0	8.0	Jiro
d3	3.0	9.0	27.0	Hanako
d4	4.0	16.0	64.0	Junko

作成した Series オブジェクト lin を DataFrame df2 の行 'd4' として追加する例である．lin のインデックスが df2 のカラムに対応している様子がわかる．

注意）

上に示したような代入による方法は慎重に行うこと．DataFrame の既存のカラムと異なる型のデータを追加すると，既存のカラムのデータ型が変更されてしまう（object 型などになる）．また，挿入する Series オブジェクトの index が，対象の DataFrame のカラム名と一致していない場合は，新たなカラムが DataFrame に追加される．

たくさんの行を既存の DataFrame に追加する場合は，後の「2.2.6 DataFrame の連結」（p.40）のところで解説する concat 関数を使用する方が良い．

loc，iloc の参照で複数行を取得する場合は DataFrame の形式で結果が得られる．（次の例）

例. 行範囲を指定して複数行を取り出す（先の例の続き）

入力：
```
df2.loc['d2':'d3']
```

出力：

	linear	square	cubic	name
d2	2.0	4.0	8.0	Jiro
d3	3.0	9.0	27.0	Hanako

入力：
```
df2.iloc[1:3]
```

出力：

	linear	square	cubic	name
d2	2.0	4.0	8.0	Jiro
d3	3.0	9.0	27.0	Hanako

例. 飛び飛びの行を指定して複数行を取り出す（先の例の続き）

入力：
```
df2.loc[ ['d1','d3'] ]
```

出力：

	linear	square	cubic	name
d1	1.0	1.0	1.0	Taro
d3	3.0	9.0	27.0	Hanako

入力：
```
df2.iloc[ [0,2] ]
```

出力：

	linear	square	cubic	name
d1	1.0	1.0	1.0	Taro
d3	3.0	9.0	27.0	Hanako

2.2.2.11 DataFrame を NumPy の配列（ndarray）に変換する方法

DataFrame のデータを NumPy の配列（ndarray）として取り出すには values プロパティを参照する.

例. DataFrame の要素を ndarray として取り出す（先の例の続き）

入力：
```
print( df2.values )          # NumPy の ndarray としてデータ配列を取り出す
print( '--------' )
print( type(df2.values) )    # データ型の調査
```

出力：
```
[[1.0 1.0 1.0 'Taro']
[2.0 4.0 8.0 'Jiro']
[3.0 9.0 27.0 'Hanako']
[4.0 16.0 64.0 'Junko']]
--------
<class 'numpy.ndarray'>
```

注意）新しい版の pandas では DataFrame オブジェクトの values プロパティの使用は**非推奨**となっており，DataFrame オブジェクトに対して to_numpy メソッドを使用して ndarray を取得するべきである.（次の例）

例. DataFrame の要素を ndarray として取り出す（先の例の続き）

入力：
```
print( df2.to_numpy() )          # NumPy の ndarray としてデータ配列を取り出す
print( '--------' )
print( type(df2.to_numpy()) )    # データ型の調査
```

出力：
```
[[1.0 1.0 1.0 'Taro']
[2.0 4.0 8.0 'Jiro']
[3.0 9.0 27.0 'Hanako']
[4.0 16.0 64.0 'Junko']]
--------
<class 'numpy.ndarray'>
```

2.2.2.12 データとしてのインデックスとカラム

DataFrame のインデックスとカラムはデータとして取り出すことができる.

例. インデックスとカラムをデータとして取り出す（先の例の続き）

入力：
```
print( df2.index )    # インデックスをデータとして取り出す
print( df2.columns )  # カラムをデータとして取り出す
```

出力：
```
Index(['d1', 'd2', 'd3', 'd4'], dtype='object')
Index(['linear', 'square', 'cubic', 'name'], dtype='object')
```

取出したデータの型は共に Index であることがわかる．Index オブジェクトはスライスによるアクセスができる.

Index 型のオブジェクトは values プロパティから ndarray 型のデータ（NumPy の配列）としてデータ列を取り出すことができる.（次の例参照）

例. インデックスとカラムを ndarray として取り出す（先の例の続き）

入力：
```
print( df2.index.values )          # インデックスを ndarray として取り出す
print( type(df2.index.values) )    # データ型の調査
print( '--------' )
print( df2.columns.values )        # カラムを ndarray として取り出す
print( type(df2.columns.values) )  # データ型の調査
```

出力：
```
['d1' 'd2' 'd3' 'd4']
<class 'numpy.ndarray'>
--------
['linear' 'square' 'cubic' 'name']
<class 'numpy.ndarray'>
```

注意）

新しい版の pandas では Index オブジェクトの values プロパティの使用は**非推奨**となっており，Index オブジェクトに対して to_numpy メソッドを使用して ndarray を取得するべきである.（次の例）

例. インデックスとカラムを ndarray として取り出す（先の例の続き）

入力：
```
print( df2.index.to_numpy() )          # インデックスを ndarray として取り出す
print( type(df2.index.to_numpy()) )    # データ型の調査
print( '--------' )
print( df2.columns.to_numpy() )        # カラムを ndarray として取り出す
print( type(df2.columns.to_numpy()) )  # データ型の調査
```

出力：
```
['d1' 'd2' 'd3' 'd4']
<class 'numpy.ndarray'>
--------
['linear' 'square' 'cubic' 'name']
<class 'numpy.ndarray'>
```

2.2.2.13 データの格納位置の調査

Index オブジェクトに対して get_loc メソッドを使用すると，指定した要素の格納位置を調べることができることを「2.1.7.2 インデックスの検索」（p.11）で示した．DataFrame においても，インデックスとカラムを Index オブジェクトとして取り出すことができるので，get_loc メソッドによって，データの格納位置を調べることができる.（次の例参照）

例. インデックス，カラムから格納位置を得る（先の例の続き）

入力：
```
y = df2.index.get_loc('d3')          # インデックスの格納位置
x = df2.columns.get_loc('square')    # カラムの格納位置
(x,y)                                # 座標で表示
```

出力：
```
(1, 2)
```

2.2.3 整列（ソート）

DataFrame の行の順序を整列（ソート）するには sort_values メソッドを使用する．このメソッドの第一引数に，整列順序のキーとなるカラムの名前を与える.（次の例参照）

例. DataFrame を昇順に整列（先の例の続き）

入力：
```
df24 = df2.sort_values( 'name' )     # カラム 'name' の値の順で整列
df24         # 内容確認
```

出力：

	linear	square	cubic	name
d3	3.0	9.0	27.0	Hanako
d2	2.0	4.0	8.0	Jiro
d4	4.0	16.0	64.0	Junko
d1	1.0	1.0	1.0	Taro

df2 の内容をカラム 'name' の値によって昇順に整列した結果を df24 に与えている．処理の結果は別の DataFrame として与えられ，元の DataFrame は変更されない[20] .（次の例参照）

例. 元のデータの内容確認（先の例の続き）

入力：
```
df2         # 内容確認
```

出力：

	linear	square	cubic	name
d1	1.0	1.0	1.0	Taro
d2	2.0	4.0	8.0	Jiro
d3	3.0	9.0	27.0	Hanako
d4	4.0	16.0	64.0	Junko

元のデータは変更されていないことがわかる．

参考）整列のキーとなるカラムは sort_values にキーワード引数「by=」の形で与えることが推奨されている．

2.2.3.1 整列順序の指定

sort_values メソッドを実行する際，キーワード引数 'ascending=' を与えることで整列の順序を指定することができる．'ascending=True' とすると昇順（暗黙設定），'ascending=False' とすると降順となる．

2.2.3.2 インデックスに沿った整列

DataFrame の内容をインデックスの順序に整列するには sort_index メソッドを使用する．

例. インデックスの順序で整列する（先の例の続き）

入力：
```
df24.sort_index()     # インデックス順に整列
```

出力：

	linear	square	cubic	name
d1	1.0	1.0	1.0	Taro
d2	2.0	4.0	8.0	Jiro
d3	3.0	9.0	27.0	Hanako
d4	4.0	16.0	64.0	Junko

sort_index の場合も元の DataFrame は変更されず，整列済みの DataFrame が新たに生成される[21] ．また，このメソッドでもキーワード引数 'ascending=' による整列順の指定ができる．

2.2.4 行，列の削除

DataFrame の中の指定した行や列を削除するには drop メソッドを使用する．

《**drop メソッド**》
書き方： DataFrame オブジェクト.drop(index=削除対象インデックス, columns=削除対象カラム)
「削除対象インデックス」,「削除対象カラム」で指定した行，列を削除する．削除対象が複数ある場合はそれぞれをリストにして与える．drop メソッドは元の DataFrame オブジェクトを変更せず，削除処理を施したものを新たな DataFrame オブジェクトとして返す．

drop メソッドのキーワード引数 'index=', 'columns=' はどちらか一方の指定でも良い．すなわち， 'index=' のみの指定の場合は行が，'columns=' のみの指定の場合は列が削除される．また，行のみを削除対象とする場合は，drop の第一引数にそのまま「削除対象インデックス」を与えても良い．

[20] sort_values 実行時にキーワード引数 'inplace=True' を与えると，整列対象の DataFrame オブジェクト自体を変更（整列）する．
[21] sort_index 実行時にキーワード引数 'inplace=True' を与えると，整列対象の DataFrame オブジェクト自体を変更（整列）する．

例. 行の削除（先の例の続き）

入力：
```
df25 = df2.drop(index='d4')  # 'd4' の行を削除
df25    # 内容確認
```

出力：

	linear	square	cubic	name
d1	1.0	1.0	1.0	Taro
d2	2.0	4.0	8.0	Jiro
d3	3.0	9.0	27.0	Hanako

例. 列の削除（先の例の続き）

入力：
```
df25 = df2.drop(columns='name')  # 'name' の列を削除
df25    # 内容確認
```

出力：

	linear	square	cubic
d1	1.0	1.0	1.0
d2	2.0	4.0	8.0
d3	3.0	9.0	27.0
d4	4.0	16.0	64.0

例. 行，列の削除（先の例の続き）

入力：
```
df25 = df2.drop(index='d4',columns='name')  # 'd4' の行と 'name' の列を削除
df25    # 内容確認
```

出力：

	linear	square	cubic
d1	1.0	1.0	1.0
d2	2.0	4.0	8.0
d3	3.0	9.0	27.0

drop によって元のデータが変更されないことを確認する.（次の例参照）

例. 元のデータの内容確認（先の例の続き）

入力：
```
df2    # 内容確認
```

出力：

	linear	square	cubic	name
d1	1.0	1.0	1.0	Taro
d2	2.0	4.0	8.0	Jiro
d3	3.0	9.0	27.0	Hanako
d4	4.0	16.0	64.0	Junko

変化が無いことが確認できる.

参考）drop メソッドにキーワード引数 'inplace=True' を与えると，対象の DataFrame 自体を変更するが，これは推奨されない.

次に，削除対象の行，列が複数ある場合の一括削除の例を示す.

例. 複数の行と列を一括削除（先の例の続き）

入力：
```
# 複数の行（d1 と d4），列（linear と name）を削除
df2.drop(index=['d1','d4'],columns=['linear','name'])
```

出力：

	square	cubic
d2	4.0	8.0
d3	9.0	27.0

2.2.4.1 カラムの抹消

del 文によって DataFrame の指定したカラムを直接抹消することができる.（次の例参照）

例. カラムを直接削除する（先の例の続き）

入力：
```
del df2['name']     # 'name' のカラムを抹消（直接削除）
df2                  # 内容確認
```

出力：

	linear	square	cubic
d1	1.0	1.0	1.0
d2	2.0	4.0	8.0
d3	3.0	9.0	27.0
d4	4.0	16.0	64.0

この方法は直接 DataFrame の内容を変更するので注意すること.

2.2.5 DataFrame の複製

既存の DataFrame オブジェクトの複製を別の DataFrame オブジェクトとして作成するには copy メソッドを使用する.

例. サンプルデータの作成（先の例の続き）

入力：
```
dOrig = pd.DataFrame([[1,2],[3,4]],index=['i1','i2'],columns=['c1','c2'])
dOrig
```

出力：

	c1	c2
i1	1	2
i2	3	4

次に，copy メソッドを使用して dOrig オブジェクトの複製を dCopy として作成する.

例. 複製の作成（先の例の続き）

入力：
```
dCopy = dOrig.copy()
dCopy
```

出力：

	c1	c2
i1	1	2
i2	3	4

このようにして得られた dCopy は，元の dOrig とは別の物であり，dCopy 側に変更を加えても dOrig には影響が無い.

例. 複製の側を変更（先の例の続き）

入力：
```
dCopy.iloc[0,0] = 11     # 複製側を変更
print('複製側');       display(dCopy)
print('オリジナル');   display(dOrig)
```

出力：

複製側

	c1	c2
i1	11	2
i2	3	4

オリジナル

	c1	c2
i1	1	2
i2	3	4

注意）既存の DataFrame オブジェクトを，イコール「=」によって別の変数に代入した場合は同一の DataFrame オブジェクトを指すので，オブジェクトの内容を変更する場合は注意が必要である.（次の例を参照）

例. 参照側の変数名を用いた DataFrame の内容の変更（先の例の続き）

入力：
```
dRef = dOrig             # 別の変数 dRef に代入
dRef.iloc[1,1] = 44      # dRef を変更すると…
print('参照側');        display(dRef)
print('オリジナル');    display(dOrig)
```

出力：

参照側

	c1	c2
i1	1	2
i2	3	44

オリジナル

	c1	c2
i1	1	2
i2	3	44

この例では DataFrame オブジェクト dOrig を別の変数名 dRef に代入しているが，この dRef は dOrig と同じ DataFrame オブジェクトを指している．従って，dRef の変数名を使用して DataFrame の内容を変更すると，dOrig の変数名で内容を確認しても同じ内容のオブジェクトが得られる．（元の DataFrame が変更されている）

2.2.6 DataFrame の連結

2.2.6.1 最も単純な連結処理

複数の DataFrame を単純に連結（行として連結）して 1 つにするには concat を用いる．

書き方： concat([DataFrame 1, DataFrame 2, …])

サンプルの DataFrame を 2 つ用意して，それらの連結作業を例にして解説する．

例. サンプルデータの作成（先の例の続き）

入力：
```
dfA = pd.DataFrame([['1a','1b'],['2a','2b']],
                   index=['d1','d2'], columns=['a','b'])
dfB = pd.DataFrame([['3a','3b'],['4a','4b']],
                   index=['d3','d4'], columns=['a','b'])
display( dfA ); display( dfB )
```

出力：

	a	b
d1	1a	1b
d2	2a	2b

	a	b
d3	3a	3b
d4	4a	4b

このようにして作成した DataFrame オブジェクト dfA，dfB を連結する例を次に示す．

例. 行として単純に連結（先の例の続き）

入力：
```
dfAB = pd.concat( [dfA,dfB] )
display( dfAB )
```

出力：

	a	b
d1	1a	1b
d2	2a	2b
d3	3a	3b
d4	4a	4b

concat の引数に与えるリストの順序通りに連結される．（次の例）

例. 逆順に連結（先の例の続き）

入力：
```
dfBA = pd.concat( [dfB,dfA] )
display( dfBA )
```

出力：

	a	b
d3	3a	3b
d4	4a	4b
d1	1a	1b
d2	2a	2b

連結しようとする DataFrame オブジェクトの間に共通しないカラムがある場合は，連結後の欠損部分の値が **NaN**（欠損値）となる.（次の例）

例. 欠損値 NaN が発生する場合（先の例の続き）

入力：
```
dfC = pd.DataFrame([['3c','3d'],['4c','4d']],
                   index=['d3','d4'], columns=['c','d'])
display( dfC )
```

出力：

	c	d
d3	3c	3d
d4	4c	4d

入力：
```
dfAC = pd.concat( [dfA,dfC] )    # dfA と dfC を連結
display( dfAC )
```

出力：

	a	b	c	d
d1	1a	1b	NaN	NaN
d2	2a	2b	NaN	NaN
d3	NaN	NaN	3c	3d
d4	NaN	NaN	4c	4d

2.2.6.2 横方向（カラム方向）の連結

共通するカラムを持たない複数の DataFrame オブジェクトを連結するには，concat にキーワード引数 ‘axis=1’ を与える．これに関する例を次に示す.

例. カラム方向の連結（先の例の続き）

入力：
```
dfD = pd.DataFrame([['1e','1f'],['2e','2f'],['3e','3f']],
                   index=['d1','d2','d3'], columns=['e','f'])
display( dfD )
```

出力：

	e	f
d1	1e	1f
d2	2e	2f
d3	3e	3f

入力：
```
dfAD = pd.concat( [dfA,dfD], axis=1 )    # dfA と dfD の連結
display( dfAD )
```

出力：

	a	b	e	f
d1	1a	1b	1e	1f
d2	2a	2b	2e	2f
d3	NaN	NaN	3e	3f

この例からもわかるように，連結対象の DataFrame オブジェクトの間に共通するインデックスを対応させる形で連結される.（対応しない部分は NaN となる）

2.2.7 データの抽出

目的の行を DataFrame から抽出する方法について説明する.

2.2.7.1 複数の格納位置を指定した一括抽出

take メソッドを用いると，複数の格納位置を指定して一度に行を抽出することができる.

例. サンプルデータの用意（先の例の続き）

入力：
```
# サンプルデータ
df3 = pd.DataFrame(
        [['taro','male',35],
         ['hanako','female',31],
         ['jiro','male',23],
         ['junko','female',21]],
    columns=['Name','gender','age'] )
df3    # 内容確認
```

出力：

	Name	gender	age
0	taro	male	35
1	hanako	female	31
2	jiro	male	23
3	junko	female	21

作成した DataFrame オブジェクト df3 は人物に関するデータで，氏名，性別，年齢のカラムから成る．この DataFrame の格納位置 1，3 の行を take メソッドで取り出す例を次に示す.

例. take による複数行の抽出（先の例の続き）

入力：
```
idx = [1,3]       # 格納位置のリスト
df3.take( idx )   # 行の抽出
```

出力：

	Name	gender	age
1	hanako	female	31
3	junko	female	21

2.2.7.2 マスクを用いた抽出

真理値 の並びに対応する行を DataFrame から抽出することができる．すなわち，DataFrame の各行に対応する真理値の並びを用意して，それをスライスのような形で DataFrame オブジェクトに与えると，真理値並びの中の真（True）に対応する行が抽出される．この場合の真理値並びにはリスト，NumPy の ndarray，Series オブジェクトが使用できる.

次のような，抽出用の真理値リストを用意して抽出処理を実行する.（次の例参照）

例. 真理値リストによる行の抽出（先の例の続き）

入力：
```
cond = [False,True,False,False] # マスク（条件リスト）
df31 = df3[cond]                # 抽出処理
df31                            # 内容確認
```

出力：

	Name	gender	age
1	hanako	female	31

真理値の列がリスト cond に作成されており，先頭から 2 番目すなわち「0 で開始する格納位置の 1 番目」を抽出することが意図されている．実際の抽出処理は `df3[cond]` で実行され，結果が別の DataFrame オブジェクト df31 として得られている.

ここに示した方法は行の抽出のための基本的なものであり，実際のデータ処理ではこの処理に根ざした更に簡便な方法を取ることになる．例えば次に説明するような，条件式から真理値列を生成して抽出処理を行う方法などがある.

2.2.7.3　条件式から真理値列を生成する方法

先に説明した抽出方法は DataFrame の基本的な機能を示すものであり，実際のデータ処理の場面では，抽出のための真理値列を手作業で記述することよりも，**条件式**から真理値列を自動的に生成する形を取ることの方が一般的である．これに関しても具体例を示して説明する．

例. 真理値列の自動生成（先の例の続き）

入力：
```
# カラム 'gender' が 'female' である行の表す真理値列
cond2 = df3['gender']=='female'
cond2    # 内容確認
```

出力：
```
0  False
1   True
2  False
3   True
Name:  gender, dtype:  bool
```

この例の中にある `df3['gender']=='female'` が真理値列を生成し，それを Series オブジェクト cond2 に与えている．比較演算子 '==' は両辺の比較結果を真理値として返すものであり，例の中の記述では DataFrame のすべての行に対して比較処理を行った結果の真理値列を取得している．

ここで得られた真理値列 cond2 を用いてデータを抽出する例を次に示す．

例. データの抽出（先の例の続き）

入力：
```
df3[cond2]       # データの抽出
```

出力：

	Name	gender	age
1	hanako	female	31
3	junko	female	21

'gender' のカラムの値が 'female' の行が抽出されていることがわかる．
以上の処理はもっと簡略化することができる．(次の例参照)

例. 更に簡便な記述（先の例の続き）

入力：
```
# もっと簡便な書き方
df3[ df3['gender']=='female' ]
```

出力：

	Name	gender	age
1	hanako	female	31
3	junko	female	21

cond2 などの中間的な変数を経ることなく，直接的に抽出処理を行っている．

2.2.7.4　論理演算子による条件式の結合

条件式の**否定**や，複数の条件式を結合（**連言，選言**）した複雑な条件によるデータ抽出ができる．条件式の結合や否定は表 4 のような記述による．

表 4: 条件式の結合や否定のための演算子

記述	解説
`p1 & p2`	条件式 p1, p2 がともに真の場合に真，それ以外は偽となる．
`p1 | p2`	条件式 p1, p2 の両方もしくはどちらかが真の場合に真，両方とも偽の場合は偽となる．
`~p`	条件式 p が偽の場合に真，真の場合に偽となる．

注意）論理演算に用いる条件式は真理値並びを返すので，括弧 '( )' で括る必要がある．

条件を連言（and）で結合してデータを抽出する例を示す．

例. ‘&’ による条件の結合（先の例の続き）

入力：
```
df3[ (df3['gender']=='female') & (df3['age']<30) ]     # 連言（and による結合）
```

出力：

	Name	gender	age
3	junko	female	21

’gender’ のカラムが ’female’ で**かつ** ’age’ のカラムが 30 未満である行を抽出している．このように，結合する条件式をそれぞれ括弧 ‘(···)’ で括る．

次に，条件の否定による抽出の例を示す．

例. ‘~’ による条件の否定（先の例の続き）

入力：
```
df3[ ~(df3['gender']=='male') ]     # 否定的な条件指定
```

出力：

	Name	gender	age
1	hanako	female	31
3	junko	female	21

’gender’ のカラムが ’male’ **でない**行を抽出している．

2.2.7.5 query による抽出

query メソッドを用いると，抽出のための条件式を文字列として記述して与えることができる．この場合，カラム名（列の見出し）を変数のように扱うことができる．先の例で使用した df3 を用いた実行例を次に示す．

例. gender 列が”female”の行を抽出（先の例の続き）

入力：
```
df3.query( 'gender=="female"' )
```

出力：

	Name	gender	age
1	hanako	female	31
3	junko	female	21

例. gender 列が”female”かつ age 列が 30 未満の行を抽出（先の例の続き）

入力：
```
df3.query( 'gender=="female" & age<30' )
```

出力：

	Name	gender	age
3	junko	female	21

例. gender 列が”male”でない行を抽出（先の例の続き）

入力：
```
df3.query( '~(gender=="male")' )
```

出力：

	Name	gender	age
1	hanako	female	31
3	junko	female	21

注意）カラム名（列の見出し）が空白文字や特殊記号を含む場合は，それらをバッククォート「‘」で括って条件式内に記述すること．

2.2.8 DataFrame に関する情報の取得

2.2.8.1 要約統計量

DataFrame オブジェクトに対して describe メソッドを使用することで，それが持つデータの**要約統計量**などが得られる．

例. 要約統計量の表示（先の例の続き）

入力：
```
df2i = df2.describe()     # 要約統計量
display( df2i )           # 内容確認
print( '--------' )
print( type(df2i) )       # データ型の調査
```

出力：

	linear	square	cubic
count	4.000000	4.000000	4.000000
mean	2.500000	7.500000	25.000000
std	1.290994	6.557439	28.225284
min	1.000000	1.000000	1.000000
25%	1.750000	3.250000	6.250000
50%	2.500000	6.500000	17.500000
75%	3.250000	10.750000	36.250000
max	4.000000	16.000000	64.000000

```
--------
<class 'pandas.core.frame.DataFrame'>
```

この例からわかるように，処理結果が要約統計量を持つ別の DataFrame として得られる．describe メソッドの出力に関しては「4.4 データの分析」（p.62）で解説する．

2.2.8.2 データ構造に関する情報の表示

DataFrame オブジェクトに対して info メソッドを使用することで，そのデータ構造に関する基本的な情報が表示される．

例. DataFrame に関する情報調査（先の例の続き）

入力：
```
df2.info()      # DataFrame の情報表示
```

出力：
```
<class 'pandas.core.frame.DataFrame'>
Index:  4 entries, d1 to d4
Data columns (total 3 columns):
linear     4 non-null float64
square     4 non-null float64
cubic      4 non-null float64
dtypes:  float64(3)
memory usage:  288.0+ bytes
```

info メソッドの戻り値は None である．

2.2.9 その他

2.2.9.1 開始部分，終了部分の取り出し

DataFrame オブジェクトに対して head メソッド，tail メソッドを使用すると，先頭から，あるいは末尾から指定した行数を取り出すことができる．結果として得られるのは DataFrame である．(次の例参照)

例. 開始部分の取り出し（先の例の続き）

入力：
```
df2.head(2)    # 開始 2 行分の取り出し
```

出力：

	linear	square	cubic
d1	1.0	1.0	1.0
d2	2.0	4.0	8.0

例. 終了部分の取り出し（先の例の続き）

入力：
```
df2.tail(2)    # 末尾 2 行分の取り出し
```

出力：

	linear	square	cubic
d3	3.0	9.0	27.0
d4	4.0	16.0	64.0

2.2.9.2 行と列の転置

DataFrame の T プロパティを参照すると，元の DataFrame の行と列を転置したものが得られる．

例. 行と列の転置（先の例の続き）

入力：
```
df2.T    # 行と列の転置
```

出力：

	d1	d2	d3	d4
linear	1.0	2.0	3.0	4.0
square	1.0	4.0	9.0	16.0
cubic	1.0	8.0	27.0	64.0

2.2.9.3 REPL での表示量の設定

DataFrame の内容を REPL（対話モードでの入出力 UI）に表示する際，DataFrame の行数が 60 行以下の場合は，その内容が省略されることなく全て表示される．また，60 行を超える行数を持つ DataFrame を表示する場合は省略形式となり，先頭 5 行，末尾 5 行が表示される．(次の例)

例. DataFrame の省略表示（先の例の続き）

入力：
```
df4 = pd.DataFrame( 'col1':range(1,71) )  # 70 行のデータ
df4    # 内容確認
```

出力：

	col1
0	1
1	2
2	3
3	4
4	5
...	...
65	66
66	67
67	68
68	69
69	70

70 rows × 1 columns

省略表示の判定基準に関しては，pandas のオプション設定項目 'display.max_rows' に設定されており，次のように get_option 関数で確認することができる．

例. 省略表示の上限値の確認（先の例の続き）

入力：
```
pd.get_option('display.max_rows')
```

出力：
```
60              ←この行数までの内容は REPL で全て表示するという設定
```

REPL で省略表示された場合の表示行数は pandas のオプション設定項目 'display.min_rows' に設定されており，次のようにして確認することができる．

例. 省略時の表示行数の確認（先の例の続き）

入力：
```
pd.get_option('display.min_rows')
```

出力：
```
10              ←省略時の表示行数
```

pandas のオプション設定の値は set_option 関数によって設定することができる．（次の例）

例. オプション設定（先の例の続き）

入力：
```
pd.set_option('display.min_rows',6)     # 省略表示の行数を 6 行に設定
df4     # 内容確認
```

出力：
```
      col1
0        1
1        2
2        3
...    ...
67      68
68      69
69      70
70 rows × 1 columns
```

この例では，省略表示の行数を 6 行（先頭 3 行，末尾 3 行）に設定している．
同様の方法で，省略表示の上限値を変更することもできる．

2.3 日付と時刻

時系列データは日付や時刻を基準とするデータ列であり，その取り扱いのために pandas は日付や時刻を表現するためのデータクラスを提供している．最も基本的なものに，単一のデータとして**タイムスタンプ**を表現するための Timestamp クラスがある．また，その列を表現するための DatetimeIndex クラスがある．

2.3.1 Timestamp クラス

Timestamp は **1 つの時点を日付と時刻で表す**ためのオブジェクトを定義するクラスである．次の例は ISO 8601 に準じて記述された日付と時刻「2019 年 6 月 17 日 15 時 14 分 31 秒」を表現する文字列を Timestamp のコンストラクタに与え，インスタンスを生成する例である．

例. Timestamp オブジェクトの生成（先の例の続き）

入力：
```
t = pd.Timestamp('2019-06-17T15:14:31')    # ISO8601 表記
print('Data type:',type(t))    # 型の調査
t    # 内容確認
```

出力：
```
Data type:  <class 'pandas._libs.tslibs.timestamps.Timestamp'>
Timestamp('2019-06-17 15:14:31')
```

入力：
```
print( t )      # 整形表示
```

出力：
```
2019-06-17 15:14:31
```

Timestamp オブジェクト t に日付と時刻を表す値が格納されている．Timestamp のコンストラクタに与える文字列には様々な表現の文字列を与えることができる．

例. 様々な表現の日付・時刻（先の例の続き）

入力：
```
t1 = pd.Timestamp('20190617151431')
t2 = pd.Timestamp('2019/06/17 15:14:31')
t3 = pd.Timestamp('2019-06-17 15:14:31')
print( 'Example:¥n', t1,'¥n', t2,'¥n', t3 )
```

出力：
```
Example:
2019-06-17 15:14:31
2019-06-17 15:14:31
2019-06-17 15:14:31
```

Timestamp オブジェクトは表 5 に示すようなプロパティを持ち，日付，時刻の部分を取り出すことができる．

表 5: Timestamp オブジェクトのプロパティ（一部）

プロパティ	値	プロパティ	値	プロパティ	値
year	年	month	月	day	日
hour	時	minute	分	second	秒
microsecond	マイクロ秒	nanosecond	ナノ秒		
day_of_week	曜日（整数：0=月曜）	is_leap_year	閏年（真理値）	quarter	第 n 四半期（1〜4）

Timestamp オブジェクトの各種プロパティを参照する例を次に示す．

例. 各種プロパティの参照（先の例の続き）

入力：
```
print( 'year:',         t.year );         print( 'month:',       t.month )
print( 'day:',          t.day );          print( 'hour:',        t.hour )
print( 'minute:',       t.minute );       print( 'second:',      t.second )
print( 'microsecond:', t.microsecond );   print( 'nanosecond:',   t.nanosecond )
print( 'week day:',     t.weekday() )     # 月曜日～日曜日を 0～6 の数値で表現
```

出力：
```
year:  2019
month:  6
day:  17
hour:  15
minute:  14
second:  31
microsecond:  0
nanosecond:  0
week day:  0
```

この例の中にある weekday() は，Timestamp オブジェクトに対するメソッドであり，それが示す日付の曜日を整数値で求める（表 6）ものである．

表 6: weekday メソッドが返す値とそれが意味する曜日

値	0	1	2	3	4	5	6
曜日	月	火	水	木	金	土	日

■ **Timestamp オブジェクト生成のための別の方法**：to_datetime

pandas の to_datetime を使用して Timestamp オブジェクトを生成する方法もある．

例. to_datetime（先の例の続き）

入力：
```
pd.to_datetime('2019-06-17 15:14:31')
```

出力：
```
Timestamp('2019-06-17 15:14:31')
```

2.3.1.1 タイムゾーン

世界の地域毎で異なる時刻の運用は**協定世界時（UTC）**を基準としている．具体的には，UTC との時間差やタイムゾーン（時間帯）を指定してその地域の時刻を表す．ここでは，タイムゾーンを明に指定して Timestamp オブジェクトを生成する方法について説明する．

UTC 時刻を明に指定する最も簡単な方法は，時刻を表現する文字列の末尾に 'Z' を付けるというものである．

例. UTC で時刻指定（先の例の続き）

入力：
```
t = pd.Timestamp('2019-06-17 15:14:31Z')    # UTC
print( t )  # 整形表示
t           # そのまま表示
```

出力：
```
2019-06-17 15:14:31+00:00
Timestamp('2019-06-17 15:14:31+0000', tz='UTC')
```

この例からわかるように，Timestamp オブジェクトの内部では，UTC との時差の情報（+00:00）と，タイムゾーンを表す「tz='UTC'」が保持されている．

日付，時刻を表す文字列に時差情報（+hh:mm）を与えて時間帯を示す方法もある．

例. 時差情報による時間帯の明示（先の例の続き）

入力：
```
pd.Timestamp('2019-06-17 15:14:31+0000')
```

出力：
```
Timestamp('2019-06-17 15:14:31+0000', tz='UTC')
```

この例も UTC 時刻を表しているが，Timestamp オブジェクトが保持するタイムゾーン情報は「tz='UTC'」となっている．このように，情報を与える方法によって Timestamp オブジェクトが保持するタイムゾーン情報「tz=」の表記

に違いが生じる点に注意すること.（次の例参照）

例. 時間帯情報の指定方法の違い（先の例の続き）

入力：
```
pd.Timestamp('2019-06-17 15:14:31', tz='Asia/Tokyo')    # タイムゾーンを指定
```

出力：
```
Timestamp('2019-06-17 15:14:31+0900', tz='Asia/Tokyo')
```

入力：
```
pd.Timestamp('2019-06-17 15:14:31+0900')    # 時差を指定
```

出力：
```
Timestamp('2019-06-17 15:14:31+0900', tz='tzoffset(None, 32400)')
```

それぞれの方法において，得られる Timestamp オブジェクトが保持するタイムゾーン情報「tz=」の表記が異なっていることがわかる.

■ タイムゾーン ID

先の例で使用した日本時間のタイムゾーンの表記に「'Asia/Tokyo'」というものがある．このように地域名などの文字列で記述したものは**タイムゾーン ID** として IANA が規定[22] している.

■ タイムゾーンの変換

Timestamp オブジェクトに対して tz_convert メソッドを実行することでタイムゾーンを変換することができる.

例. タイムゾーンの変換（先の例の続き）

入力：
```
dutc = pd.Timestamp('2022-01-01 00:00:00Z' )      # UTC の時刻
dutc.tz_convert('Asia/Tokyo')                      # 日本時間への変換
```

出力：
```
Timestamp('2022-01-01 09:00:00+0900', tz='Asia/Tokyo')
```

これは UTC の Timestamp オブジェクトを日本時間のものに変換する例である.

注意） タイムゾーン情報を持たない Timestamp オブジェクトに対して tz_convert メソッドを実行するとエラーとなる.（次の例）

例. tz_convert メソッドのエラー（先の例の続き）

入力：
```
t = pd.Timestamp('2022-01-01 00:00:00')    # タイムゾーン情報なし
t.tz_convert('Asia/Tokyo')
```

出力：
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[112], line 2
      1 t = pd.Timestamp('2022-01-01 00:00:00')    # タイムゾーン情報なし
----> 2 t.tz_convert('Asia/Tokyo')
        :
   （途中省略）
        :
TypeError: Cannot convert tz-naive Timestamp, use tz_localize to localize
```

2.3.1.2 コンストラクタのキーワード引数に日付・時刻の値を与える方法

Timestamp のコンストラクタに，表 5 に示したプロパティと同じ名前のキーワード引数を与えて日付，時刻の値を設定することができる.

例. コンストラクタにキーワード引数を与える方法（先の例の続き）

入力：
```
t = pd.Timestamp(year=2019,month=6,day=17,hour=15,minute=14,second=31,
                 microsecond=123456,nanosecond=789, tz='Asia/Tokyo')
t    # 内容確認
```

出力：
```
Timestamp('2019-06-17 15:14:31.123456789+0900', tz='Asia/Tokyo')
```

参考） 引数 microsecond と nanosecond を同時に指定することはあまり推奨されない.

[22] List of tz database time zones（https://en.wikipedia.org/wiki/List_of_tz_database_time_zones）

2.3.1.3 現在時刻の取得

Timestamp のクラスメソッド now を実行すると，その時点の日付，時刻を持つ Timestamp オブジェクトを返す．

例. 現在時刻の取得（先の例の続き）

入力：
```
pd.Timestamp.now()  # 引数なし
```

出力：
```
Timestamp('2019-06-18 13:05:14.419333')
```

入力：
```
pd.Timestamp.now( tz='Asia/Tokyo' )    # タイムゾーンを明に指定
```

出力：
```
Timestamp('2019-06-18 13:05:15.976625+0900', tz='Asia/Tokyo')
```

入力：
```
pd.Timestamp.now( tz='UTC' )    # タイムゾーンを明に指定
```

出力：
```
Timestamp('2019-06-18 04:08:15.925330+0000', tz='UTC')
```

時間帯の情報を付けた形で現在時刻を取得するには，この例のように now のキーワード引数「tz=」にタイムゾーンを与える．明に UTC のタイムゾーンで現在時刻を取得する utcnow メソッドも存在する．

2.3.2 Timestamp の差：Timedelta

2 つの Timestamp 同士の差を求めると，それらの間の経過時間を求めることができる．

例. Timestamp の差（先の例の続き）

入力：
```
t1 = pd.to_datetime('1964-10-10 14:58:00')  # 1964 東京オリンピック
t2 = pd.to_datetime('2020-07-24 20:00:00')  # 2020 東京オリンピック
dt = t2 - t1
print( type(dt) )    # 型の調査
dt    # 内容確認
```

出力：
```
<class 'pandas._libs.tslibs.timedeltas.Timedelta'>
Timedelta('20376 days 05:02:00')
```

Timestamp 同士の差は Timedelta オブジェクトとして得られる．この例では，得られた Timedelta オブジェクト dt から，t1 〜 t2 の間に「20376 日と 5 時間 2 分」の時間が経過したことがわかる．

Timestamp オブジェクトに Timedelta オブジェクトを加算することができ，その Timedelta オブジェクトの値が示す時間だけ経過した後の日付と時刻が Timestamp オブジェクトとして得られる．（次の例参照）

例. Timestamp に Timedelta を加算する（先の例の続き）

入力：
```
t2 + dt
```

出力：
```
Timestamp('2076-05-08 01:02:00')
```

2.3.2.1 Timedelta の生成

時間経過に関する情報を与えて Timedelta オブジェクトを生成することができる．次の例は「7 日と 1 時間 2 分 3.123456789 秒」の経過時間を表す Timedelta オブジェクトを生成するものである．

例. Timedelta オブジェクトの生成（先の例の続き）

入力：
```
dt = pd.Timedelta(days=7,hours=1,minutes=2,seconds=3,
                  microseconds=123456,nanoseconds=789)
dt    # 内容確認
```

出力：
```
Timedelta('7 days 01:02:03.123456')
```

Timedelta のコンストラクタに与えるキーワード引数を表 7 に示す．

Timedelta オブジェクトから値を取り出す際，days，seconds，microseconds，nanoseconds といったプロパティが参照できる．（次の例参照）

表 7: Timedelta コンストラクタのキーワード引数（一部）

キーワード引数	値	キーワード引数	値	キーワード引数	値
days	日数	hours	時間	minutes	分
seconds	秒	microseconds	マイクロ秒	nanoseconds	ナノ秒

例. Timedelta オブジェクトの値を部分的に参照する（先の例の続き）

入力：
```
print( 'days:', dt.days )
print( 'seconds:', dt.seconds )
print( 'microseconds:', dt.microseconds )
print( 'nanoseconds:', dt.nanoseconds )
```

出力：
```
days:  7
seconds:  3723
microseconds:  123457
nanoseconds:  789
```

2.3.3 Timestamp の列：date_range と DatetimeIndex

指定した間隔で並んだ Timestamp オブジェクトの列を生成するには date_range を使用する.

書き方： date_range(自, 至, freq=頻度の規則)

タイムスタンプ（Timestamp，文字列どちらも可）「自」から「至」まで,「頻度の規則」に従って Timestamp オブジェクトの列を生成する.「頻度の規則」の暗黙値は 'D' で,「1 日間隔」である.

例. Timestamp の列の生成（先の例の続き）

入力：
```
dr = pd.date_range(t1,t2)     # t1 から t2 までのタイムスタンプ列を生成
print( type(dr) )     # 型の調査
dr[:5]     # 先頭 5 個を表示
```

出力：
```
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
DatetimeIndex(['1964-10-10 14:58:00', '1964-10-11 14:58:00', '1964-10-12 14:58:00',
               '1964-10-13 14:58:00', '1964-10-14 14:58:00'],
               dtype='datetime64[ns]', freq='D')
```

この例では，先に作成した Timestamp オブジェクト t1，t2 を 1 つの期間（自-至）として与え，1 日間隔のタイムスタンプ（Timestamp）の列を生成している．date_range が生成するデータ列の型は DatetimeIndex であり，各要素の型は Timestamp である.

例. 要素の型（先の例の続き）

入力：
```
dr[0] # 先頭要素の確認
```

出力：
```
Timestamp('1964-10-10 14:58:00', freq='D')
```

2.3.3.1 他の型への変換

DatetimeIndex オブジェクトは特殊な型であるが，これを NumPy の配列（ndarray）や Series オブジェクトに変換しておくと汎用性が得られて便利である.

例. 他の型への変換（先の例の続き）

入力：
```
dr.to_numpy()[:5] # NumPy の配列として取り出す（先頭 5 個を表示）
```

出力：
```
array(['1964-10-10T14:58:00.000000000', '1964-10-11T14:58:00.000000000',
       '1964-10-12T14:58:00.000000000', '1964-10-13T14:58:00.000000000',
       '1964-10-14T14:58:00.000000000'], dtype='datetime64[ns]')
```

入力：
```
pd.Series(dr).head(5) # Series オブジェクトに変換（先頭 5 個を表示）
```

出力：
```
0    1964-10-10 14:58:00
1    1964-10-11 14:58:00
2    1964-10-12 14:58:00
3    1964-10-13 14:58:00
4    1964-10-14 14:58:00
dtype:  datetime64[ns]
```

2.3.3.2 頻度の規則

date_range に与えるキーワード引数「freq=**頻度の規則**」に与えるものとしてよく使用するものを表 8 に示す.

表 8:「頻度の規則」に与えるもの（一部）

頻度の規則	説明	頻度の規則	説明
'D'	1 日毎の日付	'W'	1 週間毎の日付
'H'	1 時間毎の時刻	'T'	1 分毎の時刻
'S'	1 秒毎の時刻	'L'	1 ミリ秒の時刻
'U'	1 マイクロ秒毎の時刻	'N'	1 ナノ秒の時刻
'Y'	1 年毎の年末の日付	'YS'	1 年毎の年始の日付
'A-MAR'	1 年毎の年度末の日付	'AS-APR'	1 年毎の年度始の日付
'M'	1 月毎の月末の日付	'MS'	1 月毎の 1 日（ついたち）の日付

例. 平成の各年度初めの日付の列（先の例の続き）

入力：
```
dr = pd.date_range( '1989-01-08 00:00:00', '2019-05-01 00:00:00', freq='AS-APR' )
print( len(dr) )    # 要素数の調査
dr[:5]    # 先頭 5 個を表示
```

出力：
```
31
DatetimeIndex(['1989-04-01', '1990-04-01', '1991-04-01', '1992-04-01',
               '1993-04-01'],
               dtype='datetime64[ns]', freq='AS-APR')
```

2.3.4 NaT（欠損値）について

Timestamp 型の**欠損値**は NaT で表される．次に示す例は，NaT を含む DataFrame を作成するものである.

例. NaT を含む DataFrame（先の例の続き）

入力：
```
dfT = pd.DataFrame()
dfT.loc['d1','a'] = pd.Timestamp('2022-01-01')
dfT.loc['d2','b'] = pd.Timestamp('2022-01-02')
dfT
```

出力：

	a	b
d1	2022-01-01	NaT
d2	NaT	2022-01-02

NaT は pandas のオブジェクト「pd.NaT」である.

2.4 データ集合に対する一括処理

Series オブジェクトや DataFrame オブジェクトの全ての要素に対して同じ関数を一斉に適用するには apply メソッドを使用する．実行例を示すために，サンプルとなる Series オブジェクトを生成し，apply で適用するための関数を 1 つ定義する．

準備. サンプルの Series 作成と関数定義（先の例の続き）

入力：
```
sr0 = pd.Series([x for x in range(4)])  # 製数列の作成
sr0      # 内容確認
```

出力：
```
0    0
1    1
2    2
3    3
dtype:  int64
```

入力：
```
def tm2(x):  return(2*x)      # 2 倍の値を計算する関数
tm2(3)  # テスト実行
```

出力：
```
6
```

ここで定義した関数 tm2 を Series オブジェクト sr0 の全要素に一斉適用する．

例. apply による関数の一斉適用（先の例の続き）

入力：
```
sr0.apply(tm2)     # 与えた関数名を全要素に適用
```

出力：
```
0    0
1    2
2    4
3    6
dtype:  int64
```

これを応用すると，DataFrame の 1 つのカラム（列）から別のカラムを生成する処理を簡素化することができる．

例. apply を用いて DataFrame を作成する（先の例の続き）

入力：
```
dfm = pd.DataFrame(columns=['linear','double','square'])
dfm['linear'] = sr0
dfm['double'] = dfm['linear'].apply( tm2 )               # 関数名を与える
dfm['square'] = dfm['linear'].apply( lambda x:x**2 )     # lambda 式を与える
dfm      # 内容確認
```

出力：

	linear	double	square
0	0	0	0
1	1	2	1
2	2	4	4
3	3	6	9

`dfm['linear']` のカラムから apply メソッドによって `dfm['double']`，`dfm['square']` が作成されているのがわかる．

※ lambda 式に関しては Python の文法に関する他の資料（公式インターネットサイト，書籍など[23] ）を参照のこと．

apply メソッドは DataFrame オブジェクトに対しても使用できる．（次の例参照）

[23]拙書「Python3 入門 - Kivy による GUI アプリケーション開発，サウンド入出力，ウェブスクレイピング」でも解説しています．

例. DataFrame オブジェクトに対する apply メソッドの使用（先の例の続き）

入力：
```
dfm.apply(tm2)  # DataFrame に対して適用
```

出力：

	linear	double	square
0	0	0	0
1	2	4	2
2	4	8	8
3	6	12	18

DataFrame オブジェクト dfm の全ての要素の値が 2 倍されていることがわかる.

3　ファイル入出力

実際のデータ処理においては，データ資源はコンピュータのデータファイルとして保存されていることが一般的である．従って，データ処理においては，必要なデータをファイルから読み込む，あるいは処理結果をファイルとして保存する処理が不可欠である．

データ処理に用いられるデータは表の形式となっていることが一般的であり，そのためのデータフォーマットとしては CSV [24] 形式が採用されることが多い．CSV のデータファイルは**テキスト形式**[25] のデータであり，多くのソフトウェアが標準的にこれの読み書きに対応している．本書では，pandas の DataFrame の内容を CSV ファイルとして出力する，あるいは CSV ファイルからデータを読み込んで DataFrame に与える方法について説明する．

3.1　DataFrame を CSV ファイルとして保存する方法

DataFrame オブジェクトに対して to_csv メソッドを使用することでその内容を CSV ファイルに保存することができる．

《to_csv メソッド》

書き方：　DataFrame オブジェクト.to_csv(ファイルのパス, encoding=エンコーディング,
index=インデックス保存指定, header=カラム保存指定)

DataFrame の内容を「ファイルのパス」で指定したファイルに保存する．‘encoding=’ には保存するデータのエンコーディング（文字コード体系）* を指定する．DataFrame のインデックスをデータとして保存するには ‘index=True’（暗黙設定）とし，保存しない場合は ‘index=False’ とする．DataFrame のカラム名を CSV の見出しレコードとして保存するには ‘header=True’（暗黙設定）とし，保存しない場合は ‘header=False’ とする．

* デフォルトのエンコーディングは utf-8 である[26]．

作成した DataFrame を CSV ファイルに保存する作業の例を示す．

例. DataFrame の作成

入力：
```
import pandas as pd                  # pandas を ’pd’ という名で読込み
lst = [[1,1,1],[2,4,8],[3,9,27]]     # 元になる 2 次元のデータ
# DataFrame の生成
df = pd.DataFrame( lst, columns=['linear','square','cubic'],
                        index=['d1','d2','d3'] )
df      # 内容確認
```

出力：

	linear	square	cubic
d1	1	1	1
d2	2	4	8
d3	3	9	27

このようにしてできた DataFrame オブジェクト df を様々な条件で CSV データとして保存する．(次の例参照)

例. CSV データの保存（先の例の続き）

入力：
```
df.to_csv( ’csv01.csv’ )                                    # デフォルト
df.to_csv( ’csv01_noindex.csv’, index=False )               # インデックス無し
df.to_csv( ’csv01_nohead.csv’, header=False )               # カラム無し
df.to_csv( ’csv01_array.csv’, index=False, header=False ) # インデックス，カラム共に無し
```

この結果としてできた CSV ファイルの内容を次に示す．

[24] **CSV**（Comma Separated Values）：データ項目をコンマ ‘,’ で区切って並べたテキスト形式のデータ．RFC 4180

[25] 人間が読むことのできる文字のみで構成されたデータのこと．各種の**エンコーディング**で規定された範囲のデータから構成される．

[26] Microsoft 社の Excel で CSV データを作成するとエンコーディングは shift-jis となることが一般的である．また，**BOM 付き UTF-8** である utf-8-sig もよく用いられる．

ファイル：csv01.csv（デフォルト）

