

Python3による データ処理の基礎

pandas / NumPy / SciPy / matplotlib /
SQLite / SQLAlchemy

第 0.4.04 版

Copyright © 2019-2021, Katsunori Nakamura

中村勝則

2021 年 09 月 10 日

免責事項

本書の内容は参考資料であり、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	1
1.1	本書を読むに当たって	1
1.2	作業環境について	1
2	データ構造	1
2.1	Series (pandas のデータ構造)	2
2.1.1	インデックスに基づくアクセス	3
2.1.1.1	ドット '.' 表記によるアクセス	4
2.1.2	格納順位に基づくアクセス	5
2.1.3	スライスを用いたアクセス方法	5
2.1.3.1	スライスに整数値を与える場合	5
2.1.3.2	スライスに非整数のインデックス項目を与える場合	6
2.1.4	要素の抽出	7
2.1.4.1	条件式から真理値の列を生成する方法	7
2.1.5	重複するインデックスを持つ Series の扱い	8
2.1.6	Series から ndarray への変換	9
2.1.6.1	データとしてのインデックス	9
2.1.6.2	インデックスの検索	10
2.1.7	ndarray から Series への変換	11
2.1.8	整列 (ソート)	12
2.1.8.1	整列順序の指定	12
2.1.8.2	インデックスに沿った整列	13
2.1.9	要素の削除	13
2.1.9.1	drop メソッド	13
2.1.9.2	del 文による削除	15
2.1.10	Series オブジェクトの変更と複製について	15
2.1.11	Series オブジェクトの連結	16
2.1.12	インデックスの再設定	17
2.1.12.1	reset_index メソッドによる方法	17
2.1.12.2	与えた項目列でインデックスで置き換える方法	18
2.1.13	マルチインデックス	18
2.1.13.1	インデックスレベルを指定した整列	19
2.1.13.2	インデックスレベルへの名前の付与	19
2.1.14	その他	20
2.1.14.1	開始部分/終了部分の取り出し	20
2.2	DataFrame (pandas のデータ構造)	21
2.2.1	DataFrame の生成	21
2.2.1.1	リストから DataFrame を生成する方法	21
2.2.1.2	NumPy の配列 (ndarray) から DataFrame を生成する方法	22
2.2.1.3	辞書から DataFrame を生成する方法	22
2.2.2	DataFrame の要素にアクセスする方法	23
2.2.2.1	at によるアクセス	23
2.2.2.2	NaN (欠損値) について	23
2.2.2.3	iat によるアクセス	24
2.2.2.4	loc による方法	25
2.2.2.5	iloc による方法	26

2.2.2.6	列（カラム）の取出しと追加	27
2.2.2.7	ドット‘.’表記による列（カラム）へのアクセス	29
2.2.2.8	行の取出しと追加	30
2.2.2.9	DataFrame を NumPy の配列（ndarray）に変換する方法	31
2.2.2.10	データとしてのインデックスとカラム	32
2.2.2.11	データの格納位置の調査	33
2.2.3	整列（ソート）	33
2.2.3.1	整列順序の指定	34
2.2.3.2	インデックスに沿った整列	34
2.2.4	行、列の削除	34
2.2.4.1	カラムの抹消	36
2.2.5	DataFrame の連結	36
2.2.5.1	最も単純な連結処理	36
2.2.5.2	横方向（カラム方向）の連結	37
2.2.6	指定した条件によるデータの抽出	38
2.2.6.1	真理値を用いた抽出	38
2.2.6.2	条件式から真理値列を生成する方法	38
2.2.6.3	論理演算子による条件式の結合	39
2.2.7	DataFrame に関する情報の取得	40
2.2.7.1	要約統計量	40
2.2.7.2	データ構造に関する情報の表示	40
2.2.8	その他	41
2.2.8.1	開始部分/終了部分の取り出し	41
2.2.8.2	行と列の転置	41
2.3	日付と時刻	42
2.3.1	Timestamp クラス	42
2.3.1.1	タイムゾーン	43
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	44
2.3.1.3	現在時刻の取得	44
2.3.2	Timestamp の差：Timedelta	44
2.3.2.1	Timedelta の生成	45
2.3.3	Timestamp の列：date_range と DatetimeIndex	46
2.3.3.1	他の型への変換	46
2.3.3.2	頻度の規則	46
2.4	データ集合に対する一括処理	48
3	ファイル入出力	50
3.1	DataFrame を CSV ファイルとして保存する方法	50
3.2	CSV ファイルを読み込んで DataFrame にする方法	51
3.2.1	CSV の指定した列をインデックスと見なす方法	51
3.2.2	CSV の先頭行をデータと見なす方法	52
4	統計処理のための基本的な操作	53
4.1	基本的なソフトウェアライブラリ	53
4.1.1	ライブラリの読み込み	53
4.2	サンプルデータの作成	53
4.3	データの分析	54
4.3.1	パーセンタイル（パーセント点）	55

4.3.1.1	中央値	55
4.3.2	基本的な統計量	55
4.3.2.1	最大値/最小値	55
4.3.2.2	合計	56
4.3.2.3	平均	56
4.3.2.4	分散（不偏分散/標本分散）	56
4.3.2.5	標準偏差（不偏標準偏差/標本標準偏差）	57
4.3.2.6	尖度/歪度	57
4.3.2.7	配列（ndarray）の統計量の算出	58
4.3.3	要素/区間毎のデータ個数の調査	59
4.3.3.1	要素の個数の調査	59
4.3.3.2	最頻値	59
4.3.3.3	区間毎の要素の個数の調査	60
4.3.3.4	最頻の区間	61
4.3.3.5	区間（Interval オブジェクト）	61
4.4	データの可視化	62
4.4.1	matplotlib による作図の手順	62
4.4.2	ヒストグラムの作成	62
4.4.3	図を画像ファイルとして保存する方法	63
4.4.4	複数の図を重ねて表示する方法	64
4.4.5	箱ひげ図の作成	64
4.4.6	折れ線グラフを作成する方法	65
4.4.7	グラフ描画に関する各種の設定	65
4.4.8	応用例：value_counts の結果をヒストグラムにする	65
4.5	集計処理	67
4.5.1	グループ集計	67
4.5.2	クロス集計	68
4.5.3	ダミー変数の取得（ワンホットエンコーディング）	70
4.6	ランダムサンプリング, シャッフル	71
4.7	変数間の関係の調査	72
4.7.1	相関係数	72
4.7.2	共分散	73
4.7.3	多項式回帰	74
4.8	統計検定	76
4.8.1	z 検定	76
4.8.2	t 検定	78
5	データベース	80
5.1	データベースについての基本的な考え方	80
5.1.1	データベースに対する基本的な操作	80
5.2	本書で取り扱うデータベース関連のソフトウェア	81
5.2.1	SQLite	81
5.2.2	SQLAlchemy	81
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	81
5.3.1	サンプルデータの作成	81
5.3.2	Engine オブジェクトの生成	82
5.3.3	SQL クエリの表示	82
5.3.4	DataFrame のテーブルへの新規保存	82

5.3.5	テーブルから DataFrame への読み込み	83
5.3.6	既存のテーブルへの追加保存	83
5.3.7	既存のテーブルを新しいデータで置き換える	84
5.3.8	指定した条件によるデータの抽出	84
5.4	データベースに対するアクセスの例 (2): トランザクション処理	85
5.4.1	トランザクションの開始	85
5.4.2	既存のレコードの変更 (データベースの更新)	85
5.4.3	既存のレコードの削除	86
5.4.4	新規レコードの追加	86
5.4.5	execute メソッドの戻り値	87
5.4.6	トランザクションの終了	87
5.5	SQL	87
5.5.1	データベースの作成	88
5.5.2	テーブルの作成	88
5.5.2.1	カラムのデータ型	88
5.5.3	テーブルの結合	90
6	各種ライブラリが提供する関数やメソッド	92
6.1	scipy.special	92
6.1.1	階乗 $n!$	92
6.1.2	順列 ${}_nP_r$, 組合せ ${}_nC_r$	93
6.2	scipy.stats	94
6.2.1	確率密度関数: PDF (Probability Density Function)	94
6.2.1.1	正規分布	94
6.2.1.2	t 分布	95
6.2.1.3	χ^2 分布	96
6.2.1.4	指数分布	97
6.2.1.5	対数正規分布	98
6.2.2	確率質量関数: PMF (Probability Mass Function)	99
6.2.2.1	二項分布	99
6.2.2.2	幾何分布	100
6.2.2.3	超幾何分布	101
6.2.2.4	ポアソン分布	102
6.2.3	累積分布関数: CDF (Cumulative Density Function)	103
6.2.4	パーセント点関数: PPF (Percent Point Function)	104
6.2.5	乱数生成: RVS (Random Variates)	105
6.2.5.1	一様乱数の生成	105
6.2.5.2	乱数生成の初期設定: random_state	106
A	統計学の用語	108
A.1	確率変数と確率を表す関数	108
A.1.1	確率に関する重要な事柄	108
A.1.2	確率質量関数	109
A.1.2.1	二項分布	109
A.1.2.2	幾何分布	109
A.1.2.3	超幾何分布	109
A.1.2.4	ポアソン分布	109
A.1.3	確率密度関数	109