```
1 ,linear,square,cubic
2 d1,1,1,1
3 d2,2,4,8
4 d3,3,9,27
```

ファイル：csv01_noindex.csv（インデックス出力せず）

```
1 linear,square,cubic
2 1,1,1
3 2,4,8
4 3,9,27
```

ファイル：csv01_nohead.csv（カラム名出力せず）

```
1 d1,1,1,1
2 d2,2,4,8
3 d3,3,9,27
```

ファイル：csv01_array.csv（インデックス, カラム名共に無し）

```
1 1,1,1
2 2,4,8
3 3,9,27
```

このように，様々な形で CSV ファイルへの保存ができる．

参考）Microsoft Excel で開くことができる CSV を作成するには，to_csv に引数「encoding='utf-8-sig'」[27] を与えると良い．これは読込み処理の read_csv 関数（次に説明する）においても同様である．

次に，ここで作成したファイルを用いて，CSV ファイルを読み込む方法について説明する．

3.2 CSV ファイルを読み込んで DataFrame にする方法

pandas の read_csv 関数を使用することで CSV ファイルの内容を読み込むことができる．

《read_csv 関数》

書き方：　read_csv(ファイルのパス, encoding=エンコーディング,
index_col=インデックス列, names=カラム名の列)

「ファイルのパス」で指定したファイルの内容を読み込んで DataFrame オブジェクトにして返す．'encoding=' には CSV ファイルのエンコーディング（文字コード体系)* を指定する．'index_col=' には，インデックスと見なす CSV ファイルの列の番号（左端は 0）を指定する．これを省略すると，すべての列を通常のカラムと見なし，DataFrame には自動的にインデックス（0 から始まる整数）を付ける．'names=' には，カラム名として与える名前の列を指定する．これを省略すると CSV ファイルの先頭行の要素をカラム名と見なす．

* デフォルトのエンコーディングは utf-8 である[28]．

CSV ファイル保存に関する先の例からもわかるように，CSV データには様々な形のものがあり，データとして読み込む際には次のような点に注意を払う必要がある．

・CSV データにインデックスと見なす列があるか．ある場合は CSV データのどの列か．
・CSV データの最初の行をカラム名の並びと見なすかどうか．

これらの点に注意しながら上で作成した CSV ファイルを読み込む例を示す．

3.2.1 CSV の指定した列をインデックスと見なす方法

先に作成した 'csv01.csv' を読み込む例を示す．

例. ファイル名のみを指定して読み込む（先の例の続き）

入力：
```
df2 = pd.read_csv( 'csv01.csv' )    # CSV ファイルの読込み
df2    # 内容確認
```

出力：

	Unnamed: 0	linear	square	cubic
0	d1	1	1	1
1	d2	2	4	8
2	d3	3	9	27

これは，先頭の行がカラム名となっているデータ 'csv01.csv' を読み込んだ例であるが，インデックス行の指定をしていないことにより，全ての列がカラムと見なされ,，左端のカラムの名前が自動的に充填（'Unnamed:0'）されている．次に，同じ CSV データにおいて左端の列をインデックスと見なして（index_col=0）読み込む例を示す．

[27] BOM 付き UTF-8

[28] Microsoft 社の Excel で CSV データを作成するとエンコーディングは shift-jis となることが一般的である．

例. インデックス列を指定して読み込む（先の例の続き）

入力：
```
df2 = pd.read_csv( 'csv01.csv', index_col=0 )     # 左端の列をインデックスと見なす
df2     # 内容確認
```

出力：

	linear	square	cubic
d1	1	1	1
d2	2	4	8
d3	3	9	27

作成時と同じ形の DataFrame が得られていることがわかる.

3.2.2 CSV の先頭行をデータと見なす方法

先に作成した 'csv01_nohead.csv' を読み込む例を示す.

例. 先頭行の扱いを無指定で読み込む（先の例の続き）

入力：
```
df2 = pd.read_csv( 'csv01_nohead.csv', index_col=0 )     # CSV ファイルの読込み
df2     # 内容確認
```

出力：

d1	1	1.1	1.2
d2	2	4	8
d3	3	9	27

これは，先頭にカラム名の行を持たないデータ 'csv01_nohead.csv' を読み込んだ例であるが，先頭行の扱いの指定をしていないことにより，先頭行がカラム名の行であると見なされて不自然な形に整形されている．次に，同じ CSV データにおいて先頭行もデータと見なしてカラム名を与えて（names=['linear','square','cubic']）読み込む例を示す.

例. カラム名を与えて読み込む（先の例の続き）

入力：
```
df2 = pd.read_csv( 'csv01_nohead.csv', index_col=0,
            names=['linear','square','cubic'] )     # CSV ファイルの読込み
df2                                                  # 内容確認
```

出力：

	linear	square	cubic
d1	1	1	1
d2	2	4	8
d3	3	9	27

3.2.2.1 インデックス及びカラム名の無い CSV ファイルの読込み

見出し行の無い CSV ファイルを読み込む際，read_csv に引数 'header=None' を与えると，先頭行をデータとみなし，カラム名が自動的に付加される.

先に作成した CSV ファイル 'csv01_array.csv' を読み込む例を次に示す.

例. カラム名の自動付加（先の例の続き）

入力：
```
df2 = pd.read_csv( 'csv01_array.csv', header=None )     # CSV ファイルの読込み
df2                                                      # 内容確認
```

出力：

	0	1	2
0	1	1	1
1	2	4	8
2	3	9	27

カラム名に，0 から始まる整数が自動的に付加されていることがわかる.

参考） pandas には，Apache Parquet 形式のファイルの入出力の機能もある．これに関しては後の
「7.2 Apache Parquet」（p.140）で解説する.

4　統計処理のための基本的な操作

4.1　基本的なソフトウェアライブラリ

これまでに，多次元の配列データを扱うための NumPy，一般的な表形式データを扱うための pandas といったライブラリを紹介したが，他にも，データ処理に必要とされる各種の機能を提供するライブラリがある．特にここでは，統計処理に関する高度な機能を提供する SciPy と，グラフの作図処理に必要となる matplotlib を取り上げる．

SciPy は NumPy を基礎にして構築されており，データ処理や各種の工学分野で使用する関数やメソッドを提供する．SciPy は各種分野毎のモジュールから成る規模の大きなライブラリであり，本書では SciPy の内，stats ライブラリ（scipy.stats ）を主として取り上げる．

統計処理を行う際のデータ構造としては，基本的に pandas の DataFrame や Series を使用し，具体的な統計処理にはそれらに対するメソッドを実行する．ただし，NumPy や SciPy は，pandas が持たない多くの機能を提供[29] しており，必要に応じて DataFrame や Series を NumPy の配列（ndarray）に変換して NumPy や SciPy のメソッドや関数を使用することもある．本書では必要に応じて NumPy や SciPy による統計処理の方法についても取り上げる．

4.1.1　ライブラリの読込み

これまでにも NumPy, pandas の 2 つのライブラリを読み込んだ形で各種処理の方法について説明してきた．ここではこれらに加えて SciPy の stats モジュールを読み込んだ形（次の例参照）でデータ処理の基本的な方法について解説する．

例. データ処理のための基本的ライブラリの読込み

入力：
```
import pandas as pd          # pandas を 'pd' という名で読込み
import numpy as np           # numpy を 'np' という名で読込み
from scipy import stats      # scipy.stats の読込み
```

ライブラリを読み込む際の「as」の記述であるが，これはライブラリの関数やメソッドを呼び出す際の記述を簡略化するためのものである．例えば NumPy の読込みに当たって，単に

```
import numpy
```

とだけ記述しても良いが，その場合は NumPy の各種関数やメソッドを呼び出す際の接頭辞として

‘**numpy. 関数呼び出し**’

のようにライブラリのフルネームを付けることになる．これに対して，as を用いて ‘np’ のように別名を付与した場合は，その別名を接頭辞とすることができ，記述上の簡略化ができる.（ライブラリの読込み方法に関する詳細は Python 言語に関する他の資料[30] や Python の公式インターネットサイトを参照のこと）

4.2　乱数生成について

SciPy の stats モジュールには各種の乱数を生成するための機能が提供されている．例えば stats モジュールが提供するクラスの 1 つに**一様乱数**の生成に関する uniform があり，このクラスのメソッド rvs を実行することで一様乱数の値（あるいは配列）が得られる．

書き方：　uniform.rvs(loc=下限, scale=乱数生成の幅, size=個数)

このメソッドは「下限」以上で「下限」+「乱数生成の幅」未満の範囲の一様乱数（浮動小数点数）を「個数」に指定した長さの配列として生成する．また引数「scale=」を省略した場合は配列ではなく 1 つの浮動小数点数の値として乱数を生成する．

素朴な意味では，乱数とはどのような値が得られるかが予測できないものである.（次の例参照）

[29]逆に，NumPy や SciPy が持たない機能が pandas 側に存在することもある．

[30]拙書「Python3 入門 - Kivy による GUI アプリケーション開発，サウンド入出力，ウェブスクレイピング」でも解説しています.

例. 一様乱数の生成（先の例の続き）

入力：
```
stats.uniform.rvs(loc=-1,scale=2,size=5)     # -1 以上，1 未満の乱数を 5 個生成
```
出力：
```
array([-0.79151565, -0.04688847, -0.47669814, 0.75754831, -0.20139237])
```

これは一様乱数 5 個の配列を生成した例である．当然のことであるが，同じ処理を再度実行すると，異なる乱数の配列が得られる．

例. 上と同じ方法で一様乱数の生成を再度実行する（先の例の続き）

入力：
```
stats.uniform.rvs(loc=-1,scale=2,size=5)
```
出力：
```
array([-0.85781518, -0.48343919, 0.08791082, -0.0199635 , -0.34880518])
```

4.2.1 得られる乱数の系列について

rvs メソッドが生成する乱数は**疑似乱数**であり，これは，固定された乱数表の値を順番に引用してえられる乱数の並びに似ている．すなわち，乱数表のある特定の位置から値の引用を開始して，以降，乱数表の値の並びに沿って数値を取得する手順に似ており，同じ開始位置から乱数の引用を開始すると同じ系列（パターン）で乱数が得られる．rvs メソッドにはキーワード引数「random_state= **種**」で乱数生成の初期状態を指定することができる．

例. 種を与えて乱数を生成する（先の例の続き）

入力：
```
stats.uniform.rvs(loc=-1,scale=2,size=5,random_state=2)     # 種は 2
```
出力：
```
array([-0.1280102 , -0.94814754, 0.09932496, -0.12935521, -0.1592644 ])
```

例. 再度同じ処理を実行する（先の例の続き）

入力：
```
stats.uniform.rvs(loc=-1,scale=2,size=5,random_state=2)     # 種は 2
```
出力：
```
array([-0.1280102 , -0.94814754, 0.09932496, -0.12935521, -0.1592644 ])
```

2 回続けて同じ乱数系列が得られていることがわかる．「種」には整数値を与える．random_state に与える「種」が異なると，得られる乱数の系列も異なったものになるが，同じ「種」からは同じ系列の乱数が得られる．

重要）
統計処理，データサイエンスの領域におけるプログラミングにおいては，実装したプログラムの動作検証の際に，再現性のあるサンプルデータを与えることが求められる場合がある．そのような場合に，種を指定した形の乱数生成が応用できる．本書でもサンプルデータの作成例を示す際に random_state を指定する方法を取る．

4.2.2 NumPy の乱数生成機能

SciPy の基盤となる NumPy にも乱数生成に関する API が提供されている[31] .（表 9）

表 9: NumPy の乱数生成 API（一部）

書　き　方	解　　説
np.random.rand(個数)	指定した個数の一様乱数を配列として返す.
np.random.normal(loc=平均, scale=標準偏差, 個数)	指定した個数の正規乱数を配列として返す.
np.random.lognormal(mean= μ , sigma= σ , 個数)	指定した個数の対数正規乱数を配列として返す． μ ， σ はそれぞれ，元になる正規分布の平均と標準偏差を意味する.
np.random.seed(種)	乱数生成の種を設定する. 種は $0 \sim 2^{32} - 1$ の整数.

基本的に，乱数の生成結果は SciPy，NumPy 共に同じである.（次の例）

[31] 基本的には SciPy の乱数生成機能は NumPy のそれを応用して実装されている.

例. NumPy，SciPy の乱数生成の比較（先の例の続き）

入力：
```
print('[一様乱数]')
np.random.seed(19)  # NumPy での種の設定
print('   NumPy :',np.random.rand(6))
print('   SciPy :',stats.uniform.rvs(size=6,random_state=19))
print('[正規乱数]')
np.random.seed(17)  # NumPy での種の設定
print('   NumPy :',np.random.normal(size=6))
print('   SciPy :',stats.norm.rvs(size=6,random_state=17))
print('[対数正規乱数]')
np.random.seed(23)  # NumPy での種の設定
print('   NumPy :',np.random.lognormal(size=6))
print('   SciPy :',stats.lognorm.rvs(loc=0,s=1,size=6,random_state=23))
```

出力：
```
[一様乱数]
  NumPy :  [0.0975336   0.76124972 0.24693797 0.13813169 0.33144656 0.08299957]
  SciPy :  [0.0975336   0.76124972 0.24693797 0.13813169 0.33144656 0.08299957]
[正規乱数]
  NumPy :  [ 0.27626589 -1.85462808 0.62390111 1.14531129 1.03719047 1.88663893]
  SciPy :  [ 0.27626589 -1.85462808 0.62390111 1.14531129 1.03719047 1.88663893]
[対数正規乱数]
  NumPy :  [1.94836012 1.02614912 0.45949859 2.58217953 2.0171221   0.34955947]
  SciPy :  [1.94836012 1.02614912 0.45949859 2.58217953 2.0171221   0.34955947]
```

NumPy，SciPy 共に同じ乱数を生成していることがわかる．

4.2.2.1 Generator API

NumPy の 1.17 の版からは乱数生成のための Generator API が導入され，2.0 の版からはこれの使用が推奨されている．古い版の NumPy では乱数生成のアルゴリズムとして MT19937 を採用していた[32] が，Generator API では PCG64 を始めとする新しい乱数生成アルゴリズムが使える．（デフォルトは PCG64）

Generator API では，まず乱数生成器（RNG）である Generator オブジェクトを作成し，それに対する各種のメソッドを実行する形で乱数を生成する．デフォルトの Generator オブジェクトの作成には `default_rng` を用いる．

書き方：　np.random.default_rng(種)

「種」で初期化された Generator オブジェクトを返す．

例. Generator オブジェクトによる乱数生成（先の例の続き）

入力：
```
rng = np.random.default_rng(13)  # Generator オブジェクトの作成
print('[一様乱数]')
print('   NumPy :',rng.random(6))
print('   SciPy :',stats.uniform.rvs(size=6,random_state=np.random.default_rng(13)))
print('[正規乱数]')
rng = np.random.default_rng(29)  # Generator オブジェクトの作成
print('   NumPy :',rng.normal(size=6))
print('   SciPy :',stats.norm.rvs(size=6,random_state=np.random.default_rng(29)))
```

出力：
```
[一様乱数]
  NumPy :  [0.86479759 0.85530251 0.8110234 0.26144636 0.07719946 0.94646578]
  SciPy :  [0.86479759 0.85530251 0.8110234 0.26144636 0.07719946 0.94646578]
[正規乱数]
  NumPy :  [-0.39186984 0.0499204 -0.16519017 -0.16796939 0.06261588 -0.22571108]
  SciPy :  [-0.39186984 0.0499204 -0.16519017 -0.16796939 0.06261588 -0.22571108]
```

NumPy，SciPy 共に同じ乱数を生成していることがわかる．この例でわかるように，Generator オブジェクトに対して乱数生成用のメソッドを実行する形式を取る．また，NumPy の旧 API では一様乱数生成は rand であるが，Generator

[32] メルセンヌ・ツイスタ

API では random メソッドである．SciPy の乱数生成関数の random_state 引数に Generator オブジェクトを与える[33] こともできる．

NumPy の Generator API に関する詳細は他の文献[34] に譲る．本書では乱数生成に SciPy を使用し，random_state には整数値を与える形で解説する．

4.3 サンプルデータの作成

この章ではデータ処理の基本的な方法について説明するが，それに先立ってサンプルデータを作成しておく．用意するデータは乱数の列であり，度数分布が**正規分布**に沿ったものと，**対数正規分布**に沿ったもの（2 種類）を作成する．（正規分布は度数分布のグラフの形状が左右対称，対数正規分布は非対称である）乱数列の生成方法を次の例に示す．

例. サンプルデータ（乱数データ）の作成（先の例の続き）

入力：
```
# 正規分布（μ=0，σ=1）に沿った乱数生成（10,000 個のデータ）
y1 = stats.norm.rvs(loc=0,scale=1,size=10000,random_state=3)
# 対数正規分布（μ=0，σ=1）に沿った乱数生成（10,000 個のデータ）
y2 = stats.lognorm.rvs(loc=0,s=1,size=10000,random_state=1)
```

この例では，scipy.stats モジュールの乱数生成関数を用いている．このモジュールは指定した度数分布に従う形で乱数列を生成する関数群を提供しており，例にある norm.rvs メソッドは正規分布に沿った乱数列を生成する．同様に lognorm.rvs メソッドは対数正規分布に沿った乱数列を生成する．

正規分布に沿った乱数の発生：

stats.norm.rvs(loc=μ, scale=σ, size=生成するデータの個数)

対数正規分布に沿った乱数の発生：

stats.lognorm.rvs(s=σ, loc=オフセット, size=生成するデータの個数)

SciPy ライブラリは NumPy ライブラリを用いて構築されている関係上，基本的に扱うデータは NumPy の配列（ndarray）である．この例では，正規分布の乱数列が y1 に，対数正規分布の乱数列が y2 に，それぞれ ndarray の 1 次元データ列として得られている．

次に，y1, y2 から pandas の DataFrame を作成する．（次の例参照）

例. DataFrame の作成（先の例の続き）

入力：
```
df = pd.DataFrame(columns=['Norm','LogNorm'])  # DataFrame を用意
df['Norm'] = y1;     df['LogNorm'] = y2          # y1, y2 を DataFrame にセット
```

これでサンプルデータの DataFrame が得られた．

4.4 データの分析

四分位数をはじめとする**要約統計量**の調査はデータ分析の初期の段階で行われることが多い．具体的には「2.2.8.1 要約統計量」（p.45）のところで示した方法（describe メソッド）を用いる．

[33] 古い版の SciPy では Generator オブジェクトが使えないので注意すること．

[34] 拙書「Python3 ライブラリブック - 各種ライブラリの基本的な使用方法」でも解説しています．

例. describe メソッドによる要約統計量の取得（先の例の続き）

入力：
```
dsum = df.describe()     # 要約統計量
display( dsum )          # 整形表示
print( '--------' )
print( type(dsum) )      # データ型の調査
```

出力：

	Norm	LogNorm
count	10000.000000	10000.000000
mean	-0.027658	1.673458
std	0.995936	2.299659
min	-3.749941	0.025824
25%	-0.699782	0.515342
50%	-0.028991	1.008490
75%	0.646364	1.957775
max	4.091393	56.083915

```
--------
<class 'pandas.core.frame.DataFrame'>
```

describe メソッドは，対象となる DataFrame の要約統計量（表 10）を各カラム毎に算出して，それらを保持する DataFrame（上記例の dsum）を返す.

表 10: 要約統計量

項目	説明	項目	説明	項目	説明	項目	説明
count	データ個数	mean	平均値	std	標準偏差	min	最小値
25%	25%点	50%	50%点 (中央値)	75%	75%点	max	最大値

この例で得られた DataFrame から更に必要な部分を取り出すことができる.（次の例参照）

例. 要約統計量の個別の値の取出し（先の例の続き）

入力：
```
dsum.loc['std','Norm']   # 通常の DataFrame としてアクセスできる
```

出力：　0.9959356956406281　　← np.float64(0.995935695640628) と表示されることもある

参考） DataFrame がカテゴリカルデータ[35] を含む場合は，describe に引数「include='all'」を与えると，非数値のデータの集計結果も得られる.

4.4.1　分位数（パーセンタイル，パーセント点）

四分位数以外の**分位数（パーセント点，パーセンタイル）**[36] を求めるには quantile メソッドを用いる.

《分位数（パーセンタイル，パーセント点）》
書き方：　DataFrame オブジェクト.quantile(q=比率)
指定した「比率」に対する分位数を DataFrame のカラム毎に求め，結果を Series オブジェクトの形式で返す.

ここで言う「比率」は統計学では「α」や「q」といった記号で表現されることが多い．要素の値を昇順に並べ，最小値から数えた要素の個数の，全要素数に対する比率を意味する．そのような α に対応するデータの値が分位数（パーセンタイル，パーセント点）である.（$0 \leq \alpha \leq 1$）
「分位数」「パーセンタイル」「パーセント点」という用語の意味は文意に沿って読み取っていただきたい.

例. 0.05 のパーセント点の算出（先の例の続き）

入力：
```
df.quantile( q=0.05 )
```

出力：
```
Norm      -1.671452
LogNorm    0.197979
Name:  0.05, dtype:  float64
```

[35] 文字データなどの**非数値のデータセット**．**カテゴリデータ**，**質的データ**と呼ぶこともある.

[36] 「A.1.5 分位数，パーセント点」（p.149）を参照のこと.

quantile メソッドのキーワード引数 ‘q=’ にはデータ列（リストなど）を与えることもでき，複数のパーセント点を求めることができる．その場合は結果が DataFrame の形で得られる．（次の例参照）

例. 0.05, 0.95 の 2 つのパーセント点の算出（先の例の続き）

入力：
```
df.quantile( q=[0.05,0.95] )
```

出力：

	Norm	LogNorm
0.05	-1.671452	0.197979
0.95	1.619248	5.327082

4.4.1.1 中央値

quantile メソッドのキーワード引数 ‘q=’ に 0.5 を指定すると**中央値**が得られる．

例. 中央値（先の例の続き）

入力：
```
df.quantile( q=0.5 ) # 中央値
```

出力：
```
Norm      -0.028991
LogNorm    1.008490
Name:  0.5, dtype:  float64
```

中央値を求めるための median メソッドもあり，‘df.median()’ を評価すると同様の結果が得られる．

4.4.2 基本的な統計量

データ個数，最大値，最小値，合計，平均，分散，標準偏差を個別に求めるメソッドがある．

count メソッドを使用するとデータの個数が得られる．

例. データ個数（先の例の続き）

入力：
```
df.count() # データ個数
```

出力：
```
Norm      10000
LogNorm   10000
dtype:  int64
```

4.4.2.1 最大値，最小値

最大値は max メソッド，最小値は min メソッドで得られる．

例. 最大値（先の例の続き）

入力：
```
df.max()     # 最大値
```

出力：
```
Norm       4.091393
LogNorm   56.083915
dtype:  float64
```

例. 最小値（先の例の続き）

入力：
```
df.min()     # 最小値
```

出力：
```
Norm      -3.749941
LogNorm    0.025824
dtype:  float64
```

4.4.2.2 合計

合計を求めるには sum メソッドを使用する．

例. DataFrame に対する合計（先の例の続き）

入力：
```
df.sum()     # DataFrame に対する合計処理
```

出力：
```
Norm       -276.584894
LogNorm   16734.578765
dtype:  float64
```

例. 特定のカラムに対する合計（先の例の続き）

入力：
```
df['Norm'].sum()     # 特定のカラムに対する合計処理
```
出力：
```
-276.5848939275271          ← np.float64(-276.5848939275271) と表示されることもある
```

4.4.2.3 平均

平均を求めるには mean メソッドを使用する．

例. DataFrame の平均値算出（先の例の続き）

入力：
```
df.mean()     # DataFrame の平均値算出
```
出力：
```
Norm      -0.027658
LogNorm    1.673458
dtype:  float64
```

例. 特定のカラムの平均値算出（先の例の続き）

入力：
```
df['Norm'].mean()     # 特定のカラムの平均値算出
```
出力：
```
-0.02765848939275271          ← np.float64(-0.02765848939275271) と表示されることもある
```

4.4.2.4 分散（不偏分散，標本分散）

分散を求めるには var メソッドを使用する．

例. DataFrame の分散算出（先の例の続き）

入力：
```
df.var()     # DataFrame の分散算出
```
出力：
```
Norm      0.991888
LogNorm   5.288432
dtype:  float64
```

例. 特定のカラムの分散算出（先の例の続き）

入力：
```
df['Norm'].var()     # 特定のカラムの分散算出
```
出力：
```
0.9918879098511819        ← np.float64(0.9918879098511817) と表示されることもある
```

var メソッドに引数を与えずに実行すると**不偏分散**を算出する．上記 2 つの例は不偏分散の例である．**標本分散**を求めるには var メソッドにキーワード引数 'ddof=0' を与えて実行する．

例. 標本分散（先の例の続き）

入力：
```
df['Norm'].var( ddof=0 )     # 標本分散
```
出力：
```
0.9917887210601968        ← np.float64(0.9917887210601967) と表示されることもある
```

var メソッドは 'ddof=1'（不偏分散）が暗黙値である．

4.4.2.5 標準偏差（不偏標準偏差/標本標準偏差）

標準偏差を求めるには std メソッドを使用する．

例. DataFrame の標準偏差（先の例の続き）

入力：
```
df.std()       # DataFrame の標準偏差
```
出力：
```
Norm      0.995936
LogNorm   2.299659
dtype:  float64
```

例. 特定のカラムの標準偏差（先の例の続き）

入力：
```
df['Norm'].std()     # 特定のカラムの標準偏差
```
出力：
```
0.9959356956406281        ← np.float64(0.995935695640628) と表示されることもある
```

std メソッドに引数を与えずに実行すると**不偏標準偏差**を算出する．上記 2 つの例は不偏標準偏差の例である．**標本**

標準偏差を求めるには std メソッドにキーワード引数 'ddof=0' を与えて実行する．

例. 標本標準偏差（先の例の続き）

入力：
```
df['Norm'].std( ddof=0 )     # 標本標準偏差
```

出力：
```
0.9958858976108642        ← np.float64(0.9958858976108642) と表示されることもある
```

std メソッドは 'ddof=1'（不偏標準偏差）が暗黙値である．

4.4.2.6 尖度，歪度

DataFrame のデータの**尖度**，**歪度**を求めるには，kurt メソッド，skew メソッドをそれぞれ用いる．（次の例参照）

例. 尖度（先の例の続き）

入力：
```
df.kurt()     # 尖度
```

出力：
```
Norm       0.003204
LogNorm   96.447775
dtype:  float64
```

例. 特定のカラムの尖度（先の例の続き）

入力：
```
df['Norm'].kurt()     # 特定のカラムの尖度
```

出力：
```
0.003203974884876448        ← np.float64(0.003203974884876448) と表示されることもある
```

例. 歪度（先の例の続き）

入力：
```
df.skew()     # 歪度
```

出力：
```
Norm      0.028590
LogNorm   6.903523
dtype:  float64
```

例. 特定のカラムの歪度（先の例の続き）

入力：
```
df['LogNorm'].skew()     # 特定のカラムの歪度
```

出力：
```
6.9035227025905455        ← np.float64(6.9035227025905455) と表示されることもある
```

参考） pandas の kurt は **Fisher 定義**（正規分布で 0）を採用している．Pearson 定義（正規分布で 3）を期待する場合は SciPy の kurtosis(..., fisher=False) を使用する．

4.4.2.7 配列（ndarray）の統計量の算出

先に説明した統計量の算出を，配列（ndarray）に対して行う方法について説明する．DataFrame や Series が保持するデータは to_numpy メソッドで NumPy の配列（ndarray）として取り出すことができる．例えば配列データのパーセント点を算出するには次のような処理を行う．

例. 0.05 のパーセント点の算出（先の例の続き）

入力：
```
print( np.quantile( df['Norm'].to_numpy(), 0.05 ) )     # 比率で指定
print( np.percentile( df['Norm'].to_numpy(), 5 ) )      # 百分率で指定
```

出力：
```
-1.6714519660353444
-1.6714519660353444
```

このように NumPy の quantile，percentile 関数を使用する．第 1 引数には配列データを，第 2 引数にはパーセント点を与える．この他にも NumPy には統計量を求める関数がある．（表 11 参照）

表 11: 統計用の NumPy の関数（一部）

関数	説明	関数	説明
quantile	パーセント点（比率指定）	percentile	パーセント点（百分率指定）
median	中央値	max	最大値
min	最小値	sum	合計
mean	平均	var	分散
std	標準偏差		

例. NumPy の関数の実行結果（先の例の続き）

入力：
```
print( '中央値:  ', np.median( df['Norm'].to_numpy() ) )          # 中央値
print( '最大値:  ',    np.max( df['Norm'].to_numpy() ) )          # 最大値
print( '最小値:  ',    np.min( df['Norm'].to_numpy() ) )          # 最小値
print( '合計:    ',    np.sum( df['Norm'].to_numpy() ) )          # 合計
print( '平均:    ',   np.mean( df['Norm'].to_numpy() ) )          # 平均
print( '分散:    ',    np.var( df['Norm'].to_numpy(), ddof=1 ) )  # 不偏分散
print( '標準偏差:',    np.std( df['Norm'].to_numpy(), ddof=1 ) )  # 不偏標準偏差
```

出力：
```
中央値:   -0.028991402777364617
最大値:   4.091392759573651
最小値:   -3.749940780113589
合計:     -276.5848939275271
平均:     -0.02765848939275271
分散:     0.9918879098511819
標準偏差:  0.9959356956406281
```

注意）NumPy の var，std にも pandas の場合と同様にキーワード引数 'ddof=' を与えることができるが，NumPy の場合は 'ddof=0' が暗黙値である．（pandas とは逆）

4.4.3 要素，区間毎のデータ個数の調査

4.4.3.1 要素の個数の調査

Series オブジェクトの要素毎の個数を調べるには value_counts メソッドを使用する．

例. 要素毎の個数の調査（先の例の続き）

入力：
```
sr = pd.Series(['a','b','b','c','c','c'])    # サンプルデータ
c = sr.value_counts()
print( c )          # 内容確認
print( '--------' )
print( type(c) )    # データ型の調査
```

出力：
```
c  3
b  2
a  1
dtype:  int64
--------
<class 'pandas.core.series.Series'>
```

Series オブジェクト sr の要素の個数の調査結果が別の Series オブジェクト c として得られていることがわかる．このオブジェクトは，調査対象の要素をインデックスとして持つ．

4.4.3.2 最頻値

mode メソッドを使用すると**最頻値**が得られる．

例. 最頻値（先の例の続き）

入力：
```
sr.mode()       # 最頻値
```

出力：
```
0  c
dtype:  object
```

要素「c」が最頻値であることがわかる．最頻値は複数得られることがあり，結果は Series オブジェクトとして得られる．

mode メソッドは DataFrame に対して使用することができる．次にその例を示す．まず次のようにしてサンプルの DataFrame を作成する．

例. サンプルの作成（先の例の続き）

入力：
```
sr2 = pd.Series(['a','b','b','b','c','c']) # サンプルデータ 2
sr3 = pd.Series(['b','b','b','c','c','c']) # サンプルデータ 3
# DataFrame を作成
df4 = pd.DataFrame( columns=['col1','col2','col3'] )
df4['col1'] = sr; df4['col2'] = sr2; df4['col3'] = sr3
df4    # 内容確認
```

出力：

	col1	col2	col3
0	a	a	b
1	b	b	b
2	b	b	b
3	c	b	c
4	c	c	c
5	c	c	c

このようにして 3 つのカラムを持つ DataFrame オブジェクト df4 ができた．これに対して mode メソッドを使用する例を次に示す．

例. DataFrame の最頻値（先の例の続き）

入力：
```
df4.mode() # DataFrame の最頻値
```

出力：

	col1	col2	col3
0	c	b	b
1	NaN	NaN	c

各カラム毎の最頻値が DataFrame として得られている．（最頻値の要素の個数が足りない部分は NaN が埋められる）

4.4.3.3 区間毎の要素の個数の調査

value_counts メソッドは値の区間毎の度数調査にも使用できる．先に作成したサンプルの DataFrame オブジェクト df（正規分布，対数正規分布に沿った乱数）を用いてそれを示す．

例. 区間毎の個数の調査（先の例の続き）

入力：
```
B0 = df['Norm'].value_counts(bins=20)     # 度数の区間を 20 等分して度数調査
print( B0.head(3) )                        # 先頭 3 要素を表示
print( '--------' )
print( 'データ個数:', len(B0) )
```

出力：
```
(-0.221, 0.171]   1517
(-0.613, -0.221]  1475
(0.171, 0.563]    1457
Name:  Norm, dtype:  int64
--------
データ個数:  20
```

この例のように value_counts メソッドにキーワード引数 'bins=**区間の分割数**' を与えると，指定した分割数で等分する形で区間の幅を算出し，それぞれの区間に属するデータの個数を集計する．得られた Series データは各区間をインデックスに持ち，それぞれの区間に対応する集計結果の値を要素として持つ．区間は表 12 に示すような括弧で上限と下限を括る形で表現される．

value_counts メソッドで得られた Series オブジェクトを区間毎に整列（インデックスでソート）すると，階級の順序に整列された度数分布のデータとなる．（次の例参照）

表 12: 区間の表現

下限	解説	上限	解説
‘(’	下限と同じ値は含まない	‘)’	上限と同じ値は含まない
‘[’	下限と同じ値を含む	‘]’	上限と同じ値を含む

例. 度数分布のデータとして整列（先の例の続き）

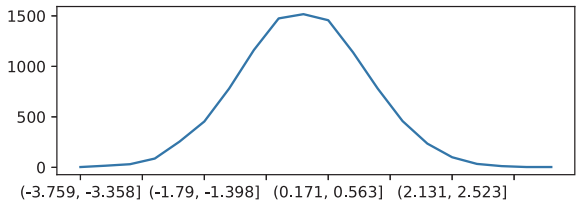
入力：
```
B0.sort_index(inplace=True)     # インデックス（区間）でソート
```

この処理で Series オブジェクト B0 が階級順に整列され，度数分布のデータとして扱える．このデータをグラフにプロットして**ヒストグラム**を作成することができる．グラフをプロットするには，そのためのライブラリを読み込み，作図に関するメソッドを実行する．(次の例参照)

例. ヒストグラムの作成（先の例の続き）

入力：
```
import matplotlib.pyplot as plt # グラフ描画ライブラリを 'plt' の名で読込み
plt.figure()                    # 描画の準備
B0.plot( figsize=(6,2) )        # 描画処理（サイズ指定）
plt.show()                      # 表示
```

出力：



グラフのプロットのためのライブラリ matplotlib から matplotlib.pyplot を読み込むことで，各種グラフの描画とそれに関連する機能が利用できる．この例では matplotlib.pyplot を ‘plt’ の名前で読み込んでいる．

ヒストグラムを描画する方法にはもっと簡便な方法があり，後の「4.5.2 ヒストグラムの作成」(p.71) で解説する．データを可視化する方法については後の「4.5 データの可視化」(p.71) で更に詳しく解説する．

4.4.3.4 最頻の区間

value_counts メソッドで区間毎の要素の数を求めた後，max メソッドによって要素数の最大値を求めることで，最頻の区間を調べることができる．先に value_counts メソッドで得た Series オブジェクト B0 から最頻の区間を調べる例を示す．

例. 最頻の区間を調べる（先の例の続き）

入力：
```
B0.max()
```

出力：
```
1517          ← np.int64(1517) と表示されることもある
```

更に iloc と組み合わせると，最頻値を持つ区間のデータを抽出することができる．(次の例参照)

例. 最頻値の区間を抽出（先の例の続き）

入力：
```
B0.iloc[ list(B0 == B0.max()) ]
```

出力：
```
(-0.221, 0.171]  1517
Name:  Norm, dtype:  int64
```

4.4.3.5 区間（Interval オブジェクト）

区間は Interval 型のオブジェクトとして与えられる．(次の例参照)

例. Interval オブジェクト（先の例の続き）

入力：
```
r = B0.index[0]                          # 最初の「区間」の取り出し
print( '区間のデータ型:', type(r) )
r        # 内容確認
```

出力：
```
区間のデータ型:  <class 'pandas._libs.interval.Interval'>
Interval(-3.759, -3.358, closed='right')
```

Interval オブジェクトは，下限（左側）を意味する left，上限（右側）を意味する right，それに閉区間を意味する closed といったプロパティを持つ.（次の例参照）

例. Interval オブジェクトのプロパティ（先の例の続き）

入力：
```
print( '下限:', r.left )
print( '上限:', r.right )
print( '閉区間:', r.closed )
```

出力：
```
下限:  -3.759
上限:  -3.358
閉区間:  right
```

closed プロパティが取る値は表 13 のようなものである.

表 13: Interval オブジェクトの closed プロパティの値

値	解説
'left'	下限を含み上限を含まない（下限は閉じている．上限は開いている）
'right'	下限を含まず上限を含む（下限は開いており，上限が閉じている）
'both'	下限, 上限ともに含む（下限, 上限ともに閉じている）
'neither'	下限, 上限ともに含まない（下限, 上限ともに開いている）

Interval オブジェクトを作成する例を次に示す.

例. $(2, 3]$ を意味する Interval オブジェクトを作成する（先の例の続き）

入力：
```
iv = pd.Interval( 2.0, 3.0, closed='right' )
print( iv )
```

出力：
```
(2.0, 3.0]
```

ある値が Interval オブジェクトが示す範囲にあるかどうかを in 演算子で調べることができる.

例. 与えた値が $(2, 3]$ にあるかどうかを調べる（先の例の続き）

入力：
```
print( '2.5 ∈ (2.0,3.0] :', 2.5 in iv )
print( '2.0 ∈ (2.0,3.0] :', 2.0 in iv )
print( '3.0 ∈ (2.0,3.0] :', 3.0 in iv )
```

出力：
```
2.5 ∈ (2.0,3.0] :  True
2.0 ∈ (2.0,3.0] :  False
3.0 ∈ (2.0,3.0] :  True
```

4.5 データの可視化

4.5.1 matplotlib による作図の手順

matplotlib ライブラリは，データを可視化するための多種多様な機能を提供する．このライブラリは使用に先立って次のようにして Python 処理系に読み込んでおく．