A.1.3.1	正規分布	110
A.1.3.2	指数分布	110
A.1.3.3	対数正規分布	110
A.1.4	尖度, 歪度	110
A.2	母集団と標本に関する重要な事柄	112
A.2.1	標本の抽出	112
A.2.1.1	標本分散と不偏分散	112
A.2.2	推定	113
A.2.2.1	点推定	113
A.2.2.2	区間推定	114
A.2.3	χ^2 分布	116
A.2.4	t 分布 (スチューデントの t 分布)	116
A.3	仮説検定	117
A.3.1	有意水準と誤りについて	117
A.3.2	母平均に関する検定 (t 検定)	118
A.3.3	母分散に関する検定 (χ^2 検定)	119

1 はじめに

本書は、Python3 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。また作図処理には 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。

● 了承事項 ●

本書では実際の処理の方法について例を示しながら説明するが、乱数の生成に基づく操作例の表示に関しては、実際に実行した場合の数値と比べて若干の差異が見られる場合がある。これは、本書の各項目の編集時点で得られた実行例のハードコピーをそのまま掲載していることによるものであり、具体的な数値は利用者各位の計算機環境で表示されるものを参照のこと。

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。

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。

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データ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ある。

¹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。

²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。

³いわゆるビッグデータ

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

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。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' という名で読み込み
      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 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。

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
```

```
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。

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
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
```

2.1.1.1 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。

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。

2.1.2 格納順位に基づくアクセス

Series の要素には格納順位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: `sr.iat[4]`

出力: 25

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。

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)

2.1.3 スライスを用いたアクセス方法

2.1.3.1 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、 $i_s \sim i_e - 1$ の範囲がアクセス対象となることに注意すること。

2.1.3.2 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、 $i_s \sim i_e$ 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 要素の抽出

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
```

2.1.4.1 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) iat, iloc, at に条件式を与えることはできないが、loc には条件式を与えることができる。(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。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。

2.1.5 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⁴ 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
```

```
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。

⁴これはあまり良いことではない。

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
```

2.1.6 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。

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'>
```

2.1.6.1 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'>
```

2.1.6.2 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**⁵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が得られる。(次の例参照)

⁵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)

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.7「2.1.4 要素の抽出」）に利用できる。

2.1.7 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 の読み込みと `ndarray` の生成（先の例の続き）

```
入力: import numpy as np                # NumPy を 'np' という名で読み込み
      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⁶を参照のこと）

⁶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. 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
```

2.1.8 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⁷、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。

2.1.8.1 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）

⁷`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. 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
```

2.1.8.2 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.

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⁸, 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.

2.1.9 要素の削除

2.1.9.1 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,

⁸`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 Series オブジェクト自体を変更 (整列) する.

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
```

```
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'、'd2' 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。

2.1.9.2 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⁹ 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。

2.1.10 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。

⁹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。

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
```

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¹⁰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。ただし、`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。

2.1.11 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。

¹⁰ リストをはじめとする変更可能なデータ構造のこと。

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 srC 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。

2.1.12 インデックスの再設定

2.1.12.1 `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.21) で解説する。

2.1.12.2 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。

2.1.13 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: 3
```

2.1.13.1 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
```

```
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。

2.1.13.2 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。

2.1.14 その他

2.1.14.1 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 2 参照)

表 2: 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列,
index=インデックスとして与えるデータ列,
dtype=NumPy のデータ型)

配列データとしては, 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

[行のデータ列 1, 行のデータ列 2, 行のデータ列 3, ...]

である. キーワード引数 'dtype=' にデータ要素の型 (整数, 浮動小数点数など) を NumPy の規則に従って指定すると処理速度が向上する場合があるが, これは省略可能である.

「配列データ」を省略するか, あるいは None を与えると「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' という名で読み込み
      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  0  1  1
1  1  2  4
2  2  3  9
```

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' の名で読み込み
      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、値の部分をもっと辞書にしている。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。

2.2.2 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。

at による要素の参照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¹¹される。これは欠損値と呼ばれる。（次の例参照）

¹¹ “Not a Number” の略。

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
```

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

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

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.

2.2.2.3 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) 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。

2.2.2.4 loc による方法

loc を使用すると DataFrame 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 環境で利用できる。

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

飛び飛びの部分の抽出するには 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'>
```

2.2.2.5 iloc による方法

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「2.1.2 格納順位に基づくアクセス」（p.5）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の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。

2.2.2.6 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
      c1 = pd.Series( ['Taro','Jiro','Hanako'], index=['d1','d2','d3'])
      df2['name'] = c1      # 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 c1 を DataFrame df2 のカラム 'name' として追加する例である。c1 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     0  1  2
1     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
```

2.2.2.7 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¹² 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。（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
```

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）

¹²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。

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
```

2.2.2.8 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。

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.

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

2.2.2.9 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'>
```

2.2.2.10 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'>
```

2.2.2.11 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. 「2.1.6.2 インデックスの検索」(p.10) 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¹³。（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。

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¹⁴。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。

¹³sort_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。

¹⁴sort_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。

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、行のみを削除対処とする場合は、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。

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し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。（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。）

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、列を削除
      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 の連結

2.2.5.1 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、連結後の当該部分の値が **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
```

2.2.5.2 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)

2.2.6 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行を DataFrame から抽出する方法について説明する。

2.2.6.1 真理値を用いた抽出

DataFrame から指定した行を抽出するための最も基本的な方法に、**真理値**による抽出がある。DataFrame の各行に対応する真理値の列を用意して、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、列の中の真 (True) に対応する行が抽出される。この場合の真理値列はリスト、NumPy の ndarray、Series オブジェクトが使用できる。

これについて例を挙げて説明する。まずサンプルデータを次のようにして用意する。

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。次に、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] # 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。

2.2.6.2 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' である行を表す真理値列
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。

2.2.6.3 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 3 のような記述による。

表 3: 条件式の結合や否定のための演算子

記述	解説
<code>p1 & p2</code>	条件式 <code>p1</code> , <code>p2</code> がともに真の場合に真、それ以外は偽となる。
<code>p1 p2</code>	条件式 <code>p1</code> , <code>p2</code> 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。
<code>~p</code>	条件式 <code>p</code> が偽の場合に真、真の場合に偽となる。

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

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

```
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 DataFrame に関する情報の取得

2.2.7.1 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、describe メソッドの処理結果が、要約統計量を持つ別の DataFrame として得られる。

2.2.7.2 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.

2.2.8 その他

2.2.8.1 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
```

2.2.8.2 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
```


2.3 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 つの時点の日付と時刻で表す**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 4 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 4: 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		

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( 'sedond:',        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
      sedond: 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 5）ものである。

表 5: `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 15:14.123:31+0000')
```

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

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='tzutc()」となっている。このように、時間帯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 15:14.123:31', 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 15:14.123:31+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。

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 4 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')
```

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。

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 6 に示す。

表 6: 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

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)

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.values[: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 7 に示す.

表 7: 「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')

```

2.4 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¹⁵）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）

¹⁵ 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

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 CSV¹⁶ 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¹⁷ のデータであり、多くのソフトウェアが標準的にこ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(ファイル名, encoding=文字コード,
index=インデックス保存指定, header=カラム保存指定)

DataFrame の内容を「ファイル名」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文字コードを指定する。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。

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' という名で読み込み
      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。

¹⁶CSV (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

¹⁷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。各種の文字コードの体系で文字として規定される範囲のデータから構成される。

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。

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(ファイル名, encoding=文字コード,
index_col=インデックス列, names=カラム名の列)

「ファイル名」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 ファイルの文字コードを指定する。‘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 ファイルの先頭行をカラム名と見なす。

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 ファイルの読み込み
      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。

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 ファイルの読み込み
      df2      # 内容確認
```

```
出力：
      1  1.1  1.2
d1
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 ファイルの読み込み
      df2      # 内容確認
```

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

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¹⁸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 ライブラリの読み込み

これまでも 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み込み

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

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 の読み込みに当たって、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。(ライブラリの読み込み方法に関する詳細は Python 言語に関する他の資料¹⁹や Python の公式インターネットサイトを参照のこと)

4.2 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 ( $\mu=0, \sigma=1$ ) 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)
      # 対数正規分布 ( $\mu=0, \sigma=1$ ) 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)
```

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

¹⁸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。

¹⁹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。

列を生成する関数群を提供しており、例にある `norm.rvs` 関数は正規分布に沿った乱数列を生成する。同様に `lognorm.rvs` 関数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= $\mu$ , scale= $\sigma$ , 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= $\sigma$ , 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。

4.3 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「2.2.7.1 要約統計量」(p.40) のところで示した方法 (describe メソッド) を用いる。

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    -0.004178    1.673434
std      1.004597    2.174391
min     -3.682918    0.014451
25%     -0.681070    0.504478
50%     -0.009957    1.013518
75%      0.667842    1.991601
max      3.553150    36.786871
-----
<class 'pandas.core.frame.DataFrame'>
```

describe メソッドは、対象となる DataFrame の要約統計量 (表 8) 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 8: 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: 1.0045974037157606
```

4.3.1 パーセンタイル（パーセント点）

四分位数以外のパーセント点（パーセンタイル）を求めるには `quantile` メソッドを用いる。

《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=累積要素数の比率)`

指定した「累積要素数の比率」のパーセント点を `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「累積要素数の比率」は統計学では「 α 」の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 α とし、そのような α を与える要素の値が「パーセンタイル」（パーセント点）である。（ $0 \leq \alpha \leq 1$ ）

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      -1.649483
      LogNorm   0.189133
      Name: 0.05, dtype: float64
```

`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 -1.649483  0.189133
0.95  1.622744  5.134822
```

4.3.1.1 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      -0.009658
      LogNorm   1.014291
      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。

4.3.2 基本的な統計量

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。

4.3.2.1 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      3.619361
      LogNorm  62.155272
      dtype: float64
```