```
import matplotlib.pyplot as plt
```

これにより，‘plt.〜’ の接頭辞の下で matplotlib のクラスや関数などを使用することができる．

例. matplotlib の読込み（先の例の続き）

入力：
```
import matplotlib.pyplot as plt
```

NumPy の ndarray や pandas の DataFrame として用意されたデータを可視化するための基本的な手順は次のようなものである．

1) 作図の準備

matplotlib の figure 関数を呼び出して，作図処理に必要となる準備を整える．このとき，グラフの描画サイズをキーワード引数 ‘figsize=(**横のサイズ**, **縦のサイズ**)’ で指定することができる．

例. `plt.figure( figsize=(6,2) )`　←描画サイズを 6×2 とする

多くの場合において，figure 関数の実行は省略できる．

1 つの描画面に複数のグラフを表示する場合は subplots を使用する．

例. `plt.subplots( 2,3, figsize=(20,5))`

これは，20 × 5（インチ）の描画面内に 2 行 3 列（合計 6 個）のグラフを作成する設定である．subplots は (Figure オブジェクト,Axes オブジェクトの配列) の組を返し，Axes オブジェクトが個々のグラフを描画する対象となる．

2) 作図に関する処理

描画するグラフのサイズやタイトルの設定，各種グラフの描画，描画したグラフのファイルへの保存といった各種の処理を行う．

3) 表示に関する処理

matplotlib の show 関数を呼び出して，作成した図を実際にウィンドウに表示する．この段階で作図処理は終了する．Jupyter Notebook の環境下ではこの処理を省略することができるが，明示的に作図を実行する際には必要である．

本書では matplotlib の使用例を示すにとどめ，その詳細に関しては言及しない．詳しくは matplotlib の公式インターネットサイトをはじめとする他の資料[37] を参照のこと．

4.5.2 ヒストグラムの作成

データの可視化の際によく作成されるものとして**ヒストグラム**（柱状の**度数分布図**）がある．pandas では，matplotlib ライブラリと連携してヒストグラムを作成するための機能が提供されている．

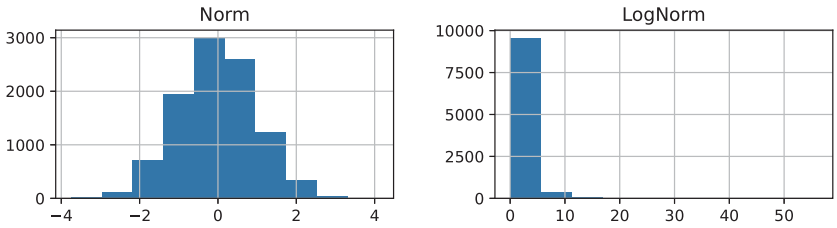
DataFrame に対して hist メソッドを使用することでヒストグラムを作成することができる．(次の例参照)

[37] 拙書「Python3 ライブラリブック - 各種ライブラリの基本的な使用方法」でも解説しています．

例. ヒストグラムの描画（先の例の続き）

入力：
```
df.hist()          # ヒストグラムの作成処理
```

出力：
```
array([[<Axes:  title='center':  'Norm'>,
        <Axes:  title='center':  'LogNorm'>]], dtype=object)
```

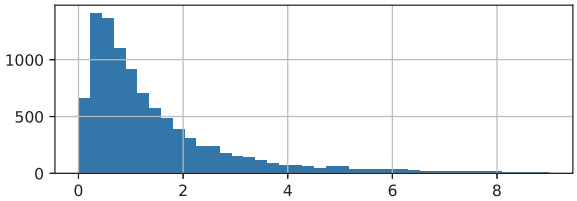


このように，全てのカラムのデータが可視化される．hist メソッドは，指定したカラムのみに対して実行することもできる．その例を次に示す．

例. カラムを指定してヒストグラムを描画（先の例の続き）

入力：
```
df['LogNorm'].hist( bins=40,          # 階級の数
                    range=(0,9),      # 範囲
                    figsize=(6,2) ) # 図の大きさ
```

出力：
```
<Axes:  >
```



この例は，DataFrame オブジェクト df の 'LogNorm'（対数正規分布のデータ）のカラムのデータを可視化したものである．また，hist メソッドにいくつかのキーワード引数を与え，可視化に関する各種の設定（表 14 参照）を施している．

表 14: hist メソッドに与えるキーワード引数（一部）

キーワード引数	説明
bins=**整数**	グラフにする階級の数（柱の数）．デフォルトは 'auto'．'fd'，'sturges'，'doane' も指定可能
range=(**下限，上限**)	グラフ化する階級値の範囲
alpha=**アルファ値**	グラフのアルファ値（不透明度）を 0〜1.0 の範囲で与える．(1.0 が暗黙値)
figsize=(**横幅，高さ**)	作成するグラフのサイズ *
layout=(**行，列**)	複数のグラフを描画する際のレイアウト．デフォルトでは自動調整．

＊ 公式ドキュメントには「単位はインチ」とあるが，実際の表示サイズは異なることがある．

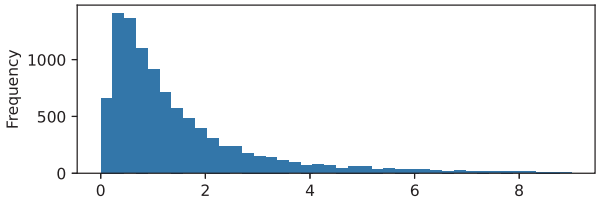
4.5.2.1 ヒストグラム作成方法のバリエーション

DataFrame，Series オブジェクトに対して hist メソッドを実行する方法を先に示したが，それ以外にも，plot メソッドや plot.hist メソッドでヒストグラムを作成する方法もある．

例. plot メソッドによるヒストグラムの描画（先の例の続き）

入力：
```
df['LogNorm'].plot( kind='hist',bins=40,range=(0,9),figsize=(6,2) )
```

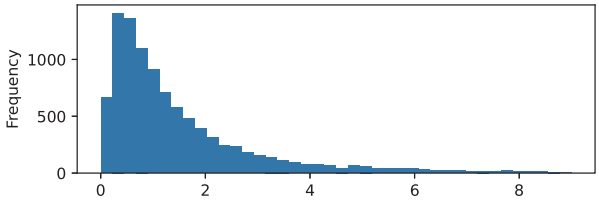
出力：
```
<Axes:  ylabel='Frequency'>
```



例. plot.hist メソッドによるヒストグラムの描画（先の例の続き）

入力：
```
df['LogNorm'].plot.hist( bins=40,range=(0,9),figsize=(6,2) )
```

出力：
```
<Axes:  ylabel='Frequency'>
```



これらの方法で作図すると自動的に縦軸のラベルがグラフに表示される．

4.5.3 図を画像ファイルとして保存する方法

matplotlib によって描画したグラフは，savefig 関数によって，各種フォーマットの画像ファイルとして保存することができる．この関数は show 関数で描画するまでに実行する．

例. グラフを画像ファイルとして保存する（先の例の続き）

入力：
```
df['LogNorm'].hist( bins=40,          # 階級の数
                    range=(0,9),      # 範囲
                    figsize=(6,2) )   # 図の大きさ
plt.savefig( 'histLogNorm01.eps' )    # 画像ファイルとして保存
```

Jupyter Notebook で実行すると，画像ファイルが保存された後，Notebook 内にグラフが表示される．この例のように，savefig 関数の引数に，保存先ファイルの名前やパスを文字列で与える．保存されるファイルのフォーマットは，ファイル名の拡張子によって判断される．

作図の結果として得られる Figure オブジェクトに対する savefig メソッド[38] もある．

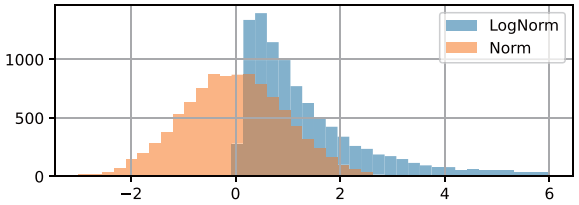
4.5.4 複数の図を重ねて表示する方法

グラフ描画処理が終了するまで（表示処理をするまで）に描画したグラフは，その順に重ねて表示される．その際，次の例のようにアルファ値（不透明度）を 1.0 未満にすると見やすい表示となる．

例. グラフを重ねて表示する（先の例の続き）

入力：
```
df['LogNorm'].hist( bins=40,           # 階級の数
                    range=(-3,6),      # 範囲
                    alpha=0.5,         # アルファ値（不透明度）
                    figsize=(6,2),     # 図の大きさ
                    label='LogNorm' )  # 凡例用ラベル
df['Norm'].hist( bins=40,              # 階級の数
                    range=(-3,6),      # 範囲
                    alpha=0.5,         # アルファ値（不透明度）
                    label='Norm' )     # 凡例用ラベル
plt.legend()  # 凡例表示
```

出力：
```
<matplotlib.legend.Legend at 0x1a41f9de0d0>
```



[38] 保存対象のグラフを明確に指定できるので，こちらの使用が推奨される．

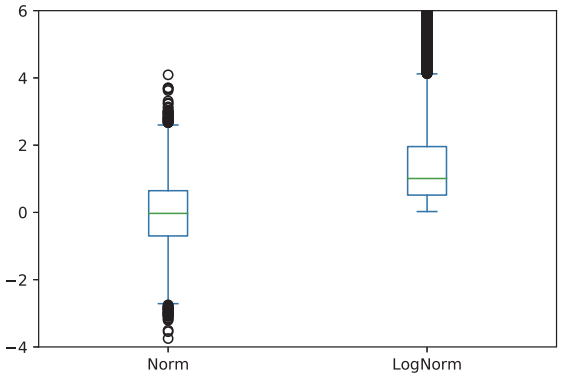
4.5.5　箱ひげ図の作成

DataFrame に対する plot メソッドにキーワード引数 'kind=box' を与えると**箱ひげ図**が描画される.

例. 箱ひげ図の作成（先の例の続き）

入力：
```
df.plot(kind='box',figsize=(6,4))        # 箱ひげ図
plt.ylim(-4,6)
```

出力：
```
(-4.0, 6.0)
```



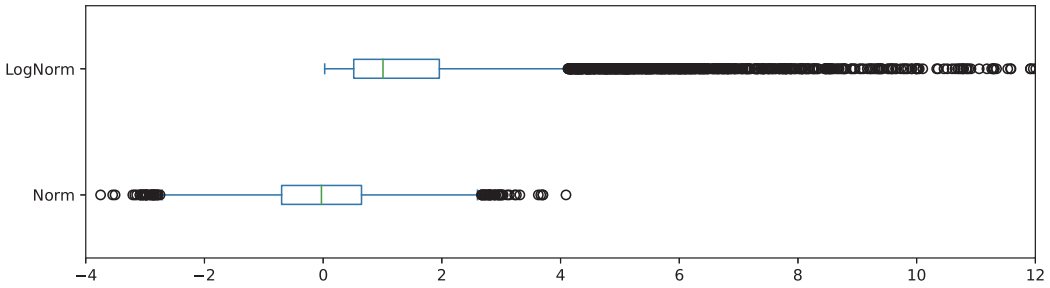
この例と同様のことが「`df.plot.box()`」という記述でも実行できる.

横向きの箱ひげ図としてプロットするには引数「`vert=False`」を与える.（次の例参照）

例. 横向きの箱ひげ図（先の例の続き）

入力：
```
df.plot.box(vert=False,figsize=(7,2))      # 箱ひげ図（横向き）
plt.xlim(-4,12)
```

出力：
```
(-4.0, 12.0)
```



【解説】箱ひげ図

箱ひげ図はデータの度数の分布を表現するものであり，25 パーセント点（$Q_{1/4}$），50 パーセント点（$Q_{2/4}$：中央値），75 パーセント点（$Q_{3/4}$）を「箱」で表示する．また，箱の長さ $Q_{3/4} - Q_{1/4}$ を IQR（interquartile range）として，有効なデータの範囲を $Q_{1/4} - 1.5{\times}IQR \sim Q_{3/4} + 1.5{\times}IQR$ であると考え，その範囲内にないデータは「外れ値」とみなす.（図 1）

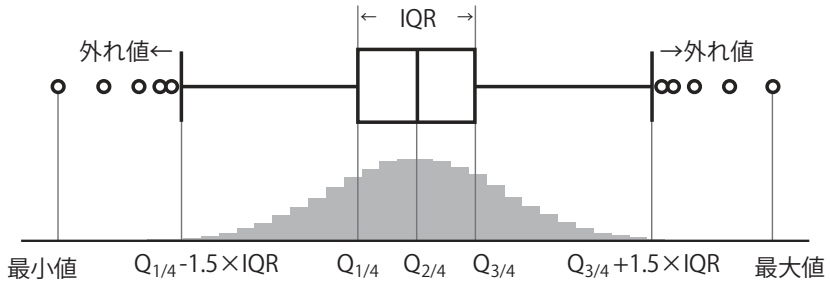


図 1: 箱ひげ図

4.5.6　折れ線グラフの作成

DataFrame に対して plot メソッドを使用することで折れ線グラフを作成することができる．まずはサンプルデータを DataFrame オブジェクト df2 として作成する．(次の例参照)

例. サンプルデータ（sin,cos）の作成（先の例の続き）

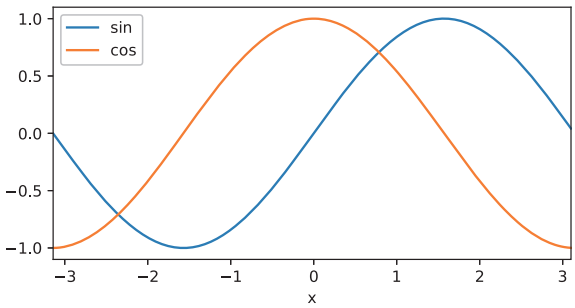
入力：
```
import numpy as np     # NumPy を 'np' という名で読込み
x = np.arange(-3.14,3.14,0.3)                          # sin,cos の定義域を生成
y1 = np.sin(x);      y2 = np.cos(x)                    # sin,cos のデータ列を生成
df2 = pd.DataFrame(columns=['x','sin','cos'])      # DataFrame の生成
df2['x']=x;      df2['sin']=y1;      df2['cos']=y2   # DataFrame にセット
```

ここで生成した df2 は $-\pi \leq x < \pi$ の定義域に対する $\sin(x), \cos(x)$ の値を持つもので，これを可視化する．(次の例参照)

例. 折れ線グラフの描画（先の例の続き）

入力：
```
df2.plot( x='x',            # 横軸のカラム
        y=['sin','cos'],    # 縦軸のカラム（複数指定可）
        figsize=(6,3))      # グラフのサイズ
```

出力：
```
<Axes:  xlabel='x'>
```



plot メソッドに与えるキーワード引数の一部を表 15 に示す．

表 15: plot メソッドに与えるキーワード引数（一部）

キーワード引数	説明
x=カラム名	グラフの横軸に与えるカラムの名前
y=カラム名	グラフの縦軸に与えるカラムの名前（複数可）

4.5.6.1　線の太さ，線種，マーカー

plot メソッドに引数「lw=**太さ**」を与えることで折れ線グラフの線の太さ（ポイント単位）を指定することができる．

例. 線の太さを 3 ポイントに指定（先の例の続き）

入力：
```
df2.plot( x='x', y=['sin','cos'], figsize=(6,3), lw=3)
```

この処理によって得られるグラフを図 2 の (a) に示す．

例. 線の太さを 7 ポイントに指定（先の例の続き）

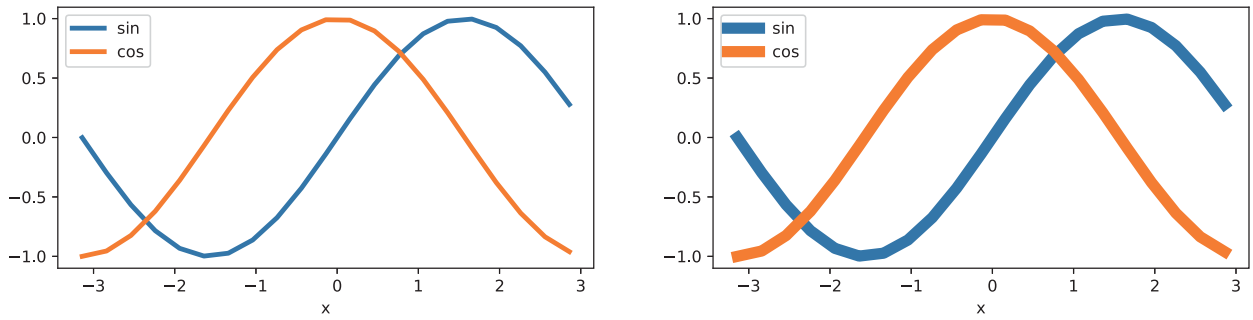
入力：
```
df2.plot( x='x', y=['sin','cos'], figsize=(6,3), lw=7)
```

この処理によって得られるグラフを図 2 の (b) に示す．

plot メソッドに引数「ls=**線種**」を与えることで折れ線グラフの線種を指定することができる．

例. 引数 ls='–' を与えて実行（先の例の続き）

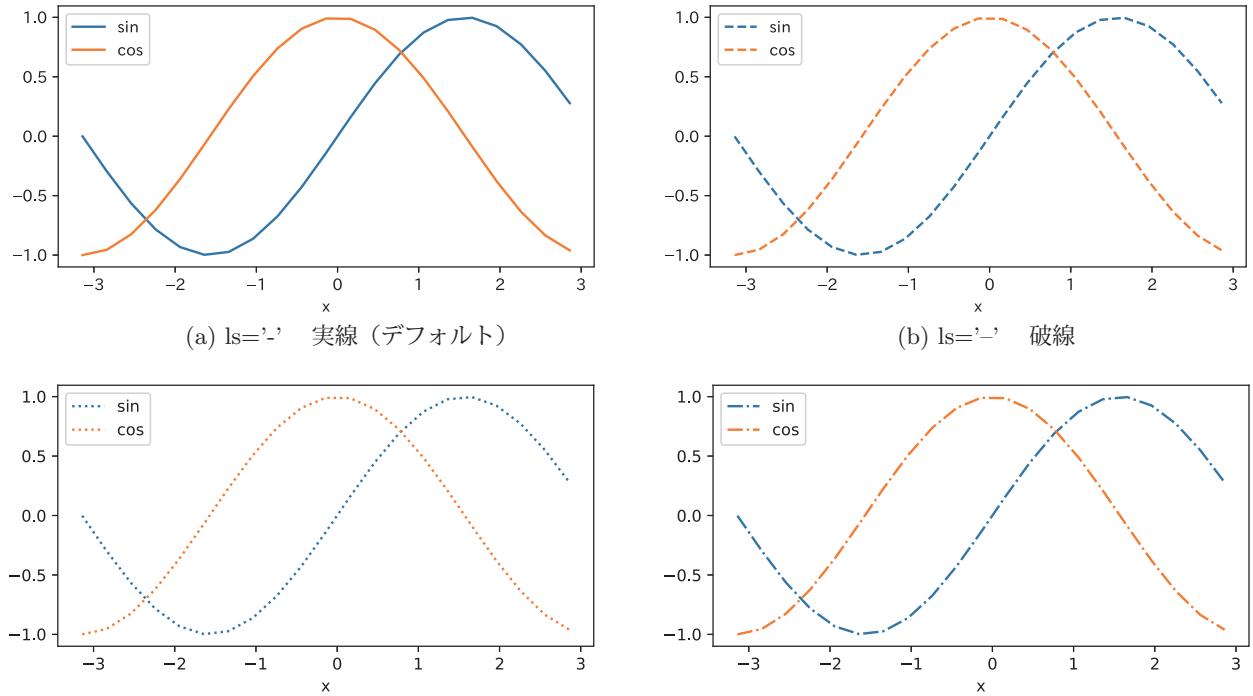
入力：
```
df2.plot( x='x', y=['sin','cos'], figsize=(6,3), ls='--')
```



(a) lw=3　　(b) lw=7

図 2: 折れ線グラフの線の太さの指定

この処理によって得られるグラフを図 3 の (b) に示す.



(a) ls='-'　実線（デフォルト）　　(b) ls='–'　破線

(c) ls=':'　点線　　(d) ls='-.'　一点鎖線

図 3: 折れ線グラフの線種の指定

引数「ls=」に与える文字列によって様々な線種になることがわかる.

plot メソッドに引数「marker=**マーカーの種類**」を与えることで折れ線グラフ上のデータ点に**マーカー**を表示することができる.

例. 引数 marker='o' を与えて実行（先の例の続き）

入力：
```
df2.plot( x='x', y=['sin','cos'], figsize=(4,2), marker='o')
```

この処理によって得られるグラフを図 4 の (a) に示す.

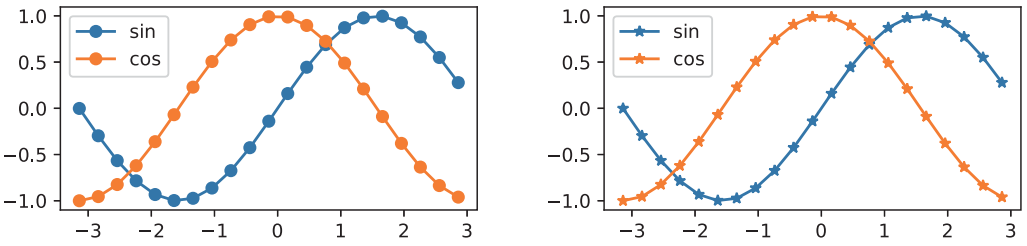
引数「marker=」に与える文字列によって様々なマーカーが表示される（図 4）ことがわかる.

参考）ここで紹介したもの以外にも多くのマーカーが指定できる. 詳しくは matplotlib の公式インターネットサイト（https://matplotlib.org/stable/api/markers_api.html）を参照[39] のこと.

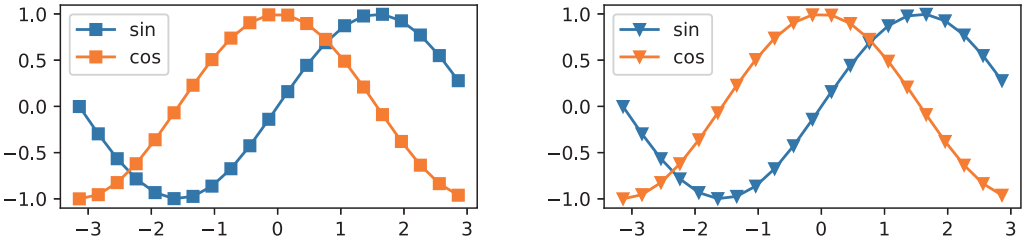
4.5.6.2　グラフの色

plot メソッドに引数「color=**色**」を与えることで折れ線グラフの色を指定することができる.

[39]拙書「Python3 ライブラリブック - 各種ライブラリの基本的な使用方法」でも解説しています.



(a) marker='o'　丸　　(b) marker='*'　星



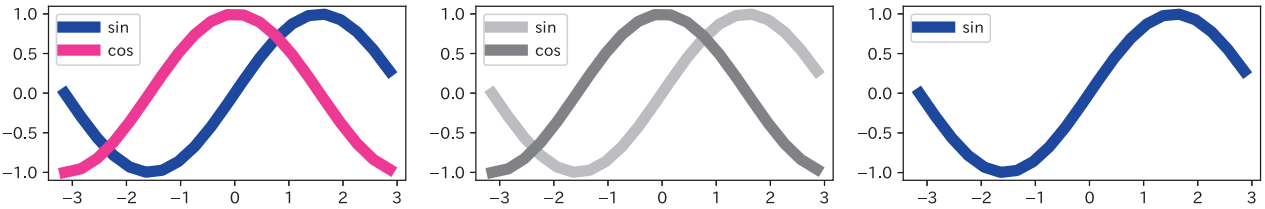
(c) marker='s'　四角　　(d) marker='v'　逆三角

図 4: データ点のマーカーの指定

例. 色を指定してプロット（先の例の続き）

入力：
```
df2.plot( x='x', y=['sin','cos'], figsize=(4,2), lw=7,color=['#0000cd','#ff1493'] )
```

この処理によって得られるグラフを図 5 の (a) に示す．色の指定は HTML における CSS の色名を使用することも可能で，上の実行例において「`color=['mediumblue','deeppink']`」としても同様の結果となる．



(a) color=['#0000cd','#ff1493']　(b) color=['0.7','0.4']　(c) color='mediumblue'

図 5: 色の指定

■ グレースケールの明るさによる指定

引数「color=」にグレースケールの明るさの値（0〜1.0：黒〜白）を文字列の形で与えることもできる．

例. グレースケールの明るさを指定してプロット（先の例の続き）

入力：
```
df2.plot( x='x', y=['sin','cos'], figsize=(4,2), lw=7,color=['0.7','0.4'] )
```

この処理によって得られるグラフを図 5 の (b) に示す．

1 つのデータ系列（1 つのグラフ）をプロットする際は色をリストではなく 1 つの値として指定できる．

例. 1 つのグラフをプロットする場合（先の例の続き）

入力：
```
df2.plot( x='x', y='sin', figsize=(4,2), lw=7,color='mediumblue' )
```

この処理によって得られるグラフを図 5 の (c) に示す．

4.5.7　グラフ描画に関する各種の設定

matplotlib の描画環境（`plt.figure()` 〜 `plt.show()` の間）でグラフ描画に関する各種の設定を行う関数を表 16 に示す．

表 16: グラフ描画に関する各種の設定を行う関数（一部）

関数	説明
xlim(下限, 上限)	描画対象とする横軸の領域を指定する.
ylim(下限, 上限)	描画対象とする縦軸の領域を指定する.
title(タイトル)	グラフのタイトルを設定する.
xlabel(横軸のラベル)	横軸のラベルを設定する.
ylabel(縦軸のラベル)	縦軸のラベルを設定する.
legend( )	描画領域に凡例を表示する.
grid( )	グリッド（格子）を表示する.

※ 関数呼出し時にはライブラリの接頭辞（‘plt.’ など）を付ける.

例. matplotlib の描画環境における各種の制御（先の例の続き）

入力：
```
df2.plot( x='x', y=['sin','cos'], figsize=(6,3))
plt.title('trigonometric functions')
plt.ylabel('value')
plt.xlim( -2.5, 2.5 )
plt.ylim( -1.4, 1.4 )
plt.grid()
```

この処理によって得られるグラフを図 6 に示す.

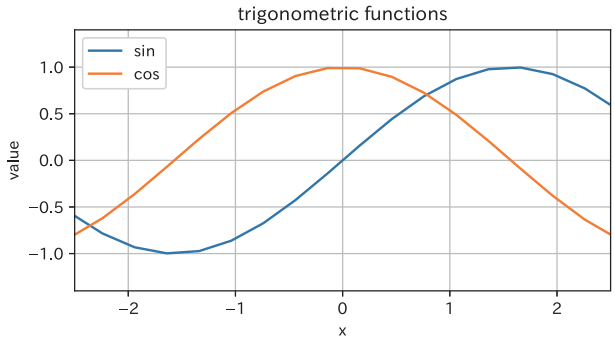


図 6: matplotlib の機能を用いたグラフ表示の設定の例

4.5.8 応用例：value_counts の結果をヒストグラムにする

先の「4.5.2 ヒストグラムの作成」(p.71) では，データの度数分布を可視化する方法について説明したが，ここでは，度数分布の調査の結果（p.67「4.4.3 要素，区間毎のデータ個数の調査」で示した方法）を元にヒストグラムを作成する方法に関して例を示す.

value_counts メソッドによって区間毎の度数の調査結果は Series オブジェクトとして得られる．これを DataFrame の形式にしておくとデータ処理の様々な局面で便利であることが多い．value_counts メソッドによって先に作成した Series オブジェクト B0 を DataFrame オブジェクトに変換する手順について考える.

まず Interval オブジェクトが持つ各種の値（下限，上限，閉区間情報）を DataFrame の各カラムに展開しておくと，それらの値が DataFrame の項目として扱えるようになる．そのために，インデックスの列に収められた Interval オブジェクトを一度に展開して，下限，上限，閉区間情報を別々のデータ列として取り出す.（次の例参照）

例. インデックス列の Interval オブジェクトを一度に展開する（先の例の続き）

入力：
```
ixl = B0.index.left
ixr = B0.index.right
ixc = B0.index.closed
print('closed :',ixc)
```

出力：
```
closed :  right
```

この作業によって，ixl, ixr, ixc にそれぞれ下限値の列，上限値の列，閉区間情報の列が得られる．次に，これらのデータ列から DataFrame を作成する．(次の例参照)

例. DataFrame の作成（先の例の続き）

入力：
```
# DataFrame の作成
dfBins = pd.DataFrame( columns=['left','right','freq'] )
# 値の設定
dfBins['left'] = list( ixl )
dfBins['right'] = list( ixr )
dfBins['freq'] = B0.to_numpy()
```

これで，度数の調査結果が DataFrame オブジェクト dfBins として得られたことになり，度数分析が行いやすくなる．

得られた dfBins を整列してヒストグラムにする例を次に示す．

例. 区間の上限値で整列する（先の例の続き）

入力：
```
dfHist = dfBins.sort_values('right')     # 区間の上限値で整列
display( dfHist.head(3) )                # 確認表示:先頭 3 行
```

出力：

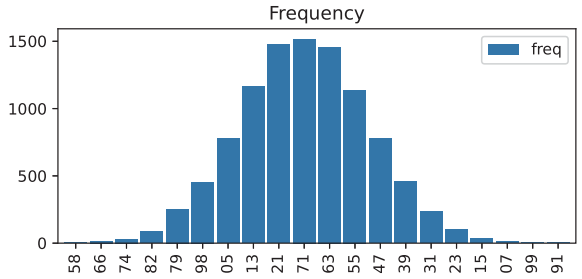
	left	right	freq
0	-3.759	-3.358	3
1	-3.358	-2.966	16
2	-2.966	-2.574	31

整列されたデータを用いてヒストグラムを描画する．(次の例参照)

例. ヒストグラムの描画（先の例の続き）

入力：
```
dfHist.plot(kind='bar',x='right',y='freq', width=0.9, figsize=(6,2.5) )
plt.title('Frequency')
```

出力：
```
Text(0.5, 1.0, 'Frequency')
```



4.5.9 円グラフの作成

円グラフは，数値データの並びを扇形で表現するものである．円グラフは DataFrame オブジェクトや Series オブジェクトに対して plot や plot.pie を実行することで作成する．

書き方 1： **DataFrame オブジェクト.plot(kind='pie', y=対象カラム)**
書き方 2： **DataFrame オブジェクト.plot.pie(y=対象カラム)**
書き方 3： **Series オブジェクト.plot(kind='pie')**
書き方 4： **Series オブジェクト.plot.pie( )**

ここでは，作成したサンプルデータを円グラフにする形で説明する．

例. サンプルデータ（国別 GDP）の作成（先の例の続き）

入力：
```
g = [ [20.894, 22.675],
      [14.867, 16.642],
      [ 5.045,  5.378],
      [ 3.843,  4.319] ]
G = pd.DataFrame(g,index=[’米国’,’中国’,’日本’,’ドイツ’],columns=[’2020’,’2021’])
G
```

出力：

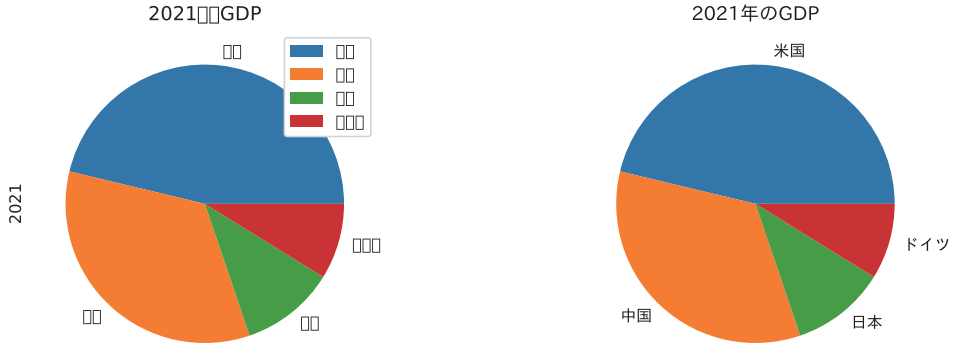
	2020	2021
米国	20.894	22.675
中国	14.867	16.642
日本	5.045	5.378
ドイツ	3.843	4.319

このようにして作成した DataFrame オブジェクト G から円グラフを作成する例を示す.

例. plot メソッドによる円グラフの作成（先の例の続き）

入力：
```
G.plot(kind=’pie’,y=’2021’,title=’2021 年の GDP’)
```

これを実行すると図 7 の (a) のような円グラフが表示される．ただし，この例では日本語フォントが正しく適用されておらず，グラフ上の各部分が正しく表示されていない.



(a) 日本語が正しく表示されていない例　　(b) 日本語フォントの設定を行った後

図 7: 円グラフの表示

4.5.9.1　グラフ作成における日本語フォントの使用

pandas の作図機能は matplotlib を用いて実装されている．従ってグラフ作成において日本語フォントを正しく表示するには matplotlib の font_manager の機能を使用して，正しく設定された FontProperties オブジェクトを作図メソッドに与える[40]ことが必要となることがある．この作業を簡潔に行うためのサードパーティのライブラリ japanize-matplotlib が存在しており，本書ではこのライブラリを使用[41] してグラフ中の日本語フォントを表示する方法を取る.

システムにインストールされている japanize-matplotlib ライブラリを使用するには

```
import japanize_matplotlib
```

として Python 上に読み込む[42] .

この作業の後に次のような処理を実行して，先に作成した DataFrame オブジェクト G を円グラフにすると図 7 の (b) のように表示される.

例. 日本語を正しく表示する円グラフ（先の例の続き）

入力：
```
G.plot(kind=’pie’,y=’2021’,title=’2021 年の GDP’,legend=False,ylabel=’’)
```

この例では plot メソッドに引数「`legend=False`」を与えて凡例を無効にしている．また，引数「`ylabel=’’`」を与

[40] これに関しては拙書「Python3 ライブラリブック - 各種ライブラリの基本的な使用方法」でも解説しています.

[41] このライブラリを pip コマンドで Python 処理系にインストールするには，インターネット接続環境下で OS のコマンド「`pip install japanize-matplotlib`」を実行する.

[42] インストール時は「`japanize-matplotlib`」と記述するが，import 時には「`japanize_matplotlib`」と記述することに注意すること.

えてカラム名の表示（グラフ左側）を無効にしている．

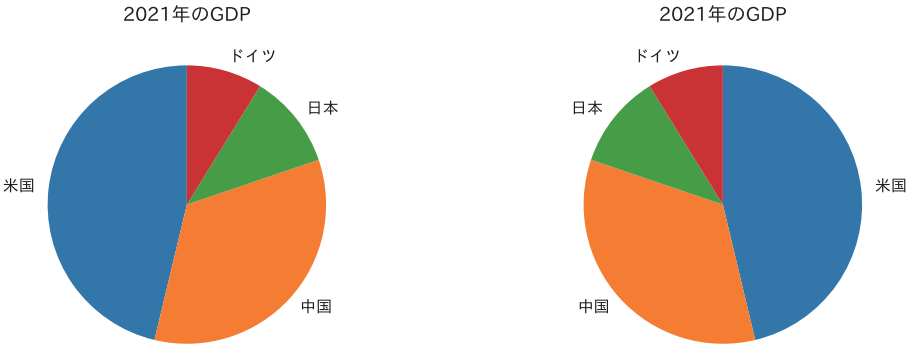
4.5.9.2　円グラフ描画の開始角度と回転方向

円グラフの描画開始の角度は暗黙で 0 °（時計の 3 時の位置）である．また角度は**反時計回り**に進む．円グラフの扇形の表示の開始角度を設定するには引数「startangle=**角度**」（単位は ° ）を与える．

例. 描画開始角度を真上（90 ° ）にする（先の例の続き）

入力：
```
G['2021'].plot.pie(title='2021 年の GDP',ylabel='',startangle=90)
```

この例では描画開始の角度を 90 ° にしており，この処理の結果，図 8 の (a) のような表示となる．



(a) 描画開始角度を真上にする　　(b) 時計回りの順に表示

図 8: 描画開始角度と表示方向

扇形の描画の方向を時計回りにするには引数「counterclock=False」を与える．

例. 時計回りの順に表示する（先の例の続き）

入力：
```
G['2021'].plot.pie(title='2021 年の GDP',ylabel='',startangle=90,counterclock=False)
```

この処理の結果，図 8 の (b) のような表示となる．

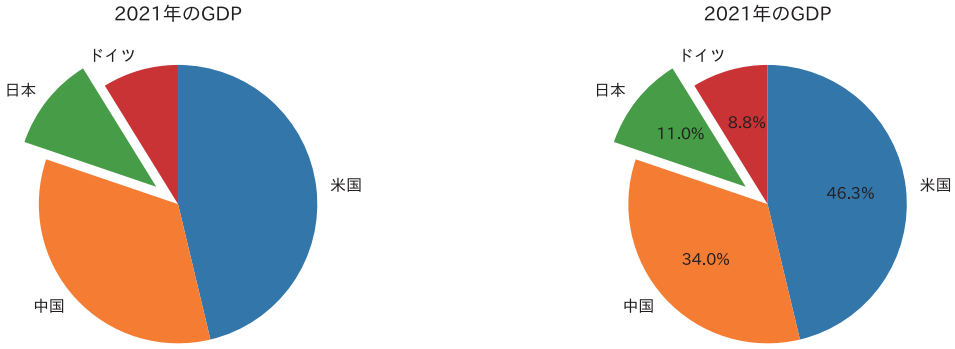
4.5.9.3　扇部の突出，百分率の表示

円グラフの扇形は外側に突出した形で表示することができる．具体的には，各扇形の突出の度合い（半径に対する比率）をリストにしたものを引数「explode=**突出率のリスト**」に与える．

例. 扇形を突出させる（先の例の続き）

入力：
```
e = [0,0,0.2,0]          # 突出率のリスト
G['2021'].plot.pie(title='2021 年の GDP',ylabel='',startangle=90,counterclock=False,
                   explode=e)
```

この例では描画開始の角度を 90 ° にしており，この処理の結果，図 9 の (a) のような表示となる．



(a) 扇形を突出させる　　(b) 百分率を表示

図 9: 扇形の突出と百分率表示

各扇形の上に当該データの百分率を表示するには引数「autopct=**書式文字列**」を与える．

例. 百分率を表示する（先の例の続き）

入力：
```
e = [0,0,0.2,0]          # 突出率のリスト
G['2021'].plot.pie(title='2021 年の GDP',ylabel='',startangle=90,counterclock=False,
                   explode=e,autopct='%4.1f%%')
```

この処理の結果，図 9 の (b) のような表示となる．引数 autopct に与える**書式文字列**は

%表示総桁数. 小数部桁数 f%%

と記述する．書式文字列の末尾の「%%」を外すと，扇形の上の表示の「%」表示が無くなる．また，整数型の表示にするには「f」の代わりに「d」と記述する．

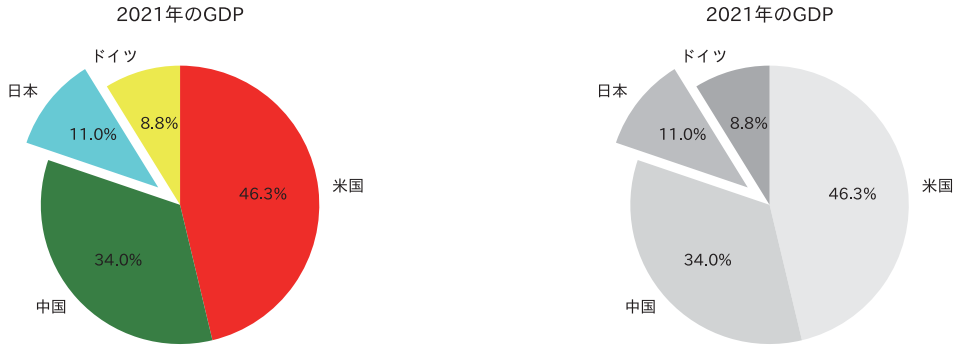
4.5.9.4 扇部の色の設定

円グラフの各扇形の色を設定するには引数「colors=**色のリスト**」を与える．

例. 扇部の色を設定する（先の例の続き）

入力：
```
e = [0,0,0.2,0]
c = ['red','green','cyan','yellow']     # 色名のリスト
G['2021'].plot.pie(title='2021 年の GDP',ylabel='',startangle=90,counterclock=False,
                   explode=e,autopct='%4.1f%%',colors=c)
```

この例では色名のリストを c に作成しており，この処理の結果，図 10 の (a) のような表示となる．



(a) 扇形の色を指定する　　(b) 扇形の明るさを指定する

図 10: 扇形の色，明るさの設定

引数 colors に与えるリストの要素には **'#RRGGBB'** の形式（RRGGBB は 16 進数）の文字列を与えることもできる．

各扇形の色をグレースケールにするには，引数 colors に与えるリストの要素を 0 ～ 1.0 の値（0:黒，1.0:白）とする．またこの場合の要素は文字列とする．

例. 扇部の明るさを設定する（先の例の続き）

入力：
```
e = [0,0,0.2,0]
c = ['0.9','0.8','0.7','0.6']        # 明るさのリスト
G['2021'].plot.pie(title='2021 年の GDP',ylabel='',startangle=90,counterclock=False,
                   explode=e,autopct='%4.1f%%',colors=c)
```

この処理の結果，図 10 の (b) のような表示となる．

4.5.10　棒グラフの作成

棒グラフは，数値データの並びを棒の長さで表現するものである．棒グラフは DataFrame オブジェクトや Series オブジェクトに対して plot や plot.bar ，あるいは plot.barh を実行することで作成する．

書き方 1：　DataFrame オブジェクト.plot(kind='bar', y=対象カラム)
書き方 2：　DataFrame オブジェクト.plot.bar(y=対象カラム)
書き方 3：　Series オブジェクト.plot(kind='bar')
書き方 4：　Series オブジェクト.plot.bar( )

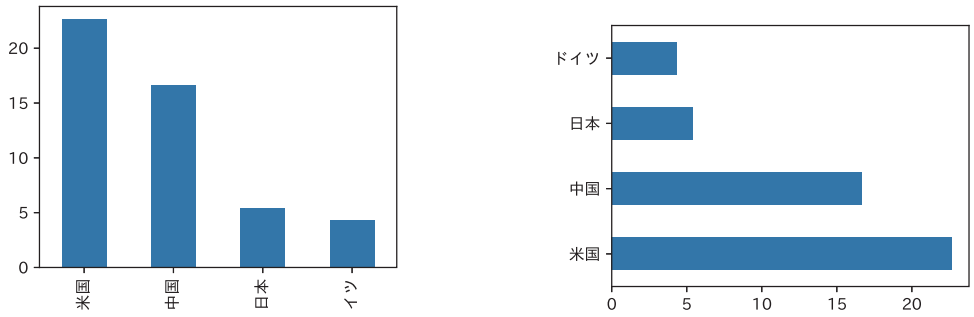
横向きの棒グラフを作成する場合は，上記の引数の記述の部分を kind='barh' とするか，あるいは plot.barh メソッドを使用する．

ここでは，先に作成した国別 GDP のサンプルデータを棒グラフにする形で説明する．

例. 棒グラフの作成（先の例の続き）

入力：
```
G.plot(kind='bar',y='2021',legend=False,figsize=(4,3))
```

この処理の結果，図 11 の (a) のような表示となる．引数を kind='barh' とすると図 11 の (b) のような表示となる．



(a) kind='bar'　　(b) kind='barh'

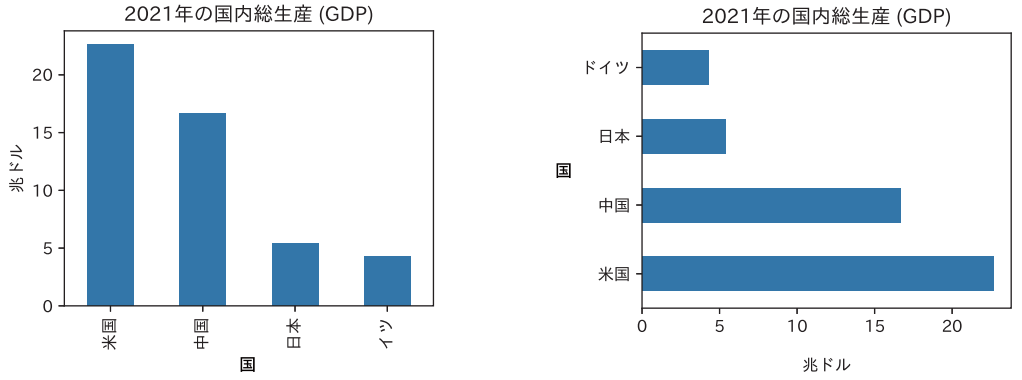
図 11: 縦棒グラフ，横棒グラフ

同様のことが，plot.bar メソッド，plot.barh メソッドでも可能である．これらメソッドには描画に関する各種の引数を与えることができる．次にその例を示す．

例. bar メソッドに様々な引数を与えて描画する（先の例の続き）

入力：
```
G['2021'].plot.bar(figsize=(4,3),
                   xlabel='国',ylabel='兆ドル',title='2021年の国内総生産（GDP)')
```

これは縦横の軸のラベルとグラフのタイトルを指定した例である．この処理の結果，図 12 の (a) ような表示となる．



(a) bar メソッドによる縦棒グラフ　　(b) barh メソッドによる横棒グラフ

図 12: 軸ラベル，タイトルの表示

plot.barh メソッドで横棒グラフを作成する場合は，引数 xlabel，ylabel の役割が変わることに注意すること．次の例の実行結果は図 12 の (b) ようになる．

例. barh メソッドで横棒グラフを描画する（先の例の続き）

入力：
```
G['2021'].plot.bar(figsize=(4,3),
                   ylabel='国',xlabel='兆ドル',title='2021 年の国内総生産 (GDP)')
```

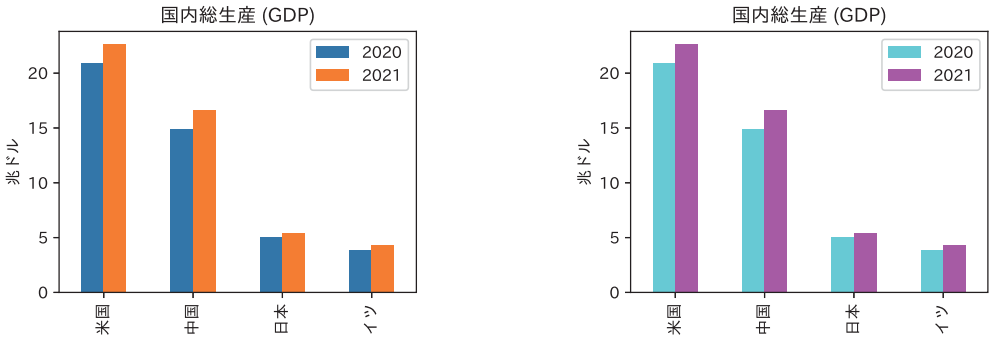
4.5.10.1 複数のカラムを棒グラフにする

DataFrame オブジェクトの複数のカラムを棒グラフにすることができる.

例. DataFrame の複数のカラムを棒グラフにする（先の例の続き）

入力：
```
G.plot.bar(figsize=(4,3),xlabel='国',ylabel='兆ドル',title='国内総生産 (GDP)')
```

この処理の結果，図 13 の (a) のような表示となる.



(a) '2020' のカラムと '2021' のカラムを同時にプロット　　(b) 色を指定してプロット

図 13: 複数のカラムを同時に棒グラフにする

引数「color=」に色のリストを与えると，系列毎の色を指定できる.

例. DataFrame の複数のカラムを棒グラフにする（先の例の続き）

入力：
```
c = ['cyan','magenta']          # 色のリスト
G.plot.bar(figsize=(4,3),xlabel='国',ylabel='兆ドル',title='国内総生産 (GDP)',
           color=c)
```

この処理の結果，図 13 の (b) のような表示となる.

4.5.11 散布図の作成

DataFrame の指定した 2 つのカラムのデータから**散布図**を作成する方法について解説する.

書き方 (1)：　DataFrame オブジェクト.plot(kind='scatter', x=横軸カラム, y=縦軸カラム)
書き方 (2)：　DataFrame オブジェクト.plot.scatter(x=横軸カラム, y=縦軸カラム)

「横軸カラム」,「縦軸カラム」にはカラムの名前を文字列で与える.

以下に，作成したサンプルデータから散布図を作成する流れを示す.

例. サンプルデータの作成

入力：
```
# データ系列 1
x1 = stats.norm.rvs(loc=1,scale=1,size=200,random_state=1)
x2 = stats.norm.rvs(loc=8,scale=2,size=800,random_state=2)
x12 = np.append(x1,x2)
# データ系列 2
y1 = stats.norm.rvs(loc=1,scale=1,size=200,random_state=3)
y2 = stats.norm.rvs(loc=8,scale=2,size=800,random_state=4)
y12 = np.append(y1,y2)
```

入力：
```
df5 = pd.DataFrame()
df5['x'] = x12; df5['y'] = y12
```

この実行により，サンプルデータの DataFrame オブジェクト df5 が作成される．これに対して次のような処理を行

いヒストグラムを表示してデータの分布を確認すると図 14 のようになる．

例． サンプルデータの分布を調べる（先の例の続き）

入力：
```
df5.hist(bins=20,figsize=(12,3))
```

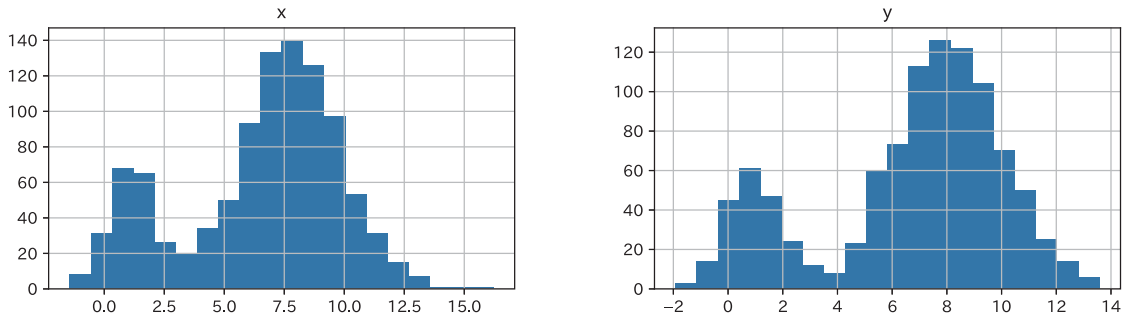


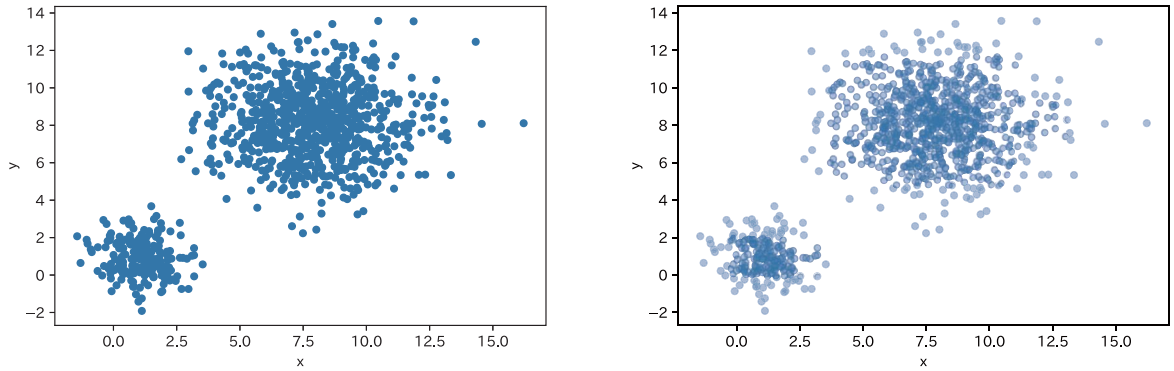
図 14: サンプルデータの分布をヒストグラムで確認

次に df5 のカラム 'x' を横軸に，カラム 'y' を縦軸にして散布図を作成する．

例． 散布図（その 1）の作成（先の例の続き）

入力：
```
df5.plot(kind='scatter',x='x',y='y')
```

これによって図 15 の (a) のような散布図が表示される．



(a) alpha 値の設定なし　　(b) alpha=0.4

図 15: 散布図

散布図作成メソッドに引数「alpha=**不透明度**」を与えることで，散布図のデータ点に不透明度（0〜1.0）を設定することができる．

例． 散布図（その 2）の作成（先の例の続き）

入力：
```
df5.plot.scatter(x='x',y='y',alpha=0.4)
```

これによって図 15 の (b) のような散布図が表示される．

4.6 集計処理

4.6.1 グループ集計

データ集合を「ある属性」毎にまとめた形で統計処理するには，その属性毎にデータを**グループ化**する．ここでは作業の例を示しながら，DataFrame オブジェクトのグループ化による処理について解説する．

まず最初にサンプルデータを準備する．次の例は，$N[-3,3^2]$, $N[0,1^2]$, $N[1,0.5^2]$ の 3 種類のデータ集合を準備するものである．

例. サンプルデータの準備：ライブラリの読込みとデータ列の生成

入力：
```
import pandas as pd          # pandas を pd の名で読込み
from scipy import stats      # scipy.stats の読込み
import numpy as np           # NumPy の np の名で読込み
```

入力：
```
rA = stats.norm.rvs( loc=-3, scale=3,   size=10000, random_state=1 )  # μ=-3, σ=3
rB = stats.norm.rvs( loc=0, scale=1,    size=10000, random_state=2 )  # μ=0, σ=1
rC = stats.norm.rvs( loc=1, scale=0.5, size=10000, random_state=3 )  # μ=1, σ=0.5
```

この例では，$N[-3,3^2]$, $N[0,1^2]$, $N[1,0.5^2]$ をそれぞれ配列 rA, rB, rC として生成している．次に，これらに「種類のラベル」'A', 'B', 'C' を付けた形の DataFrame にする．

例. サンプルデータの準備：データフレームにする（先の例の続き）

入力：
```
df = pd.DataFrame(columns=['種類','値'])
df['種類'] = ['A']*len(rA) + ['B']*len(rB) + ['C']*len(rC)
df['値'] = np.concatenate([rA,rB,rC])
# シャッフルする
dfR = df.sample(frac=1,random_state=4).reset_index(drop=True)
dfR.head(4)
```

出力：

	種類	値
0	A	-4.392193
1	A	-4.406021
2	B	-1.907950
3	C	0.697232

'A', 'B', 'C' のラベルを付けて配列 rA, rB, rC を DataFrame にしたものが dfR である．この DataFrame は sample メソッド[43] によってシャッフルされ，3 種類のデータが入り乱れた形で得られている．当然であるが，dfR が持つデータは全体としては配列 rA, rB, rC とは異なる統計的特徴を持つ．（次の例参照）

例. データ全体の統計（先の例の続き）

入力：
```
print( '平均:', dfR['値'].mean() )
print( '標準偏差:', dfR['値'].std() )
```

出力：
```
平均:  -0.667900972250482
標準偏差:  2.4958058728041013
```

このような 3 種類のデータが入り交じる DataFrame を，'種類' のカラムが持つ値で分類してグループ化する処理を次に示す．

例. グループ化処理（先の例の続き）

入力：
```
g = dfR.groupby('種類')     # グループ化処理
print( type(g) )
display( g.mean() )
display( g.std() )
```

[43]「4.7 ランダムサンプリング，シャッフル」（p.90）で解説する．

出力： <class 'pandas.core.groupby.generic.DataFrameGroupBy'>

種類	値
A	-2.970682
B	-0.019192
C	0.986171

種類	値
A	2.996507
B	1.000467
C	0.497968

この例では，groupby メソッドで dfR をグループ化して，それを g として得ている．
このオブジェクト g は DataFrameGroupBy クラスのオブジェクトであり，これに対して mean, std などのメソッドで統計値を取得するとグループ毎の値が得られる．

DataFrameGroupBy オブジェクトに対して describe メソッドを使用することもできる．(次の例参照)

例. describe による要約統計量の調査（先の例の続き）

入力：
```
g.describe()
```

出力：

種類	count	mean	std	min	25%	50%	75%	max
A	10000.0	-2.970682	2.996507	-13.969320	-4.988775	-2.974638	-0.984573	9.080547
B	10000.0	-0.019192	1.000467	-3.582359	-0.688269	-0.020273	0.641505	4.133362
C	10000.0	0.986171	0.497968	-0.874970	0.650109	0.985504	1.323182	3.045696

4.6.2 クロス集計

DataFrame の指定したカラムに基づいてデータをクロス集計するには crosstab 関数を使用する．また pivot_table メソッドを使用するとピボット表が作成できる．ここでは作業例を挙げてクロス集計やピボット表の作成の方法[44] について説明する．

まず最初にサンプルデータを準備する．次の例は「出身地」「年齢」「食べ物の好み」（好物）に関するアンケートデータを模したものを生成する例である．

例. サンプルデータの準備（先の例の続き）

入力：
```
dtab = pd.DataFrame(columns=['出身地','年齢','好物'])  # データフレームの作成
```

入力：
```
import random       # リストのシャッフルに使用するモジュールの読込み
p = ['大阪','京都','東京']*100
m = ['お好み焼き']*80 + ['カレーライス']*120 + ['おでん']*50 + ['肉うどん']*50
random.seed(1)
random.shuffle(p)         # '出身地' の列をシャッフル
random.seed(2)
random.shuffle(m)         # '好物' の列をシャッフル
dtab['出身地'] = p
dtab['好物'] = m
# 年齢には適当に乱数を設定
ages = stats.norm.rvs(loc=35,scale=10,size=300,random_state=5)
dtab['年齢'] = np.clip(ages,0,None).astype('uint32')  # 負値にならないための処理
dtab.head(5)
```

出力：

	出身地	年齢	好物
0	東京	39	カレーライス
1	大阪	31	おでん
2	京都	59	カレーライス
3	京都	32	おでん
4	大阪	36	肉うどん

この例では，random モジュールの shuffle を用いてデータをシャッフルしている．

[44] 多くの表計算ソフトウェアが**ピボットテーブル**と呼ばれる集計機能を提供しているが，ここで説明する集計機能はそれらに類似するものである．

《**random.shuffle**》

リストなどの，順序がありかつ**ミュータブル**（変更可能）なデータ構造を shuffle の引数に与えることでそれをシャッフルできる．

書き方：　shuffle(データオブジェクト)

shuffle の動作に再現性を持たせるには，random モジュールの seed を使用する．

書き方：　seed(種)

random モジュールが提供する乱数発生機能やシャッフルの機能の初期状態を「種」で指定することができる．この場合の「種」は「4.2.1 得られる乱数の系列について」（p.60）で解説した事柄と類似の考え方によるものである．

4.6.2.1　crosstab

関数 crosstab は指定した行と列の項目毎のデータ件数を集計し，集計結果の DataFrame を返す．

書き方：　crosstab(行の並び, 列の並び)

「行の並び」に与えたデータ列を行のインデックス,「列の並び」に与えたデータ列をカラム名としてデータ件数を集計し，結果を DataFrame として返す．

例. ’出身地’ と’好物’ 毎にデータ件数を集計する．（先の例の続き）

入力：
```
ct = pd.crosstab( dtab['出身地'], dtab['好物'] )
ct
```

出力：

出身地＼好物	おでん	お好み焼き	カレーライス	肉うどん
京都	15	27	43	15
大阪	21	25	41	13
東京	14	28	36	22

crosstab 関数に引数「margins=True」を与えると，行，列毎の集計結果を追加する．

例. ’出身地’ と’好物’ 毎にデータ件数を集計する．（先の例の続き）

入力：
```
ct = pd.crosstab( dtab['出身地'], dtab['好物'], margins=True )
ct
```

出力：

出身地＼好物	おでん	お好み焼き	カレーライス	肉うどん	All
京都	15	27	43	15	100
大阪	21	25	41	13	100
東京	14	28	36	22	100
All	50	80	120	50	300

4.6.2.2　pivot_table

DataFrame に対して pivot_table メソッドを使用すると，データ件数以外の集計処理ができる．

《**DataFrame のピボット表**》

pivot_table(　index=ピボット表のインデックスとするためのカラム,
columns=ピボット表のカラムとするためのカラム,　values=集計対象のカラム,
aggfunc=集計処理の種類,　margins=[True/False])

‘index=’ に指定したカラムの値と ‘columns=’ に指定したカラムの値をそれぞれ行ラベル，列ラベルとするピボット表を作成する．集計対象の値は ‘values=’ に指定したカラムである．‘aggfunc=’ には 'sum', 'max', 'min', 'count', 'mean' といった，集計処理の種類を意味する文字列を与える．

先のデータの ’年齢’ の値を ’出身地’ と ’好物’ のピボット表にする例を示す．

例. 年齢の平均値のピボット集計（先の例の続き）

入力：
```
pt = dtab.pivot_table(index='出身地',columns='好物',values='年齢',
                      aggfunc='mean', margins=True)
pt
```

出力：

出身地＼好物	おでん	お好み焼き	カレーライス	肉うどん	All
京都	33.866667	34.481481	37.534884	30.333333	35.08
大阪	37.809524	35.160000	33.048780	33.461538	34.63
東京	34.071429	33.642857	35.083333	31.545455	33.76
All	35.580000	34.400000	35.266667	31.680000	34.49

例. 年齢の最大値のピボット集計（先の例の続き）

入力：
```
pt = dtab.pivot_table(index='出身地',columns='好物',values='年齢',
                      aggfunc='max',margins=True)
pt
```

出力：

出身地＼好物	おでん	お好み焼き	カレーライス	肉うどん	All
京都	56	50	60	42	60
大阪	57	57	55	49	57
東京	50	49	48	52	52
All	57	57	60	52	60

例. データ個数のピボット集計（先の例の続き）

入力：
```
pt = dtab.pivot_table(index='出身地',columns='好物',values='年齢',
                      aggfunc='count',margins=True)
pt
```

出力：

出身地＼好物	おでん	お好み焼き	カレーライス	肉うどん	All
京都	15	27	43	15	100
大阪	21	25	41	13	100
東京	14	28	36	22	100
All	50	80	120	50	300

4.6.3 ダミー変数の取得（ワンホットエンコーディング）

数値でない値を持つカラムを**ダミー変数**として展開（**ワンホットエンコーディング**：One-Hot encoding ）するには get_dummies を使用する.

次のような DataFrame をダミー変数に展開する例を考える.

例. サンプルデータ（先の例の続き）

入力：
```
# サンプルデータ
d = pd.DataFrame(columns=['氏名','出身地'])
d['氏名'] = ['中村 勝則','田中 由恵','平田 洋子']
d['出身地'] = ['大阪府','京都府','東京都']
display(d)
```

出力：

	氏名	出身地
0	中村 勝則	大阪府
1	田中 由恵	京都府
2	平田 洋子	東京都

この DataFrame の「出身地」のカラムをワンホットエンコーディングしてダミー変数に展開する例を次に示す.

例. ワンホットエンコーディング（先の例の続き）

入力：
```
dmy = pd.get_dummies(d,columns=['出身地'])
display(dmy)
```

出力：

	氏名	出身地_京都府	出身地_大阪府	出身地_東京都
0	中村 勝則	0	1	0
1	田中 由恵	1	0	0
2	平田 洋子	0	0	1

このように get_dummies のキーワード引数 columns= に展開するカラムを指定（複数可）する．展開されたカラムの名前は「元のカラム名_元の値」として与えられ，新しいカラムの値には，元の値の有無を意味する数値（1 か 0）もしくは真理値（True/False）が与えられる[45]．DataFrame をこのような形に変換することで，集計処理が容易になる場合[46] がある．

get_dummies のキーワード引数 columns= を省略すると全てのカラムが展開される．

例. 全てのカラムの展開（先の例の続き）

入力：
```
dmy = pd.get_dummies(d)
display(dmy)
```

出力：

	氏名_中村 勝則	氏名_平田 洋子	氏名_田中 由恵	出身地_京都府	出身地_大阪府	出身地_東京都
0	1	0	0	0	1	0
1	0	0	1	1	0	0
2	0	1	0	0	0	1

4.7 ランダムサンプリング，シャッフル

sample メソッドを用いると，DataFrame からデータをランダムにサンプリング（**無作為抽出**）することができる．ここでは，実際の作業を例に示しながら解説する．

例. ライブラリの読込みとサンプルデータの作成

入力：
```
import numpy as np
import pandas as pd
```

入力：
```
df = pd.DataFrame(None,columns=['n','even','odd'])
df['n'] = [n for n in range(10)]
df['even'] = [2*n for n in range(10)]
df['odd'] = [2*n+1 for n in range(10)]
df.head(3)
```

出力：

	n	even	odd
0	0	0	1
1	1	2	3
2	2	4	5

偶数と奇数の列をカラムとして持つ DataFrame オブジェクト df ができたので，ここからランダムサンプリングを試みる．

例. ランダムに 1 行取り出す（先の例の続き）

入力：
```
df.sample()
```

出力：

	n	even	odd
3	3	6	7

このように引数を与えずに sample メソッドを実行すると，ランダムに 1 行取り出す．

[45] pandas，NumPy の新しい版では**真理値**（True/False）となる．

[46] 数値でないデータを機械学習で利用する際もワンホットエンコーディングがしばしば用いられる．

例. 指定した行数のランダムサンプリング（先の例の続き）

入力：
```
df.sample(2)      # df.sample(n=2) としても同じ
```

出力：

	n	even	odd
6	6	12	13
3	3	6	7

このように，第一引数に（あるいはキーワード引数 ‘n=’ に）取り出す行数を指定することができる．

例. 比率を指定してランダムサンプリング（先の例の続き）

入力：
```
df.sample(frac=0.2)
```

出力：

	n	even	odd
1	1	2	3
6	6	12	13

キーワード引数 ‘frac=’ に取り出す比率（全体の行数に対する）を指定することもできる．比率として 1 を指定すると全件ランダムに取り出すことになり，これは DataFrame 全体を**シャッフル**したことになる．

例. シャッフル（先の例の続き）

入力：
```
df.sample(frac=1)      # ランダムに全件取り出し（シャッフル）
```

出力：

	n	even	odd
2	2	4	5
4	4	8	9
8	8	16	17
7	7	14	15
3	3	6	7
1	1	2	3
0	0	0	1
9	9	18	19
6	6	12	13
5	5	10	11

sample メソッドは Series に対しても使用できる．

参考） Sample メソッドによる抽出動作に再現性を持たせるには引数「random_state= **種**」を与える．この場合の「種」は「4.2.1 得られる乱数の系列について」（p.60）で解説した事柄と類似の考え方によるものである．

4.8 変数間の関係の調査

4.8.1 相関係数

pandas の DataFrame には，2 つの変数の間の相関係数を求める corr メソッドが使用できる．ここでは，作業の例を示しながら相関係数を求める方法について解説する．

まず最初に，次のようにして必要なライブラリを読み込む．

例. パッケージの読込み

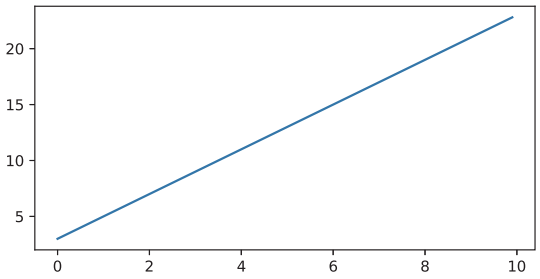
入力：
```
import pandas as pd                  # pandas を pd の名で読込み
from scipy import stats              # scipy.stats の読込み
import numpy as np                   # NumPy の np の名で読込み
import matplotlib.pyplot as plt      # matplotlib を plt の名で読込み
```

次に，サンプルのデータとして 1 次式 $y = 2x + 3$ に**回帰**する 2 つのデータ列（変数 x, y）を作成する．そのために，この 1 次式を満たすデータ列 X, Y を NumPy の配列として作成する．(次の例参照)

例. サンプルデータの作成と確認 (1)（先の例の続き）

入力：
```
X = np.arange( 0, 10, 0.1 )     # データ列 X
Y = 2*X + 3                     # データ列 Y
plt.figure(figsize=(6,3))       # プロット
plt.plot(X,Y)
```

出力：
```
[<matplotlib.lines.Line2D at 0x1f61fbd5bd0>]
```

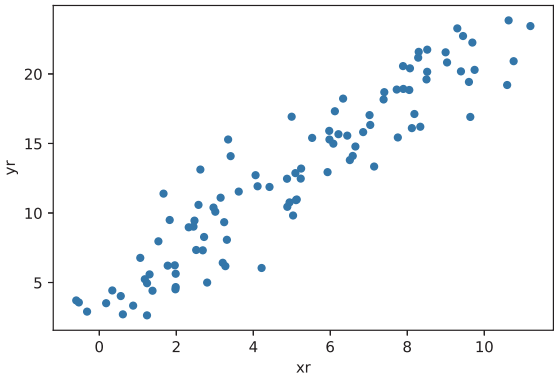


このようにして得られた X, Y に乱数を加えて撹乱したデータ Xr, Yr を作成し，それらを元に DataFrame を作成する．(次の例参照)

例. サンプルデータの作成と確認 (2)（先の例の続き）

入力：
```
Xr = X + stats.norm.rvs( size=len(X) )  # データ列 X を撹乱
Yr = Y + stats.norm.rvs( size=len(Y) )  # データ列 Y を撹乱
# DataFrame の作成
df = pd.DataFrame(columns=('x','y','xr','yr'))
df['x'] = X; df['y'] = Y
df['xr'] = Xr; df['yr'] = Yr
plt.figure(figsize=(6,3))       # プロット
df.plot( kind='scatter', x='xr', y='yr' )
```

出力：
```
<Figure size 432x216 with 0 Axes>
```



これでサンプルデータが DataFrame オブジェクト df として用意できた．次に，corr メソッドを使用して，カラム間の相関係数を求める．(次の例参照)

例. カラム間の相関係数（先の例の続き）

入力：
```
cr = df.corr()   # 相関係数の算出
display( cr )    # 内容確認
type( cr )       # データ型の調査
```

出力：

	x	y	xr	yr
x	1.000000	1.000000	0.949148	0.986042
y	1.000000	1.000000	0.949148	0.986042
xr	0.949148	0.949148	1.000000	0.936856
yr	0.986042	0.986042	0.936856	1.000000

```
pandas.core.frame.DataFrame
```

このように，カラム間の相関係数が**相関行列**を表す DataFrame オブジェクトとして得られる．

相関係数の値は -1〜1 の範囲をとり，1 に近いほど強い正の相関，-1 に近いほど強い負の相関を意味する．

4.8.2 共分散

DataFrame のカラムの間の**共分散**を求めるには cov メソッドを用いる．

例. 共分散を算出（先の例の続き）

入力：
```
df.cov()
```

出力：

	x	y	xr	yr
x	8.416667	16.833333	8.412701	16.723137
y	16.833333	33.666667	16.825403	33.446273
xr	8.412701	16.825403	9.371583	16.821442
yr	16.723137	33.446273	16.821442	34.240424

DataFrame の cov メソッドは暗黙で**不偏共分散**を算出する．このメソッドに引数「ddof=0」を与えると標本共分散を算出する．(ddof の暗黙値は 1)

NumPy の cov 関数でも共分散を求めることができる．

例. NumPy の cov 関数による標本共分散の算出（先の例の続き）

入力：
```
print( 'Xr の分散:', Xr.var(ddof=0) )
print( 'Yr の分散:', Yr.var(ddof=0) )
print( '分散・共分散行列:' )
display( np.cov( Xr, Yr, ddof=0 ) )
```

出力：
```
Xr の分散:  9.277867067756937
Yr の分散:  33.898020069069474
分散・共分散行列:
array([[ 9.27786707, 16.65322718],
       [16.65322718, 33.89802007]])
```

NumPy の cov 関数にキーワード引数 'ddof=0' を与えると変数間の標本共分散を算出する．また，'ddof=1' を与えると変数間の不偏共分散を算出する．

例. NumPy の cov 関数による不偏共分散の算出（先の例の続き）

入力：
```
print( 'Xr の分散:', Xr.var(ddof=1) )
print( 'Yr の分散:', Yr.var(ddof=1) )
print( '分散・共分散行列:' )
display( np.cov( Xr, Yr, ddof=1 ) )
```

出力：
```
Xr の分散:  9.37158289672418
Yr の分散:  34.24042431219139
分散・共分散行列:
array([[ 9.3715829 , 16.82144159],
       [16.82144159, 34.24042431]])
```

4.8.3 多項式回帰

NumPy ライブラリは多項式回帰のための関数 polyfit を提供している．先に作成したデータを例に用いて多項式回帰の方法について説明する．

多項式回帰：　polyfit(データ列 1, データ列 2, deg=多項式の次数)

この関数は，データ列 1,2 が回帰する多項式の係数を配列（降べきの順）として返す．

この関数を用いて，先に作成したデータ Xr, Yr が回帰する 1 次式を求める．

例. 多項式回帰（1 次）（先の例の続き）

入力：
```
np.polyfit(Xr,Yr,deg=1)
```

出力：
```
array([1.82296057, 3.62008113])
```

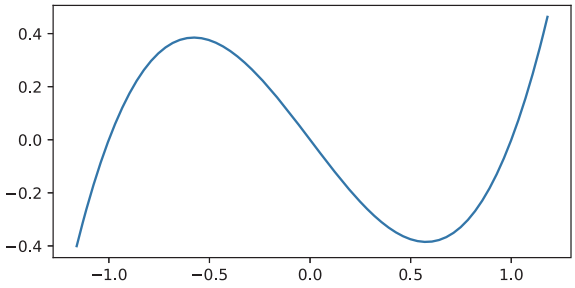
先に作成したデータ Xr, Yr は 1 次式 $y = 2x+3$ を撹乱して作成したものであり，この処理の結果は，$y = 1.82296057x+3.62008113$ に回帰することを示している．

次に高次の多項式回帰の例を示す．サンプルとして $y = x^3 - x$ に回帰するデータ列を作成する．まず，この多項式に沿ったデータ列 X3, Y3 を作成する．（次の例参照）

例. サンプルデータの作成と確認 (1)（先の例の続き）

入力：
```
X3 = np.arange(-1.16,1.19,0.01)     # データ列 Y3
Y3 = X3**3 - X3                      # データ列 Y3
plt.figure(figsize=(6,3))     # プロット
plt.plot(X3,Y3)
```

出力：
```
<Figure size 600x300 with 0 Axes>
```

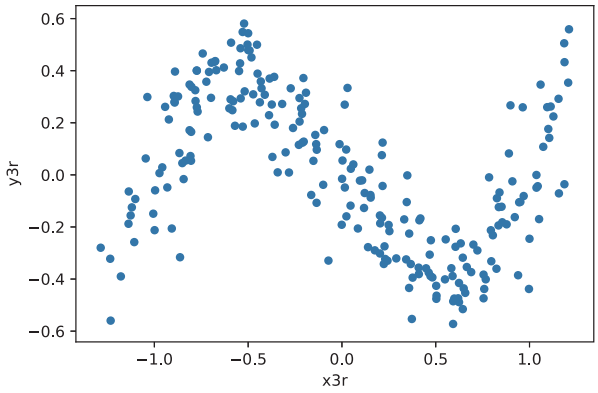


このようにして得られた X3, Y3 に乱数を加えて撹乱したデータ X3r, Y3r を作成し，それらを元に DataFrame を作成する．（次の例参照）

例. サンプルデータの作成と確認 (2)（先の例の続き）

入力：
```
X3r = X3 + 0.1*stats.norm.rvs( size=len(X3) )     # データ列 X3 を撹乱
Y3r = Y3 + 0.1*stats.norm.rvs( size=len(Y3) )     # データ列 Y3 を撹乱
# DataFrame の作成
df3 = pd.DataFrame(columns=('x3','y3','x3r','y3r'))
df3['x3'] = X3; df3['y3'] = Y3
df3['x3r'] = X3r; df3['y3r'] = Y3r
plt.figure(figsize=(6,3))        # プロット
df3.plot( kind='scatter', x='x3r', y='y3r' )
```

出力：
```
<Figure size 432x216 with 0 Axes>
```



これでサンプルデータが DataFrame オブジェクト df3 として用意できた．次に，polyfit 関数によって，データが回帰する 3 次多項式を求める．(次の例参照)

例. 多項式回帰（3 次）（先の例の続き）

入力：
```
np.polyfit(X3r,Y3r,deg=3)
```

出力：
```
array([ 0.81718282, 0.00861416, -0.86622252, -0.00648668])
```

処理の結果は，$y = 0.81718282x^3 + 0.00861416x^2 - 0.86622252x - 0.00648668$ に回帰することを示している．

4.9 統計検定

4.9.1 z 検定

標本の平均が母平均から有意に外れているかどうかを z 検定で調べることができる．z 検定のための前提として次の 2 つがある．

1) 母集団は正規分布に従う[47]

2) 母平均と母分散が既知である

ここではサンプルの母集団データを生成して，z 検定の作業を例示する．

例. ライブラリの読込み

入力：
```
import pandas as pd          # pandas を 'pd' という名で読込み
from scipy import stats     # scipy.stats の読込み
import numpy as np           # numpy を 'np' という名で読込み
```

数値計算用の関数を使用するために NumPy ライブラリを読み込んでいる．これで必要なライブラリの読込みが完了した．次にサンプルの母集団データを生成する．

例. 母集団データの生成（先の例の続き）

入力：
```
# 正規分布（μ=0，σ=1）に沿った乱数生成（10,000 個のデータ）
y1 = stats.norm.rvs(loc=0,scale=1,size=10000, random_state=0)
# 対数正規分布（μ=0，σ=1）に沿った乱数生成（10,000 個のデータ）
y2 = stats.lognorm.rvs(loc=0,s=1,size=10000, random_state=0)
df = pd.DataFrame(columns=['Norm','LogNorm'])  # DataFrame を用意
df['Norm'] = y1; df['LogNorm'] = y2             # y1, y2 を DataFrame にセット
```

母集団とするための正規分布に沿った乱数を生成して `df['Norm']` に用意している．これと比較して平均値が有意に外れているデータ集合（対数正規分布に沿った乱数）を `df['LogNorm']` に用意している．また，実行例の再現性を確保するために，乱数の生成に種を設定（random_state=0）している．

次に，既知の母数として μ と σ を設定する．

例. 既知の母数の設定（先の例の続き）

入力：
```
mu = 0        # 既知の母平均
sigma = 1     # 既知の標準偏差
```

これで mu に母平均が，sigma に母集団の標準偏差が設定された．次に，母集団から標本（30 件）を無作為に抽出（再現性確保のために種を設定している）して標本平均を算出する．

例. 標本（30 件）の抽出（先の例の続き）

入力：
```
n = 30                      # 標本数
df2 = df.sample(n, replace=False, random_state=0)  # 標本の無作為抽出
x = df2['Norm'].mean()  # 標本の平均
print( '標本平均:', x )
```

出力：　標本平均:　-0.0669052850624805

これで標本が df2 に得られた．

z 検定は，**標準誤差**から求めた **z スコア**（下記）を用いる検定方法である．

標準誤差：　$S_e = \dfrac{\sigma}{\sqrt{n}}$　　（σ は母集団の標準偏差，n は標本の件数）

z スコア：　$z = \dfrac{\bar{X} - \mu}{S_e}$　　（μ は母平均，$\bar{X}$ は標本の平均）

[47]ただし，サンプルのサイズが十分に大きい場合はこの前提を外しても良い場合がある．

次に，これらの値を求める．

例. 標本の z スコアの算出（先の例の続き）

入力：
```
se = sigma / np.sqrt(n)     # 標準誤差の取得
z = (x - mu) / se           # 検定のための「z スコア」の算出
print('z スコア:',z)
```

出力：
```
z スコア:  -0.3664553384503401
```

この z スコアが信頼できる範囲内にあるかどうかを検査する．そのために必要となる下側の限界値 L と，上側の信頼値 U を求める．（この作業に関する考え方については p.152「A.2.2.2 区間推定」を参照のこと）
実際の方法としては，正規分布のパーセント点関数 stats.norm.ppf を用いる．

例. 信頼区間の算出（先の例の続き）

入力：
```
# 信頼度 95%(α=0.05) で信頼できる限界値
L = stats.norm.ppf( q=0.025, loc=0, scale=1 ); print(' 下側の限界値:',L)
U = stats.norm.ppf( q=0.975, loc=0, scale=1 ); print(' 上側の限界値:',U)
```

出力：
```
下側の限界値:  -1.9599639845400545
上側の限界値:  1.959963984540054
```

stats.norm.ppf 関数は正規分布 $N(\mu, \sigma^2)$ のパーセント点を求めるもので，キーワード引数 'q=' には累積確率を，'loc=' には μ，'scale=' には σ を指定する．

最後に，z スコアが信頼できる区間（採択域）に入っているかどうかを調べる．

例. z スコアを検査（先の例の続き）

入力：
```
L < z and z < U # 採択域に入っているか
```

出力：
```
True            ←環境によっては np.True_ と表示される場合がある
```

抽出した標本 df2['Norm'] の平均は有意なずれが無く，信頼できることがわかった．

両側の **p 値**

$$p = 2\left(1 - cdf\left(|z|\right)\right) \quad : \quad cdf \text{ は正規分布の累積分布関数}$$

による検定も次に示す．

例. p 値による検定（先の例の続き）

入力：
```
# 信頼度 95%(α=0.05) での判定
p = 2 * (1 - stats.norm.cdf(abs(z)))
print('p 値 :',p,'->',end='')
if p > 0.05:
    print(' 採択域内')
else:
    print(' 採択域外')
```

出力：
```
p 値 :  0.714025332970222 ->採択域内
```

次に，母集団と明らかに特徴が異なる標本 df2['LogNorm'] の平均値を z 検定で調べる．

例. df2['LogNorm'] の検定（先の例の続き）

入力：
```
x = df2['LogNorm'].mean()     # 標本の平均（対数正規分布）
z = (x - mu) / se             # 検定のための「z スコア」の算出
print('z スコア:',z)
L < z and z < U               # 採択域に入っているか
```

出力：
```
z スコア:  12.219216179169237
False           ←環境によっては np.False_ と表示される場合がある
```

z スコアが信頼できる範囲に無く，標本が母集団から有意に外れていることがわかる．(対数正規分布は正規分布とは分布の形が異なるので当然のことである)

本来，分布の形が著しく異なる場合，平均の比較だけで判断することは危険（母集団の形状に依存する）である点に注意すること．

p 値による df2['LogNorm'] の検定の例も次に示す．

例. p 値による検定（先の例の続き）

入力：
```
# 信頼度 95%(α=0.05) での判定
p = 2 * (1 - stats.norm.cdf(abs(z)))
print('p 値 :',p,'->',end='')
if p > 0.05:
    print(' 採択域内')
else:
    print(' 採択域外')
```

出力：　p 値 : 0.0 ->採択域外

注意)

対数正規分布 stats.lognorm のパラメータ (μ, σ) は，その分布の平均や標準偏差を表すものではなく，元になっている正規分布の「平均」「標準偏差」を表すパラメータである．本節では，分布の「形」が母集団と大きく異なる場合に，z 検定が大きなズレを示すことを確認するため，あえてこの例を用いた．

4.9.2 t 検定

母集団が正規分布に従い，母分散が未知であると仮定する場合に，採取した少量のサンプルのみから母平均に関する仮説を検証する方法として t **検定**が有効である．

t 検定は，母集団の平均 μ が，仮定された値 μ_0 に等しいという帰無仮説 $H_0 : \mu = \mu_0$ が妥当であるかどうかを，標本平均と標本標準偏差に基づいて判断するための検定方法である。

ここでは，t 検定によって，仮定した μ_0 が妥当であるかどうかを調べる（**1 群の t 検定**）作業の例を示す．

先の例と同じくライブラリを読み込み，サンプルの母集団データを作成する．

例. ライブラリの読み込みと母集団データの作成

入力：
```
import pandas as pd         # pandas を 'pd' という名で読込み
from scipy import stats     # scipy.stats の読込み
import numpy as np          # numpy を 'np' という名で読込み
```

入力：
```
# 正規分布（μ=0, σ=1）に沿った乱数生成（10,000 個のデータ）
y1 = stats.norm.rvs(loc=0,scale=1,size=10000, random_state=0)
df = pd.DataFrame(y1,columns=['Norm'])     # DataFrame にする
```

母集団のデータが df に作成された．次にここから少量（25 件）の標本を抽出する．

例. 標本の抽出（先の例の続き）

入力：
```
n = 25                      # 標本数
df2 = df.sample(n, replace=False, random_state=1)     # 標本の無作為抽出
x = df2['Norm'].mean()  # 標本の平均
s = df2['Norm'].std(ddof=1)    # 標本の標準偏差（不偏）
print(' 標本の平均:    ',x)
print(' 標本の標準偏差:',s)
```

出力：　標本の平均:　　-0.20128598519841298
　　標本の標準偏差:　1.0854390441954571

t 検定は，仮定した母平均 $\hat{\mu}$ と標本の統計量から得られた t **統計量**が，t 分布上の信頼できる区間に含まれるかどう

か[48] を調べる検定方法である．

t 統計量： $t = \dfrac{\bar{X} - \hat{\mu}}{\dfrac{s}{\sqrt{n}}}$ （n は標本の個数，$\bar{X}$ は標本の平均，s は標本の標準偏差，$\hat{\mu}$ は仮定した母平均）

次に，t 統計量の値を求める．

例. t 統計量の算出（先の例の続き）

入力：
```
mu = 0.7     # 母平均の仮定
t = (x-mu)/(s/np.sqrt(n))     # t 統計量
print('t 統計量：',t)
```

出力：　t 統計量： -4.151711650774728

この例では，母平均を 0.7 と仮定して，それが妥当かどうかを検定する．具体的には t 統計量が信頼できる区間にあるかどうかを検査する．そのために必要となる下側の限界値 L と，上側の限界値 U を求める．（この作業の考え方に関しては p.152「A.2.2.2 区間推定」を参照のこと）
実際の方法としては，t 分布のパーセント点関数 stats.t.ppf を用いる．

例. 信頼区間の算出（先の例の続き）

入力：
```
# 信頼度 95%(α=0.05) で信頼できる限界値
L = stats.t.ppf( q=0.025, loc=0, scale=1, df=n-1 )
U = stats.t.ppf( q=0.975, loc=0, scale=1, df=n-1 )
print(' 下側の限界値:',L);     print(' 上側の限界値:',U)
```

出力：　下側の限界値:　-2.063898561628021
上側の限界値:　2.0638985616280205

stats.t.ppf 関数は t 分布のパーセント点を求めるもので，キーワード引数 ‘q=’ には累積確率を，‘loc=’ には t 分布のオフセットを，‘scale=’ には縮尺を，‘df=’ には自由度を指定する．

最後に，t 統計量が信頼区間に入っているかどうかを調べる．

例. t 統計量を検査（先の例の続き）

入力：
```
L<t and t<U
```

出力：　False　　←環境によっては np.False_ と表示されることがある

t 統計量が信頼できる範囲に無く，仮定した母平均が妥当でないことがわかる．このことを p 値によっても確かめる．（次の例）

例. p 値を使って確かめる（先の例の続き）

入力：
```
p = 2 * (1 - stats.t.cdf(abs(t), df=n-1))
print('p 値:', p)
print(' 判定:', ' 棄却' if p < 0.05 else ' 採択')
```

出力：　p 値:　0.00035887741731599476
判定:　棄却

[48] 詳しくは「A.3.2 母平均に関する検定（t 検定）」（p.156）参照のこと．

次に母平均を $\hat{\mu} = 0.05$ と仮定して同様の検定を試みる．

例. $\hat{\mu} = 0.05$ で再度検定（先の例の続き）

入力：
```
mu = 0.05     # 母平均の仮定
t = (x-mu)/(s/np.sqrt(n))     # t 統計量
print('t 統計量：',t)
L<t and t<U
```

出力：
```
t 統計量： -1.157531537778198
True          ←環境によっては np.True_ と表示されることがある
```

t 統計量が信頼できる範囲にあり，仮定した母平均が妥当であることがわかる．このことを p 値によっても確かめる．（次の例）

例. p 値を使って確かめる（先の例の続き）

入力：
```
p = 2 * (1 - stats.t.cdf(abs(t), df=n-1))
print('p 値:', p)
print(' 判定:', ' 棄却' if p < 0.05 else ' 採択')
```

出力：
```
p 値:  0.25844590095076936
判定:  採択
```

ここでの例からわかるように，母平均の仮定 $\hat{\mu} = 0.7$ が棄却され，$\hat{\mu} = 0.05$ が採択されるという，かなり有効な検定手段であることが伺える．

5　データベース

大量のデータを扱うには**データベース**の取り扱いが必須となる．データベースとしてデータを蓄積管理するためのシステムは DBMS [49]（データベース管理システム）と呼ばれる独立したシステムである．データベースの機能を使用するアプリケーションは，この DBMS を介してデータを抽出したり，データの登録や更新などを行う．この際の DBMS に対する処理の依頼を**クエリ**（query）と呼び，クエリのための言語として SQL [50] がある．本書では DBMS として主にパブリックドメインのフリーソフトウェアである SQLite の扱いを例に挙げて SQL によるデータベースの使用方法について説明する．SQLite は充分な性能がある上に導入方法も簡単であるため，データベースの機能についての学習や，アプリケーションへの組み込み，あるいは個人的な運用に適している．

5.1　データベースについての基本的な考え方

本書では**関係モデル**[51] に基づく**リレーショナル・データベース**（関係データベース，RDB）を取り扱う．RDB では，データを**対象**，**属性**，**値**の組として扱う．具体的には，これら組の 1 つを 1 行として扱い，複数の列（**項目**）の値を横方向に書き並べて 1 つの**レコード**とする．これは表計算ソフトウェアによる表の扱いに似ている．すなわち，左端の列に対象物の名称や記号を記述し，それより右側には各種の項目の値を書き並べる形式である．関係データベースは表を参照することによって対象の属性の値を返す，すなわち「…の〜は何？」という問い合わせ（クエリ）に対して，「△△△です」と値を返すシステムであると見做すことができる．

RDB では，複数の**表**（table）をまとめて 1 つの**データベース**（database）とする．表計算ソフトウェアの場合では，1 つのデータファイル（ブック）は複数の表（スプレッドシート）を束ねた形[52] になっており，この意味でも，RDB と表計算ソフトウェアの類似性（表 17）[53] が見られる．

表 17: RDB と表計算ソフトウェアの類似性

RDB	表計算ソフトウェア
table	シート
database	シートを束ねたデータファイル

RDB では特定の項目（列）を**主キー**（primary key）として定め，主キーによる高速なデータ検索を可能にしている．また，table の検索の結果として得られた値を**外部キー**（foreign key）として，別の table の主キーを検索することができ，複雑な情報検索を可能にする．

RDB では表の構成を設計する際に，どのような項目の集まりとするかに注意を払う必要があり，可能な限り，同じような項目が繰り返し現れることのないように配慮する必要がある．このような，無駄や矛盾が起こらないような配慮のために**正規化**と呼ばれる設計上の工夫が重要であるが，これに関しては本書では触れない．

5.1.1　データベースに対する基本的な操作

RDB に対する基本的な操作は**選択**，**射影**，**結合**の 3 種類である．データベースは多量のデータを保持するものであり，必要な行（レコード）や列（項目）を絞り込んでデータを抽出することが重要であり，選択は「必要な行の抽出」，射影は「必要な列の抽出」を意味する．また，RDB では 1 つのデータベース（database）に複数の表（table）があり，それら複数の表を連結して 1 つの表のように扱う機能が「結合」である．これらに関しては後に事例を挙げて具体的に説明する．

[49] Microsoft 社の Access や SQL Server，オープンソースの PostgreSQL や MySQL，パブリックドメインの SQLite などがある．

[50] **SQL**: データベースにアクセスする際の標準的な言語であり，多くの DBMS がその名称に「SQL」という語を冠している．本書では最も基本的な部分に関して「5.5 SQL」（p.111）で解説する．

[51] IBM のエドガー・F・コッドによって考案された現在もっとも広く用いられているデータモデル．

[52] Microsoft 社の Excel における標準的なデータファイル形式である「ブック形式」などがその例である．

[53] この類似性は，RDB を最初に理解する際の解釈の 1 つであり，厳密には RDB と表計算ソフトウェアは，仕組みと取扱方法が異なることに留意すること．

5.2 本書で取り扱うデータベース関連のソフトウェア

5.2.1 SQLite

SQLite はパブリックドメインの DBMS であり，ソフトウェア本体と関連情報が公式インターネットサイト

https://www.sqlite.org/

から入手できる．一般的な DBMS は**ミドルウェア**[54] として常時稼働し，複数の利用者から同時に**トランザクション**[55] を受け付けるという形態で運用される．これに対して SQLite はソフトウェアライブラリの形で提供され，アプリケーションに組み込んで[56] 使用する形態のソフトウェアである．SQLite が扱うデータベースは単一のファイルであり，運用と管理が極めて単純である．このため実際に多くのアプリケーションシステムに組み込まれて[57] 利用されている．

本書ではデータベースの DBMS としてこの SQLite を前提とする．

5.2.2 SQLAlchemy

SQLAlchemy は Python 処理系でデータベースを扱うためのオープンソースのソフトウェアライブラリであり，データベースを使用するための API を提供する．本書ではデータベースの取扱において SQLAlchemy を前提とする．このソフトウェアに関する情報は公式インターネットサイト

https://www.sqlalchemy.org/

で公開されている．

SQLAlchemy は PSF の標準ライブラリではないので，pip などのライブラリ管理ツール[58] を用いて別途インストールする必要がある．

例. pip コマンドによる SQLAlchemy のインストール