例. 最小値（先の例の続き）

```
入力: df.min()    # 最小値
```

```
出力:  Norm      -4.055946
      LogNorm    0.019120
      dtype: float64
```

4.3.2.2 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      44.318932
      LogNorm 16553.450680
      dtype: float64
```

例. 特定の列に対する合計（先の例の続き）

```
入力: df['Norm'].sum()    # 特定の列に対する合計処理
```

```
出力: 44.31893207196165
```

4.3.2.3 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      0.004432
      LogNorm  1.655345
      dtype: float64
```

例. 特定の列の平均値算出（先の例の続き）

```
入力: df['Norm'].mean()    # 特定の列の平均値算出
```

```
出力: 0.0044318932071961735
```

4.3.2.4 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      1.010548
      LogNorm  4.417688
      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: 1.010547687832727
```

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: 1.0104466330639437
```

var メソッドは 'ddof=1'（不偏分散）が暗黙値である。

4.3.2.5 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      1.00526
      LogNorm   2.10183
      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: 1.00526001006343
```

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: 1.0052097458062887
```

std メソッドは 'ddof=1'（不偏標準偏差）が暗黙値である。

4.3.2.6 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      0.001349
      LogNorm   67.286293
      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: 0.0013494374654290375
```


例. 歪度（先の例の続き）

```
入力： df.skew()      # 歪度
```

```
出力：  Norm      0.004293
      LogNorm  5.554591
      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：  5.554590753829899
```

4.3.2.7 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プロパティ values が保持しており、それを参照することでデータ列を 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'].values, 0.05 ) )      # 比率で指定
      print( np.percentile( df['Norm'].values, 5 ) )      # 百分率で指定
```

```
出力： -1.6180661298464882
      -1.6180661298464882
```

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 9 参照）

表 9: 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'].values ) )      # 中央値
      print( '最大値: ', np.max( df['Norm'].values ) )      # 最大値
      print( '最小値: ', np.min( df['Norm'].values ) )      # 最小値
      print( '合計: ', np.sum( df['Norm'].values ) )      # 合計
      print( '平均: ', np.mean( df['Norm'].values ) )      # 平均
      print( '分散: ', np.var( df['Norm'].values, ddof=1 ) )      # 不偏分散
      print( '標準偏差:', np.std( df['Norm'].values, ddof=1 ) )      # 不偏標準偏差
```

```
出力： 中央値:    0.013086767631907819
      最大値:    3.645192009575384
      最小値:   -4.19948688951108
      合計:    52.30769890030817
      平均:    0.005230769890030817
      分散:    0.9673035932152788
      標準偏差: 0.9835159343982581
```

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)

4.3.3 要素/区間毎のデータ個数の調査

4.3.3.1 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。

4.3.3.2 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)

4.3.3.3 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=10) # 度数の区間を 10 等分して度数調査
      print( B0.head(3) )                  # 先頭 3 要素を表示
      print( 'データ個数:', len(B0) )
```

```
出力: (-0.542, 0.235]    3002
      (0.235, 1.012]    2470
      (-1.319, -0.542]  2085
      Name: Norm, dtype: int64
      -----
      データ個数: 10
```

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 10 に示すような括弧で上限と下限を括る形で表現される。

表 10: 区間の表現

下限	解説	上限	解説
'('	下限と同じ値は含まない (开区間))'	上限と同じ値は含まない (开区間)
'['	下限と同じ値を含む (闭区间)	']'	上限と同じ値を含む (闭区间)

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)

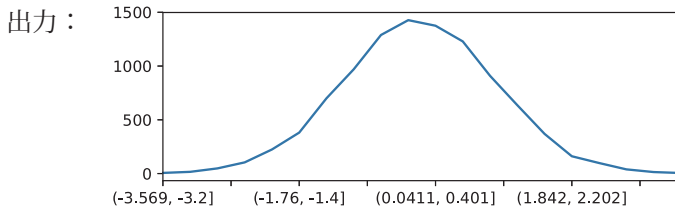
例. 度数分布のデータとして整列 (先の例の続き)

```
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' の名で読み込み
      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
```



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「4.4.2 ヒストグラムの作成」(p.62) で解説する。データを可視化する方法については後の「4.4 データの可視化」(p.62) で更に詳しく解説する。

4.3.3.4 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: 1499

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: (-0.218, 0.165] 1499
Name: Norm, dtype: int64

4.3.3.5 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.lib.interval.Interval'>
Interval(-0.542, 0.235, 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:  -0.542
      上限:   0.235
      閉区間:  right
```

closed プロパティが取る値は表 11 のようなものである。

表 11: 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）

4.4 データの可視化

4.4.1 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。

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、作図処理に必要な準備を整える。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。

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。

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²⁰を参照のこと。

4.4.2 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。

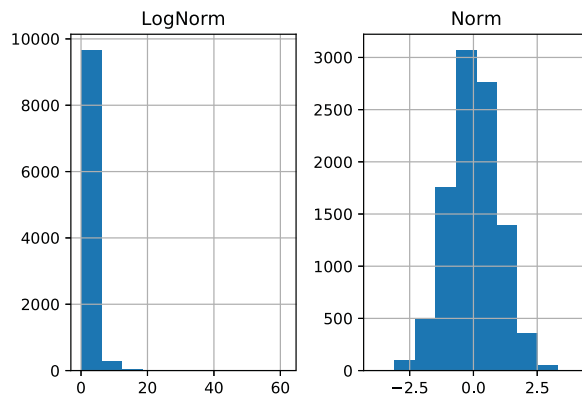
²⁰ 拙書「Python3 ライブラリブック - 各種ライブラリの基本的な使用方法」でも解説しています。

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)

例. ヒストグラムの描画 (先の例の続き)

```
入力: plt.figure()    # 描画の準備
      df.hist()       # ヒストグラムの作成処理
      plt.show()      # 描画の実行
```

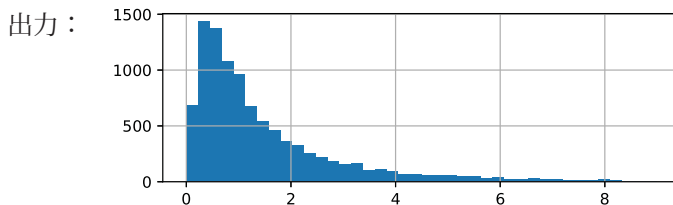
出力: <Figure size 432x288 with 0 Axes>



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: plt.figure()
      df['LogNorm'].hist( bins=40,      # 階級の数
                           range=(0,9), # 範囲
                           figsize=(6,2) ) # 図の大きさ
      plt.show()
```



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 12 参照) を施している。

表 12: 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)
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 *

* 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。

4.4.3 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: plt.figure()
      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
```

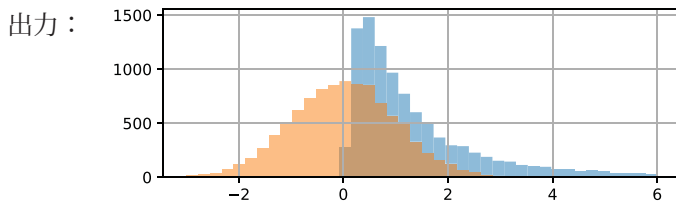
Jupyter Notebook で実行すると、画像ファイルが保存された後、Notebook 内にグラフが表示される。

4.4.4 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: plt.figure()
      df['LogNorm'].hist( bins=40,          # 階級の数
                          range=(-3,6),     # 範囲
                          alpha=0.6,        # アルファ値（不透明度）
                          figsize=(6,2) )   # 図の大きさ
      df['Norm'].hist( bins=40,            # 階級の数
                       range=(-3,6),      # 範囲
                       alpha=0.6 )        # アルファ値（不透明度）
      plt.show()
```



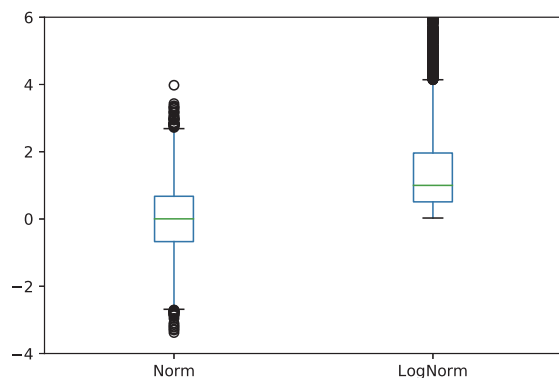
4.4.5 箱ひげ図の作成

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を描画する。

例. 箱ひげ図の作成（先の例の続き）

```
入力: plt.figure()          # 描画開始
      df.plot(kind='box')    # 箱ひげ図
      plt.ylim(-4,6)
      plt.show()
```

出力: <Figure size 432x288 with 0 Axes>



4.4.6 折れ線グラフを作成する方法

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)

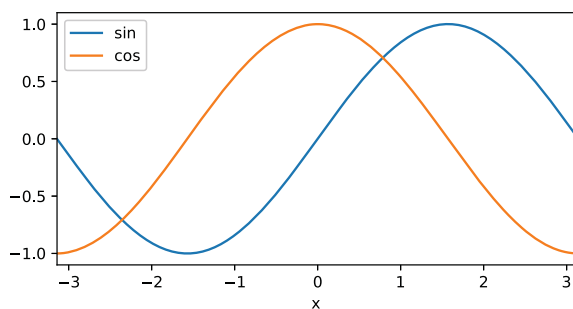
```
入力: import numpy as np                # NumPy を 'np' という名で読み込み
      x = np.arange(-3.14,3.14,0.04)    # sin,cos の定義域を生成
      y1 = np.sin(x);    y2 = np.cos(x)  # sin,cos のデータ列を生成
      df2 = pd.DataFrame(columns=['x','sin','cos']) # DataFrame の生成
      df2['x']=x;    df2['sin']=y1;    df2['cos']=y2 # 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: plt.figure()
      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
      plt.show()
```

出力: <Figure size 432x288 with 0 Axes>



plot メソッドに与えるキーワード引数の一部を表 13 に示す。

表 13: 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)

4.4.7 グラフ描画に関する各種の設定

グラフ描画に関する各種の設定を行う関数を表 14 に示す。

4.4.8 応用例: value_counts の結果をヒストグラムにする

先の「4.4.2 ヒストグラムの作成」(p.62) 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.59 「4.3.3 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。

表 14: グラフ描画に関する各種の設定を行う関数

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.

※ 関数呼出し時にはライブラリの接頭辞 (‘plt.’ など) を付ける.

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: # Interval からプロパティを取り出す関数
def getL(v): return( v.left )
def getR(v): return( v.right )
def getC(v): return( v.closed )
# Index オブジェクトの全要素を展開
ixl = map( getL, B0.index.values )
ixr = map( getR, B0.index.values )
ixc = map( getC, B0.index.values )

```

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', 'closed', 'freq'] )
# 値の設定
dfBins['left'] = list( ixl )
dfBins['right'] = list( ixr )
dfBins['closed'] = list( ixc )
dfBins['freq'] = B0.values

```

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')    # 区間の下限値で整列
      display( dfHist.head(3) )              # 確認表示:先頭 3 行

```

```

出力:
   left  right  closed  freq
0 -4.237 -3.831   right    2
1 -3.831 -3.434   right    1
2 -3.434 -3.036   right   13

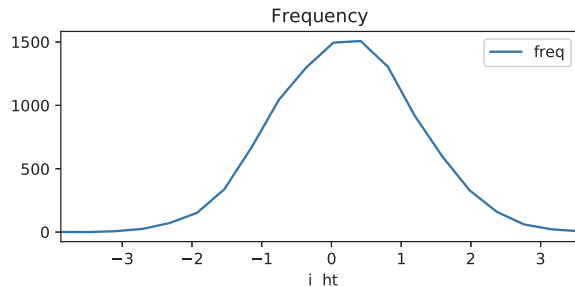
```

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: plt.figure()
      dfHist.plot(x='right',y='freq')
      plt.title('Frequency')
      plt.show()
```

出力: <Figure size 432x288 with 0 Axes>



4.5 集計処理

4.5.1 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：ライブラリの読み込みとデータ列の生成

```
入力: import pandas as pd          # pandas を pd の名で読み込み
      from scipy import stats      # scipy.stats の読み込み
      import numpy as np          # NumPy の np の名で読み込み
```

```
入力: # テストデータの作成
      rA = stats.norm.rvs( loc=-3, scale=3, size=10000 )    #  $\mu=-3, \sigma=3$ 
      rB = stats.norm.rvs( loc=0, scale=1, size=10000 )     #  $\mu=0, \sigma=1$ 
      rC = stats.norm.rvs( loc=1, scale=0.5, size=10000 )   #  $\mu=1, \sigma=0.5$ 
```

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: # DataFrame にする
      df = pd.DataFrame(None, 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).reset_index(drop=True)
      dfR.head(4)
```

出力:

	種類	値
0	C	0.903994
1	B	-0.346105
2	C	0.938629
3	A	-2.387583

'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²¹ 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.mean() )
      print( '標準偏差:', dfR.std() )
```

```
出力: 平均: 値          -0.66558
      dtype: float64
      標準偏差: 値       2.506053
      dtype: float64
```

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() )
```

```
出力: <class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

種類	値
A	-2.983229
B	-0.019839
C	1.006329

種類	値
A	3.002240
B	0.996119
C	0.501850

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	-2.996943	2.979435	-14.902420	-4.992959	-2.953977	-1.002697	8.905233
B	10000.0	-0.007944	0.992593	-3.859665	-0.689127	-0.003120	0.659043	4.001650
C	10000.0	0.998092	0.499101	-0.880872	0.669355	1.002958	1.328639	2.840786

4.5.2 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 pivot_table メソッドを使用する。ここでは作業例を挙げてクロス集計の方法²² について説明する。

²¹ 「4.6 ランダムサンプリング, シャッフル」(p.71) で解説する。

²² 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。

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
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.shuffle(p) # '出身地' の列をシャッフル
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
dtab['年齢'] = stats.norm.rvs(loc=35,scale=10,size=300).astype('uint32')
dtab.head(5)
```

```
出力：   出身地  年齢  好物
0   東京   22  肉うどん
1   東京   40  肉うどん
2   大阪   44  カレーライス
3   京都   41  肉うどん
4   京都   34  お好み焼き
```

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。

次に pivot_table メソッドについて説明する。

《DataFrame のクロス集計》

```
pivot_table( index=クロス表のインデックスとするためのカラム,
              columns=クロス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、列ラベルとするクロス表を作成する。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’ のクロス表にする例を示す。

例. 年齢の最大値のクロス集計（先の例の続き）

```
入力： cr = dtab.pivot_table(index='出身地',columns='好物',values='年齢',
                              aggfunc='max',margins=True)
cr
```

```
出力：   好物  おでん  お好み焼き  カレーライス  肉うどん  A11
出身地
京都      49      58          52          52      58
大阪      60      57          47          42      60
東京      50      65          54          46      65
A11      60      65          54          52      65
```

例. データ個数のクロス集計（先の例の続き）

```
入力: cr = dtab.pivot_table(index='出身地',columns='好物',values='年齢',
                             aggfunc='count',margins=True)
cr
```

```
出力: 好物    おでん    お好み焼き    カレーライス    肉うどん    All
出身地
京都      12      32      37      19    100
大阪      21      24      35      20    100
東京      17      24      48      11    100
All       50      80     120     50    300
```

4.5.3 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）が与えられる。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²³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
```

²³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。

4.6 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. ライブラリの読み込みとサンプルデータの作成

```
入力: import numpy as np
import pandas as pd
import scipy.stats as stats
```

```
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。

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。

4.7 変数間の関係の調査

4.7.1 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。

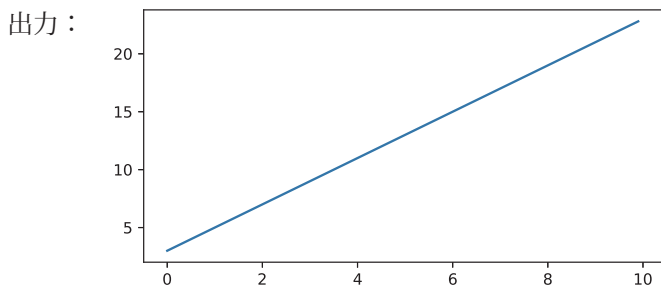
例. パッケージの読み込み

```
入力: import pandas as pd          # pandas を pd の名で読み込み
      from scipy import stats      # scipy.stats の読み込み
      import numpy as np          # NumPy の np の名で読み込み
      import matplotlib.pyplot as plt  # matplotlib を plt の名で読み込み
```

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)
      plt.show()
```

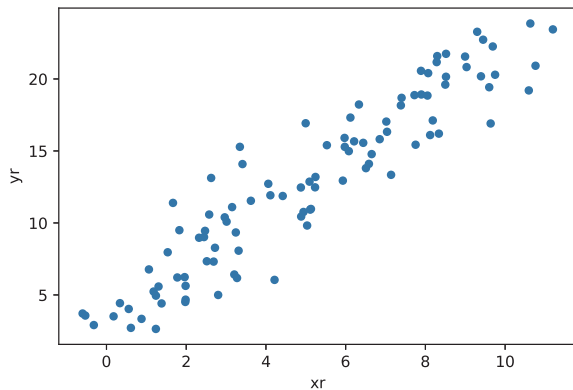


このようにして得られた X, Y に乱数を加えて攪乱したデータ Xr, Yr を作成し、それらを元に DataFrame を作成する。（次の例参照）

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 を攪乱
      Yr = Y + stats.norm.rvs( size=len(Y) ) # データ列 Y を攪乱
      # 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' )
      plt.show()
```

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のクロス表が DataFrame オブジェクトとして得られる。

4.7.2 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 メソッドでは**不偏共分散**を算出する。SciPy の cov 関数を使用すると**標本共分散**を求めることが

できる.