```
1）pip install sqlalchemy                ←最も標準的な操作
2）python -m pip install sqlalchemy       ← macOS や Linux での安全な操作
3）py -m pip install sqlalchemy           ← Windows 用 PSF 版 Python での安全な操作
```

PSF の Python の場合，計算機環境に合わせて上記 1)〜3) のいずれかの方法でインストールできる．

5.3 データベースに対するアクセスの例 (1)：DataFrame を基本とする処理

データベースを新規に作成して，pandas の DataFrame をデータベースのテーブルに登録する方法の最も簡単な例を示す．また既存のデータベース，テーブルに対する基本的な操作方法に関しても例示する

5.3.1 サンプルデータの作成

データベースに保存するためのサンプルデータを作成する．そのためにまず，必要となるソフトウェアライブラリを次のようにして読み込む．

例. ライブラリの読み込み

入力：
```
import sqlalchemy as sa  # SQLAlchemy を 'sa' の名で読込み
import pandas as pd      # pandas を 'pd' という名で読込み
```

この後 pandas と SQLAlchemy を使用することができる．この例では，SQLAlchemy の接頭辞を 'sa' としている．

次に，サンプルの DataFrame を作成する．

---

[54] OS とアプリケーションの中間に位置するもので，各種アプリケーションからのサービス要求に応えるといった役割を持つソフトウェア．

[55] データベースに対する一連の処理の手続きで，アクセスの基本的な流れとなる．

[56] SQLite は動的リンクライブラリ（Windows では DLL）の形で提供される．

[57] SQLite 本体（Windows の場合は sqlite3.dll）を対象アプリケーションのディレクトリに配置するなど，導入が単純である．
PSF 版 Python の sqlite3 モジュールではこの DLL は同梱されており，特に意識する必要はない．

[58] Anaconda ディストリビューションの場合は conda コマンドなどがこれにあたる．

例. サンプルの DataFrame 作成（先の例の続き）

入力：
```
df = pd.DataFrame( columns=[’番号’,’氏名’,’英語’,’化学’] ) # DataFrame の作成
df[’番号’] = [1,2,3]                                    # ’番号’ のカラム
df[’氏名’] = [’山田 太郎’,’田中 花子’,’中村 勝則’]      # ’氏名’ のカラム
df[’英語’] = [52,61,89]                                 # ’英語’ のカラム
df[’化学’] = [81,72,64]                                 # ’化学’ のカラム
df       # 内容確認
```

出力：

	番号	氏名	英語	化学
0	1	山田 太郎	52	81
1	2	田中 花子	61	72
2	3	中村 勝則	89	64

このようにしてできた DataFrame オブジェクト df を SQLite データベースに保存する作業を以下に例示する.

5.3.2 データベースへの接続

5.3.2.1 Engine オブジェクト

Engine オブジェクトは，データベースへのアクセスのための基本機能を提供するものであり，SQLAlchemy では，データベースの利用に際して最初にこれを作成する.

Engine オブジェクトの生成には SQLAlchemy の create_engine メソッドを使用する.（次の例参照）

例. Engine オブジェクトの生成（先の例の続き）

入力：
```
egn = sa.create_engine( ’sqlite:///testdb01.db’, echo=False ) # DB 接続エンジンの生成
```

この例では create_engine メソッドの第 1 引数にデータベースの URI を与えている．この処理で ‘testdb01.db’ という名前のデータベースにアクセスするための Engine オブジェクト egn を生成している.

この処理は単に Engine オブジェクトを作成するだけであり，ファイル資源としてのデータベースに対する処理は何も行われない.

■ SQL クエリの表示の有無

データベースへの実際のアクセスは，SQLAlchemy が DBMS に対して SQL クエリを発行することで実現される．先の例では create_engine メソッドにキーワード引数 ‘echo=False’ を与えているが，’echo=True’ を与えると次の例のように，発行された SQL クエリが表示され確認することができる.

例. ’echo=True’ によって表示されるクエリ

```
2025-11-21 19:41:55,021 INFO sqlalchemy.engine.Engine BEGIN (implicit)
2025-11-21 19:41:55,023 INFO sqlalchemy.engine.Engine PRAGMA main.table_info("tbl01")
2025-11-21 19:41:55,024 INFO sqlalchemy.engine.Engine [raw sql] ()
2025-11-21 19:41:55,025 INFO sqlalchemy.engine.Engine PRAGMA temp.table_info("tbl01")
2025-11-21 19:41:55,026 INFO sqlalchemy.engine.Engine [raw sql] ()
2025-11-21 19:41:55,027 INFO sqlalchemy.engine.Engine
CREATE TABLE tbl01 (
   "番号" BIGINT,
   "氏名" TEXT,
   "英語" BIGINT,
   "化学" BIGINT
)

2025-11-21 19:41:55,028 INFO sqlalchemy.engine.Engine [no key 0.00060s] ()
2025-11-21 19:41:55,038 INFO sqlalchemy.engine.Engine INSERT INTO tbl01
                         ("番号", "氏名", "英語", "化学") VALUES (?, ?, ?, ?)
2025-11-21 19:41:55,039 INFO sqlalchemy.engine.Engine [generated in 0.00086s]
                         [(1, ’山田 太郎’, 52, 81), (2, ’田中 花子’, 61, 72), (3, ’中村 勝則’, 89, 64)]
2025-11-21 19:41:55,040 INFO sqlalchemy.engine.Engine COMMIT
```

5.3.2.2 Connection オブジェクト

ファイル資源としてのデータベースに対する実際のアクセスには Connection オブジェクトを用いる．これは，Engine オブジェクトに対して connect メソッドを実行することで生成する.

書き方： Engine オブジェクト.connect( )

connect メソッドは Connection オブジェクトを返す.

例. Connection オブジェクトの生成（先の例の続き）

入力：
```
cn = egn.connect()  # データベースへの接続（Connection オブジェクトの生成）
```

この例では Connection オブジェクト cn を得ている．以後，この cn を介してデータベースにアクセスする．

データベースへのアクセスが終了した後は，close メソッドで接続を閉じる．

書き方：　Connection オブジェクト.close( )

「Connection オブジェクト」が保持する接続を終了する．

■ データベースへの接続に対する考え方

一般的な DBMS は同時に複数のプロセスからの接続を受け付け，排他制御によりデータの内容の一貫性を保持している．したがって，1 回のアクセス（ここでは**トランザクション**[59] を意味する）は可能な限り短時間で終了することが望ましい．これはトランザクション中の接続が排他制御の対象となるためであり，長時間の接続は他の処理を待たせる原因となるからである．したがって，各トランザクションは迅速に完了し，都度，接続を終了することが推奨される．

5.3.3　DataFrame のテーブルへの新規保存

DataFrame をデータベースに保存するには to_sql メソッドを DataFrame オブジェクトに対して実行する．

書き方：　DataFrame オブジェクト.to_sql(テーブル名, con=Connection オブジェクト,
index=True/False)

「DataFrame オブジェクト」の内容を指定した「テーブル名」でデータベースに保存する．対象となるデータベースは「Connection オブジェクト」が示すものである．to_sql メソッドは値を返さない（None）[60] .（次の例参照）

例. Connection オブジェクトを介して DataFrame を SQLite のテーブルに保存する（先の例の続き）

入力：
```
df.to_sql( 'tbl01', con=cn, index=False )    # DataFrame をデータベースに保存
cn.close()    # 接続の終了
```

この作業で DataFrame オブジェクト df の内容が，データベース testdb01 の新規のテーブル tbl01 として保存される．to_sql メソッドに与えているキーワード引数 'index=False' は，DataFrame のインデックスを保存の対象としないことを意味する．

▲注意▲

to_sql による上の例での保存処理では，対象のデータベース内に同名のテーブルが既に存在する場合，エラーとなるので注意すること．

5.3.4　テーブルから DataFrame への読込み

pandas の read_sql 関数でデータベースのテーブルを読み込むことができる．

書き方：　pd.read_sql(SQL 文, con=Connection オブジェクト)

「Connection オブジェクト」が示すデータベースに対して「SQL 文」（文字列形式）に記述された処理を実行し，得られたデータを DataFrame として返す．

先に作成したデータベースのテーブルから DataFrame にデータを読み込む例を示す．

例. SQLite の table からデータを読み込んで DataFrame にする（先の例の続き）

入力：
```
cn = egn.connect()  # データベースへの接続
q = 'SELECT * FROM tbl01'          # SQL クエリ
df2 = pd.read_sql( q, con=cn )    # 読込みの実行
cn.close()          # 接続の終了
df2                  # 内容確認
```

[59]「5.4 データベースに対するアクセスの例 (2)：SQL によるトランザクション処理」（p.107）で解説する．

[60]処理環境によっては to_sql メソッドは書き込んだデータ件数を返す．

出力：

	番号	氏名	英語	化学
0	1	山田 太郎	52	81
1	2	田中 花子	61	72
2	3	中村 勝則	89	64

この例では q に読込み処理のための SQL 文を与え，pandas の read_sql 関数に与えている．

この例で用いられている SQL 文は「テーブル tbl01 から（FROM）全てのカラム（'*' で表記）を対象にしてレコードを選択（SELECT）する」ことを意味する．(詳しくは p.111「5.5 SQL」を参照のこと)

■ with 構文によるアクセスの簡素な定型化

Connection オブジェクトの生成から接続の終了までの流れは with 構文により簡素な形で定型化できる．

《**with 構文の応用**》

記述例：　with egn.connect() as cn:
（データベースに対する処理の記述）

Engine オブジェクト egn によってデータベースに接続し，Connection オブジェクト cn を生成している．この cn による接続の下で「データベースに対する処理の記述」を実行し，それが終われば自動的に「cn.close()」を実行する．

以後はこの形式で実行例を示す．

5.3.5　既存のテーブルへの追加保存

DataFrame の内容を既存のテーブルに追加保存する方法を例示する．まず，追加用のデータを用意する．(次の例参照)

例. 追加用データの作成（先の例の続き）

入力：
```
df3 = pd.DataFrame( columns=['番号','氏名','英語','化学'] )  # DataFrame の作成
df3['番号'] = [4,5,6]                                        # '番号' のカラム
df3['氏名'] = ['吉田 たか子','ジョン スミス','リサ シェパード']  # '氏名' のカラム
df3['英語'] = [85,0,0]                                       # '英語' のカラム
df3['化学'] = [52,0,0]                                       # '化学' のカラム
df3    # 内容確認
```

出力：

	番号	氏名	英語	化学
0	4	吉田 たか子	85	52
1	5	ジョン スミス	0	0
2	6	リサ シェパード	0	0

このデータを to_sql メソッドを使用して，既存のテーブル tbl01 に追加する例を示す．

例. 既存のテーブルへのデータの追加（先の例の続き）

入力：
```
with egn.connect() as cn:
    df3.to_sql( 'tbl01', con=cn, index=False, if_exists='append' )
```

この例のように to_sql メソッドの引数にキーワード引数 'if_exists='append'' を与えることで，既存のテーブルにデータを追加保存することができる．

この処理の後のテーブルの内容を確認するために，次のような作業を行う．

例. テーブルを読み込んで内容を確認する（先の例の続き）

入力：
```
q = 'SELECT * FROM tbl01'            # SQL クエリ
with egn.connect() as cn:
    df2 = pd.read_sql( q, con=cn )   # 読込みの実行
df2    # 内容確認
```

出力：

	番号	氏名	英語	化学
0	1	山田 太郎	52	81
1	2	田中 花子	61	72
2	3	中村 勝則	89	64
3	4	吉田 たか子	85	52
4	5	ジョン スミス	0	0
5	6	リサ シェパード	0	0

データが追加保存されていることが確認できる.

5.3.6 既存のテーブルを新しいデータで置き換える

to_sql メソッドの引数にキーワード引数 'if_exists='replace'' を与えて DataFrame に対して実行すると，既存のテーブルをその DataFrame の内容で置き換える.（**注意**：実行前のテーブルの内容は失われる）
この処理の例を次に示す.

例. 既存のテーブルの置き換え（先の例の続き）

入力：
```
with egn.connect() as cn:
    df3.to_sql( 'tbl01', con=cn, index=False, if_exists='replace' )
```

この処理で既存のテーブル tbl01 の内容が一旦消去され，DataFrame オブジェクト df3 の内容で置き換えられる.（次の例参照）

例. テーブルを読み込んで内容を確認する（先の例の続き）

入力：
```
q = 'SELECT * FROM tbl01'            # SQL クエリ
with egn.connect() as cn:
    df2 = pd.read_sql( q, con=cn )   # 読込みの実行
df2    # 内容確認
```

出力：

	番号	氏名	英語	化学
0	4	吉田 たか子	85	52
1	5	ジョン スミス	0	0
2	6	リサ シェパード	0	0

テーブル tbl01 の内容が df3 の内容に置き換えられていることが確認できる.

5.3.7 指定した条件によるデータの抽出

指定した条件を満たすデータのみを読み込むには，read_sql メソッドでデータを読み込む際の SQL 文にそれを記述する.（次の例参照）

例. 条件を指定して読込み（先の例の続き）

入力：
```
q = 'SELECT * FROM tbl01 WHERE 氏名="ジョン スミス" OR 氏名="リサ シェパード"'
with egn.connect() as cn:
    df2 = pd.read_sql( q, con=cn )
df2    # 内容確認
```

出力：

	番号	氏名	英語	化学
0	5	ジョン スミス	0	0
1	6	リサ シェパード	0	0

該当するデータのみが読み込まれていることが確認できる．この例の SQL 文