例. SciPy の cov 関数による標本共分散の算出 (先の例の続き)

```
入力: import scipy as sp      # SciPy の読み込み
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( sp.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]])
```

SciPy 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. SciPy 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( sp.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]])
```

4.7.3 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)
```

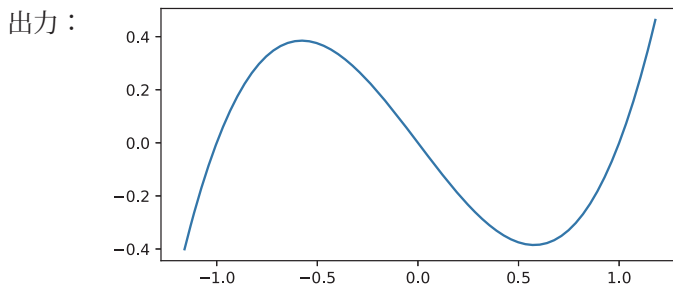
```
出力:  array([1.82296057, 3.62008113])
```

先に作成したデータ Xr, Yr は 1 次式 $y = 2x + 3$ を撓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)

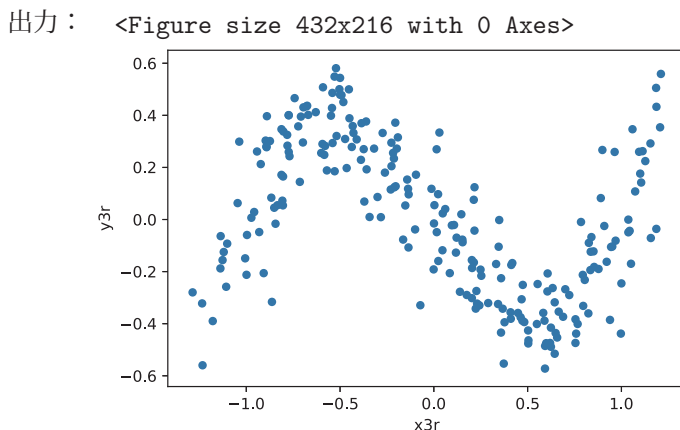
```
入力: X3 = np.arange(-1.16,1.19,0.01)    # データ列 Y3
      Y3 = X3**3 - X3                    # データ列 Y3
      plt.figure(figsize=(6,3))        # プロット
      plt.plot(X3,Y3)
      plt.show()
```



このようにして得られた X3, Y3 に乱数を加えて攪乱したデータ 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 を攪乱
      Y3r = Y3 + 0.1*stats.norm.rvs( size=len(Y3) )    # データ列 Y3 を攪乱
      # 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' )
      plt.show()
```



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。

4.8 統計検定

4.8.1 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²⁴
- 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. ライブラリの読み込み

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

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み込みが完了した。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 ( $\mu=0, \sigma=1$ ) 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)
      # 対数正規分布 ( $\mu=0, \sigma=1$ ) 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)
      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。

次に、既知の母数として `df['Norm']` の標準偏差と平均を取得する。

例. 既知の母数の取得（先の例の続き）

```
入力: sigma = df['Norm'].std()    # 母集団の不偏標準偏差
      mu = df['Norm'].mean()      # 母平均
      print( '  $\mu$ :', mu )
      print( '  $\sigma$ :', sigma )
```

```
出力:  $\mu$ : -0.002719692432291954
       $\sigma$ : 0.9929845090673574
```

これで `sigma` に母集団の標準偏差が、`mu` に母平均が得られた。次に、母集団から標本（30 件）を抽出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.iloc[0:n]          # 標本の抽出
      x = df2['Norm'].mean()       # 標本の平均
      print( ' 標本平均:', x )
```

```
出力: 標本平均: -0.27252209806587246
```

これで標本が `df2` に得られた。

²⁴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。

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。

$$\text{標準誤差: } S_e = \frac{\sigma}{\sqrt{n}} \quad (\sigma \text{ は母集団の標準偏差, } n \text{ は標本の件数})$$

$$z \text{ スコア: } z = \frac{\bar{X} - \mu}{S_e} \quad (\mu \text{ は母平均, } \bar{X} \text{ は標本の平均})$$

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: -1.4882091541736906

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.114 「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

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。

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: 7.028559588238926
False

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。

4.8.2 t 検定

母集団が正規分布に従うと仮定する場合に、採取した少量のサンプルのみから母平均を推定する方法として t 検定が有効である。すなわち t 検定は、標本の個数、標本平均、標本の標準偏差から母平均を推定する方法となる。ここでは t 検定によって母平均を推定（**1 群の t 検定**）する作業の例を示す。

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. ライブラリを読み込みと母集団データの作成

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

```
入力: # 正規分布 ( $\mu=0, \sigma=1$ ) 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)
      df = pd.DataFrame(columns=['Norm'])    # DataFrame を用意
      df['Norm'] = y1                        # y1, y2 を DataFrame にセット
```

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.iloc[0:n]           # 標本の抽出
      x = df2['Norm'].mean()        # 標本の平均
      s = df2['Norm'].std()         # 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:      -0.07791639022918537
      標本の標準偏差:  0.9946417533383528
```

t 検定は、母平均の推定値 $\hat{\mu}$ 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²⁵ を調べる検定方法である。

$$t \text{ 統計量: } t = \frac{\bar{X} - \hat{\mu}}{\frac{s}{\sqrt{n}}} \quad (n \text{ は標本の個数, } \bar{X} \text{ は標本の平均, } s \text{ は標本の標準偏差, } \hat{\mu} \text{ は母平均の推定値})$$

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                    # 母平均の推定
      t = (x-mu)/(s/np.sqrt(n))    # t 統計量
      print('t 統計量:', t)
```

```
出力:  t 統計量:  -3.910535565283862
```

この例では、母平均を 0.7 と推定して、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.114 「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。

²⁵ 詳しくは「A.3.2 母平均に関する検定 (t 検定)」(p.118) 参照のこと。

例. 信頼区間の算出（先の例の続き）

```
入力： # 信頼度 95% ( $\alpha=0.05$ ) 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：  2.063898561628025
```

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
```

t 統計量が信頼できる範囲に無く、推定した母平均が妥当でないことがわかる。次に母平均を $\hat{\mu} = 0.05$ と推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      # 母平均の推定
      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： 0.7390285058400614
      True
```

ここでの例からわかるように、母平均の推定 $\hat{\mu} = 0.7$ 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 ²⁶（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 ²⁷ 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 は十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** ²⁸ 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 ²⁹ 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 15）が見られる。

表 15: 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。

²⁶Microsoft 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。

²⁷**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.87) で解説する。

²⁸IBM 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。

²⁹Microsoft 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。

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³⁰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³¹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³²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³³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、トランザクションに関する 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。

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 SQLA # SQLAlchemy を 'SQLA' の名で読み込み
      import pandas as pd      # pandas を 'pd' という名で読み込み
```

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 'SQLA' としている。

次に、サンプルの DataFrame を作成する。

³⁰OS 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。

³¹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。

³²SQLite は動的リンクライブラリ（Windows では DLL）の形で提供される。

³³SQLite 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。Python スクリプトから利用する場合は、スクリプトと同一のディレクトリに SQLite 本体を配置するだけでよい。

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 Engine オブジェクトの生成

SQLAlchemy では Engine オブジェクトを介してデータベースにアクセスする。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: engn = SQLAlchemy.create_engine( 'sqlite:///testdb01', echo=False ) # DB 接続エンジンの生成
```

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。以後は生成した Engine オブジェクトを介してデータベースにアクセスする。この例では 'testdb01' という名前のデータベースに接続して (データベースが無ければ作成して)、それにアクセスするための Engine オブジェクト engn を生成している。

5.3.3 SQL クエリの表示

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。今回の例では 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。

```
SELECT CAST('test plain returns' AS VARCHAR(60)) AS anon_1
SELECT CAST('test unicode returns' AS VARCHAR(60)) AS anon_1
PRAGMA table_info("tbl01")
CREATE TABLE tbl01 (
  "番号" BIGINT,
  "氏名" TEXT,
  "英語" BIGINT,
  "化学" BIGINT
)
COMMIT
BEGIN (implicit)
INSERT INTO tbl01 ("番号", "氏名", "英語", "化学") VALUES (?, ?, ?, ?)
((1, ' 山田 太郎', 52, 81), (2, ' 田中 花子', 61, 72), (3, ' 中村 勝則', 89, 64))
COMMIT
```

5.3.4 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。その際、引数に Engine オブジェクトを与える。(次の例参照)

例. Engine オブジェクトを用いて DataFrame を SQLite の table に登録する（先の例の続き）

```
入力: df.to_sql( 'tbl01', egn, index=False )      # DataFrame をデータベースに保存
```

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。

5.3.5 テーブルから DataFrame への読み込み

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: q = 'SELECT * FROM tbl01'      # SQL クエリ
      df2 = pd.read_sql( q, egn )    # 読み込みの実行
      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

この例では q に読み込み処理のための SQL 文を与え、pandas の read_sql メソッド（データベースからデータを読み込むメソッド）に与えている。この際、先に作成した Engine オブジェクト egn を用いている。

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.87 「5.5 SQL」を参照のこと）

5.3.6 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。先に作成した Engine オブジェクト egn を使用する例を示す。

例. 既存のテーブルへのデータの追加（先の例の続き）

```
入力: df3.to_sql( 'tbl01', egn, 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
      df2 = pd.read_sql( q, egn )    # 読み込みの実行
      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。

5.3.7 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： df3.to_sql( 'tbl01', egn, 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
      df2 = pd.read_sql( q, egn )    # 読み込みの実行
      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。

5.3.8 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. 条件を指定して読み込み（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"'
      df2 = pd.read_sql( q, egn )    # 読み込みの実行
      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を使用できる。

5.4 データベースに対するアクセスの例 (2)：トランザクション処理

データベースへのアクセスはトランザクションと呼ばれる一連の手続きに基づいた形で行われる。トランザクションは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。

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

データベースへの接続のために作成した Engine オブジェクトに対して connect メソッドを実行することでトランザクションが開始する。(次の例参照)

例. トランザクションの開始 (先の例の続き)

```
入力: cn = eng.connect()    # トランザクションの開始
      print( type(cn) )    # データ型の調査
```

出力: <class 'sqlalchemy.engine.base.Connection'>

connect メソッドの処理が正常に終了すると Connection オブジェクトが返される。この例では Connection オブジェクト cn が得られており、以後はこのオブジェクトに対して SQL 文を発行することになる。

SQL 文の発行には execute メソッドを使用する。(下記参照)

書き方: Connection オブジェクト.execute(SQL 文)

execute メソッドの戻り値は ResultProxy 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.87) で説明する。

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: r1 = cn.execute('UPDATE tbl01 SET 英語=99, 化学=24 WHERE 氏名="ジョン スミス"')
      r2 = cn.execute('UPDATE tbl01 SET 英語=100, 化学=41 WHERE 氏名="リサ シェパード"')
```

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'
      df2 = pd.read_sql( q, egn )    # 読み込みの実行
      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   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: r = cn.execute('DELETE FROM tbl01 WHERE 氏名="ジョン スミス"')
```

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'
      df2 = pd.read_sql( q, egn )    # 読み込みの実行
      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: r = cn.execute('''INSERT INTO tbl01 (番号, 氏名, 英語, 化学)
                      VALUES (5, "斉藤 ジェシカ", 93, 82)''')
```

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'
      df2 = pd.read_sql( q, egn )    # 読み込みの実行
      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
      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 ResultProxy 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 ResultProxy 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: r = cn.execute('SELECT * FROM tbl01')
```

この例は単にデータを選択するのみである。得られている ResultProxy オブジェクト r は Python のイテラブルな形式であり、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. ResultProxy 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:
          print( rec['番号'], rec['氏名'], rec['英語'], rec['化学'])
```

```
出力:  ['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。ResultProxy オブジェクトから取り出された要素 rec は RowProxy オブジェクトである。これは SELECT 文で選択されたデータに含まれる各行と見なすことができ、[...] でカラム名を指定して値を取り出すことができる。また、ResultProxy オブジェクトに keys メソッドを使用することでレコードを構成するカラム名をリストの形で取得することができる。

5.4.6 トランザクションの終了

一連のトランザクションが終了した後は、Connection オブジェクトに対して close メソッドを実行する。（次の例参照）

例. トランザクションの終了（先の例の続き）

```
入力: cn.close() # トランザクションの終了
```

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 SQLA    # SQLAlchemy を 'SQLA' の名で読み込み
      import pandas as pd         # pandas を 'pd' という名で読み込み
```

サンプルとして作成するテーブルを表 16 に示す。

表 16 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 16 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。

表 16: 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)	都道府県 (文字列)	面積 (整数)
主キー	-	主キー	-
13	東京都	東京都	2191
23	愛知県	神奈川県	2416
26	京都府	愛知県	5172
27	大阪府	大阪府	1905

テーブル名: tblPref テーブル名: tblGeom

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、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 接続エンジンの生成
      egn = SQLAlchemy.create_engine( 'sqlite:///testJPNdb', echo=False, encoding='utf-8' )
      cn = egn.connect() # トランザクションの開始
```

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。

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 17 に挙げる。

表 17: 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
r = cn.execute( '''CREATE TABLE tblPref( コード INTEGER, 都道府県 TEXT,
                        PRIMARY KEY(コード) )''' )

# レコードの挿入
r = cn.execute( '''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")''' )
```

これでテーブル tblPref が作成され、レコードが 3 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 ResultProxy 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³⁴ 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。

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
r = cn.execute( '''CREATE TABLE tblGeom( 都道府県 TEXT, 面積 INTEGER,
                        PRIMARY KEY(都道府県) )''' )

# レコードの挿入
r = cn.execute( '''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)''' )
```

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。

³⁴SQLite 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。

《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 **ResultProxy** 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.87) のところで説明したが, **read_sql** メソッドによる読み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み込みができる.

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
      df = pd.read_sql( q, egn )      # 読み込みの実行
      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
      df = pd.read_sql( q, egn )    # 読み込みの実行
      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
```

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'''
df = pd.read_sql( q, egn )      # 読み込みの実行
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'''
df = pd.read_sql( q, egn )      # 読み込みの実行
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。

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` の読み込み

入力: `import scipy.special as special` # `scipy.special` を 'special' の名で読み込み

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データ列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： 9332621544394415268169923885626670049071596826438162146859296389521759999322991
56089414639761565182862536979208272237582511852109168640000000000000000000000000
int
```

計算結果が Python 本来の整数型 (int)³⁵ で得られている。ただし、この方法による計算は、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。

6.1.2 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
```

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: 112157625266646245087810168410491392091465702506712196420110542438400000000000
      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)

³⁵Python の整数 (int) 型は長い桁の値が扱える。

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
```

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: 555974423571664033815804589243553849851258056649719919687842027223208475
      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み込み

```
入力: import scipy.stats as stats    # scipy.special を 'special' の名で読み込み
      import numpy as np            # NumPy を 'np' の名で読み込み
      import matplotlib.pyplot as plt  # グラフ描画ライブラリを 'plt' の名で読み込み
```

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。

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。

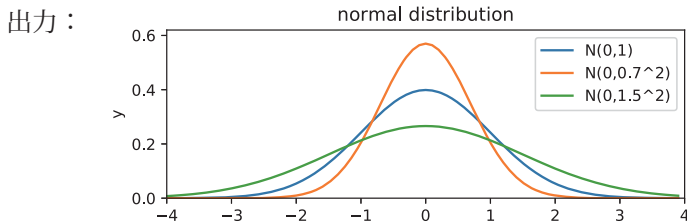
例. 正規分布の確率密度関数 (2) (先の例の続き)

```
入力: ax = np.arange( -4, 4, 0.1 )      # -4 以上 4 未満の数値を 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
```



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。

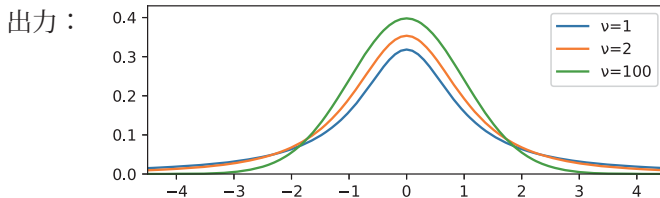
例. t 分布の確率密度関数 (2) (先の例の続き)

```
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 )    #  $\nu=1$ 
      ay2 = stats.t.pdf( ax, 2, loc=0, scale=1 )    #  $\nu=2$ 
      ay3 = stats.t.pdf( ax, 100, loc=0, scale=1 )  #  $\nu=100$ 
```

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='  $\nu=1$ ' )    # グラフのプロット 1
      plt.plot( ax, ay2, label='  $\nu=2$ ' )    # グラフのプロット 2
      plt.plot( ax, ay3, label='  $\nu=100$ ' )  # 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
```



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。

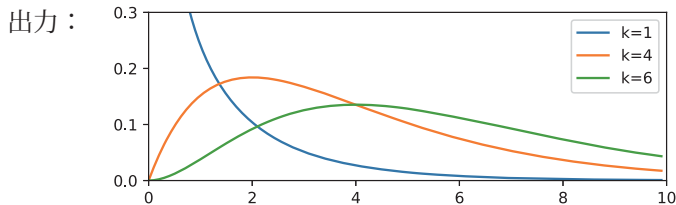
例. χ^2 分布の確率密度関数 (2) (先の例の続き)

```
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
```



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。

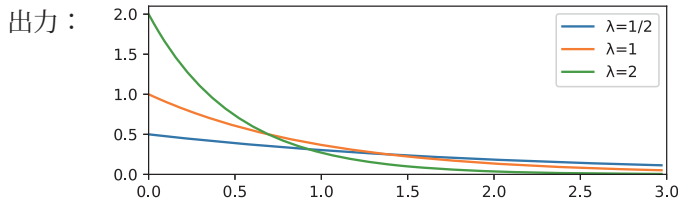
例. 指数分布の確率密度関数 (2)（先の例の続き）

```
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
```



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、

$$f(x) = \frac{1}{\sqrt{2\pi} \sigma x} \exp\left(-\frac{(\log x - \mu)^2}{2\sigma^2}\right) \quad (x > 0)$$

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。

$$\text{lognorm.pdf}(x, s) = \frac{1}{s \cdot x \cdot \sqrt{2\pi}} \cdot \exp\left\{-\frac{1}{2} \cdot \left(\frac{\log(x)}{s}\right)^2\right\}$$

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。

$$x_2 = \frac{x - \text{loc}}{\text{scale}} \rightarrow \text{計算結果} : \frac{\text{lognorm.pdf}(x_2, s)}{\text{scale}}$$

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)

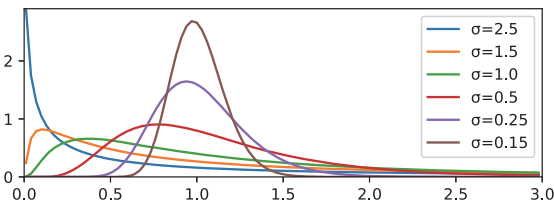
```
入力: ax = np.arange( 0.01, 3, 0.03 )      # 0.01 以上 3 未満の数列を 0.03 間隔で生成
      ay1 = stats.lognorm.pdf( ax, 2.5, loc=0, scale=1 )  #  $\sigma=2.5$ 
      ay2 = stats.lognorm.pdf( ax, 1.5, loc=0, scale=1 )  #  $\sigma=1.5$ 
      ay3 = stats.lognorm.pdf( ax, 1.0, loc=0, scale=1 )  #  $\sigma=1.0$ 
      ay4 = stats.lognorm.pdf( ax, 0.5, loc=0, scale=1 )  #  $\sigma=0.5$ 
      ay5 = stats.lognorm.pdf( ax, 0.25, loc=0, scale=1 ) #  $\sigma=0.25$ 
      ay6 = stats.lognorm.pdf( ax, 0.15, loc=0, scale=1 ) #  $\sigma=0.15$ 
```