```
SELECT * FROM tbl01 WHERE 氏名="ジョン スミス" OR 氏名="リサ シェパード"
```

は「WHERE 句」の記述を含んでおり，これでデータ抽出の条件を記述する．WHERE 句の条件式には AND，OR，NOT といった論理演算子が使用できる．

5.3.8　read_sql，to_sql の con 引数に関すること

本書では DBMS として SQLite を前提としているが，pandas と SQLAlchemy は対象の DBMS として MySQL [61]，PostgreSQL [62] をはじめとする多くのものに対応しており，使用する DBMS における**接続オブジェクト**を read_sql，to_sql の con 引数に与えることで，本書で解説している処理が基本的に実行可能となる．

SQLite を使用する場合は，read_sql，to_sql の con 引数に Engine オブジェクトを与えることもできるが，それは推奨されない．

5.4　データベースに対するアクセスの例 (2)：SQL によるトランザクション処理

データベースへのアクセスは SQL で記述された一連の手続きに基づいた形で行われる．この流れを概略的に見ると次のような 3 つの段階から成る．

1．データベース利用の開始
2．データの抽出，追加，更新，削除といった各種の作業
3．データベース利用の終了

先に pandas の DataFrame の扱いを基本にしてデータベースにアクセスする方法を例示してきたが，データベースのレコード単位での更新や削除といった細かい作業をするには，その都度 SQL 文で記述した**クエリ**を DBMS に対して発行することになる．

■ トランザクション

トランザクションとは，一連の SQL の処理のまとまりのことであり，より正確には，正しく完了することのできる作業単位である．また，1 つのトランザクションは COMMIT （確定）の処理と，ROLLBACK（取り消し）の処理の対象となる作業単位[63] である．

ここではトランザクション処理の最も基本的な部分について説明する．

5.4.1　トランザクションの開始

Connection オブジェクトを生成してデータベースとの接続ができると，その後，トランザクションの処理が可能となる．1 つのトランザクションは begin（開始）から commit（確定）までの一連の処理単位であり，Connection オブジェクトに対して begin メソッドを実行することで開始する．

書き方：　Connection オブジェクト.begin( )

この処理の結果，Transaction オブジェクトが返され，それに対して commit メソッドを実行することでトランザクションの処理が確定する．ただし，SQLAlchemy の Connection オブジェクトは auto-begin（自動トランザクション開始）機能を持ち，最初の SQL を実行した時点で，自動的にトランザクションが開始される．begin メソッドの実行を省略した場合は，commit メソッドは Connection オブジェクトに対して実行する．

トランザクション開始後は，当該 Connection オブジェクトに対して execute メソッドで SQL を実行する処理を記述する．

書き方：　Connection オブジェクト.execute(SQL 文)

execute メソッドの戻り値は CursorResult オブジェクトである．このオブジェクトに関しては「5.4.5 execute メソッドの戻り値」（p.109）で説明する．「SQL 文」には，文字列（str 型）で記述した SQL を，実行可能な Executable SQL（**構文木オブジェクト**）に変換したものを与える．これには SQLAlchemy が提供する text 関数を用いる．

[61] https://www.mysql.com/

[62] https://www.postgresql.org/

[63] 全て正しく完了することができ，また，まとめて処理を取り消すことのできる作業単位．

■ データベースに変更を加える処理に関すること

データベースの参照ではなく，追加や削除をはじめとする**更新系の処理**を実行する場合は，先の execute メソッドの実行の後に commit メソッドを Connection オブジェクトに対して実行して，トランザクションを確定する必要がある．

書き方：　Connection オブジェクト.commit( )

このメソッドは値を返さない．(None を返す)　更新系でない処理（例．SELECT）の場合は commit の実行を省略することができる．また，pandas の DataFrame の SQL 関連のメソッドにおいては，必要に応じて commit の処理を自動的に実行するので，明示的に commit メソッドを実行する必要はない．

5.4.2　既存のレコードの変更（データベースの更新）

データベースの既存のレコードの指定したカラムの値を変更するには SQL の UPDATE 文を発行する．(次の例参照)

例． 既存のレコードの更新（先の例の続き）

入力：
```
s1 = sa.text('UPDATE tbl01 SET 英語=99, 化学=24 WHERE 氏名="ジョン スミス"')
s2 = sa.text('UPDATE tbl01 SET 英語=100, 化学=41 WHERE 氏名="リサ シェパード"')
with egn.connect() as cn:
    r1 = cn.execute(s1)
    r2 = cn.execute(s2)
    cn.commit()
```

これで ‘ジョン スミス’ 氏と ‘リサ シェパード’ 氏のレコードが更新される．更新対象のレコードが WHERE 句に記述された条件で絞り込まれ，SET 句で新規の値を設定している．

例． 更新後の確認（先の例の続き）

入力：
```
q = 'SELECT * FROM tbl01'
with egn.connect() as cn:
    df2 = pd.read_sql( q, con=cn )
df2    # 内容確認
```

出力：

	番号	氏名	英語	化学
0	4	吉田 たか子	85	52
1	5	ジョン スミス	99	24
2	6	リサ シェパード	100	41

5.4.3　既存のレコードの削除

データベースの既存のレコードを削除するには SQL の DELETE 文を発行する．(次の例参照)

例． 既存のレコードの削除（先の例の続き）

入力：
```
s = sa.text('DELETE FROM tbl01 WHERE 氏名="ジョン スミス"')
with egn.connect() as cn:
    r = cn.execute(s)
    cn.commit()
```

これで ‘ジョン スミス’ 氏のレコードが削除される．更新対象のレコードが WHERE 句に記述された条件で絞り込まれている．

例． 更新後の確認（先の例の続き）

入力：
```
q = 'SELECT * FROM tbl01'
with egn.connect() as cn:
    df2 = pd.read_sql( q, con=cn )     # 読込みの実行
df2    # 内容確認
```

出力：

	番号	氏名	英語	化学
0	4	吉田 たか子	85	52
1	6	リサ シェパード	100	41

5.4.4 新規レコードの追加

データベースに新規のレコードを追加するには SQL の INSERT 文を発行する.（次の例参照）

例. レコードの新規追加（先の例の続き）

入力：
```
s = sa.text('''INSERT INTO tbl01 (番号, 氏名, 英語, 化学)
                        VALUES (5, "斉藤 ジェシカ", 93, 82)''')
with egn.connect() as cn:
    r = cn.execute(s)
    cn.commit()
```

これで '斉藤 ジェシカ' 氏のレコードが新規に追加される．新規レコードに設定する値は

(カラム名の列) VALUES (値の列)

とそれぞれ対応させて記述する.

例. 新規追加後の確認（先の例の続き）

入力：
```
q = 'SELECT * FROM tbl01'
with egn.connect() as cn:
    df2 = pd.read_sql( q, con=cn )
df2    # 内容確認
```

出力：

	番号	氏名	英語	化学
0	4	吉田 たか子	85	52
1	6	リサ シェパード	100	41
2	5	斉藤 ジェシカ	93	82

5.4.5 execute メソッドの戻り値

execute メソッドの戻り値である CursorResult オブジェクトを用いてデータベースのレコードの内容を参照することができる．例えば SQL の SELECT 文はデータベースのレコードを選択するためのものであり，先にも read_sql メソッドの引数として与える例を示したが，ここでは execute メソッドで SELECT 文を実行し，その結果として得られる CursorResult オブジェクトを用いてデータベースのレコードの内容を参照する方法を紹介する.

例. SELECT 文の実行（先の例の続き）

入力：
```
s = sa.text('SELECT * FROM tbl01')
with egn.connect() as cn:
    r = cn.execute(s)
```

この例は単にデータを選択するのみである.（commit メソッドを実行する必要はない）得られている CursorResult オブジェクト r に対して mappings メソッドを実行すると，Python の**イテラブル**な形式のオブジェクト（MappingResult オブジェクト）が得られ，繰り返し制御のためのデータ列として扱うことができる.（次の例参照）

例. CursorResult を用いたデータの参照（先の例の続き）

入力：
```
print( r.keys() )
for rec in r.mappings():
    print( rec['番号'], rec['氏名'], rec['英語'],rec['化学'])
```

出力：
```
RMKeyView(['番号', '氏名', '英語', '化学'])
4 吉田 たか子 85 52
6 リサ シェパード 100 41
5 斉藤 ジェシカ 93 82
```

データベースのレコードが 1 行ずつ取り出されていることがわかる．この例では CursorResult オブジェクトに対して keys メソッドを実行してレコードのキー（カラム）の名前の並びを RMKeyView オブジェクトとして取得して出力している.

注意）CursorResult や MappingResult は**消費型のオブジェクト**であり，for 文などで一度繰り返し処理を行うと内容を再利用することはできない．keys メソッドは安全に使用できるが，実際のレコード内容を参照する処理は一回限りである点に注意すること．

上の例では，mappings メソッドによって取り出された MappingResult オブジェクトを for 文に使用し，その要素（RowMapping オブジェクト）を rec に受け取りながら，SELECT で選択されたレコードの情報を出力している．RowMapping オブジェクトは辞書として使用することができ，スライス（添字）‘[...]’ でカラム名を指定して値を取り出すことができる．

5.4.6 データベースの使用の終了

データベースの使用が終了した後，開いたままの Connection オブジェクトがあれば，それに対して close メソッドを実行する．また，データベースの使用を完全に終了するには，Engine オブジェクトに対して dispose メソッドを実行する．(次の例参照)

例. データベース使用の終了（先の例の続き）

入力：
```
cn.close()      # トランザクションの終了
egn.dispose()   # Engine の解放
```

既に接続を終了した Connection オブジェクトに close メソッドを実行しても何も起こらない．

参考）Engine オブジェクトに対して dispose メソッドを実行すると，内部で保持しているコネクションプールを破棄することができるが，通常の利用では必須ではない．SQLite のデータベースファイルを完全に閉じたい場合など，特殊な用途で使用する．

5.5 SQL

SQL はデータベースに対するクエリを記述するための言語として最も多く用いられており，多くの DBMS で標準的に採用されている．本書では，Python でデータ処理を行う際に必要となる最低限の範囲で SQL について例を示す形で説明する．

SQLite と SQLAlchemy を用いた実行例を示すに当たって，次のようにしてライブラリを読み込んでおく．

例. サンプルを実行するための準備

入力：
```
import sqlalchemy as sa     # SQLAlchemy を 'sa' の名で読込み
import pandas as pd         # pandas を 'pd' という名で読込み
```

サンプルとして作成するテーブルを表 18 に示す．

表 18: 作成するテーブルのサンプル

コード（整数）	都道府県（文字列）
主キー	-
13	東京都
23	愛知県
26	京都府
27	大阪府

テーブル名：tblPref

都道府県（文字列）	面積（整数）
主キー	-
東京都	2191
神奈川県	2416
愛知県	5172
大阪府	1905

テーブル名：tblGeom

表 18 の tblPref は都道府県コードに対する都道府県名を保持するもので，tblGeom は都道府県の面積を保持するものである．表 18 にある**主キー**とは，当該テーブルから目的のレコードを見つけ出すための検索キーであり，関係データベースの重要な要素である．

関係データベースでは，情報のカテゴリや利用目的に沿った形のテーブルを複数用意して，それらを関連付けて高度な情報検索を実現する．ここでは，上記 2 つのテーブルを**結合**して，両方のテーブルの全ての情報を 1 つのテーブルのように扱うことができることを例示する．

5.5.1 データベースの作成

複数のテーブルを保持するデータベースを新規に作成するには，SQL の文 'CREATE DATABASE' をクエリとして発行する．具体的には作成するデータベースの名前を指定して

CREATE DATABASE IF NOT EXISTS **データベース名**

と記述する．SQLite と SQLAlchemy ではこの処理は必要ではなく[64]，データベース名を表す URI を指定して Engine オブジェクトを生成し，トランザクションのための Connection オブジェクトを生成することでデータベースが作成される．（データベースが既存の場合はそれが使用される）

SQLite と SQLAlchemy による実行例を次に示す．

例. データベースの作成と利用開始（先の例の続き）

入力：
```
# DB 用エンジンの生成
egn = sa.create_engine( 'sqlite:///testJPNdb.db', echo=False )
```

データベースにアクセスするための Engine オブジェクト egn と，トランザクション処理のための Connection オブジェクト cn が生成されている．この処理の結果としてデータベース testJPNdb が作成される．（既存の場合はそれが使用される）　SQLAlchemy では，この Connection オブジェクトに対してクエリを発行する．

[64]SQLite は CREATE DATABASE 文をサポートしていない（文法として存在しない）ので，敢えて実行を試みるとエラーとなる．

5.5.2 テーブルの作成

テーブルを新規に作成するには，‘CREATE TABLE’ 文をクエリとして発行する．

《テーブルの作成》
書き方：　CREATE TABLE テーブル名 (カラム名 1　型, カラム名 2　型,　…,
PRIMARY KEY(対象のカラム))
「テーブル名」で指定したテーブルを作成する．この際，保持するカラムとそのデータ型を指定する．
PRIMARY KEY に主キーとするカラムを指定する．(複数可)

5.5.2.1 カラムのデータ型

カラムのデータ型を表 19 に挙げる．

表 19: カラムのデータ型（SQLite の場合）

型	解説	型	解説
INTEGER	整数	REAL, NUMERIC	浮動小数点数
TEXT	文字列	NONE	型なし

サンプルとして示したテーブル tblPref を作成するには次のように実行する．

例. サンプルのテーブル tblPref の作成（先の例の続き）

入力：
```
# テーブル tblPref の生成
s1 = sa.text('''CREATE TABLE tblPref( コード INTEGER, 都道府県 TEXT,
                PRIMARY KEY(コード) )''')
# レコードの挿入
s2 = sa.text('''INSERT INTO tblPref ( コード, 都道府県 )
                VALUES (13,"東京都"), (23,"愛知県"), (27,"大阪府"), (26,"京都府")''')
with egn.connect() as cn:
    r = cn.execute( s1 )
    r = cn.execute( s2 )
    cn.commit()    # 処理を DB に反映
```

これでテーブル tblPref が作成され，レコードが 4 件登録される．

Connection オブジェクトに対して execute メソッドを実行すると，戻り値として CursorResult オブジェクトが得られる．上の例では r にそれが得られている．

注意）既存のテーブルと同じ名前のテーブルを作成するクエリを発行するとエラーとなる[65] ので注意すること．

テーブルへのレコードの挿入には ‘INSERT INTO’ 文を発行する．

《レコードの挿入》
書き方：　INSERT INTO テーブル名 (カラム 1,　カラム 2,　…)
VALUES (カラム 1 の値, カラム 2 の値),
(カラム 1 の値, カラム 2 の値), …)
「テーブル名」で指定したテーブルにレコードを挿入（追加）する．その際の各カラムの値を記述する．
VALUES 句の後ろには複数の値の列を記述することができ，複数のレコードを挿入することができる．

同様の方法でテーブル tblGeom を次のようにして作成する．

[65] SQLite ではデータベースが単一のファイルとして作成されるので，トランザクションが開いていない状態であれば，データベースのファイルを削除することで全てのテーブルを破棄することができる．

例. サンプルのテーブル tblGeom の作成（先の例の続き）

入力：
```
# テーブル tblGeom の生成
s1 = sa.text('''CREATE TABLE tblGeom( 都道府県 TEXT, 面積 INTEGER,
                PRIMARY KEY(都道府県) )''')
# レコードの挿入
s2 = sa.text('''INSERT INTO tblGeom ( 都道府県, 面積 )
                VALUES ("東京都",2191), ("神奈川県",2416),
                       ("愛知県",5172), ("大阪府",1905)''')
with egn.connect() as cn:
    r = cn.execute( s1 )
    r = cn.execute( s2 )
    cn.commit()    # 処理を DB に反映
```

以上の処理で，2 つのテーブル tblPref, tblGeom が作成されて値が設定された．この後は ‘SELECT’ 文によるクエリを発行してテーブルのレコードを抽出（選択処理）することができる．

《レコードの選択》
書き方： SELECT カラム 1, カラム 2, … FROM テーブル名 WHERE 条件式

「テーブル名」で指定したテーブルから,「カラム 1, カラム 2,…」のカラムを指定してレコードを選択する．このとき，WHERE 句にレコードを選択するための条件を記述する．WHERE 句を省略すると，全てのレコードが選択される．カラムの記述をアスタリスク ‘*’ とすると，全てのカラムが対象となる．

SELECT 文の発行によって得られた CursorResult オブジェクトを介して選択されたレコードを読み取ることができる．これに関しては「5.4.5 execute メソッドの戻り値」（p.109）のところで説明したが，read_sql メソッドによる読込み処理では独自のトランザクション処理をするため，Connection オブジェクトを明に生成することなくデータベースの読込みができる．

ここで，作成したテーブルの内容を確認する.（次の例参照）

例. tblPref の内容確認（先の例の続き）

入力：
```
q = 'SELECT * FROM tblPref'        # SQL クエリ
with egn.connect() as cn:
    df = pd.read_sql( q, con=cn )   # 読込みの実行
df      # 内容確認
```

出力：

	コード	都道府県
0	13	東京都
1	23	愛知県
2	26	京都府
3	27	大阪府

例. tblGeom の内容確認（先の例の続き）

入力：
```
q = 'SELECT * FROM tblGeom'        # SQL クエリ
with egn.connect() as cn:
    df = pd.read_sql( q, con=cn )   # 読込みの実行
df    # 内容確認
```

出力：

	都道府県	面積
0	東京都	2191
1	神奈川県	2416
2	愛知県	5172
3	大阪府	1905

pandas の DataFrame に対する SQL 関連メソッドでは，SQL 文を文字列（str）与えることができる[66] ．これは，

[66] メソッド内で，SQLAlchemy の text 関数が使用されているため.

Connection オブジェクトに対する execute メソッドの場合と異なる点である．

5.5.3 テーブルの結合

データベースの複数の表を**結合**して 1 つのテーブルのように扱うことができる．実際のデータベースシステムでは，データのカテゴリや使用目的毎に複数のテーブルが作られており，それらを結合して高度な情報抽出を行う．具体的には SELECT 文の INNER JOIN 句などを記述して複数のテーブルの結合を指定する．

《テーブルの結合》
書き方： SELECT カラムの記述 FROM 主たるテーブル名 INNER JOIN 結合するテーブル ON 結合するカラムの対応

ここで扱っているサンプルでは，tblPref のテーブルに「都道府県」の名前を持つカラムがあり，これを tblGeom の主キーに結合することができる．それによって，2 つのテーブルを結合して 1 つのテーブルのように扱うことができる．以下にその例を示す．

例. INNER JOIN（内部結合）（先の例の続き）

入力：
```
q = '''SELECT tblPref.コード, tblPref.都道府県, tblGeom.面積 FROM tblPref
            INNER JOIN tblGeom ON tblPref.都道府県 = tblGeom.都道府県'''
with egn.connect() as cn:
    df = pd.read_sql( q, con=cn )     # 読込みの実行
df     # 内容確認
```

出力：

	コード	都道府県	面積
0	13	東京都	2191
1	23	愛知県	5172
2	27	大阪府	1905

テーブル tblPref に tblGeom が結合している様子がわかる．ここで注意しなければならないこととして，tblPref の「京都府」のレコードが得られていないことがある．これは，tblGeom に対応するレコードが存在しないことが理由である．INNER JOIN はデータベースの**内部結合**を意味し，選択処理の結果として対応が取れないレコードを除外する．tblGeom にレコードが存在しない場合も除外せずに結合結果を得るには次のようにする．

例. LEFT OUTER JOIN（左外部結合）（先の例の続き）

入力：
```
q = '''SELECT tblPref.コード, tblPref.都道府県, tblGeom.面積 FROM tblPref
          LEFT OUTER JOIN tblGeom ON tblPref.都道府県 = tblGeom.都道府県'''
with egn.connect() as cn:
    df = pd.read_sql( q, con=cn )     # 読込みの実行
df     # 内容確認
```

出力：

	コード	都道府県	面積
0	13	東京都	2191.0
1	23	愛知県	5172.0
2	26	京都府	NaN
3	27	大阪府	1905.0

これは**左外部結合**によるもので，LEFT OUTER JOIN で結合を指定する．対応が取れずに値が得られなかったカラムの部分には**非数**（欠損値 NaN）が設定される．

※ この他にも様々な結合形態が存在するが本書では割愛する．

処理の最後に，Engine オブジェクトを解放しておく．

例. 終了処理（先の例の続き）

入力：
```
egn.dispose()   # Engine を解放
```

6　各種ライブラリが提供する関数やメソッド

SciPy ライブラリは統計学，確率論，解析学に関する多くの関数を提供する．ここではその 1 部を紹介する．

6.1　scipy.special

scipy.special は多くの数学関数を提供する．このライブラリを利用するには次のようにして読み込む．

例. scipy.special の読込み

入力：
```
import scipy.special as special    # scipy.special を 'special' の名で読込み
```

このように別名 'special' を与えておくと，以後はこれを接頭辞として各種関数を呼び出すことができる．

6.1.1　階乗 $n!$

階乗を求めるには scipy.special.factorial 関数を使用する．(次の例参照)

例. 10! の計算（先の例の続き）

入力：
```
a1 = special.factorial(10)  # 10!  の算出
print( a1 )
type( a1 )  # データ型の調査
```

出力：
```
 3628800.0
numpy.ndarray
```

この例は $n!$ を算出するものである．計算結果のデータ型は NumPy の配列 ndarray であることがわかる．この例のように 1 つの値（スカラー）を与えて 1 つの値を返す場合は，計算結果を float(a1) などとして通常の数値（スカラー）に変換しておくと安全である．

SciPy の関数群は NumPy ライブラリを用いて構築されており，多くの関数において，スカラーだけでなく配列（ndarray）を与えることができる．(次の例参照)

例. $1!, 2!, 3!, 4!, 5!$ を一度に算出（先の例の続き）

入力：
```
special.factorial([1,2,3,4,5]) # 複数の値について計算
```

出力：
```
array([ 1., 2., 6., 24., 120.])
```

算出する値が大きくなる場合は，仮数部を丸めた形で結果を返す．(次の例参照)

例. 100! の計算（先の例の続き）

入力：
```
a1 = special.factorial(100) # 100!  の算出
print( a1 )
```

出力：
```
9.332621544394415e+157
```

仮数部を丸めた形で得られており，この計算結果は**誤差**を含む．正確な値を求める場合は factorial 関数にキーワード引数 'exact=True' を与える．(次の例参照)

例. 100! を正確に求める（先の例の続き）

入力：
```
a1 = special.factorial(100, exact=True)  # 100!  の算出
print( a1 )
type( a1 )  # データ型の調査
```

出力：
```
 933262154439441526816992388562667004907159682643816214685929638952175999932299 1
 560894146397615651828625369792082722375825118521091686400000000000000000000000
int
```

計算結果が Python 本来の整数型（int）で得られている[67]．ただし，この方法による計算は，NumPy 独特の大きな

[67] Python の整数（int）型は長い桁の値が扱える．

計算速度が得られないことに注意すること.

6.1.2 順列 ${}_nP_r$，組合せ ${}_nC_r$

順列を求めるには scipy.special.perm 関数を使用する.（次の例参照）

例. ${}_5P_2$ の計算（先の例の続き）

入力：
```
a2 = special.perm(5,2)       # 5P2 の計算
print( a2 )
type( a2 )  # データ型の調査
```

出力：
```
  20.0
numpy.float64
```

計算結果は NumPy の数値（float64）として得られる．計算結果が大きくなる場合は仮数部を丸めた形で結果を返す.（次の例参照）

例. ${}_{100}P_{40}$ の計算（先の例の続き）

入力：
```
special.perm(100,40)     # 100P40 の計算
```

出力：
```
  1.1215762526664621e+76      ←環境によっては np.float64(1.1215762526664621e+76) と表示されることがある
```

仮数部を丸めた形で得られており，この計算結果は**誤差**を含む．正確な値を求める場合は perm 関数にキーワード引数 ‘exact=True’ を与える.（次の例参照）

例. ${}_{100}P_{40}$ を正確に求める（先の例の続き）

入力：
```
a2 = special.perm(100,40, exact=True)     # 100P40 の計算
print( a2 )
type( a2 )  # データ型の調査
```

出力：
```
  11215762526664624508781016841049139209146570250671219642011054243840000000000
int
```

計算結果が Python 本来の整数型（int）で得られている．ただし，この方法による計算では，NumPy 独特の大きな計算速度が得られないことに注意すること.

組合せの値を求めるには scipy.special.comb 関数を使用する．取り扱い方法は順列の計算の場合と同様である.（次の例参照）

例. ${}_5C_2$ の計算（先の例の続き）

入力：
```
a3 = special.comb(5,2)  # 5C2 の計算
print( a3 )
type( a3 )  # データ型の調査
```

出力：
```
  10.0
numpy.float64
```

計算結果は NumPy の数値（float64）として得られる．計算結果が大きくなる場合は仮数部を丸めた形で結果を返す.（次の例参照）

例. ${}_{1000}C_{40}$ の計算（先の例の続き）

入力：
```
special.comb(1000,40)     # 1000C40 の計算
```

出力：
```
  5.559744235716389e+71      ←環境によっては np.float64(5.559744235716389e+71) と表示されることがある
```

この計算結果は丸めによる**誤差**を含む．正確な値を求める場合は comb 関数にキーワード引数 ‘exact=True’ を与える.（次の例参照）

例. ${}_{1000}C_{40}$ の計算（先の例の続き）

入力：
```
a3 = special.comb(1000,40, exact=True)  # 1000C40 の計算
print( a3 )
type( a3 )  # データ型の調査
```

出力：
```
55597442357166403381580458924355384985125805664971991968784202722320847 5
int
```

計算結果が Python 本来の整数型（int）で得られている．ただし，この方法による計算では，NumPy 独特の大きな計算速度が得られないことに注意すること．

6.2 scipy.stats

scipy.stats は統計学，確率論に関する多くの関数を提供する．ここでは，実行例を示しながらこのライブラリの使用方法について説明する．

NumPy，matplotlib と共に scipy.stats ライブラリを次のようにして読み込む．

例. scipy.stats をはじめとするライブラリの読込み

入力：
```
import scipy.stats as stats      # scipy.stats の読込み
import numpy as np               # NumPy を 'np' の名で読込み
import matplotlib.pyplot as plt  # グラフ描画ライブラリを 'plt' の名で読込み
```

scipy.stats に別名 'stats' を与えておくと，以後はこれを接頭辞として各種関数を呼び出すことができる．また，この部分を「`from scipy import stats`」として読み込んでも良い．

6.2.1 確率密度関数：PDF（Probability Density Function）

各種の確率密度関数の使用方法について説明する．SciPy では確率密度関数は 'pdf' である．

6.2.1.1 正規分布

正規分布に関する関数群は scipy.stats.norm に含まれ，正規分布の確率密度関数 pdf を呼び出す場合は scipy.stats.norm.pdf と記述する．（別名の接頭辞により記述を省略できる）

書き方：　scipy.stats.norm.pdf(定義域の値, loc=μ, scale=σ)

「定義域の値」には 1 つの数値（スカラー）あるいはデータ列を与えることができる．

例. 正規分布の確率密度関数 (1)（先の例の続き）

入力：
```
y = stats.norm.pdf( -1, loc=0, scale=1 )
print( y )
type( y )    # データ型の調査
```

出力：
```
0.24197072451914337
numpy.float64
```

結果が NumPy の数値として得られている．次に定義域の値をデータ列として与えて，それに対する値域をデータ列として取得する例を示す．

例. 正規分布の確率密度関数 (2)（先の例の続き）

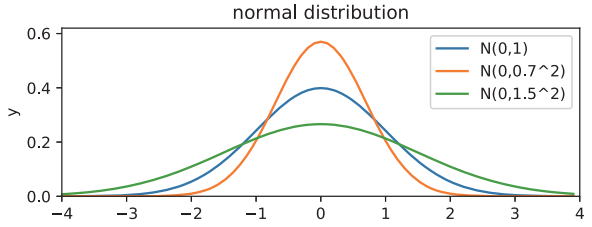
入力：
```
ax = np.arange( -4, 4, 0.1 )    # -4 以上 4 未満の数列を 0.1 間隔で生成
ay1 = stats.norm.pdf( ax, loc=0, scale=1 )    # N(0,1)
ay2 = stats.norm.pdf( ax, loc=0, scale=0.7 )  # N(0,0.7^2)
ay3 = stats.norm.pdf( ax, loc=0, scale=1.5 )  # N(0,1.5^2)
```

この例では定義域のデータ列を ax に作成し，それに対する値域を 3 種類の標準偏差毎（1.0，0.7，1.5）に ay1，ay2，ay3 として取得している．それらを matplotlib によって可視化する例を次に示す．

例. 可視化処理（先の例の続き）

入力：
```
plt.figure( figsize=(6,2) )              # サイズを指定して描画作業の開始
plt.plot( ax, ay1, label='N(0,1)' )      # グラフのプロット 1
plt.plot( ax, ay2, label='N(0,0.7^2)' )  # グラフのプロット 2
plt.plot( ax, ay3, label='N(0,1.5^2)' )  # グラフのプロット 3
plt.xlim(-4,4)                           # 描画範囲（横軸）の設定
plt.ylim(0,0.62)                         # 描画範囲（縦軸）の設定
plt.title('normal distribution')         # グラフのタイトルの設定
plt.xlabel('x')                          # 横軸ラベル
plt.ylabel('y')                          # 縦軸ラベル
plt.legend()                             # 凡例の表示
plt.show()                               # 描画の実行
```

出力：



6.2.1.2 t 分布

t 分布に関する関数群は scipy.stats.t に含まれ，t 分布の確率密度関数 pdf を呼び出す場合は scipy.stats.t.pdf と記述する．(別名の接頭辞により記述を省略できる)

書き方：　scipy.stats.t.pdf(定義域の値, 自由度, loc=オフセット, scale=比率)

「定義域の値」には 1 つの数値（スカラー）あるいはデータ列を与えることができる．

例. t 分布の確率密度関数 (1)（先の例の続き）

入力：
```
y = stats.t.pdf( 0, 1, loc=0, scale=1 )
print( y )
type( y )     # データ型の調査
```

出力：
```
0.31830988618379075
numpy.float64
```

結果が NumPy の数値として得られている．次に定義域の値をデータ列として与えて，それに対する値域をデータ列として取得する例を示す．

例. t 分布の確率密度関数 (2)（先の例の続き）

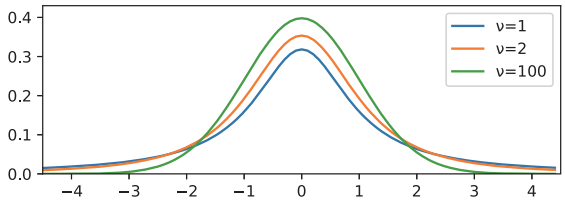
入力：
```
ax = np.arange( -4.5, 4.5, 0.1 )     # -4.5 以上 4.5 未満の数列を 0.1 間隔で生成
ay1 = stats.t.pdf( ax, 1,   loc=0, scale=1 )     # ν=1
ay2 = stats.t.pdf( ax, 2,   loc=0, scale=1 )     # ν=2
ay3 = stats.t.pdf( ax, 100, loc=0, scale=1 )     # ν=100
```

この例では定義域のデータ列を ax に作成し，それに対する値域を 3 種類の自由度毎（1.0，2.0，100.0）に ay1，ay2，ay3 として取得している．それらを matplotlib によって可視化する例を次に示す．

例. 可視化処理（先の例の続き）

入力：
```
plt.figure( figsize=(6,2) )              # サイズを指定して描画作業の開始
plt.plot( ax, ay1, label='ν=1' )         # グラフのプロット 1
plt.plot( ax, ay2, label='ν=2' )         # グラフのプロット 2
plt.plot( ax, ay3, label='ν=100' )       # グラフのプロット 3
plt.xlim(-4.5,4.5)                       # 描画範囲（横軸）の設定
plt.ylim(0,0.43)                         # 描画範囲（縦軸）の設定
plt.legend()                             # 凡例の表示
plt.show()                               # 描画の実行
```

出力：



6.2.1.3 χ^2 分布

χ^2 分布に関する関数群は scipy.stats.chi2 に含まれ，χ^2 分布の確率密度関数 pdf を呼び出す場合は scipy.stats.chi2.pdf と記述する．(別名の接頭辞により記述を省略できる)

書き方：　scipy.stats.chi2.pdf(定義域の値, 自由度, loc=オフセット, scale=比率)

「定義域の値」には 1 つの数値（スカラー）あるいはデータ列を与えることができる．

例. χ^2 分布の確率密度関数 (1)（先の例の続き）

入力：
```
y = stats.chi2.pdf( 2, 1, loc=0, scale=1 )
print( y )
type( y )     # データ型の調査
```

出力：
```
0.10377687435514868
numpy.float64
```

結果が NumPy の数値として得られている．次に定義域の値をデータ列として与えて，それに対する値域をデータ列として取得する例を示す．

例. χ^2 分布の確率密度関数 (2)（先の例の続き）

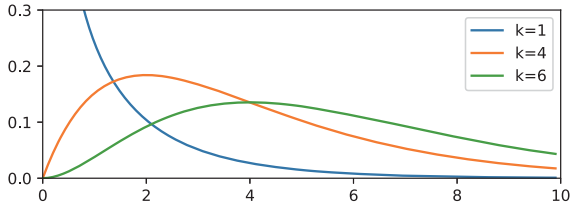
入力：
```
ax = np.arange( 0, 10, 0.1 )     # 0 以上 10 未満の数列を 0.1 間隔で生成
ay1 = stats.chi2.pdf( ax, 1, loc=0, scale=1 )     # k=1
ay2 = stats.chi2.pdf( ax, 4, loc=0, scale=1 )     # k=4
ay3 = stats.chi2.pdf( ax, 6, loc=0, scale=1 )     # k=6
```

この例では定義域のデータ列を ax に作成し，それに対する値域を 3 種類の自由度毎（1.0，4.0，6.0）に ay1，ay2，ay3 として取得している．それらを matplotlib によって可視化する例を次に示す．

例. 可視化処理（先の例の続き）

入力：
```
plt.figure( figsize=(6,2) )          # サイズを指定して描画作業の開始
plt.plot( ax, ay1, label='k=1' )     # グラフのプロット 1
plt.plot( ax, ay2, label='k=4' )     # グラフのプロット 2
plt.plot( ax, ay3, label='k=6' )     # グラフのプロット 3
plt.xlim(0,10)                       # 描画範囲（横軸）の設定
plt.ylim(0,0.3)                      # 描画範囲（縦軸）の設定
plt.legend()                         # 凡例の表示
plt.show()                           # 描画の実行
```

出力：



6.2.1.4 指数分布

指数分布に関する関数群は scipy.stats.expon に含まれ，指数分布の確率密度関数 pdf を呼び出す場合は scipy.stats.expon.pdf と記述する．(別名の接頭辞により記述を省略できる)

書き方：　scipy.stats.expon.pdf(定義域の値, loc=オフセット, scale=1/λ)

「定義域の値」には 1 つの数値（スカラー）あるいはデータ列を与えることができる．

例. 指数分布の確率密度関数 (1)（先の例の続き）

入力：
```
y = stats.expon.pdf( 0, loc=0, scale=1 )
print( y )
type( y ) # データ型の調査
```

出力：
```
1.0
numpy.float64
```

結果が NumPy の数値として得られている．次に定義域の値をデータ列として与えて，それに対する値域をデータ列として取得する例を示す．

例. 指数分布の確率密度関数 (2)（先の例の続き）

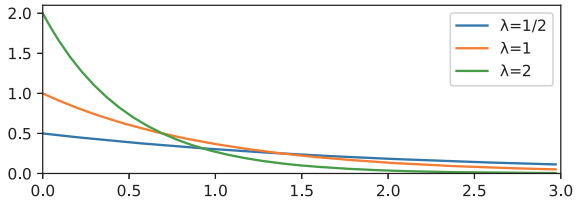
入力：
```
ax = np.arange( 0, 3, 0.03 )     # 0 以上 3 未満の数列を 0.03 間隔で生成
ay1 = stats.expon.pdf( ax, loc=0, scale=2 )       # λ=1/2
ay2 = stats.expon.pdf( ax, loc=0, scale=1 )       # λ=1/1=1
ay3 = stats.expon.pdf( ax, loc=0, scale=0.5 )     # λ=1/0.5=2
```

この例では定義域のデータ列を ax に作成し，それに対する値域を 3 種類の λ（0.5，1.0，2.0）毎に ay1，ay2，ay3 として取得している．それらを matplotlib によって可視化する例を次に示す．

例. 可視化処理（先の例の続き）

入力：
```
plt.figure( figsize=(6,2) )              # サイズを指定して描画作業の開始
plt.plot( ax, ay1, label='λ=1/2' )   # グラフのプロット 1
plt.plot( ax, ay2, label='λ=1' )     # グラフのプロット 2
plt.plot( ax, ay3, label='λ=2' )     # グラフのプロット 3
plt.xlim(0,3)                            # 描画範囲（横軸）の設定
plt.ylim(0,2.1)                          # 描画範囲（縦軸）の設定
plt.legend()                             # 凡例の表示
plt.show()                               # 描画の実行
```

出力：



6.2.1.5 対数正規分布

対数正規分布に関する関数群は scipy.stats.lognorm に含まれ，対数正規分布の確率密度関数 pdf を呼び出す場合は scipy.stats.lognorm.pdf と記述する．（別名の接頭辞により記述を省略できる）

書き方： **scipy.stats.lognorm.pdf(定義域の値, σ, loc=オフセット, scale=比率)**

「定義域の値」には 1 つの数値（スカラー）あるいはデータ列を与えることができる．

対数正規分布の確率密度関数 $f(x)$ の定義は，

$$f(x) = \frac{1}{\sqrt{2\pi}\,\sigma\,x} \exp\left(-\frac{(\log x - \mu)^2}{2\sigma^2}\right) \qquad (x > 0)$$

であるが，SciPy の lognorm.pdf 関数の定義はこれをもう少し簡略化したものであり，次のような定義となっている．

$$lognorm.pdf(x, s) = \frac{1}{s \cdot x \cdot \sqrt{2\pi}} \cdot \exp\left\{-\frac{1}{2} \cdot \left(\frac{\log(x)}{s}\right)^2\right\}$$

更に loc=**オフセット**，scale=**比率** は，処理結果を次のように変換する．

$$x_2 = \frac{x - loc}{scale} \quad \rightarrow \quad \text{計算結果：} \frac{lognorm.pdf(x_2, s)}{scale}$$

例. 対数正規分布の確率密度関数 (1)（先の例の続き）

入力：
```
y = stats.lognorm.pdf( 0.5, 1.0, loc=0, scale=1 )
print( y )
type( y ) # データ型の調査
```

出力：
```
  0.6274960771159245
numpy.float64
```

結果が NumPy の数値として得られている．次に定義域の値をデータ列として与えて，それに対する値域をデータ列として取得する例を示す．

例. 対数正規分布の確率密度関数 (2)（先の例の続き）

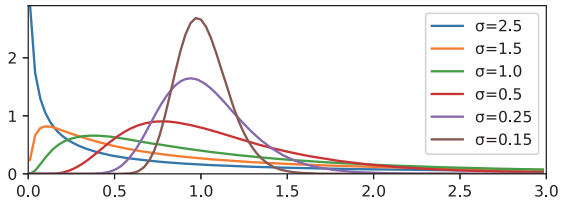
入力：
```
ax = np.arange( 0.01, 3, 0.03 )     # 0.01 以上 3 未満の数列を 0.03 間隔で生成
ay1 = stats.lognorm.pdf( ax, 2.5, loc=0, scale=1 )  # σ=2.5
ay2 = stats.lognorm.pdf( ax, 1.5, loc=0, scale=1 )  # σ=1.5
ay3 = stats.lognorm.pdf( ax, 1.0, loc=0, scale=1 )  # σ=1.0
ay4 = stats.lognorm.pdf( ax, 0.5, loc=0, scale=1 )  # σ=0.5
ay5 = stats.lognorm.pdf( ax, 0.25, loc=0, scale=1 )  # σ=0.25
ay6 = stats.lognorm.pdf( ax, 0.15, loc=0, scale=1 )  # σ=0.15
```

この例では定義域のデータ列を ax に作成し，それに対する値域を 6 種類の σ（2.5，1.5，1.0, 0.5, 0.25, 0.15）毎に ay1，ay2，…，ay6 として取得している．それらを matplotlib によって可視化する例を次に示す．

例. 可視化処理（先の例の続き）

入力：
```
plt.figure( figsize=(6,2) )            # サイズを指定して描画作業の開始
plt.plot( ax, ay1, label=' σ=2.5' )  # グラフのプロット 1
plt.plot( ax, ay2, label=' σ=1.5' )  # グラフのプロット 2
plt.plot( ax, ay3, label=' σ=1.0' )  # グラフのプロット 3
plt.plot( ax, ay4, label=' σ=0.5' )  # グラフのプロット 3
plt.plot( ax, ay5, label=' σ=0.25' ) # グラフのプロット 3
plt.plot( ax, ay6, label=' σ=0.15' ) # グラフのプロット 3
plt.xlim(0,3)                          # 描画範囲（横軸）の設定
plt.ylim(0,2.9)                        # 描画範囲（縦軸）の設定
plt.legend()                           # 凡例の表示
plt.show()                             # 描画の実行
```

出力：



6.2.1.6　三角分布

三角分布は，現実の現象から自然に導かれる分布ではなく，仮定を明示するために用いられる便宜的なモデルである．scipy.stats が次のような関数として提供している．

書き方：　scipy.stats.triang.pdf(定義域の値, c=最頻値の相対位置, loc=オフセット, scale=底辺の幅)

「定義域の値」には 1 つの数値（スカラー）あるいはデータ列を与えることができる．この場合の「最頻値の相対位置」とは，「オフセット」からの相対位置であり，「底辺の幅」に対する比率（0.0〜1.0）で与える．引数と関数の概形との関係を図 16 に示す．

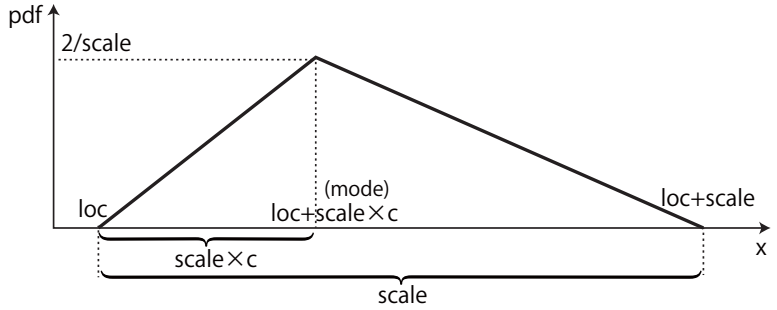


図 16: 三角分布

例. 三角分布の確率密度関数 (1)（先の例の続き）

入力：
```
y = stats.triang.pdf( 0.25, c=0.5, loc=0.0, scale=1 )
print( y )
type( y )     # データ型の調査
```

出力：
```
1.0
numpy.float64
```

結果が NumPy の数値として得られている．次に定義域の値をデータ列として与えて，それに対する値域をデータ列として取得する例を示す．

例. 三角分布の確率密度関数 (2)（先の例の続き）

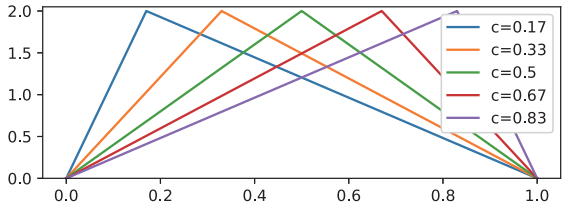
入力：
```
ax = np.linspace( 0.0, 1.0, 101 )       # 0.0 以上 1 以下の数列を 101 個生成
ay1 = stats.triang.pdf( ax, c=0.17, loc=0, scale=1 )  # c=0.17
ay2 = stats.triang.pdf( ax, c=0.33, loc=0, scale=1 )  # c=0.33
ay3 = stats.triang.pdf( ax, c=0.5,  loc=0, scale=1 )  # c=0.5
ay4 = stats.triang.pdf( ax, c=0.67, loc=0, scale=1 )  # c=0.67
ay5 = stats.triang.pdf( ax, c=0.83, loc=0, scale=1 )  # c=0.83
```

この例では定義域のデータ列を ax に作成し，それに対する値域を 5 種類の c の値（0.17, 0.33, 0.5, 0.67, 0.83）毎に ay1，ay2，…，ay5 として取得している．それらを matplotlib によって可視化する例を次に示す．

例. 可視化処理（先の例の続き）

入力：
```
plt.figure( figsize=(6,2) )          # サイズを指定して描画作業の開始
plt.plot( ax, ay1, label='c=0.17' ) # グラフのプロット 2
plt.plot( ax, ay2, label='c=0.33' ) # グラフのプロット 3
plt.plot( ax, ay3, label='c=0.5' )  # グラフのプロット 4
plt.plot( ax, ay4, label='c=0.67' ) # グラフのプロット 5
plt.plot( ax, ay5, label='c=0.83' ) # グラフのプロット 6
plt.xlim(-0.05,1.05)                 # 描画範囲（横軸）の設定
plt.ylim(0,2.05)                     # 描画範囲（縦軸）の設定
plt.legend()                         # 凡例の表示
```

出力：



6.2.2 確率質量関数：PMF（Probability Mass Function）

各種の確率質量関数の使用方法について説明する．SciPy では確率質量関数は ‘pmf’ である．

6.2.2.1　二項分布

二項分布に関する関数群は scipy.stats.binom に含まれ，二項分布の確率質量関数 pmf を呼び出す場合は scipy.stats.binom.pmf と記述する．(別名の接頭辞により記述を省略できる)　二項分布 $B_i(n,p)$ の値は次のようにして算出する．

書き方：　scipy.stats.binom.pmf(定義域の値, n, p)

「定義域の値」には 1 つの数値（スカラー）あるいはデータ列を与えることができる．

例. 二項分布の確率質量関数 (1)（先の例の続き）

入力：
```
y = stats.binom.pmf(5,10,0.5)  # Bi(10,0.5) の 5 に対する値
print( y )
type( y ) # データ型の調査
```

出力：
```
0.24609375000000025
numpy.float64
```

結果が NumPy の数値として得られている．次に定義域の値をデータ列として与えて，それに対する値域をデータ列として取得する例を示す．

例. 二項分布の確率質量関数 (2)（先の例の続き）

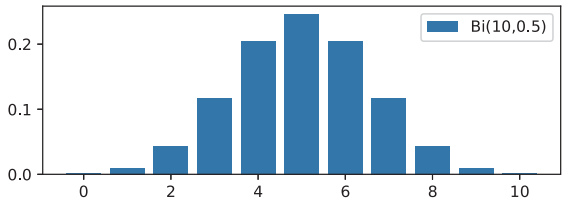
入力：
```
ax = np.arange(0,11)                # 0〜10 の数列
ay1 = stats.binom.pmf(ax,10,0.5)    # Bi(10,0.5) の 0〜10 までの値
```

この例では定義域のデータ列を ax に作成し，それに対する値域を ay1 として取得している．それらを matplotlib によって可視化する例を次に示す．

例. 可視化処理（先の例の続き）

入力：
```
plt.figure( figsize=(6,2) )              # サイズを指定して描画作業の開始
plt.bar( ax, ay1, label='Bi(10,0.5)' )   # グラフのプロット
plt.legend()                             # 凡例の表示
plt.show()                               # 描画の実行
```

出力：



6.2.2.2　幾何分布

幾何分布に関する関数群は scipy.stats.geom に含まれ，幾何分布の確率質量関数 pmf を呼び出す場合は scipy.stats.geom.pmf と記述する．(別名の接頭辞により記述を省略できる)　確率 p の幾何分布の値は次のようにして算出する．

書き方：　scipy.stats.geom.pmf(定義域の値, p)

「定義域の値」には 1 つの数値（スカラー）あるいはデータ列を与えることができる．

例. 幾何分布の確率質量関数 (1)（先の例の続き）

入力：
```
y = stats.geom.pmf(1,0.5)    # p=0.5
print( y )
type( y )    # データ型の調査
```

出力：
```
0.5
numpy.float64
```

結果が NumPy の数値として得られている．次に定義域の値をデータ列として与えて，それに対する値域をデータ列として取得する例を示す．

例. 幾何分布の確率質量関数 (2)（先の例の続き）

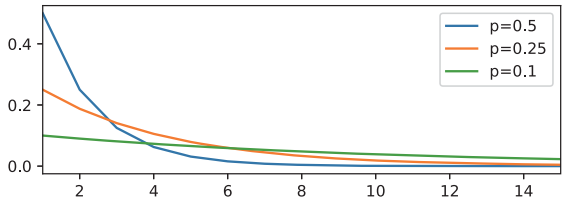
入力：
```
ax = np.arange(1,16)            # 1〜15 の数列
ay1 = stats.geom.pmf(ax,0.5)    # p=0.5
ay2 = stats.geom.pmf(ax,0.25)   # p=0.25
ay3 = stats.geom.pmf(ax,0.1)    # p=0.1
```

この例では定義域のデータ列を ax に作成し，3 つの p（0.5, 0.25, 0.1）に対して値域の列をそれぞれ ay1, ay2, ay3 として取得している．それらを matplotlib によって可視化する例を次に示す．

例. 可視化処理（先の例の続き）

入力：
```
plt.figure( figsize=(6,2) )          # サイズを指定して描画作業の開始
plt.plot( ax, ay1, label='p=0.5' )   # グラフのプロット 1
plt.plot( ax, ay2, label='p=0.25' ) # グラフのプロット 2
plt.plot( ax, ay3, label='p=0.1' )   # グラフのプロット 3
plt.xlim(1,15)                       # プロット範囲の設定
plt.legend()                         # 凡例の表示
plt.show()                           # 描画の実行
```

出力：



6.2.2.3　超幾何分布

超幾何分布に関する関数群は scipy.stats.hypergeom に含まれ，超幾何分布の確率質量関数 pmf を呼び出す場合は scipy.stats.hypergeom.pmf と記述する．（別名の接頭辞により記述を省略できる）

超幾何分布は，母集団の大きさを N，母集団中の成功数を K，そこから無作為に n 個を抽出したときに，成功とみなされる要素が k 個含まれる確率を与える離散分布である。一般的には次のように表される．

$$f_X(k;N,K,n) = \frac{\binom{K}{k}\binom{N-K}{n-k}}{\binom{N}{n}}$$

超幾何分布の値は次のようにして算出する．

書き方：　**scipy.stats.hypergeom.pmf(** k, M, K, n **)**

k には 1 つの数値（スカラー）あるいはデータ列を与えることができる．

注意） scipy.stats.hypergeom.pmf の第 2 引数 M は，統計学で一般に用いられる母集団サイズ N に対応する点に注意すること．

例. 超幾何分布の確率質量関数 (1)（先の例の続き）

入力：
```
y = stats.hypergeom.pmf(20,100,40,50)     # fx(20;100,40,50)
print( y )
type( y )     # データ型の調査
```

出力：
```
0.16158335607970392
numpy.float64
```

結果が NumPy の数値として得られている．次に定義域の値をデータ列として与えて，それに対する値域をデータ列として取得する例を示す．

例. 超幾何分布の確率質量関数 (2)（先の例の続き）

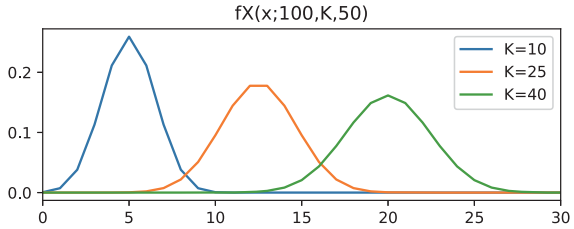
入力：
```
ax = np.arange(0,101)              # 0〜100 の数列
ay1 = stats.hypergeom.pmf(ax,100,10,50)     # fx(x;100,10,50)
ay2 = stats.hypergeom.pmf(ax,100,25,50)     # fx(x;100,25,50)
ay3 = stats.hypergeom.pmf(ax,100,40,50)     # fx(x;100,40,50)
```

この例では $f_X(x;100,K,50)$ の超幾何分布のデータを作成している．定義域のデータ列を ax に作成し，3 つの K（10, 25, 40）に対して値域をそれぞれ ay1, ay2, ay3 として取得している．それらを matplotlib によって可視化する例を次に示す．

例. 可視化処理（先の例の続き）

入力：
```
plt.figure( figsize=(6,2) )          # サイズを指定して描画作業の開始
plt.plot( ax, ay1, label='K=10' )    # グラフのプロット 1
plt.plot( ax, ay2, label='K=25' )    # グラフのプロット 2
plt.plot( ax, ay3, label='K=40' )    # グラフのプロット 3
plt.xlim(0,30)                        # プロット範囲の設定
plt.title('fX(x;100,K,50)')          # タイトル
plt.xlabel('x')                       # 横軸ラベル
plt.legend()                         # 凡例の表示
plt.show()                           # 描画の実行
```

出力：



6.2.2.4　ポアソン分布

ポアソン分布に関する関数群は scipy.stats.poisson に含まれ，ポアソン分布の確率質量関数 pmf を呼び出す場合は scipy.stats.poisson.pmf と記述する．（別名の接頭辞により記述を省略できる）　ポアソン分布の値は次のようにして算出する．

書き方：　scipy.stats.poisson.pmf(定義域の値, λ)

定義域の値には 1 つの数値（スカラー）あるいはデータ列を与えることができる．

例. ポアソン分布の確率質量関数 (1)（先の例の続き）

入力：
```
y = stats.poisson.pmf(10,10)     # λ=10
print( y )
type( y )     # データ型の調査
```

出力：
```
0.12511003572113372
numpy.float64
```

結果が NumPy の数値として得られている．次に定義域の値をデータ列として与えて，それに対する値域をデータ列として取得する例を示す．

例. ポアソン分布の確率質量関数 (2)（先の例の続き）

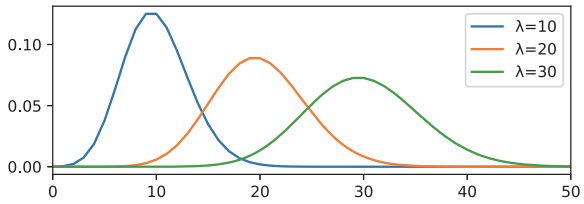
入力：
```
ax = np.arange(0,101)              # 0〜100 の数列
ay1 = stats.poisson.pmf(ax,10)   # λ=10
ay2 = stats.poisson.pmf(ax,20)   # λ=20
ay3 = stats.poisson.pmf(ax,30)   # λ=30
```

この例ではポアソン分布のデータを作成している．定義域のデータ列を ax に作成し，3 つの λ（10, 20, 30）に対して値域をそれぞれ ay1, ay2, ay3 として取得している．それらを matplotlib によって可視化する例を次に示す．

例. 可視化処理（先の例の続き）

入力：
```
plt.figure( figsize=(6,2) )          # サイズを指定して描画作業の開始
plt.plot( ax, ay1, label='λ=10' )    # グラフのプロット 1
plt.plot( ax, ay2, label='λ=20' )    # グラフのプロット 2
plt.plot( ax, ay3, label='λ=30' )    # グラフのプロット 3
plt.xlim(0,50)                       # プロット範囲の設定
plt.legend()                         # 凡例の表示
plt.show()                           # 描画の実行
```

出力：



6.2.3 累積分布関数：CDF（Cumulative Distribution Function）

確率密度関数 $f(x)$ を次のように積分した関数を**累積分布関数**（図 17）という．

$$cdf(x) = \int_{-\infty}^{x} f(u)\,du$$

累積分布関数は次のような性質を持つ．

$$\lim_{x\to\infty} cdf(x) = 1$$

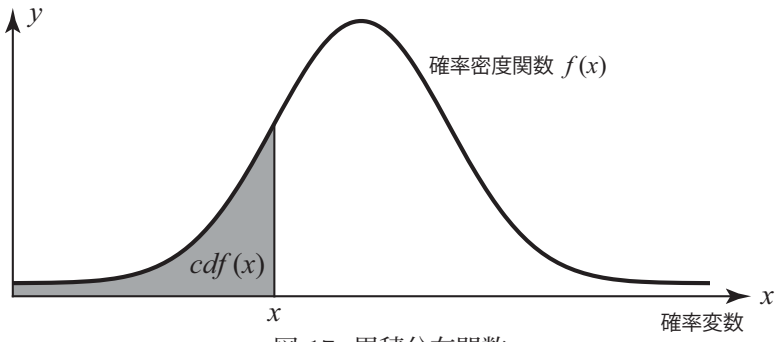


図 17: 累積分布関数

確率質量関数 $P(x)$ に関しても累積分布関数が定義できる．x の最小値を x_{min}，最大値を x_{max} とすると，

$$cdf(x) = \sum_{x_i=x_{min}}^{x} P(x_i)$$

となる．また，$cdf(x_{max}) = 1$ である．

SciPy では各種の分布関数毎に CDF があり，関数の名前は cdf である．ここでは 1 つの例として，正規分布の CDF（stats.norm.cdf）の使用方法を示す．

例. stats.norm.cdf の使用例 (1)（先の例の続き）

入力：
```
y = stats.norm.cdf( 0, loc=0, scale=1 ) # N(0,1) の x=0 における CDF
print( y )
type( y )     # データ型の調査
```

出力：
```
0.5
numpy.float64
```

結果が NumPy の数値として得られている．次に定義域の値をデータ列として与えて，それに対する値域をデータ列として取得する例を示す．

例. stats.norm.cdf の使用例 (2)（先の例の続き）

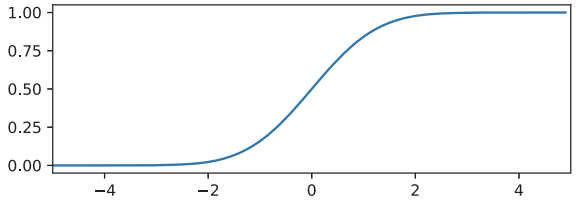
入力：
```
ax = np.arange(-5,5,0.1)                    # -5 以上 5 未満の数列を 0.1 間隔で生成
ay1 = stats.norm.cdf(ax,loc=0,scale=1)   # 上記に対する CDF
```

この例では定義域のデータ列を ax に作成し，それに対する CDF の値域の列を ay1 として取得している．それを matplotlib によって可視化する例を次に示す．

例. 可視化処理（先の例の続き）

入力：
```
plt.figure( figsize=(6,2) )     # サイズを指定して描画作業の開始
plt.plot( ax, ay1 )      # グラフのプロット
plt.xlim(-5,5)           # 描画範囲（横軸）の設定
plt.show()               # 描画の実行
```

出力：



6.2.4 パーセント点関数：PPF（Percent Point Function）

累積分布関数の逆関数として**パーセント点関数**（図 18）がある．この関数は「累積確率が q となる確率変数 x の値」を求める場合に利用する．

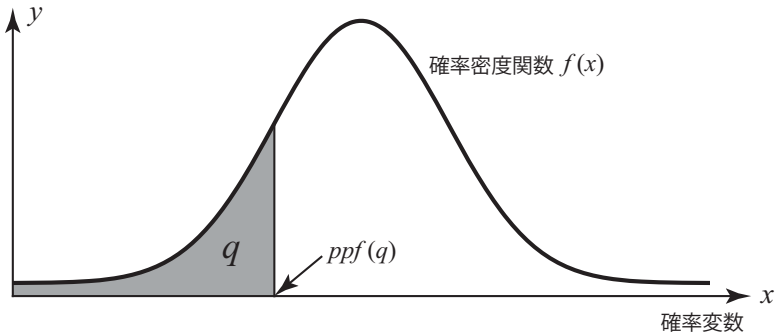


図 18: パーセント点関数

SciPy では各種の分布関数毎に PPF があり，関数の名前は ppf である．ここでは 1 つの例として，正規分布の PPF（stats.norm.ppf）の使用方法を示す．

例. stats.norm.ppf の使用例 (1)（先の例の続き）

入力：
```
y = stats.norm.ppf( 0.5, loc=0, scale=1 )  # 累積確率が 0.5 のパーセント点
print( y )
type( y ) # データ型の調査
```

出力：
```
 0.0
numpy.float64
```

結果が NumPy の数値として得られている．次に累積確率の値をデータ列として与えて，それに対するパーセント点をデータ列として取得する例を示す．

例. stats.norm.ppf の使用例 (2)（先の例の続き）

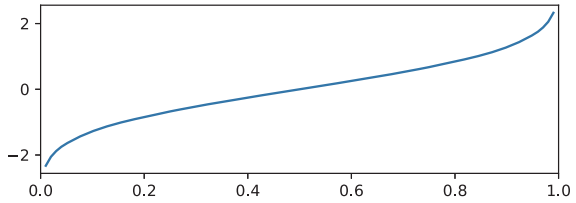
入力：
```
ax = np.arange(0.01,1,0.01)                 # 0.01 以上 1 未満の数列を 0.01 間隔で生成
ay1 = stats.norm.ppf(ax,loc=0,scale=1)   # 上記に対する PPF
```

この例では累積確率のデータ列を ax に作成し，それに対する PPF の列を ay1 として取得している．それを matplotlib によって可視化する例を次に示す．

例. 可視化処理（先の例の続き）

入力：
```
plt.figure( figsize=(6,2) ) # サイズを指定して描画作業の開始
plt.plot( ax, ay1 )         # グラフのプロット
plt.xlim(0,1)               # 描画範囲（横軸）の設定
plt.show()                  # 描画の実行
```

出力：



6.2.5 乱数生成：RVS（Random Variates）

SciPy は各種の分布関数に沿った形の乱数を生成する機能を提供する．乱数を生成するメソッドの名前は rvs である．ここでは 1 つの例として，対数正規分布の RVS（stats.lognorm.rvs）の使用方法を示す．

例. stats.lognorm.rvs の使用例（先の例の続き）

入力：
```
# 対数正規分布（σ=1, μ=0）に沿った乱数を 10,000 個生成
r = stats.lognorm.rvs(1,size=10000)
type( r )     # データ型の調査
```

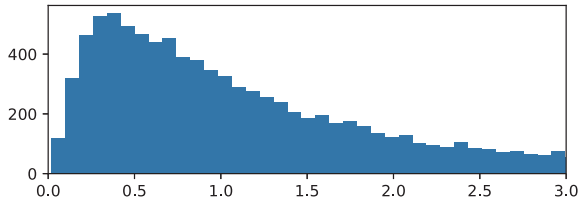
出力：
```
numpy.ndarray
```

結果が NumPy の配列として得られている．これを matplotlib によって可視化する例（ヒストグラム）を次に示す．

例. 可視化処理（先の例の続き）

入力：
```
plt.figure( figsize=(6,2) ) # サイズを指定して描画作業の開始
plt.hist(r,bins=500)      # ヒストグラムの作成
plt.xlim(0,3)             # 描画範囲（横軸）の設定
plt.show()                # 描画の実行
```

出力：



6.2.5.1 一様乱数の生成

uniform.rvs メソッドは一様乱数を生成する．

書き方：　uniform.rvs(size=個数)

このメソッドは 0 以上 1 未満の乱数を生成する．「個数」には生成する乱数の個数を与える．

例. uniform.rvs メソッドによる一様乱数の生成：その 1（先の例の続き）

入力：
```
# 一様乱数の生成 (1)
r = stats.uniform.rvs(size=10000)  # 0 以上 1 未満の一様乱数を 1 万個生成
print( 'len:', len(r) )      # 長さ
print( 'min:', r.min() )     # 最小値
print( 'max:', r.max() )     # 最大値
```

出力：
```
len:  10000
min:  0.0003431206390183128
max:  0.9998096595271029
```

生成する乱数の下限と変動幅をキーワード引数 'loc=**下限**'，'scale=**変動幅**'，に指定することができる．（「下限」以上「下限」＋「変動幅」未満の乱数生成）

例. 下限と変動幅を指定した一様乱数の生成（先の例の続き）

入力：
```
# 一様乱数の生成 (2)
r = stats.uniform.rvs(loc=-1,scale=2,size=10000)  # -1 以上 1 未満の一様乱数を 1 万個生成
print( 'len:', len(r) )      # 長さ
print( 'min:', r.min() )     # 最小値
print( 'max:', r.max() )     # 最大値
```

出力：
```
len:  10000
min:  -0.9997483995888579
max:  0.9999336790442426
```

6.2.5.2 乱数生成の初期設定：random_state

scipy.stats が提供する乱数生成用の関数は，基本的に確定的な過程で乱数を生成する．このことは，紙面に書かれた乱数表を引用する作業に似ている．当然のことであるが，乱数表の同じ位置から引用を開始すると，決まった（同じ）乱数列が得られる．このことについて例を挙げて説明する．

次の例は，初期状態を指定せずに乱数生成を行うものでる．

例. 通常の乱数生成（先の例の続き）

入力：
```
# 一様乱数の生成（連続実行：初期状態の指定なし）
print( stats.uniform.rvs(size=6) )
print( stats.uniform.rvs(size=6) )
print( stats.uniform.rvs(size=6) )
```

出力：
```
[0.89249932 0.22325052 0.41200067 0.28257124 0.81956365 0.68454985]
[0.01294233 0.30247601 0.62241121 0.86658365 0.94314688 0.81336499]
[0.00474479 0.63538474 0.51469667 0.2309236 0.13998039 0.14201854]
```

6 個の乱数列を立て続けに 3 回生成している．当然であるが各回で生成される乱数列は互いに異なるものになっている．次に，毎回同じ初期状態を指定して乱数列を生成する例を示す．

例. 同じ初期状態による乱数列の生成（先の例の続き）

入力：
```
# 一様乱数の生成（連続実行：初期状態を指定）
print( stats.uniform.rvs(size=6,random_state=0) )
print( stats.uniform.rvs(size=6,random_state=0) )
print( stats.uniform.rvs(size=6,random_state=0) )
```

出力：
```
[0.5488135 0.71518937 0.60276338 0.54488318 0.4236548 0.64589411]
[0.5488135 0.71518937 0.60276338 0.54488318 0.4236548 0.64589411]
[0.5488135 0.71518937 0.60276338 0.54488318 0.4236548 0.64589411]
```

この例のようにキーワード引数 'random_state=**種**' を与えることで，乱数生成の初期状態（**種**：seed）を指定することができる．同じ seed からは同じ乱数列が生成される．

統計処理や機械学習のためのプログラムを開発する際には，乱数を元にしたテストデータを作成して使用することが多い．そのような場合には，この例で示したような方法で確定的なテストデータを生成して，プログラムの動作に再現性を持たせることができる．

注意）暗号学的な処理を行う際は，確定的な乱数を使用してはならない．

■ Generator による乱数生成

新しい版の NumPy からは Generator API が導入され[68]，生成する乱数の品質が向上している．Generator API では**乱数生成器**（RNG）である **Generator オブジェクト**を作成し，それを用いて乱数を生成する．(次の例)

[68] NumPy 1.17 から導入され，2.0 からは乱数生成にこれを使用することが推奨されている．

例. RNG の作成と乱数生成（先の例の続き）

入力：
```
rng = np.random.default_rng(1)     # 種「1」を指定して RNG を作成
print( rng )                       # 確認
r = rng.random( 5 )  # RNG を用いて一様乱数を 5 個生成
print( r )           # 表示
```

出力：
```
Generator(PCG64)          ← Generator オブジェクト
[0.51182162 0.9504637 0.14415961 0.94864945 0.31183145]     ←生成された 5 個の乱数
```

この例のように，RNG（Generator オブジェクト）に対して乱数生成用のメソッド（この例では random）を実行する形で乱数を作る．

この例では「np.random.default_rng(1)」として RNG を作成しているが，これは PCG64 というアルゴリズムによって乱数を生成する（種は「1」）ものである．NumPy はこれ以外の乱数生成アルゴリズムも提供しているが，詳細に関しては NumPy の公式インターネットサイトを始めとする他の文献[69] を参照のこと．

上の例では，種として「1」を与えており，これによって，生成される乱数に再現性が得られる．（次の例）

例. 同じ種で再度実行して再現性を確認（先の例の続き）

入力：
```
rng = np.random.default_rng(1)
print( rng.random(5) )
```

出力：
```
[0.51182162 0.9504637 0.14415961 0.94864945 0.31183145]     ←先の例と同じ乱数系列
```

上の例で示した random メソッドは NumPy が提供するものであるが，SciPy の乱数生成用 API に NumPy の Generator オブジェクトを使用することができる[70] ．具体的には，rvs 関数の random_state 引数に RNG を与える．（次の例）

例. SciPy の乱数生成 API で RNG を使用する（先の例の続き）

入力：
```
rng = np.random.default_rng(3)  # RNG の作成
print( stats.uniform.rvs(size=5,random_state=rng) )     # 乱数生成と表示
rng = np.random.default_rng(3)  # 再度同じ種で RNG を作成
print( stats.uniform.rvs(size=5,random_state=rng) )     # 同じ乱数が生成される
```

出力：
```
[0.08564917 0.23681051 0.80127447 0.58216204 0.09412864]
[0.08564917 0.23681051 0.80127447 0.58216204 0.09412864]     ←乱数生成の再現性が確認できる
```

6.2.6 確率変数オブジェクトによる効率的な処理の方法

先に解説した各種の関数（pdf，pmf，cdf，ppf，rvs など）は，ある**確率分布**（確率密度，確率質量）に基づいて処理結果を求めるものである．これまでの解説に従い，例えば正規分布に基づく各種の関数は次のようにして求める．

例. 正規分布に基づく各種の関数（先の例の続き）

入力：
```
print( stats.norm.pdf(      4, loc=4, scale=2 ) )     # PDF：確率密度関数
print( stats.norm.cdf(      4, loc=4, scale=2 ) )     # CDF：累積分布関数
print( stats.norm.ppf(    0.5, loc=4, scale=2 ) )     # PPF：パーセント点関数
print( stats.norm.rvs( size=5, loc=4, scale=2 ) )     # RVS：乱数生成関数
```

出力：
```
0.19947114020071635
0.5
4.0
[5.30821988 3.18032601 1.76689272 3.35107792 4.34685275]     ←乱数
```

この例では，各種の関数を実行する度に同じパラメータ「loc=4，scale=2」を指定している．このように，同一のパラメータの確率分布に基づいて各種の関数の値を求める場合は，**確率変数オブジェクト**[71] （**rv オブジェクト**）を先に作成しておき，それに対して各種メソッドを実行するという形式を取るのが良い．（次の例）

[69] 拙書「Python3 ライブラリブック - 各種ライブラリの基本的な使用方法」でも解説しています．

[70] SciPy 1.7 以降，random_state 引数が NumPy の Generator オブジェクトを受理する機能が正式にサポートされた．

[71] **frozen 分布オブジェクト**と呼ぶこともある．

例. rv オブジェクトの作成（先の例の続き）

入力：
```
rv = stats.norm(loc=4,scale=2)     # rv オブジェクトを作成
print( type(rv) )     # 型の確認
```

出力：
```
<class 'scipy.stats._distn_infrastructure.rv_continuous_frozen'>     ←このような型
```

例. rv オブジェクトに対するメソッドによる処理（先の例の続き）

入力：
```
print( rv.pdf( 4 ) )          # PDF：確率密度関数
print( rv.cdf( 4 ) )          # CDF：累積分布関数
print( rv.ppf( 0.5 ) )        # PPF：パーセント点関数
print( rv.rvs( size=5 ) )     # RVS：乱数生成関数
```

出力：
```
0.19947114020071635
0.5
4.0
[ 2.8854176 4.57750975 5.26980769 2.09623916 -0.11231771]     ←乱数（先の例とは異なる）
```

この形式の方が，先の例よりも記述が簡潔になるだけでなく，分布のパラメータ設定が一度で済むので処理の効率も高くなる．

同様のことが，離散的確率変数の場合（確率質量関数の場合）にも言える．例えば，二項分布に関する各種の処理を次のように記述することができる．

例. 二項分布に基づく各種の関数（先の例の続き）

入力：
```
print( stats.binom.pmf(      5, n=10, p=0.5 ) )     # PMF：確率質量関数
print( stats.binom.cdf(      5, n=10, p=0.5 ) )     # CDF：累積分布関数
print( stats.binom.ppf(  0.623, n=10, p=0.5 ) )     # PPF：パーセント点関数
print( stats.binom.rvs( size=5, n=10, p=0.5 ) )     # RVS：乱数生成関数
```

出力：
```
0.24609375
0.623046875
5.0
[5 7 6 5 4]          ←乱数
```

これと同等の処理を，rv オブジェクトを用いて実行する例を次に示す．

例. rv オブジェクトによる処理（先の例の続き）

入力：
```
rv = stats.binom(n=10,p=0.5)   # rv オブジェクトを作成
print( type(rv) )              # 型の確認
print( rv.pmf( 5 ) )           # PMF：確率質量関数
print( rv.cdf( 5 ) )           # CDF：累積分布関数
print( rv.ppf( 0.623 ) )       # PPF：パーセント点関数
print( rv.rvs( size=5 ) )      # RVS：乱数生成関数
```

出力：
```
<class 'scipy.stats._distn_infrastructure.rv_discrete_frozen'> ←この場合の rv オブジェクトの型
0.24609375
0.623046875
5.0
[4 7 5 3 5]      ←乱数（先の例とは異なる）
```

乱数生成以外の処理は，同じ結果が得られていることがわかる．

7　Apache 製データ処理基盤との連携

Apache Software Foundation [72]（ASF）は，膨大なデータ[73] を効率的に保存，管理，処理するためのソフトウェアツール群を開発して公開している．それらソフトウェアの目的は，1 台のコンピュータでは処理しきれない問題を，たくさんのコンピュータに分担させて解決（分散処理）することにある．

従来のデータベースシステムでは，データ量が著しく大きく（数 TB〜）なったり，処理の方法が複雑になると性能が大きく低下する．ASF のソフトウェア群は，この問題を解決するために開発された．具体的には，以下の 2 つの役割を専門のツールに分担させる．

- **データの保存：ストレージ**
  データの分散保存のための機能（Hadoop，HDFS）や，処理を容易にするためのデータフォーマット（Arrow，Parquet）の取り扱いに関する機能．その他の高機能なストレージのための機能（Hive，Kafka）．
- **データの処理**
  保存されたデータを効率よく読み込み，並列的に計算処理を行う機能（Spark）．データのストリーミングやリアルタイム処理のための機能（Flink）．

本書では Arrow，Parquet の基本的な事柄について取り上げ，pandas ライブラリでそれらを使用するための方法について解説する．

7.1　Apache Arrow

pandas の DataFrame は，複数のカラム（列）を束ねたデータ構造であり，このような形式は多くのデータ処理ソフトウェアに共通するものである．しかし各種ソフトウェアは，仮想記憶上でそのようなデータ構造を独自のデータ型として実現しており，ソフトウェア間でのデータ交換には変換処理が必要となる．この労力を省き，仮想記憶上での取り扱いを効率的にするために Apache Arrow が開発された．

Apache Arrow（以後，Arrow と略す）は複数のカラム（列）を束ねたデータ構造を仮想記憶上で効率的に表現するものであり，多くのソフトウェア（Python，GNU R，Apache Spark）がこの形式をサポートしている．Arrow は，後述の Parquet を中心とした大規模データ処理の高速化を強く意識した設計を持つ一方で，CSV，JSON，Feather など多数のファイル形式との入出力をサポートし，データ基盤としての汎用性も高い．

Python で Arrow を使用するには pyarrow ライブラリを用いる．このライブラリは Python の標準ライブラリではなく[74]，使用に際しては，別途インストールする必要がある．

例. pip コマンドによる pyarrow のインストール（OS のコマンド処理）