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='  $\sigma=2.5$ ' ) # グラフのプロット 1
      plt.plot( ax, ay2, label='  $\sigma=1.5$ ' ) # グラフのプロット 2
      plt.plot( ax, ay3, label='  $\sigma=1.0$ ' ) # グラフのプロット 3
      plt.plot( ax, ay4, label='  $\sigma=0.5$ ' ) # グラフのプロット 3
      plt.plot( ax, ay5, label='  $\sigma=0.25$ ' ) # グラフのプロット 3
      plt.plot( ax, ay6, label='  $\sigma=0.15$ ' ) # 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:



6.2.2 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: 0.246093750000000025
      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。

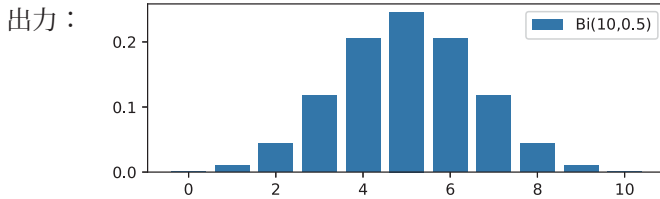
例. 二項分布の確率質量関数 (2) (先の例の続き)

```
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
```



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。

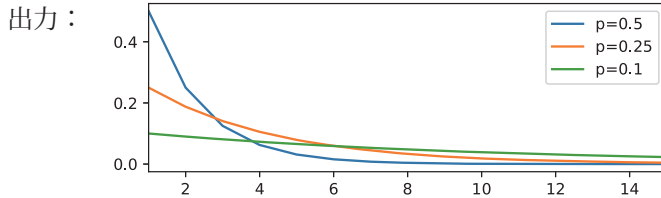
例. 幾何分布の確率質量関数 (2) (先の例の続き)

```
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
```



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）

$f_x(k; N, K, n)$ の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, N, n, K)`

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。

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;100,50,40)
      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。

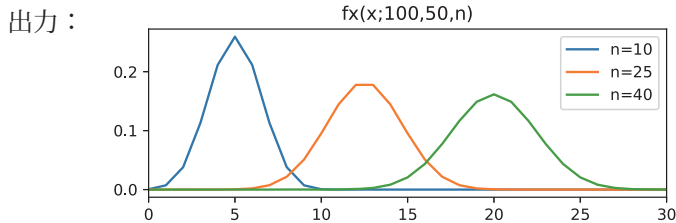
例. 超幾何分布の確率質量関数 (2)（先の例の続き）

```
入力： ax = np.arange(0,101)                  # 0～100 の数列
      ay1 = stats.hypergeom.pmf(ax,100,10,50)   # fx(x;100,50,10)
      ay2 = stats.hypergeom.pmf(ax,100,25,50)   # fx(x;100,50,25)
      ay3 = stats.hypergeom.pmf(ax,100,40,50)   # fx(x;100,50,40)
```

この例では $f_x(x; 100, 50, n)$ 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 n (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='n=10' ) # グラフのプロット 1
      plt.plot( ax, ay2, label='n=25' ) # グラフのプロット 2
      plt.plot( ax, ay3, label='n=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;100,50,n)')      # 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
```



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)    #  $\lambda=10$ 
      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。

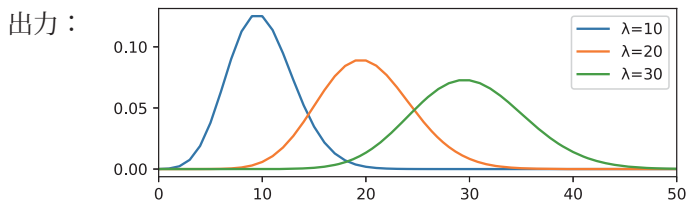
例. ポアソン分布の確率質量関数 (2)（先の例の続き）

```
入力: ax = np.arange(0,101)           # 0~100 の数列
      ay1 = stats.poisson.pmf(ax,10)   #  $\lambda=10$ 
      ay2 = stats.poisson.pmf(ax,20)   #  $\lambda=20$ 
      ay3 = stats.poisson.pmf(ax,30)   #  $\lambda=30$ 
```

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
```



6.2.3 累積分布関数：CDF（Cumulative Density 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 1）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。

$$\lim_{x \rightarrow \infty} cdf(x) = 1$$

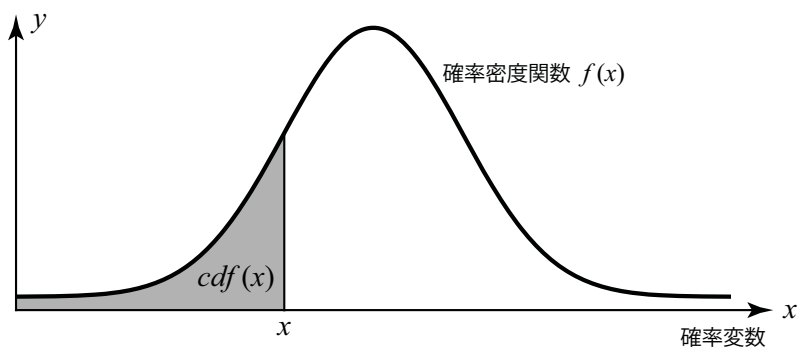


図 1: 累積分布関数

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。

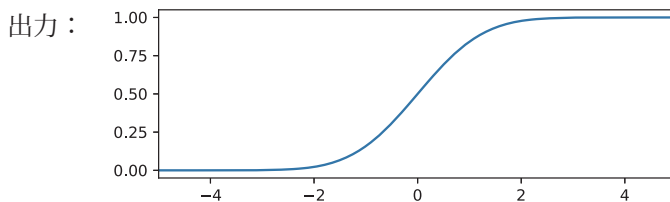
例. stats.norm.cdf の使用例 (2) (先の例の続き)

```
入力: ax = np.arange(-5,5,0.1)          # -5 以上 5 未満の数値を 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
```



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 2) 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。

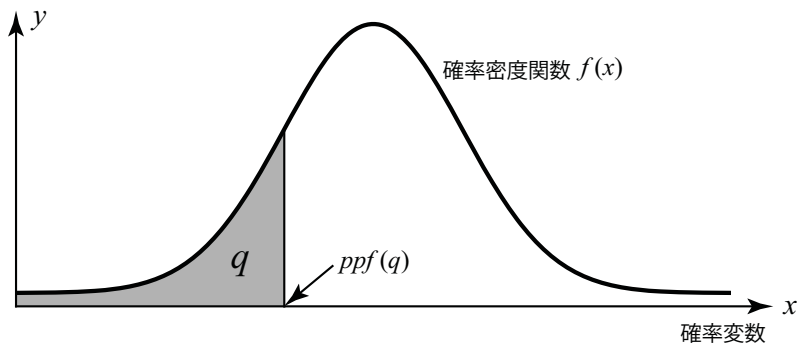


図 2: パーセント点関数

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。

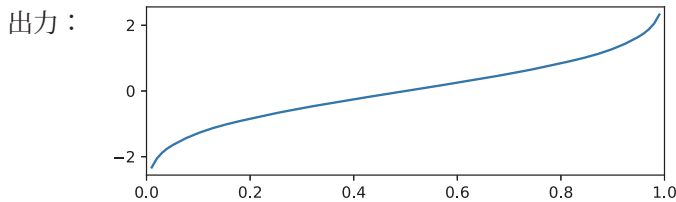
例. stats.norm.ppf の使用例 (2) (先の例の続き)

```
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
```



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。乱数を生成する関数の名前は 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)

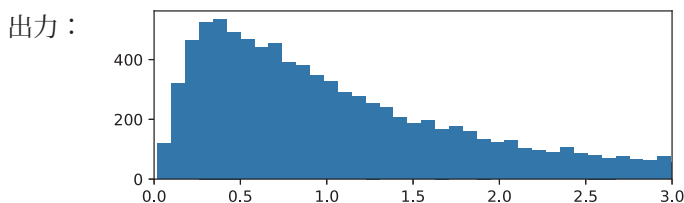
```
入力: # 対数正規分布 ( $\sigma=1$ ,  $\mu=0$ ) に沿った乱数を 10,000 個生成
      r = stats.lognorm.rvs(1,size=10000)
      type( r )    # データ型の調査
```