```
pip install pyarrow              ←標準的なインストール
py -m pip install pyarrow        ← Windows 用の PSF 版 Python の場合
python -m pip install pyarrow    ← macOS，Linux のシェルで実行する場合
```

7.1.1　基本的なデータ構造

Arrow が提供する基本的なデータ構造は Array 型（とその派生型），Table 型である．Array は 1 次元のデータの並びであり，それらをカラム（列）として複数束ねたものが Table である．また，構造を持たない単一の値は Scalar 型とその派生型である．

Array オブジェクトが要素として保持できるものとして，Scalar 型の値の他に，ListArray ，FixedSizeListArray，StructArray ，MapArray といった複雑なオブジェクトもある．ただし，1 つの Array オブジェクトが保持する要素は全て同じ型でなければならない．

Arrow が提供するデータ構造は，仮想記憶上でデータセットを効率良く表現するためのもの（低レベルのデータ構造）であり，利用者が直接的にデータ処理に用いるものではない．すなわち，pandas の Series や DataFrame に内部表現として Arrow のオブジェクトを与え，利用者は pandas の API を使用して実際のデータ処理を行うことになる．

Arrow のデータ構造に関する基礎的理解を促すための実行例を以下に示す．

[72] https://www.apache.org/

[73] いわゆる**ビッグデータ**

[74] https://pypi.org/project/pyarrow/

7.1.1.1 Array

Array オブジェクトを作成する例を示す．まずは，次のようにして必要なライブラリを読み込む．

例. Arrow ライブラリの読み込み

入力：
```
import pyarrow as pa
```

このように「pa」という短縮形のエイリアスを与えるのが一般的である．

次に Array オブジェクト作成の例を示す．

例. 整数要素の Array の作成（先の例の続き）

入力：
```
ar1 = pa.array([1,2,3])     # 整数要素の Array
print( type(ar1) )          # 型を調べる
print( ar1 )                # 確認のための表示
```

出力：
```
<class 'pyarrow.lib.Int64Array'>      ← ar1 オブジェクトの型
[
  1,              ←カラム指向なので
  2,              ←縦に並ぶ形で表示
  3               ←される
]
```

この例では Int64Array 型（Array 型の派生型）のオブジェクト ar1 が得られている．Array オブジェクトはリストの場合と同じように，インデックスを指定して要素を参照することができる．（次の例）

例. インデックス指定による要素の参照（先の例の続き）

入力：
```
ar1[0]     # インデックス 0 の要素（先頭要素）の参照
```

出力：
```
<pyarrow.Int64Scalar:  1>        ← Arrow 独特の型
```

この例では ar1 の先頭要素（インデックス 0 の要素）を参照しているが，Python 元来の int 型や NumPy のスカラー値ではなく，Int64Scalar 型（Scalar 型の派生型）の値として得られていることがわかる．また，スライスにインデックス範囲を指定する（例．ar1[0:2]）と，参照結果は元のオブジェクトと同じ型の Array オブジェクトとなる．

注意） Arrow 独自の Scalar 型の値は，Python や NumPy が提供する型の値とは異なり，基本的には，通常の処理（計算など）には使えない．（次の例）

例. Scalar に対して通常の計算を試みる（先の例の続き）

入力：
```
ar1[0] + 3     # Arrow 独自のデータ型であるため，計算処理できない
```

出力：
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[15], line 1
----> 1 ar1[0] + 3    #

TypeError:unsupported operand type(s) for +:'pyarrow.lib.Int64Scalar' and 'int'
```

Arrow のデータ構造は，仮想記憶上で効率的にデータセットを表現するためのものであり，データに対する実際の処理は，適切な形式（pandas の DataFrame など）に変換する必要がある．上の例では，Array の要素を「+」演算子で計算しようとした例であるが，Scalar 型の値はこれに対応していない[75] ため，エラーとなっている．もちろん，Arrow の Scalar 型の値も適切に変換することで演算処理に使用することができる．（次の例）

例. Scalar 値を演算可能な値に変換して使用する（先の例の続き）

入力：
```
ar1[0].as_py() + 3     # Python の型（int 型）に変換してから計算に使用する
```

出力：
```
4
```

この例では，as_py メソッドで Scalar 値を通常の int 型に変換[76] して演算を実行している．

Scalar 型を Python で使用できるデータ型に変換するには，それぞれの型のコンストラクタを使用することもできるが，as_py メソッドは型を適切に判断して変換するため，これの使用が推奨される．

[75] lazy representation（遅延表現）と表現されることもある．

[76] このように，Arrow 独自の型の値を通常の処理に使用できるように変換することを「materialize」と表現することがある．

Scalar には，浮動小数点数や文字列のための派生型もある．

例. 浮動小数点数の Array（先の例の続き）

入力：
```
arf = pa.array([1.0,2.0,3.0])
arf
```

出力：
```
<pyarrow.lib.DoubleArray object at 0x00000294ED4F5960>
[
  1,          ←小数点以下が表示されていないが
  2,          ←内部では浮動小数点数として
  3           ←保持されている
]
```

浮動小数点数の Array は DoubleArray クラスであり，その要素は DoubleScalar クラスの値である．

例. 文字列の Array（先の例の続き）

入力：
```
ars = pa.array(['apple','orange','grape'])
ars
```

出力：
```
<pyarrow.lib.StringArray object at 0x00000294ED9D8EE0>
[
  "apple",
  "orange",
  "grape"
]
```

文字列の Array は StringArray クラスであり，その要素は StringScalar クラスの値である．

7.1.1.2 Table

Array をカラム（列）としてそれらを複数束ね，表のような形式である Table オブジェクトを作ることができる．Table オブジェクトを作成する例を示す．

例. Table オブジェクトの作成（先の例の続き）

入力：
```
tbl = pa.table({
    'linear'  :  ar1,
    'square'  :  pa.array([1.0,4.0,9.0]),
    'cubic'   :  pa.array([1.0,8.0,27.0]),
    'fruits'  :  ars
})
tbl
```

出力：
```
pyarrow.Table                 ←型は Table
linear:  int64                 ←'linear' カラムは 64 ビット整数
square:  double                 ←'square' カラムは double 型（64 ビット浮動小数点数）
cubic:  double                 ←'cubic' カラムも同様
fruits:  string                 ←'fruits' カラムは文字列
----
linear:  [[1,2,3]]              ←それぞれの
square:  [[1,4,9]]              ←カラムの内容が
cubic:  [[1,8,27]]             ←↓表示されている
fruits:  [["apple","orange","grape"]]
```

この例では，'linear'，'square'，'cubic'，'fruits' の各キーに対する値として Array を持つ辞書オブジェクトから Table オブジェクト tbl を作成している．出力の '—-' の上側にデータ型に関する情報が，下側に各カラムの内容が表示されている．

各カラムの内容は二重の括弧 '[[…]]' で表示されているが，これは後に説明する ChunkedArray の形式である．各カラムを参照するには，Table オブジェクトにカラム名の文字列をスライスの形で指定する．（次の例）

例. スライス指定によるカラムの参照（先の例の続き）

入力：
```
tbl['square']     # square カラムの参照
```

出力：
```
<pyarrow.lib.ChunkedArray object at 0x000001A4782BD180>  ← ChunkedArray であるとの表示
[
  [    ←第 1 チャンクの Array オブジェクトの内容の開始
    1,
    4,
    9
  ]
]
```

7.1.1.3 ChunkedArray

仮想記憶上の連続した領域に確保された配列状のデータ構造は，そのサイズを拡大することができないことが多く，他のデータを連結して拡大する際は，データの完全な複製と連結処理が求められる．これは，規模の大きなデータ[77]を扱う際に問題（処理時間，記憶域の消費など）となる．

Arrow では，仮想記憶上で連続した配列構造である Array を複数，データの複製を行わず，メタデータの管理によって，連結した形の ChunkedArray とすることができ，これは 1 つの連続的なデータ並びとして扱うことができる．例えば先に示した例で得られた tbl において，'square' カラムの要素を次のようにして自然な形で参照できる．

例. ChunkedArray の要素への順次アクセス（先の例の続き）

入力：
```
for x in tbl['square']:  # 連続的に要素を順次参照するループ
    print(x,end=', ')
```
出力：
```
1.0, 4.0, 9.0,          ←順番に参照できている
```

複数の Array オブジェクトを chunked_array 関数で ChunkedArray にすることができる．(次の例)

例. 複数の Array から ChunkedArray を作る（先の例の続き）

入力：
```
ar1 = pa.array([1, 2])     # 第 1 チャンクの Array
ar2 = pa.array([3, 4])     # 第 2 チャンクの Array
ch = pa.chunked_array( [ar1, ar2] ) # それらをチャンクとして連結
ch     # 内容確認
```
出力：
```
<pyarrow.lib.ChunkedArray object at 0x000001A4782BD480>
[
  [          ←第 1 チャンクとしての ar1
    1,
    2
  ],
  [          ←第 2 チャンクとしての ar2
    3,
    4
  ]
]
```

得られた ch の要素に順次アクセスする例を次に示す．

例. 要素への順次アクセス（先の例の続き）

入力：
```
for x in ch:
    print(x,end=', ')
```
出力：
```
1, 2, 3, 4,           ←順番に参照できている
```

ChunkedArray オブジェクト内の，指定した位置（インデックス）のチャンク（Array オブジェクト）を参照するには chunk メソッドを使用する．(次の例)

例. チャンクの参照（先の例の続き）

入力：
```
ch.chunk(0)     # インデックス 0（先頭）のチャンクを参照
```
出力：
```
<pyarrow.lib.Int64Array object at 0x000001A4782BCBE0>
[            ← ar1 の内容が表示される
  1,
  2
]
```

注意）型の異なる Array オブジェクトを同一の ChunkedArray オブジェクトに含めることはできない．

[77] いわゆるビッグデータ．

注意）Arrow が提供するデータ構造は基本的には**イミュータブル**であり，作成後に内容を変更することはできない．

7.1.2 pandas のバックエンドに Arrow を使用する方法

pandas が提供する Series や DataFrame のバックエンドのデータ構造（内部のデータ表現）は，通常は NumPy の配列オブジェクトであるが，Arrow が提供する Array や Table を pandas のバックエンドに使用することもできる．これは特に，巨大なサイズのデータを，内容の変更が伴わない形（参照系の処理）で取り扱う際に，記憶資源の使用効率の面で有利になることがある．

注意）Arrow バックエンドの場合でも，DataFrame に対する各種処理のパフォーマンスが，通常の NumPy バックエンドの場合と比べて高くなるとは限らないという点[78] に留意すること．

ここでは，Arrow をバックエンドにして pandas の Series，DataFrame を作成する例を示す．まず，次のようにして Arrow，NumPy，pandas のライブラリ群を読み込む．

例. ライブラリの読み込み

入力：
```
import pyarrow as pa     # PyArrow の読み込み
import numpy as np       # NumPy の読み込み
import pandas as pd      # pandas の読み込み
```

7.1.2.1 Table ベースの DataFrame の作成

サンプルデータの Table オブジェクトを作成する．

例. サンプルデータの作成（先の例の続き）

入力：
```
# サンプルデータ（Table）の作成
a1 = np.arange(1000,dtype=np.float64)    # 0.0〜999.0
tbl = pa.table(
    'linear'  :  a1,
    'square'  :  a1**2,
    'cubic'   :  a1**3
)
tbl    # 内容確認
```

出力：
```
pyarrow.Table
linear:  double
square:  double
cubic:  double
----
linear:  [[0,1,2,3,4,...,995,996,997,998,999]]
square:  [[0,1,4,9,16,...,990025,992016,994009,996004,998001]]
cubic:  [[0,1,8,27,64,...,985074875,988047936,991026973,994011992,997002999]]
```

次に，to_pandas メソッドで，tbl をバックエンドに持つ DataFrame を作成する．これには to_pandas メソッドに引数「types_mapper=pd.ArrowDtype」を与えて実行する．

例. Table をバックエンドにした DataFrame を作成（先の例の続き）

入力：
```
dfArw = tbl.to_pandas(types_mapper=pd.ArrowDtype)
dfArw
```

[78] このことは後の「7.1.2.3 性能評価」（p.138）で示す．

出力：

	linear	square	cubic
0	0.0	0.0	0.0
1	1.0	1.0	1.0
2	2.0	4.0	8.0
3	3.0	9.0	27.0
4	4.0	16.0	64.0
...	...	...	...
995	995.0	990025.0	985074875.0
996	996.0	992016.0	988047936.0
997	997.0	994009.0	991026973.0
998	998.0	996004.0	994011992.0
999	999.0	998001.0	997002999.0

1000 rows × 3 columns

この DataFrame オブジェクト dfArw の dtypes プロパティを確認する.（次の例）

例. dtypes プロパティの確認（先の例の続き）

入力：
```
dfArw.dtypes
```

出力：
```
linear    double[pyarrow]        ←バックエンドが
square    double[pyarrow]        ← Arrow に
cubic     double[pyarrow]        ←なっている
dtype:  object
```

この後，dfArw は通常の DataFrame として扱える.（次の例）

例. dfArw の要素を計算に使用する（先の例の続き）

入力：
```
n = dfArw.loc[999,'square']  # 要素の取り出し
print( type(n) )    # 要素の値の型を調べる
print( n )          # 要素を表示
print( n**0.5 )     # 計算に使用する
```

出力：
```
<class 'float'>        ← Python 元来の浮動小数点数
998001.0               ←要素の値
999.0                  ← √n の計算結果
```

7.1.2.2 Array，ChunkedArray ベースの Series の作成

サンプルデータの Array オブジェクト，ChunkedArray オブジェクトを作成する.

例. Array，ChunkedArray オブジェクトの作成（先の例の続き）

入力：
```
ar1 = pa.array( [1.2, 3.4] )          # Array の作成 (1)
ar2 = pa.array( [5.6, 7.8] )          # Array の作成 (2)
ch = pa.chunked_array( [ar1, ar2] )   # ChunkedArray の作成
```

この例で作成した Array オブジェクト ar1 をバックエンドに持つ Series オブジェクトを作成する.（次の例）

例. Array をバックエンドにした Series の作成（先の例の続き）

入力：
```
sr1 = ar1.to_pandas(types_mapper=pd.ArrowDtype)
sr1
```

出力：
```
0  1.2
1  3.4
dtype:  double[pyarrow]        ←バックエンドが Arrow になっている
```

同じ方法で，ChunkedArray オブジェクト ch をバックエンドに持つ Series オブジェクトを作成する.（次の例）

例. ChunkedArray をバックエンドにした Series の作成（先の例の続き）

入力：
```
sr12 = ch.to_pandas(types_mapper=pd.ArrowDtype)
sr12
```

出力：
```
0  1.2
1  3.4
2  5.6
3  7.8
dtype:  double[pyarrow]       ←バックエンドが Arrow になっている
```

7.1.2.3 性能評価

PyArrow の Array や Table はイミュータブルな列指向データ構造である．個々の要素を 1 つずつ参照する用途よりも，列としてまとめて処理する用途に適している．このことを示すサンプルプログラムを示す．まずはじめに，次のようにしてサンプルデータを作成する．

例. サンプルデータの用意などの準備作業（先の例の続き）

入力：
```
# サンプルデータの作成
import time                          # 時間計測用モジュール
N = 1000000                          # データ件数
rng = np.random.default_rng(1)   # RNG の作成
val = rng.normal(size=N)          # 正規乱数の作成
idx = rng.permutation(N)          # ランダムなインデックス
sr_np = pd.Series(val)           # NumPy ベースの Series
sr_ar = pd.Series(val, dtype="float64[pyarrow]") # Arrow ベースの Series
```

正規乱数の配列 val と，その値を保持する Series オブジェクト sr_np（NumPy バックエンド），sr_ar（Arrow バックエンド）ができた．

ランダムなインデックスの列 idx を用いて，sr_np，sr_ar に対して要素に対するランダムアクセスにかかる時間を測定する例を次に示す．また，そのために使用する関数 testf1 を次のように定義する．

例. 実行時間測定用関数 testf1（先の例の続き）

入力：
```
def testf1(s,idx):
    acc = 0.0
    t0 = time.perf_counter() # 開始時刻
    for i in idx:
        acc += float(s.iat[i])
    t1 = time.perf_counter() # 終了時刻
    print(f'実行時間:{t1-t0:6.4f}(sec)，集計結果:{acc:8.4f}')
```

この関数は，計測対象の Series オブジェクトを第 1 引数 s に，アクセス用のインデックス並びを第 2 引数 idx に受け取る．Series オブジェクトへのアクセスは iat による値の参照で，要素を 1 つずつ参照する．全ての要素の値の合計を求めた後，実行時間を出力する．

この関数を複数回呼び出して実行し，Series オブジェクトへのアクセス時間を測定する例を次に示す．

例. NumPy バックエンドの Series オブジェクトに対するアクセスの時間の計測（先の例の続き）

入力：
```
for _ in range(3):
    testf1(sr_np,idx)
```

出力：
```
実行時間:1.1978(sec)，集計結果:-208.9982
実行時間:1.2139(sec)，集計結果:-208.9982
実行時間:1.1289(sec)，集計結果:-208.9982
```

例. Arrow バックエンドの Series オブジェクトに対するアクセスの時間の計測（先の例の続き）

入力：
```
for _ in range(3):
    testf1(sr_ar,idx)
```

出力：
```
実行時間:2.8843(sec)，集計結果:-208.9982
実行時間:2.9097(sec)，集計結果:-208.9982
実行時間:2.7937(sec)，集計結果:-208.9982
```

iat で個別の要素を参照する処理では，Arrow バックエンドの Series オブジェクトの場合の方が処理に時間（2 倍以上）がかかっていることがわかる．これは意外なことに見えるが，take メソッドによる一括アクセスを行う場合は異なる結果となる．このことを示すために，実行時間測定用の関数 testf2 を次のように定義する．

例. 実行時間測定用関数 testf2（先の例の続き）

入力：
```
def testf2(s,idx):
    t0 = time.perf_counter()          # 開始時刻
    acc = float(s.take(idx).sum())  # まとめて取り出して合計
    t1 = time.perf_counter()          # 終了時刻
    print(f'実行時間:{t1-t0:6.4f}(sec)，集計結果:{acc:8.4f}')
```

先の testf1 と同様の働きをするが，take メソッドでデータを一括して取り出した後で合計値を算出している．

この testf2 を用いて sr_np，sr_ar へのアクセスにかかる時間を測定する例を次に示す．

例. NumPy バックエンドの Series オブジェクトに対するアクセスの時間の計測（先の例の続き）

入力：
```
for _ in range(3):
    testf2(sr_np,idx)
```

出力：
```
実行時間:0.0130(sec)，集計結果:-208.9982
実行時間:0.0151(sec)，集計結果:-208.9982
実行時間:0.0126(sec)，集計結果:-208.9982
```

例. Arrow バックエンドの Series オブジェクトに対するアクセスの時間の計測（先の例の続き）

入力：
```
for _ in range(3):
    testf2(sr_ar,idx)
```

出力：
```
実行時間:0.0126(sec)，集計結果:-208.9982
実行時間:0.0118(sec)，集計結果:-208.9982
実行時間:0.0116(sec)，集計結果:-208.9982
```

この方法によるアクセスでは，Arrow バックエンドの Series オブジェクトの場合の方が処理にかかる時間が若干短いことがわかる．また，Python の for ループによる逐次処理は，バックエンドの違いよりも Python 処理系のオーバーヘッドの影響を強く受けることも留意する必要がある．

pandas のデータ構造を介さずに，直接的に Array オブジェクトにアクセスする場合は，更に実行時間が短くなる．このことを示すための準備を次に示す．

例. 実行時間測定のための準備（先の例の続き）

入力：
```
import pyarrow.compute as pc  # 計算に必要なモジュール
val_ar = pa.array(val)         # 先の val を Array オブジェクトに変換
idx_ar = pa.array(idx)         # 先の idx を Array オブジェクトに変換
def testf3(val,idx):           # 時間計測用関数
    t0 = time.perf_counter()                  # 開始時刻
    acc = pc.sum(pc.take(val,idx)).as_py()    # まとめて取り出して合計
    t1 = time.perf_counter()                  # 終了時刻
    print(f'実行時間:{t1-t0:7.5f}(sec)，集計結果:{acc:8.4f}')
```

この準備処理で，先の val，idx から Array オブジェクト val_ar，idx_ar を作成し，処理時間測定用の関数 testf3 を定義している．(pyarrow.compute モジュールに関する詳細は，公式インターネットサイトをはじめとする他の文献を参照のこと)

関数 testf3 を用いて，処理時間の測定を複数回行う例を次に示す．

例. Array オブジェクトに対するアクセスの時間の計測（先の例の続き）

入力：
```
for _ in range(3):
    testf3(val_ar,idx_ar)
```

出力：
```
実行時間:0.00519(sec)，集計結果:-208.9982
実行時間:0.00568(sec)，集計結果:-208.9982
実行時間:0.00458(sec)，集計結果:-208.9982
```

この例では，Arrow オブジェクトへのアクセスは，sr_np に対するアクセスの 2 倍を超える速度が得られた．また，今回の性能評価では sr_ar へのアクセスが最も遅いという結果となった．

注意）当然であるが，pandas が提供する豊富な機能を使用するには，フロントエンドとして DataFrame や Series オブジェクトの形式としてデータを扱う必要がある．上の例はあくまで性能評価のための参考的な試みである．

以上のことから，Arrow バックエンドの Series では，各要素への個別のアクセスがスカラーアクセス用に最適化されていないことがわかる．また一方で，take や sum のように列としてまとめて処理する場合には高い性能を発揮する．

7.2 Apache Parquet

データ処理で取り扱う表形式のデータは，1 つのデータ系列を 1 つの列として扱う形式のものが多い．例えば pandas の DataFrame においても，1 つの列は，型を同じくする 1 つのデータ系列を表す．これは**列指向のデータセット**であると表現され，CSV のような**行指向のデータセット**と対比される．

列指向のデータセットを効率よくストレージに格納するファイルフォーマットに Apache Parquet （以後 Parquet と略す）がある．Parquet は実際に，多くのデータ処理系ソフトウェアで使用可能であり，機械可読な形の（表形式の）ファイル資源を大規模に貯蔵する[79] のに適している．また Parquet では，CSV などのデータファイルと異なり，必要な列データを選択的に高速に読み取ることができる．

Parquet は列指向フォーマットであるため，既存ファイルの一部だけを更新する処理は効率的ではない．実運用においては，更新が必要なデータは別途管理し，ある時点でまとめて Parquet ファイルとして書き出す，という使い方が一般的である．すなわち Parquet は，データをまとめて保存し，必要な列を高速に読み取る用途に適した形式であり，個々の行や列を頻繁に更新する用途にはあまり向いていない．この意味では，CSV ファイルの完全な代替となるものではないので，プログラミングの意図に合わせてファイル形式を適切に選択することが肝要である．

本書では，pandas の DataFrame と Parquet ファイルの間の入出力の方法について解説する．

7.2.1 DataFrame の内容を Parquet ファイルに保存する方法

ここでは，作成した DataFrame のサンプルを Parquet ファイルに保存する方法を解説する．まず次のようにして，必要なライブラリを読み込み，サンプルデータを作成する．

例. ライブラリの読み込みとサンプルデータの作成（先の例の続き）

入力：
```
import numpy as np      # NumPy の読み込み
import pandas as pd     # pandas の読み込み
pd.set_option('display.min_rows',6)     # DataFrame の省略表示行数を 6 行に設定
```

入力：
```
rng = np.random.default_rng(1)     # 乱数生成器の作成
r1 = (rng.random(10000) * 10000).astype(np.int64)   # 乱数系列の作成 (1)
r2 = rng.random( 10000 )                            # 乱数系列の作成 (2)
df0 = pd.DataFrame( 'r1':r1, 'r2':r2 )     # DataFrame にする
df0     # 内容確認
```

[79] この様子を**データレイク**（Data Lake）と表現することがある．

出力：

	r1	r2
0	5118	0.572126
1	9504	0.031302
2	1441	0.145296
...	...	...
9997	1165	0.417162
9998	3198	0.945928
9999	4814	0.281505

10000 rows × 2 columns

DataFrame の内容を Parquet 形式のファイルに保存するには to_parquet メソッドを使用する.

書き方： DataFrame オブジェクト.to_parquet(出力ファイルのパス, engine=使用するエンジン)

「出力ファイルのパス」は文字列で与える.「使用するエンジン」は省略可能で，その場合は，システムが自動的にエンジンを選択する．複数のエンジンがシステムのインストールされている場合は，明示的に「使用するエンジン」を指定するのが安全である.

例. DataFrame の内容を Parquet ファイルに保存する（先の例の続き）

入力：
```
df0.to_parquet('Parquet01.parquet', engine='pyarrow')
```

これは，先に作成した DataFrame オブジェクト df0 を，カレントディレクトリの 'Parquet01.parquet' というファイルに保存する例である．エンジンには PyArrow を指定（'pyarrow'）しているが，これは事前にインストールされている必要がある．ファイル名の拡張子は任意であるが,「.parquet」を付けるのが一般的である.

7.2.2 Parquet ファイルの内容を DataFrame に読み込む方法

pandas の read_parquet 関数によって，Parquet ファイルの内容を読み込むことができる．この関数は，読み取った内容を DataFrame として返す.

書き方： read_parquet(対象ファイルのパス, engine=使用するエンジン)

「対象ファイルのパス」は文字列で与える.「使用するエンジン」は省略可能で，その場合は，システムが自動的にエンジンを選択する．複数のエンジンがシステムのインストールされている場合は，明示的に「使用するエンジン」を指定するのが安全である.

例. Parquet ファイルの読み込み（先の例の続き）

入力：
```
df_numpy = pd.read_parquet('Parquet01.parquet', engine='pyarrow')
df_numpy    # 内容確認
```

出力：

	r1	r2
0	5118	0.572126
1	9504	0.031302
2	1441	0.145296
...	...	...
9997	1165	0.417162
9998	3198	0.945928
9999	4814	0.281505

10000 rows × 2 columns

これは，先に作成した Parquet ファイル 'Parquet01.parquet' を，カレントディレクトリから読み込み，その内容を DataFrame オブジェクト df_numpy として取得する例である．エンジンには PyArrow を指定（'pyarrow'）している.

得られた df_numpy のバックエンドはデフォルトでは NumPy 配列である．これを確かめる例を次に示す.

例. 得られた DataFrame オブジェクトのバックエンドを調べる（先の例の続き）

入力：
```
df_numpy.dtypes
```

出力：
```
r1     int64
r2   float64
dtype:  object
```

■ 指定したカラムのみを読み込む方法

read_parquet 関数に引数「columns=[カラム 1，カラム 2，…]」を与えることで，指定したカラムのみを読み込むことができる．次に示す例は，先と同じ 'Parquet01.parquet' の中の 'r1' のカラムのみを読み込むものである．

例. カラム 'r1' のみを読み込む（先の例の続き）

入力：
```
df_r1 = pd.read_parquet('Parquet01.parquet', engine='pyarrow',
                        columns=['r1'])
df_r1    # 内容確認
```

出力：
```
        r1
0     5118
1     9504
2     1441
...    ...
9997  1165
9998  3198
9999  4814
10000 rows × 1 columns
```

7.2.2.1 Parquet の読み取り結果を Arrow ベースの DataFrame にする方法

read_parquet 関数にキーワード引数 dtype_backend を与え，これに Arrow 系エンジンを指定すると，Arrow をバックエンドとする DataFrame オブジェクトとして読み取り結果を取得することができる．次の例は

```
dtype_backend='pyarrow'
```

を指定して読み込むものである．

例. 読み取り結果を Arrow ベースの DataFrame にする方法（先の例の続き）

入力：
```
df_arrow = pd.read_parquet('Parquet01.parquet',
                           engine='pyarrow', dtype_backend='pyarrow')
df_arrow    # 内容確認
```

出力：
```
        r1        r2
0     5118  0.572126
1     9504  0.031302
2     1441  0.145296
...    ...       ...
9997  1165  0.417162
9998  3198  0.945928
9999  4814  0.281505
10000 rows × 2 columns
```

入力：
```
df_arrow.dtypes    # 型を調べる
```

出力：
```
r1    int64[pyarrow]
r2   double[pyarrow]
dtype:  object
```

この例では，読み取り結果が PyArrow をバックエンドとする DataFrame として得られていることがわかる．

7.2.2.2 Parquet の内容を選択的に読み込む方法

巨大な Parquet ファイルを読み込む際は，指定した条件を満たす部分のみを選択的に読み込むことが推奨される．読み込み時の条件は，read_parquet 関数に引数 filters として指定する．ここでは，巻末付録「B.1 疑似データセットの作成」(p.159) に示す方法で作成した Parquet02.parquet というファイルから，内容を選択的に読み込む例を挙げて解説する．

例. Parquet02.parquet の読み込みと内容確認（先の例の続き）

入力：
```
df_All = pd.read_parquet('Parquet02.parquet')
display(df_All.iloc[:16:4])
print('レコード数 :',len(df_All))
```

出力：

	city	gender	age
0	大阪	男	30
4	東京	女	84
8	大阪	男	77
12	福岡	男	51

レコード数 : 10000

このように Parquet02.parquet は 10,000 件のレコードから成るデータであり，city（都市），gender（性別），age（年齢）のカラムから成る．次は，city が「大阪」であるレコードのみを選択して読み込む例を示す．

例. city カラムが「大阪」のみのデータを読み込む（先の例の続き）

入力：
```
df_Osaka = pd.read_parquet('Parquet02.parquet',
                filters = [ ('city','==','大阪') ])
df_Osaka
```

出力：

	city	gender	age
0	大阪	男	30
1	大阪	男	31
2	大阪	男	77
...	...	...	...
2497	大阪	女	58
2498	大阪	男	51
2499	大阪	男	74

2500 rows × 3 columns

この例では read_parquet に

```
filters = [ ('city','==','大阪') ])
```

という引数を与えている．最も内側のタプルは

(カラム, オペレータ, オペランド)

の形式であり，指定した「カラム」が「オペレータ」と「オペランド」で表される条件を満たすレコードのみを読み込みの対象とする．また，このようなタプルを要素とするリストは，それらが全て満たされる条件（AND 結合）を意味する[80] ．指定した複数の条件を全て満たすレコードのみを読み込む例を次に示す．

例. 「大阪」でかつ「女性」でかつ「10 才代」のデータを読み込む（先の例の続き）

入力：
```
df_OsakaWT = pd.read_parquet('Parquet02.parquet',
           filters = [ ('city','==','大阪'), ('gender','==','女'),
                       ('age','>=',10), ('age','<',20) ])
df_OsakaWT
```

出力：

	city	gender	age
0	大阪	女	14
1	大阪	女	12
2	大阪	女	13
...	...	...	...
72	大阪	女	11
73	大阪	女	11
74	大阪	女	13

75 rows × 3 columns

上で示したAND 結合形式のリストを，更に要素として持つリストは OR 結合[81] の条件記述となる．次に示す例は

「大阪」でかつ「女性」でかつ「「未成年」もしくは「85 才以上」」

という条件を展開して filters 引数に与えるものである．

[80] **連言標準形**（CNF：Conjunctive Normal Form）

[81] **選言標準形**（DNF：Disjunctive Normal Form）

例. 「大阪」でかつ「女性」でかつ「「未成年」もしくは「85 才以上」」のデータを読み込む（先の例の続き）

入力：
```
df_OsakaWYO = pd.read_parquet('Parquet02.parquet',
    filters = [ [('city','==',' 大阪'), ('gender','==',' 女'), ('age','<',20)],
                [('city','==',' 大阪'), ('gender','==',' 女'), ('age','>=',85)] ])
df_OsakaWYO
```

出力：

	city	gender	age
0	大阪	女	87
1	大阪	女	0
2	大阪	女	14
...	...	...	...
216	大阪	女	0
217	大阪	女	99
218	大阪	女	89

219 rows × 3 columns

参考）filters 引数に与える値は「全階層がリスト」あるいは「全階層がタプル」であっても read_parquet は正しく機能するが，そのような形式は推奨されない．

付録

A　統計学の用語

統計解析の目的を素朴な形で表現すると,「与えられたデータの集合の特徴を調べる」ことである．ここで重要な用語として**母集団**[82]（population）と**標本**（sample）がある．母集団は調査対象の集団全体のことを指す．例えば，インターネット上に公開されている日本語の Web コンテンツの中に占める単語の出現頻度を調べる場合は,「インターネット上に公開されている日本語の Web コンテンツの全ての単語」が母集団となる．実際の統計処理においては母集団を完全に調査することは困難であることが多く，部分的に標本を**抽出**（sampling）して調べることになる．このとき，調査の対象として取得した要素（とその属性）のことを標本と呼ぶ.

抽出された標本から算出される各種の値は，その標本のみに関するものであり，母集団の統計的特徴とは差異がある．従って，母集団の特徴を調査するには，抽出する標本の要素の数を多く取るといった様々な工夫をしながら**推定**（estimation）することになる．母集団の推定をする方法に関しても各種の手法がある.

実際の統計解析は，各種の**要約統計量**[83] を調べることからはじまる．主な要約統計量としては**平均値**[84]（mean），**最頻値**（mode），**中央値**（median）を含む**四分位数**（quartile points）[85] があるが，これらに加えて**最大値**，**最小値**，**標準偏差**（あるいは**分散**）なども含める.

採取した標本から「ある**値**」（あるいは**階級**[86] ）に対する「その**度数**」（frequency）を調べたものが**度数分布**（frequency distribution）である．またそれを柱状のグラフにしたものを**ヒストグラム**（histogram）あるいは**度数分布図**という．度数分布の調査において，サンプルの総数に対する各階級値の出現頻度から，その階級値の出現確率を算出することができる.

階級値毎の標本の出現確率の特徴を調べるには，それがどのような**確率分布**となっているかを考えることが基本的な作業となる.

A.1　確率変数と確率を表す関数

確率分布（probability distribution）は，**確率変数**に対する確率を表したものである．日本工業規格では,「確率変数がある値となる確率，又はある集合に属する確率を与える関数」と定義している．実際の統計処理において，例えばサイコロやルーレットの試行においては「結果の値」（出た目の番号）が，あるいは世帯年収に対する世帯数の調査では「世帯年収の金額」などが確率変数の値となる.

A.1.1　確率に関する重要な事柄

全事象の確率の合計は必ず 1 となる．例えば，サイコロの試行において，各目が出る確率はそれぞれ 1/6 で，全ての目の確率を合計すると $6 \times 1/6 = 1$ となることが挙げられる．次に，確率変数は離散的なものと連続的なものの 2 種類が存在するとうことが重要である．実際の統計処理では離散的な確率変数を扱う場合が多い.

離散的な確率変数 x には，それが取る値毎に確率の値 $P(x)$ があり，この P を**確率質量関数**という．全ての x に対する $P(x)$ の合計は 1 になる．すなわち,

$$\sum_x P(x) \;=\; 1$$

である.

連続的な確率変数 x では，それが取る値に対して確率の値があるのではなく，**確率密度**の値 $f(x)$ が定義される．この f を**確率密度関数**という．確率密度関数を確率変数の全領域で定積分すると 1 になる．すなわち,

[82] 日本工業規格では,「考察の対象となる特性をもつ全てのものの集団」と定義している.

[83] **代表値**（measures of central tendency），基本統計量ともいう.

[84] 「平均値」は通常**算術平均**を意味するが，他にも**調和平均**，**幾何平均**があり，それぞれ定義が異なる.

[85] 「A.1.5 分位数，パーセント点」（p.149）を参照のこと.

[86] 値の範囲．通常はその範囲の中央の値を**階級値**（class value）とする.

$$\int_{-\infty}^{\infty} f(x)\,dx \;=\; 1$$

である．

確率質量関数として基本的で重要なものに，**二項分布**が，確率密度関数として重要なものに**正規分布**がある．

A.1.2　確率質量関数

離散的な確率変数に対する確率の値を与える**確率質量関数**として代表的なものを挙げる．

A.1.2.1　二項分布

各試行において成功か失敗かの 2 状態をとり，各試行において成功する確率が p であるという事象を考える．例えば，当たり／はずれのくじがあり，当たりが出る確率が p であるという状況を考える．このような状況で，n 回の試行[87] 中 x 回成功となる確率の分布（確率質量関数） $f(x)$ は**二項分布**となる．具体的には次のような式になる．

$$f(x) = {}_nC_x \cdot p^x(1-p)^{n-x}$$

成功確率 p の試行を n 回行う場合の二項分布を $Bi(n,p)$ と書く．

A.1.2.2　幾何分布

成功確率 p のベルヌーイ試行を繰り返したとき，初めて成功するまでの回数 x の分布は**幾何分布**となる．幾何分布の定義は次の通り．

$$P(x) = p(1-p)^{x-1} \qquad \text{（定義 a）}$$

これとは別に，初めて成功するまでに失敗した回数の分布として幾何分布が定義されることもあり，その場合は

$$P(x) = p(1-p)^{x} \qquad \text{（定義 b）}$$

となる．統計解析に幾何分布を用いる際は，どちらの定義によるものかを示す必要がある．

A.1.2.3　超幾何分布

非復元抽出による確率事象は**超幾何分布**に従うものが多い．サンプル抽出に伴い母集団の要素数は減少する（母集団が変化する）が，そのまま引き続いてサンプル抽出を行う形のものを非復元抽出と呼ぶ．非復元抽出による確率事象として現実的な事例としては「複数の当たりを含むくじ」である．例えば，抽選会などで使用される「当たり玉くじ」などが代表的な例である．この「当たり玉くじ」の状況をまとめると次のようになる．

「N 個の玉のくじがあり，K 個の当たりを含んでいる」

このようなくじから，n 個の玉を取り出した際に k 個の当たりが出る確率は超幾何分布となる．この分布は $f_x(k;N,K,n)$ と書き，定義は次の通りである．

$$f_x(k;N,K,n) \;=\; \frac{{}_nC_k \cdot {}_{N-n}C_{K-k}}{{}_NC_K}$$

A.1.2.4　ポアソン分布

ある確率事象が，定められた時間内に x 回生起する確率の分布は**ポアソン分布**となる．ポアソン分布の定義は次の通り．

$$P(x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

ポアソン分布は，発生確率の低い確率事象を表現するもので，これに沿う現象の例としては，交通事故に遭う回数や，熟達したタイピストが起こすタイプミスの発生回数などが挙げられる．すなわち，ポアソン分布に従う事象の特徴として，発生確率 p が小さく，十分に大きな試行回数 n の下で $np =$ **一定** となることである．上記の関数定義において，$\lambda = np$ であり，確率変数 x は整数（発生回数）を取る．

[87] このような試行を**ベルヌーイ試行**という．

A.1.3　確率密度関数

連続的な確率変数に対する確率密度の値を与える**確率密度関数**として代表的なものを挙げる．

A.1.3.1　正規分布

正規分布 $f(x)$ の定義は次の通り．

$$f(x) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \qquad (x \text{ は実数})$$

ここで μ は x の**平均（期待値）**，σ は**標準偏差**である．$f(x)$ のプロットを図 19 に示す．

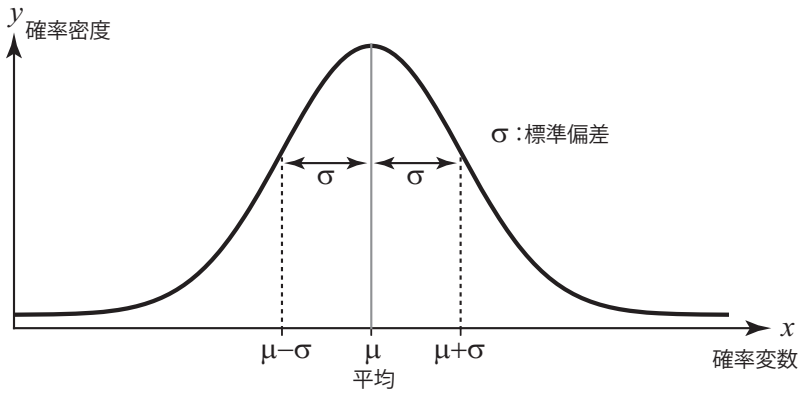


図 19: 正規分布

平均が μ，分散が σ^2 の正規分布を $N(\mu, \sigma^2)$ と書く．

A.1.3.2　指数分布

単位時間内に平均して λ 回発生する確率事象があるとき，その事象が最後に発生してから次に生起するまでの時間を x とすると，時間 x 後にその事象が発生する確率の密度は**指数分布**となる．指数分布の定義は次の通りである．

$$f(x) = \begin{cases} x \geq 0 & \rightarrow & \lambda e^{-\lambda x} \\ x < 0 & \rightarrow & 0 \end{cases}$$

A.1.3.3　対数正規分布

経済学や石油資源開発などの分野では**対数正規分布**がしばしば用いられる．この分布の定義は次の通りである．

$$f(x) = \frac{1}{\sqrt{2\pi}\,\sigma\, x} \exp\left(-\frac{(\log x - \mu)^2}{2\sigma^2}\right) \qquad (x > 0)$$

確率密度関数としては，ここで挙げたもの以外にも t **分布**，χ^2 **分布**が重要である．これらに関しては，「A.2 母集団と標本に関する重要な事柄」（p.150）の所で，応用例を示しながら解説する．

A.1.4　尖度，歪度

正規分布よりも「尖った」分布（図 20）を「**尖度**（kurtosis）が大きい」と表現する．

尖度は正規分布を基準とし，正規分布の尖度を 0 [88] とする．尖度の大きい分布では，山の近くでは尖りが強く，裾が厚い形となる．

確率変数の左右（大小）で非対称な分布は「**歪度**（skewness）が大きい」と表現する．左右対称な分布は歪度が 0 である．例えば図 21 に示す対数正規分布は歪度が大きい分布である．

[88] 正規分布の尖度を 3 とする文献もあるので注意すること．

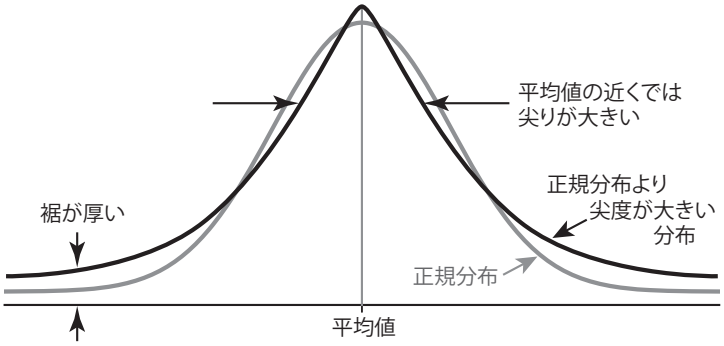


図 20: 尖度の大きい分布

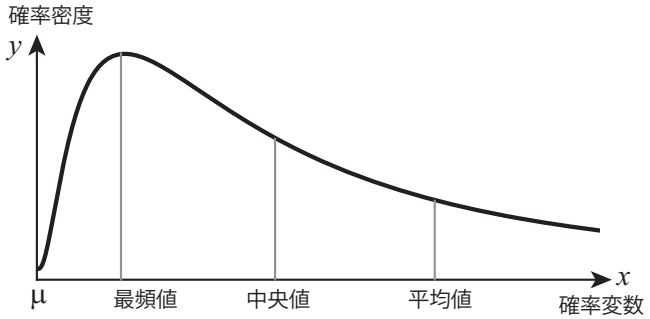


図 21: 対数正規分布
注）確率密度関数を構成する μ, σ と，結果としての
平均値（期待値），標準偏差は異なる．

対数正規分布では平均値，中央値，最頻値が全て異なる値となり，結果として μ，σ も「平均」（期待値）,「標準偏差」とは異なる意味を持つ．

【μ，σ が意味するもの】

統計調査で用いる確率質量関数，確率密度関数の多くのものが，正規分布に変形，変換を施したものとして解釈できる．その意味では正規分布は確率密度関数のもっとも基本的なものであるということができる．実際に，多くの確率密度関数が μ，σ を用いて記述される．ただし，正規分布以外の確率密度関数においては，結果的な平均値と標準偏差は μ，σ とは必ずしも同じものとはならないことに注意すること．先の対数正規分布で μ と平均値が異なることが 1 つの例である．

注意）

実際の統計学の文献や調査報告において「μ」,「平均値」,「期待値」の用語の意味が統一されていないことがあるので注意すること．特に期待値 $E(X)$ は「標本（あるいは確率変数）の確率の重みを付けた合計」であり，確率変数 $\{x_1, x_2, \cdots, x_n\} \in X$ に対する確率を $P(x_i)$ とすると次のように定義される．

$$E(X) = \sum_i x_i \cdot P(x_i) \qquad \text{（離散分布の場合）}$$

期待値を実質的な意味で「平均値」として取り扱う文献もあるので留意しておくこと．

尖度，歪度は「**モーメント**[89]（moment）」を用いて計算する．

【α 周りの n 次のモーメント】

基準となるある値 α を考え，平均値（期待値）の計算を E と書き，

$$E\,(X-\alpha)^n$$

[89] **積率**ともいう．

を,「標本集合 X の α 周りの n 次のモーメント」と定義する．更に $E\,(X)^n$ を「原点周りの n 次のモーメント」[90] という．

平均値周りの n 次のモーメントを m_n と書くと，尖度は　$\dfrac{m_4}{\sigma^4} - 3$

として定義される.（この定義では正規分布の尖度は 0 となる）また歪度は　$\dfrac{m_3}{\sigma^3}$

として定義される．対数正規分布では歪度が正の値を取り，これは「右の裾が長く，山が左に寄っている」ことを意味する．逆に歪度が負の値を取る分布は「左の裾が長く，山が右に寄っている」形となる．

モーメントを用いると，平均値は原点周りの 1 次のモーメントとして，また分散は平均値周りの 2 次のモーメント m_2 として定義できる．

A.1.5　分位数，パーセント点

連続的な確率変数 x に対して確率密度関数 $f(x)$ があるとき，$q \in [0,1]$ に対して

$$\int_{-\infty}^{Q_q} f(x)dx \;=\; q$$

を満たす Q_q を $f(x)$ の q **分位数**という[91] .（図 22）

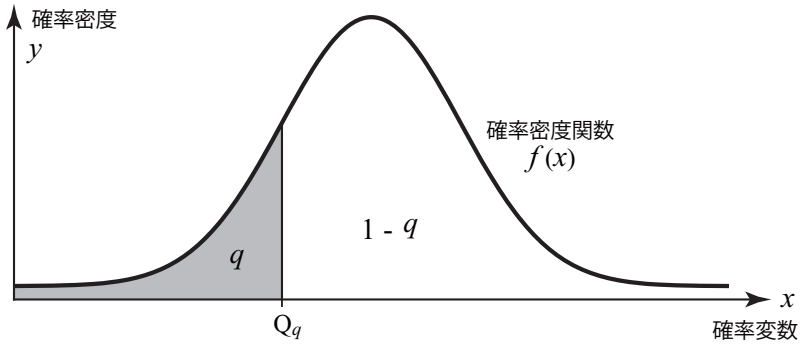


図 22: q 分位数
確率密度関数の面積を左から $q : 1-q$ に分割する点 Q_q

離散的な確率変数に対しても分位数が定義されている．この場合は，採取された確率変数の値（サンプル）を昇順に整列し，最小の値から数えたサンプル数 n の全サンプル数 N に対する割合で q 分位数を算出する[92] .

四分位数の各点はそれぞれ $q = 0.25$, $q = 0.5$, $q = 0.75$ に対応する.（表 20）

表 20: 各四分位数と分位数の対応

四分位数（パーセント点）	q	備考
第 1 四分位数（25%点）	$q = 0.25$	
第 2 四分位数（50%点）	$q = 0.5$	中央値
第 3 四分位数（75%点）	$q = 0.75$	

パーセント点とは分位数を百分率で表現したものである.

[90] **参考**）　$E\left(\dfrac{X-\mu}{\sigma}\right)^n$ を**標準化された**モーメントという.

[91] このような方程式の解 Q_q が存在することを前提としている.

[92] 離散的な確率変数に対する分位数の厳密な定義は割愛する.

A.2　母集団と標本に関する重要な事柄

母集団の平均を**母平均**，取り出した標本の平均を**標本平均**という．ある母集団から無作為抽出で標本を取り出す場合，母平均と標本平均の間には差異（**誤差**）がある．抽出する標本の数を多くするほど得られる標本平均は母平均に近づくが，この際の誤差の分布は**正規分布**に近づく．これが**中心極限定理**である.（証明は他の文献に譲り割愛する）

母集団の分散を**母分散**という．母平均，母分散など，母集団の統計量を**母数**（parameter）という．

A.2.1　標本の抽出

単純無作為抽出（単純ランダムサンプリング）で抽出された標本から統計量を算出する場合について考える．抽出した標本の集合を X と書き，標本の数を n とすると，**標本平均** $\bar{X}$ は次のような式で書く．

$$\bar{X} = \frac{x_1 + x_2 + \cdots + x_n}{n} \qquad (x_1, x_2, \cdots, x_n \in X)$$

標本平均は，**母平均**（母集団の平均）μ とは異なり，誤差を含んだ値である．標本平均の期待値 $E(\bar{X})$ は母平均 μ に等しい．例えば，同一の母集団に対して複数回の統計調査（ランダムサンプリング）を行い，各回の $\bar{X}$ の平均を取るとその値は母平均 μ に近づくという事実がこれに当たる．もちろんこの行為は n を母集団の数 N に近づけることに等しい．

A.2.1.1　標本分散と不偏分散

採取した標本から求めた分散は**標本分散**と呼ばれ，次の式で定義される．

$$S^2 = \frac{(x_1 - \bar{X})^2 + (x_2 - \bar{X})^2 + \cdots + (x_n - \bar{X})^2}{n}$$

標本分散は母分散（母集団の分散）σ^2 とは異なり，誤差を含んだ値である．次に，**不偏分散** s^2 というもの（次の式）を定義する．

$$s^2 = \frac{(x_1 - \bar{X})^2 + (x_2 - \bar{X})^2 + \cdots + (x_n - \bar{X})^2}{n-1}$$

分母を $n-1$ とする不偏分散を定義する理由としては， S^2 の期待値 $E(S^2)$ を算出すると，

$$E(S^2) = \frac{n-1}{n} \cdot \sigma^2$$

となるということがある．このことは，s^2 が S^2 に比べて**不偏性**[93] があり，σ^2 に近い形に補正されたものであることを意味する．実際の統計処理において，分散として s^2 が採用されることが多い．

S を**標本標準偏差**，s を**不偏標準偏差**と呼ぶ．

参考.

実際の統計処理では，計算の簡便性のために標本の値の 2 乗の平均を求めておくことがある．これにより，標本分散 S^2 を次の形で求めることができる．

$$S^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \bar{X}^2$$

（証明）

$$S^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{X})^2 = \frac{1}{n}\sum_{i=1}^{n}\left(x_i^2 - 2x_i\bar{X} + \bar{X}^2\right) = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - 2\bar{X}\cdot\frac{1}{n}\sum_{i=1}^{n} x_i + \bar{X}^2 \cdot \frac{1}{n}\sum_{i=1}^{n} 1$$

$$= \frac{1}{n}\sum_{i=1}^{n} x_i^2 - 2\bar{X}\cdot\bar{X} + \bar{X}^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - 2\bar{X}^2 + \bar{X}^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \bar{X}^2$$

A.2.2　推定

母集団の統計量に関する推定には次の 2 種類の場合がある．1 つは母集団の確率がどのような分布に沿っているかが予め分かっている場合である．このような場合は，母集団の分布を決めている確率質量関数や確率密度関数から μ,

[93] **不偏性**と**一致性**に関する解説は他の文献を参照のこと.

σ などを推定する．母集団の分布が予め分かっているという前提での推定を**パラメトリック（parametric）な推定**という．
2 つ目は，母集団が従う確率分布が予め分からない場合の推定である．このような場合は，得られた標本から各種の統計量を算出して，統計処理を更なる段階へと進めるための判断材料とする．このように，母集団の分布が予め分かっていない前提での推定処理を**ノン・パラメトリック（non-parametric）な推定**という．

以上のように，抽出した標本を元にして母数を近似的に**推定**[94]（estimation）することになるが，抽出した標本から得られた統計量は，母数を推定するための**推定量**（estimator）であるという．これには，標本から算出した**標本平均**や**標本分散**などが含まれる．推定処理を行うために標本から各種の値を算出するが，それらをまとめて**統計量**（statistic）と呼ぶ．

A.2.2.1　点推定

統計調査において十分な数の標本が採取できる場合は**点推定**と呼ばれる方法で母数を推定することがある．点推定では，母数（母集団が持つ各種統計量）の各値を 1 通りに定める．点推定とは別に，母数の各値を「ある範囲にある」として推定する方法を**区間推定**といい，点推定とは区別する．（区間推定については後で説明する）

最も単純な点推定の方法としては，得られた標本の統計量をそのまま母数として採用するものがある．例えば n 個の標本の集合 X から算出した標本平均と不偏分散を，それぞれ母平均 μ の推定値 $\hat{\mu}$，母分散 σ^2 の推定値 $\hat{\sigma}^2$ とみなすものである．すなわち，

$$\hat{\mu} = \frac{x_1 + x_2 + \cdots + x_n}{n}, \quad \hat{\sigma}^2 = \frac{(x_1 - \bar{X})^2 + (x_2 - \bar{X})^2 + \cdots + (x_n - \bar{X})^2}{n-1}$$

とする．このように，母数として推定された値（推定値）にはハット ‘ˆ’ を付けて表記する．

【点推定の例】最尤法

成功する（実現値が 1 である）確率が p，失敗する（実現値が 0 である）確率が $1-p$ である**ベルヌーイ試行**を考え，この場合の母数 p を最尤法で推定する方法を例を挙げて説明する．この試行を 5 回行った結果，実現値（標本）が次のようになったとする．

$$1, \quad 1, \quad 1, \quad 0, \quad 1$$

すなわち，標本抽出の結果,「4 回の成功，1 回の失敗」という標本が得られたことになる．この事象が発生する確率は $p^4(1-p)$ となる．以上のことを元にして最尤法で p を推定する．最尤法では「最も確率の高い事象が発生した」と仮定して母数を推定する．この仮定を**最尤原理**（principle of maximum likelihood）と呼ぶ．今回の例では，最尤なる p を探すために，事象の発生確率の関数 $L(p)$ を定義してこれが最大となる p を探す方法を取る．すなわち，

$$L(p) = p^4(1-p)$$

が最大値を与える p を探す．この例では $0 \leq p \leq 1$ の範囲で母数 p を探すが，想定される母数の範囲（集合）を**母数空間**（parameter space）という． $L(p)$ のプロットを図 23 に示す．

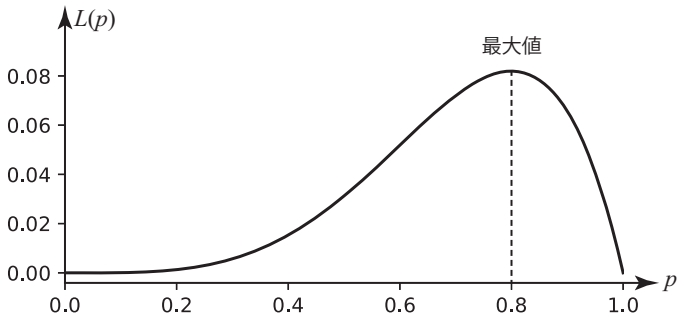


図 23: $L(p)$ のプロット

[94] これは統計学の用語である．統計調査の結果などを用いて各種の事象に関して推論することを**推測**（surmise）というが，「推定」という語とは異なる語であることを意識すること.「推定」とは母数を求めるという限定的な意味を持つ．

今回の場合，$\frac{d}{dp}L(p) = p^3(4-5p)$ となり，これが 0 となりかつ $L(p)$ が極大となる点，すなわち $\hat{p} = 0.8$ が推定値となる．

（最尤法に関する今回の例は文献 [1] を参考にしている）

A.2.2.2　区間推定

実際の統計調査においては，母集団よりも小さなサイズの標本を抽出して母数を推定する．ここでは，抽出した標本の統計量がどの程度の確からしさで母集団の母数を表しているかについて考える．具体的には「ある母数がある範囲内に含まれる条件」を求め，抽出した標本がその条件を満たしているかを判定して，標本の統計量の信憑性について評価する．

■ 信頼区間（confidence interval）

ある母数 θ がある区間 $[L, U]$ に $1-\alpha$ の確率（確からしさ／信頼性）で含まれるとする．例えば，抽出した標本の統計量が「〜%の信頼性で母数を表している」という評価を行う場合に $1-\alpha$ を満たす L と U を定める．これは，抽出した標本がどの程度の確からしさで母数を表しているかを判定する基準にもなり，信頼できる統計調査のための標本数の策定などに応用できる．この $[L, U]$ を「母数 θ の確率 $1-\alpha$ の**信頼区間**（confidence interval）」という．また，$1-\alpha$ を**信頼係数**（confidence coefficient）と呼び，L，U をそれぞれ**下側信頼限界**（lower confidence limit），**上側信頼限界**（upper confidence limit）と呼ぶ．

ここでは，正規分布に従う母集団（**正規母集団**）において信頼区間を求める方法について説明するが，そのための準備として，正規母集団からの標本抽出に関して基礎的な内容について説明しておく．

I. **標準正規分布**

平均が μ，分散が σ^2 の正規分布を $N(\mu, \sigma^2)$ と書き，平均が 0，分散が 1 の正規分布，すなわち $N(0,1)$ を**標準正規分布**という．$N(0,1)$ は確率変数 x の関数 $\phi(x)$ と書かれることがある．すなわち，

$$\phi(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)$$

である．これは $N(\mu, \sigma^2)$ の正規分布

$$f(x) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

を，**標準化変数** $x' = \frac{x-\mu}{\sigma}$ を用いて書き換えたもの ($f(x')$) と見ることができる．

II. **誤差関数**（ガウスの誤差積分）

標準正規分布 $\phi(x)$ に従う事象において，ある z に対して $x \leq z$ の範囲の標本が得られる確率は，$\phi(x)$ の累積分布関数

$$\Phi(z) = \int_{-\infty}^{z} \phi(x)\,dx$$

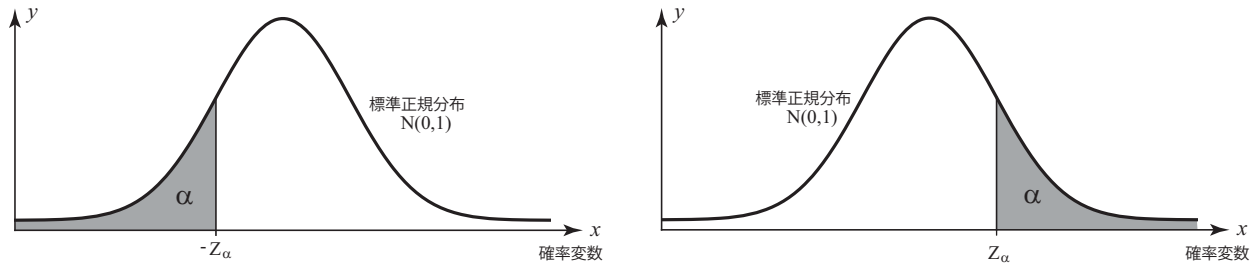
を用いて表される．これを**誤差関数**（ガウスの誤差積分）と呼び，$erf(z)$ と記す．

III. **パーセント点**[95]

$\phi(x)$ に従う事象において，ある x よりも大きな確率変数に対応する標本の発生確率が α であるとすると，その x の点を Z_α と書き，これを「発生確率が α の**パーセント点**」と呼ぶ．（図 24）

α と Z_α の間には $\alpha = 1 - \Phi(Z_\alpha)$ という関係があり，これを解くことでパーセント点を求めることができる．

[95]「A.1.5 分位数，パーセント点」（p.149）で述べたものとは別の定義であることに留意されたい．



(a) 下側確率　　(b) 上側確率

図 24: 発生確率が α のパーセント点 Z_α

【信頼区間の算出】

考え方 1.

パーセント点の考え方を応用して信頼区間を算出することができる．例えば正規母集団（簡単のため，標準正規分布に従うとする）から採取した標本が，平均値 0 を中心とする信頼係数 $1-\alpha$ の範囲内にある区間は図 25 における $\left[-Z_{\frac{\alpha}{2}}, Z_{\frac{\alpha}{2}}\right]$ である．

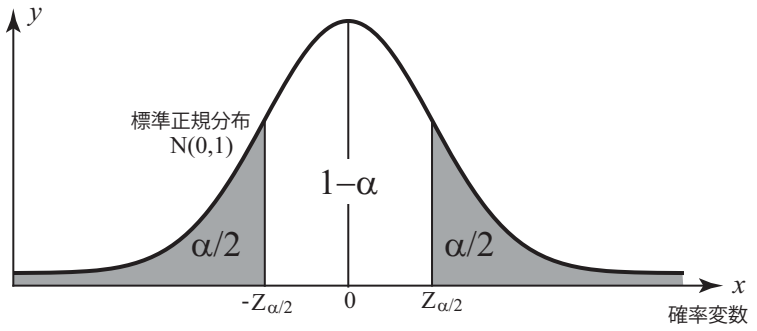


図 25: 信頼区間の作り方

考え方 2.

正規母集団 $N(\mu, \sigma^2)$ から n 個の標本を取り出したときの標本平均を $\bar{X}$ とする．この $\bar{X}$ は本当の平均値（母平均）μ からある誤差を持って離れており，$\bar{X}$ は別の正規分布 $N\left(\mu, \dfrac{\sigma^2}{n}\right)$ に従う.[96]　このことから，$\bar{X}$ が信頼係数 $1-\alpha$ で $N\left(\mu, \dfrac{\sigma^2}{n}\right)$ に重なる確率は（標準化された形で）

$$P\left(-Z_{\frac{\alpha}{2}} \leq \frac{\sqrt{n}(\bar{X}-\mu)}{\sigma} \leq Z_{\frac{\alpha}{2}}\right) = 1-\alpha$$

と表現される．更に μ が明にわかる形に変形すると，

$$P\left(\bar{X} - Z_{\frac{\alpha}{2}} \cdot \frac{\sigma}{\sqrt{n}} \leq \mu \leq \bar{X} + Z_{\frac{\alpha}{2}} \cdot \frac{\sigma}{\sqrt{n}}\right) = 1-\alpha$$

となり，信頼区間は，

$$\left[\bar{X} - Z_{\frac{\alpha}{2}} \cdot \frac{\sigma}{\sqrt{n}}, \bar{X} + Z_{\frac{\alpha}{2}} \cdot \frac{\sigma}{\sqrt{n}}\right]$$

となる．

考え方 3.

母集団が持つ真の分散（母分散）は未知であることが多く（従って母集団の標準偏差 σ も未知），上で示した信頼区間の式をそのまま用いることができる場合は少ない．そこで実際の統計調査では標本から得られた**不偏分散** s^2 を何らかの形で利用することになる．ただし，標本の不偏分散をそのまま母分散とするべきではなく，ここで t **統計量**を導入して信頼区間を決める．

[96]「A.2 中心極限定理と正規分布」を参照のこと．

■ **t 統計量，t 分布，標準誤差について**

n 個の標本 X の分布が正規分布 $N(\mu, \sigma^2)$ に従うとする．このとき，

$$Z = \frac{\bar{X} - \mu}{\sigma/\sqrt{n}}$$

は標準正規分布 $N(0,1)$ に従う．この Z の σ を s で置き換えた **t 統計量**

$$t = \frac{\bar{X} - \mu}{s/\sqrt{n}}$$

を導入する．t 統計量は **t 分布**に従う．

● **t 分布**

$N(0,1)$ に従う Z と $\chi^2(k)$ に従う Y があり[97]，Z と Y は独立であるとするとき，次の t が従う分布を**自由度 k の t 分布**[98] といい，$t(k)$ と書く．

$$t = \frac{Z}{\sqrt{Y/k}}$$

先の t 統計量 $t = \dfrac{\bar{X} - \mu}{s/\sqrt{n}}$ は自由度 $n-1$ の t 分布 $t(n-1)$ に従う．

また，t 統計量の分母である $s/\sqrt{n}$（$\bar{X}$ の標準偏差）を「標本平均の**標準誤差**」という．

t 分布は標本数が小さい場合に「正規分布の代用品」として使用されることがある．すなわち，抽出した標本数が n の場合，標本平均 $\bar{X}$ は自由度 $n-1$ の t 分布 $t(n-1)$ に従うとする．

t 分布のパーセント点も正規分布のパーセント点と同様に定義され，$t_\alpha(k)$ と書く．実際の統計調査では，t 分布のパーセント点を用いて，母数の信頼区間を

$$\left[\bar{X} - t_{\frac{\alpha}{2}}(n-1) \cdot \frac{s}{\sqrt{n}},\ \bar{X} + t_{\frac{\alpha}{2}}(n-1) \cdot \frac{s}{\sqrt{n}}\right]$$

とすることがある．

参考）実際の統計調査においては信頼係数は 99%（$1-\alpha = 0.99$） や 95% （$1-\alpha = 0.95$） といった値が採用されることが多い．

A.2.3 χ^2 分布

$N(0,1)$ に従う k 個の独立な確率変数 $Z_1, Z_2, \cdots, Z_k$ があるとき，

$$\chi^2 = Z_1^2 + Z_2^2 + \cdots + Z_k^2$$

が従う分布を **χ^2 分布**という．特に変数の個数 k によって「**自由度が k の χ^2 分布である**」という．
確率変数 x に対する χ^2 分布の確率密度関数 $f(x)$ は，ガンマ関数を用いて次のように定義される．

$$f(x) = \frac{x^{\frac{k}{2}-1}}{2^{\frac{k}{2}}\,\Gamma\left(\frac{k}{2}\right)} \exp\left(-\frac{x}{2}\right)$$

自由度 k の χ^2 分布を $\chi^2(k)$ と書く．

A.2.4 t 分布（スチューデントの t 分布）

自由度 k の t 分布の確率密度関数 $t(x)$ 定義は次の通り．

$$t(x) = \frac{\Gamma\left(\frac{k+1}{2}\right)}{\sqrt{k\,\pi}\,\Gamma\left(\frac{k}{2}\right)} \left(1 + \frac{x^2}{k}\right)^{-\frac{k+1}{2}}$$

[97]「A.2.3 χ^2 分布」を参照のこと．
[98]「A.2.4 t 分布」を参照のこと．

A.3 仮説検定

仮説検定[99]（hypothesis testing）とは，立てた仮説の真偽を統計的手法で評価することを意味する．ここでいう仮説検定は論理学的な証明とは異なり，“確からしさ”や “疑わしさ” ということを統計調査を元にして，ある規準（**有意水準**）を設けて判定する行為を指す．

例. コインを 20 回投げる試行における表が出る回数に関する仮説検定[100]

コインを 20 回投げて，表側が 14 回出たとする．このとき「表と裏の出る確率は等しくない」という仮説を統計的に検定するために,「表と裏の出る確率は等しい」という仮説を設定し，この仮説を統計的データを用いて**棄却**（reject）するという方法（背理法）を取ることにする．このように，棄却したい仮説を**帰無仮説**（null hypothesis）と呼び，それに対する仮説を**対立仮説**（alternative hypothesis）と呼ぶ．帰無仮説，対立仮説という名称には特に意味はなく，検定作業をする状況に応じて設定される「互いに逆になっている命題」である．

今回の例では,

帰無仮説：「表と裏の出る確率は等しい」

対立仮説：「表と裏の出る確率は等しくない」

と設定する．また今回の試行は「確率 p の事象が n 回の試行で x 回発生する」と言い換えることができ，この x は二項分布 $Bi(n,p)$ に従う．この分布の確率密度とその累積値を表 21 に示す．これを元に検定の作業を進める．

表 21: $Bi(20,1/2)$ の分布

x	値	累積値
0	9.54×10^{-7}	9.54×10^{-7}
1	1.91×10^{-5}	2.00×10^{-5}
2	0.000181198	0.000201225
3	0.001087189	0.001288414
4	0.004620552	0.005908966
5	0.014785767	0.020694733
6	0.036964417	0.057659149
7	0.073928833	0.131587982
8	0.120134354	0.251722336
9	0.160179138	0.411901474
10	0.176197052	0.588098526
11	0.160179138	0.748277664
12	0.120134354	0.868412018
13	0.073928833	0.942340851
14	0.036964417	0.979305267
15	0.014785767	0.994091034
16	0.004620552	0.998711586
17	0.001087189	0.999798775
18	0.000181198	0.999979973
19	1.91×10^{-5}	0.999999046
20	9.54×10^{-7}	1

$Bi(20,1/2)$ の分布において，コインの表側が 14 回以上出る確率 $P(x \geq 14)$ を表 21 から求める．コインの表側が 13 回以下の回数で出る確率 $P(x \leq 13)$ は 0.942340851 であることから，$P(x \geq 14)$ は $1 - 0.942340851 = 0.057659149$ となる．これは発生確率としては低いと見做され，今回の試行のように，コインを 20 回投げて表側が 14 回出たことで,「表と裏の出る確率は等しい」（$p = 1/2$）という仮説は棄却され，逆の対立仮説が**採択**（accept）される．

A.3.1 有意水準と誤りについて

棄却／採択のための規準を**有意水準**（significance level）といい，通常，記号 α で記される．先の例において $\alpha = 0.1$ と設定すると，0.057659149 という数値は α を下回っており，帰無仮説が棄却されるが，$\alpha = 0.01$ と設定すると帰無仮説は棄却されない.（帰無仮説のずれが有意ではないとする）

[99] より厳密に**統計的仮説検定**ともいう.
[100] この例は参考文献 [1] からの引用である.

有意水準は統計調査の意向（厳格さなど）によって設定されるが，この設定が適切でないと，棄却／採択の判定に**誤り**が起こる．この場合の誤りには 2 種類のものがあり,「帰無仮説として設定された命題が正しいにもかかわらず棄却してしまう」誤りを**第一種の誤り**,「誤った帰無仮説を採択してしまう」誤りを**第二種の誤り**という．

A.3.2　母平均に関する検定（t 検定）

実際の統計調査では十分に大きな標本数が得られないことが多く，母数が従う分布として正規分布の替わりに t 分布を用いることがあると先に述べた．ここでは t 統計量を使用して母平均に関して検定する t **検定**について，例を挙げて説明する．

例. $N(\mu, \sigma^2)$ に従う正規母集団がある．ここから $n = 25$ の数の標本を抽出して $\bar{X} = 13.7$，$s = 2.3$ を得た．この状況で次のような仮説を立てて検定する．

帰無仮説 H_0：「母平均は 15 である」（ $\mu = 15$ ）
対立仮説 H_1：「母平均は 15 でない」（ $\mu \neq 15$ ）

有意水準は $\alpha = 0.05$ とする.[101]

まず，$\mu = 15$ とする t 統計量を求めると，

$$t = \frac{13.7 - 15}{2.3/\sqrt{25}} = -2.826\cdots$$

となる．この例の対立仮説は $\mu < 15 \vee 15 < \mu$ と同値なので，$\alpha = 0.05$ となる $t(24)$ のパーセント点を求めると，

$$t_{\frac{0.05}{2}}(24) = 2.06389856\cdots$$

となり，先の t 統計量の絶対値はこの値を越えるので，帰無仮説は棄却される．

この検定では，求めた t 統計量が図 26 の t 分布における両端の部分（グレーの部分）に属していることから帰無仮説を棄却したということができる．この図のグレーの部分を**棄却域**（rejection region）といい，それ以外の部分を**採択域**（acceptance region）という．

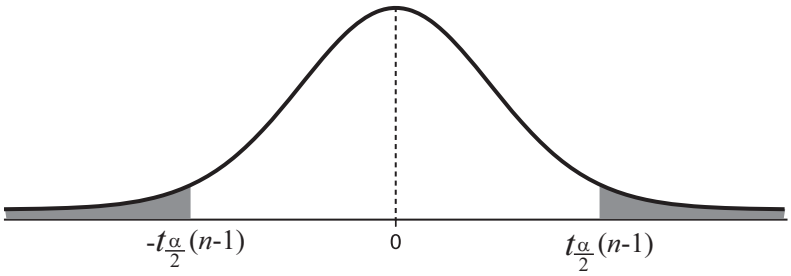


図 26: 両側検定の棄却域（t 検定）

両側検定

ここに示した t 検定では，帰無仮説の t 統計量が t 分布のパーセント点より遠い領域（0 と比べて）にあることで棄却する．また μ を別の値に設定する帰無仮説を立てた際，その t 統計量が t 分布のパーセント点より原点に近い場合は帰無仮説を採択することになる．このように，t 分布における両端の部分（グレーの部分）に属していることで帰無仮説を棄却する検定方法を**両側検定**（two-sided test）という．今回設定した対立仮説のように $\mu \neq 15$ という命題は，$\mu < 15$ という命題と $\mu > 15$ という命題の論理和であり，両側検定を適用するケースである．また今回のような対立仮説を**両側対立仮説**（two-sided alternative hypothesis）という．

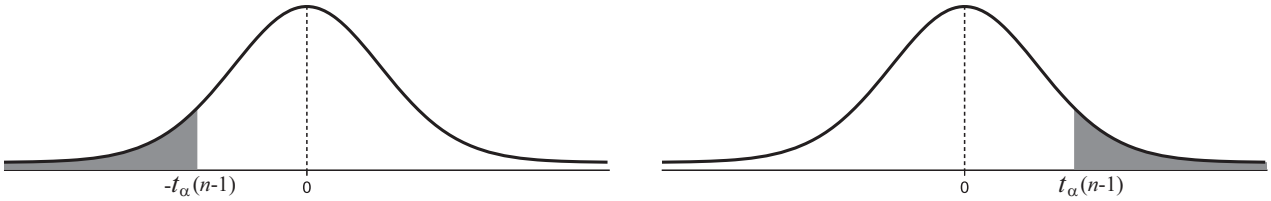
片側検定

帰無仮説 $\mu = 15$ に対して対立仮説 $\mu < 15$ を設定して検定すると，棄却域は図 27 の (a) の領域となる．この場合のパーセント点を求めると，

$$t_{0.05}(24) = 1.710882\cdots$$

[101] この例は参考文献 [1] からの引用である．

となり，この値は未だ帰無仮説の t 統計量の絶対値より大きい，従ってこの設定でも帰無仮説は棄却される．



(a) 左側対立仮説における棄却域　　(b) 右側対立仮説における棄却域

図 27: 片側検定の棄却域

次に，対立仮説として $\mu > 15$ を設定して検定すると棄却域は図 27 の (b) の領域となる．この場合は帰無仮説の t 統計量は採択域に入り，帰無仮説は棄却されない．この結論は「$\mu > 15$ という仮説よりは $\mu = 15$ という仮説の方がより確からしい」と解釈される．

この例では，$\mu = 15$ に対して，$\mu < 15$ や $\mu > 15$ といった対立仮説を立てたが，これらをそれぞれ**片側対立仮説**（one-sided alternative hypothesis）といい，これによる検定を**片側検定**という．

片側検定においては，帰無仮説と対立仮説は論理的には互いに逆の関係になっていないことがあるので注意しなければならない．

A.3.3　母分散に関する検定（χ^2 検定）

母分散 σ^2 がある値 σ_0^2 と等しいかどうかを検定するには，帰無仮説 $\sigma^2 = \sigma_0^2$ に関する下記の統計量を用いる．

$$\chi^2 = \frac{(n-1)\,s^2}{\sigma_0^2}$$

すなわち，これが標本分散 s^2 によって自由度 $(n-1)$ の χ^2 分布に従うという性質を利用する．

【手順】

1. 有意水準 α を定める．

2. χ^2 分布のパーセント点を求めて棄却か採択かを判定する．

両側検定の場合は $\chi^2_{1-\frac{\alpha}{2}}(n-1)$ と $\chi^2_{\frac{\alpha}{2}}(n-1)$ の 2 つのパーセント点の値を求め，

$$\chi^2_{1-\frac{\alpha}{2}}(n-1) < \chi^2 < \chi^2_{\frac{\alpha}{2}}(n-1)$$

の場合は帰無仮説を採択し，そうでなければ棄却する．

対立仮説が $\sigma^2 > \sigma_0^2$ の場合は $\chi^2_\alpha(n-1) \leq \chi^2$ による**右片側検定**で帰無仮説を棄却し，対立仮説が $\sigma^2 < \sigma_0^2$ の場合は $\chi^2 \leq \chi^2_{1-\alpha}(n-1)$ による**左片側検定**で帰無仮説を棄却する．

このような検定方法を χ^2 **検定**という．

例. ある学力考査では，例年平均点は 50 点，分散 36 であった．本年度もこの学力考査を実施し，ランダムサンプリングで受験者 25 人の成績を抽出したところ，平均点は 53 点，分散 48 が得られた．本年度の受験者の学力の散らばりは例年よりも大きいと見るべきかどうかを

帰無仮説 H_0 : $\sigma^2 = 36$
帰無仮説 H_1 : $\sigma^2 \neq 36$
有意水準 $\alpha = 0.1$

として検定する．

χ^2 統計量を算出すると，

$$\chi^2 = \frac{(25-1)\cdot 48}{36} = 32$$

となる．次に χ^2 分布のパーセント点を求めると，

$$\chi^2_{0.05}(24) = 13.8484\cdots$$
$$\chi^2_{0.95}(24) = 36.4150\cdots$$

となる．(図 28 参照)

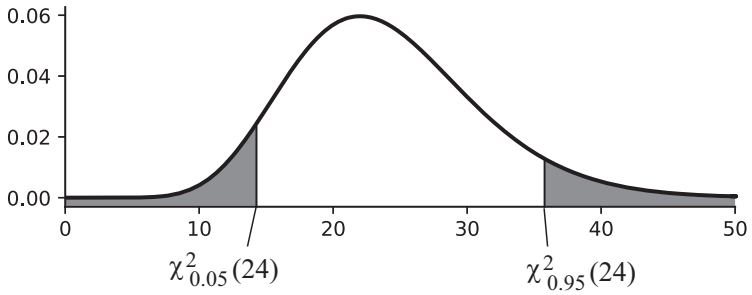


図 28: 両側検定の棄却域（χ^2 検定）

従って，帰無仮説の χ^2 統計量は採択域にあり，棄却されない．

B　サンプルプログラム

B.1　疑似データセットの作成

以下に，本書で使用したサンプルデータ Parquet02.parquet を作成するプログラムを示す.

例. ライブラリ読込みなどの準備

入力：
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import japanize_matplotlib     # matplotlib で日本語を有効にする
```

これで，必要なライブラリの読込と，matplotlib での日本語フォントの使用を有効化する．japanize_matplotlib は事前にインストールしておく必要があり，pip で導入する場合は，OS のコマンド作業で例えば

```
py -m pip install japanize-matplotlib
```
（Windows 用 PSF 版 Python の場合）

などとする.

性別毎の年齢の分布の疑似データを次のようにして作成する.

例. 年齢の分布の疑似データの作成（先の例の続き）

入力：
```
rng = np.random.default_rng(2025)     # RNG 作成
#--- 男性 ---
r1_m = rng.normal(77, 4.0, size=730)
r2_m = rng.normal(53, 4.0, size=1000)  # 男性ピーク
# ベースは mode=60 で少し高齢側に寄せるが，比率を下げる
base_m = rng.triangular(-35, 60, 105, size=3600)
population_m = np.concatenate([r1_m, r2_m, base_m])     # データを合併
population_m = population_m[(population_m >= 0) & (population_m <= 100)] # 年齢範囲制限
rng.shuffle(population_m)     # シャッフル
print('男性人数 :',len( population_m ))
#--- 女性 ---
r1_w = rng.normal(77, 4.0, size=1000)  # 女性は 77 歳をメインに
r2_w = rng.normal(53, 4.0, size=680)
# base の mode を 60 程度に留める
base_w = rng.triangular(left=-35, mode=60, right=110, size=3672)
population_w = np.concatenate([r1_w, r2_w, base_w])     # データを合併
population_w = population_w[(population_w >= 0) & (population_w <= 100)] # 年齢範囲制限
rng.shuffle(population_w)     # シャッフル
print('女性人数 :',len( population_w ))
```

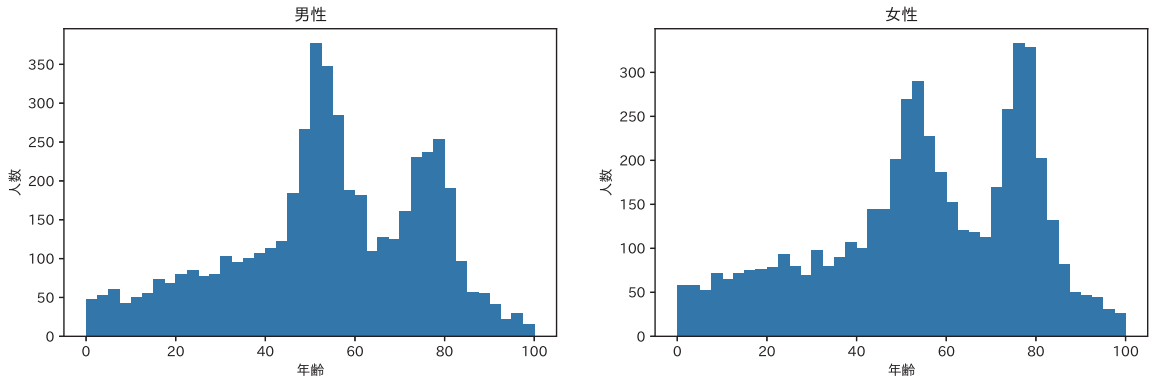
出力：
```
男性人数 :  5000
女性人数 :  5000
```

この処理で得られたデータ population_w，population_m をヒストグラムで確認する.（次の例）

例. 上記で得られたデータのヒストグラム作成（先の例の続き）

入力：
```
fig,ax = plt.subplots( 1,2, figsize=(14,4) )
ax[0].hist( population_m, bins=40 )
ax[0].set_title('男性')
ax[0].set_xlabel('年齢')
ax[0].set_ylabel('人数')
ax[1].hist( population_w, bins=40 )
ax[1].set_title('女性')
ax[1].set_xlabel('年齢')
ax[1].set_ylabel('人数')
```

出力：



性別，都市のラベル（質的データ）を次のようにして作成する．

例. 性別ラベルの作成（先の例の続き）

入力：
```
gender = ['男']*len(population_m)+['女']*len(population_w)  # リストとして生成
gender = np.array(gender,dtype='object')     # NumPy 配列に変換
rng.shuffle(gender)     # シャッフル
```

例. 都市ラベルの作成（先の例の続き）

入力：
```
p = len(population_m) + len(population_w)     # 全データ数
n = np.ceil(p/4).astype(np.int64)
city = ['東京']*n+['大阪']*n+['愛知']*n+['福岡']*n   # リストとして生成
city = np.array(city,dtype='object')     # NumPy 配列に変換
rng.shuffle(city)     # シャッフル
city = city[:p]       # データ数調整
```

ここまでに得られたデータを次のようにして DataFrame に統合する．

例. DataFrame に統合（先の例の続き）

入力：
```
# 年齢データ統合
age = np.zeros(p).astype(np.int64)      # ベース作成
age[gender=='男'] = population_m        # 「男性」の部分の設定
age[gender=='女'] = population_w        # 「女性」の部分の設定
# DataFrame 作成
df = pd.DataFrame(
    'city'   :  city,
    'gender' :  gender,
    'age'    :  age
)
```

入力：
```
# 内容確認
pd.set_option('display.min_rows',8)
df
```

出力：

	city	gender	age
0	大阪	男	30
1	愛知	男	79
2	福岡	男	78
3	東京	男	51
...	...	...	...
9996	大阪	男	74
9997	愛知	男	8
9998	東京	女	78
9999	愛知	女	71

10000 rows × 3 columns

得られた DataFrame を次のようにして Parquet ファイルに保存する.

例. Parquet ファイルに保存（先の例の続き）

入力：
```
# Parquet ファイルに保存
df.to_parquet('Parquet02.parquet')
```

▲注意▲

ここで解説した方法で作成されたデータはあくまで「疑似データ」であり，実際の日本国民の状況を反映しているわけではないことに注意されたい.

参考文献

[1] 松原 望, 縄田 和満, 中井検裕,
“「統計学入門」東京大学教養学部統計学教室 編”, 東京大学出版会, 1991

[2] 中村 勝則,
“「Python3 入門」 − Kivy による GUI アプリケーション開発，サウンド入出力，ウェブスクレイピング”,
IDEJ 出版 ISBN978-4-9910291-0-3 C3004, 2019 年 3 月 14 日
（PDF を http://www.k-techlabo.org/www_python/python_main.pdf で公開中）

[3] 中村 勝則,
「Python3 ライブラリブック」各種ライブラリの基本的な使用方法, 2019 年
（PDF を http://www.k-techlabo.org/www_python/python_modules.pdf で公開中）

索 引

「Python3 によるデータ処理の基礎」

著者：中村勝則

発行：2026 年 2 月 12 日

テキストの最新版と更新情報

本書の最新版と更新情報を，プログラミングに関する情報コミュニティ Qiita で配信しています．

→ https://qiita.com/KatsunoriNakamura/items/cf1664da8d891bc3c4bf

上記 URL の QR コード

● 連絡先

nkatsu2012@gmail.com

中村勝則