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
```



6.2.5.1 一様乱数の生成

uniform.rvs 関数は一様乱数を生成する。

書き方: uniform.rvs(size=個数)

この関数は 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 関数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.

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ある.

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、プログラムの動作の再現性を示すことができる。

参考) 暗号的な処理を行う際は、確定的な乱数を使用することは避けるべきである。

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**³⁶ (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**³⁷ を調べることから始まる。主要な要約統計量としては**平均値**³⁸ (mean)、**最頻値** (mode)、**中央値** (median)、**四分位数** (quartile points) 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**³⁹）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

まず最初に、全事象の確率の合計は必ず 1 となることがある。サイコロの試行を例に挙げると、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 $6 \times 1/6 = 1$ となることがある。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、

$$\int_{-\infty}^{\infty} f(x) dx = 1$$

³⁶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。

³⁷代表値 (measures of central tendency)、基本統計量ともいう。

³⁸「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。

³⁹値の範囲。通常はその範囲の中央値を**階級値** (class value) とする。

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**⁴⁰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⁴⁰ 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。

$$f(x) = {}_n C_x \cdot p^x (1-p)^{n-x}$$

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。

$$P(x) = p(1-p)^{x-1} \quad (\text{定義 a})$$

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

$$P(x) = p(1-p)^x \quad (\text{定義 b})$$

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。

$$f_x(k; N, K, n) = \frac{{}_n C_k \cdot {}_{N-n} C_{K-k}}{{}_N C_K}$$

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 $np = \text{一定}$ 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。

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。

⁴⁰このような試行をベルヌーイ試行という。

A.1.3.1 正規分布

正規分布 $f(x)$ の定義は次の通り.

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \quad (x \text{ は実数})$$

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 3 に示す.

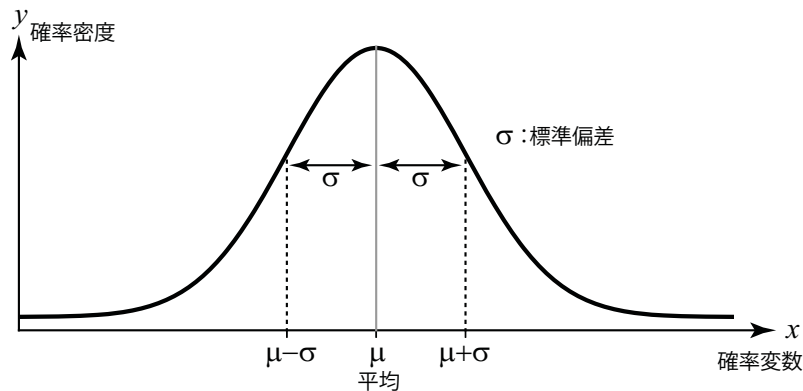


図 3: 正規分布

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.

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma x} \exp\left(-\frac{(\log x - \mu)^2}{2\sigma^2}\right) \quad (x > 0)$$

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.112) 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 4) を「**尖度** (kurtosis) が大きい」と表現する.

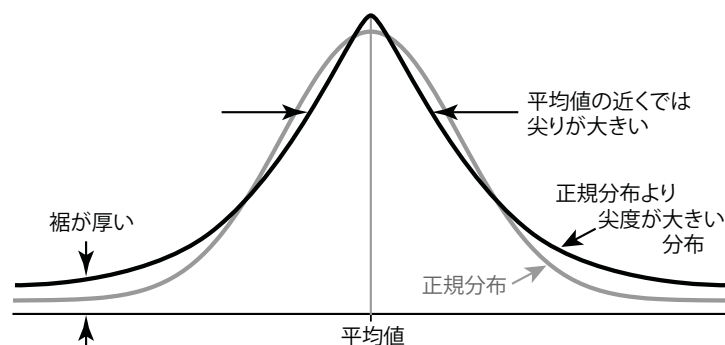


図 4: 尖度の大きい分布

尖度は正規分布を基準とし、正規分布の尖度を 0⁴¹ 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 5 に示す対数正規分布は歪度が大きい分布である。

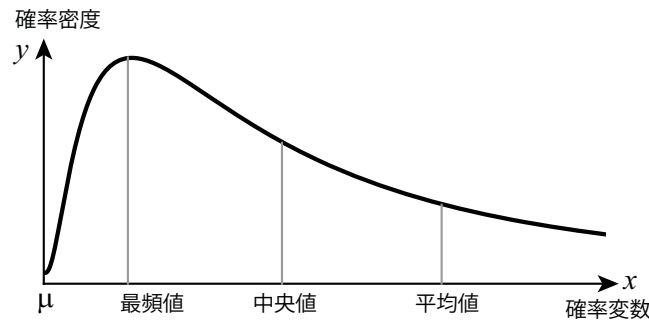


図 5: 対数正規分布
注) 確率密度関数を構成する μ, σ と、結果としての平均値（期待値）、標準偏差は異なる。

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出来る。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 $\{x_1, x_2, \dots, x_n\} \in X$ に対する確率を $P(x_i)$ とすると次のように定義される。

$$E(X) = \sum_i x_i \cdot P(x_i) \quad (\text{離散分布の場合})$$

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⁴²（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、

$$E(X - \alpha)^n$$

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」⁴³ 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 $\frac{m_4}{\sigma^4} - 3$

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 $\frac{m_3}{\sigma^3}$

⁴¹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。

⁴²積率ともいう。

⁴³参考) $E\left(\frac{X - \mu}{\sigma}\right)^n$ を標準化されたモーメントという。

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.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。

$$\bar{X} = \frac{x_1 + x_2 + \cdots + x_n}{n} \quad (x_1, x_2, \cdots, x_n \in X)$$

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**⁴⁴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）

$$\begin{aligned} S^2 &= \frac{1}{n} \sum_{i=1}^n (x_i - \bar{X})^2 = \frac{1}{n} \sum_{i=1}^n (x_i^2 - 2x_i\bar{X} + \bar{X}^2) = \frac{1}{n} \sum_{i=1}^n x_i^2 - 2\bar{X} \cdot \frac{1}{n} \sum_{i=1}^n x_i + \bar{X}^2 \cdot \frac{1}{n} \sum_{i=1}^n 1 \\ &= \frac{1}{n} \sum_{i=1}^n x_i^2 - 2\bar{X} \cdot \bar{X} + \bar{X}^2 = \frac{1}{n} \sum_{i=1}^n x_i^2 - 2\bar{X}^2 + \bar{X}^2 = \frac{1}{n} \sum_{i=1}^n x_i^2 - \bar{X}^2 \end{aligned}$$

⁴⁴不偏性と一致性に関する解説は他の文献を参照のこと。

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 , 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る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**⁴⁵ (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 '^' 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、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6に示す。

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)

⁴⁵ 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。

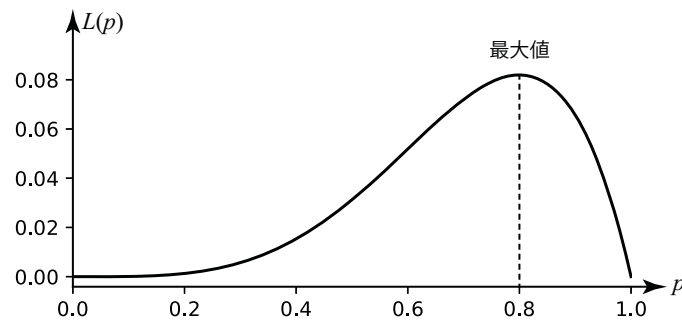


図 6: $L(p)$ のプロット

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、 $\text{erf}(z)$ と記す。

III. パーセント点

$\phi(x)$ 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、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 7)

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。

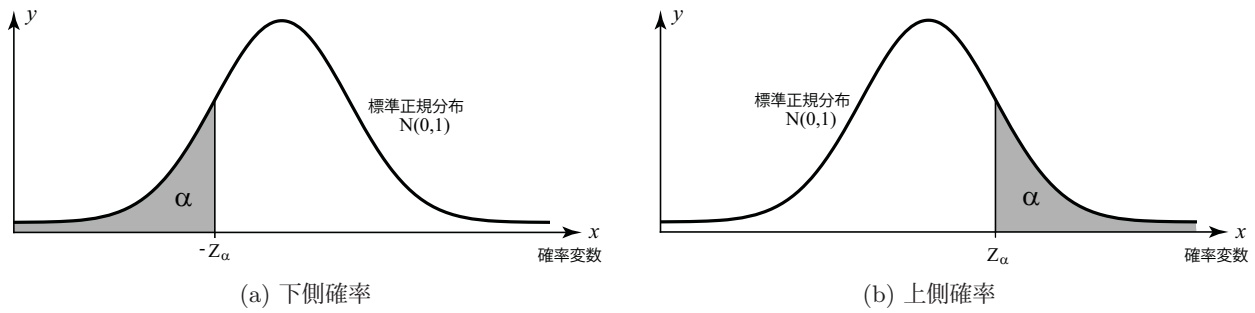


図 7: 発生確率が α のパーセント点 Z_α

【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 8 における $[-Z_{\frac{\alpha}{2}}, Z_{\frac{\alpha}{2}}]$ である。

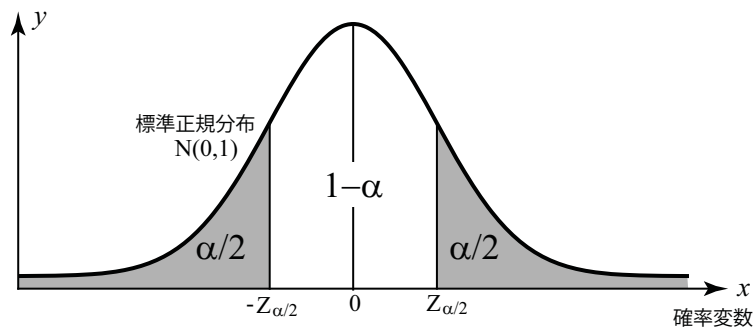


図 8: 信頼区間の作り方

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 $N\left(\mu, \frac{\sigma^2}{n}\right)$ に従う。⁴⁶ このことから、 \bar{X} が信頼係数 $1 - \alpha$ で $N\left(\mu, \frac{\sigma^2}{n}\right)$ 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。

⁴⁶ 「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⁴⁷, 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⁴⁸ といい, $t(k)$ と書く.

$$t = \frac{Z}{\sqrt{Y/k}}$$

先の t 統計量 $t = \frac{\bar{X} - \mu}{s/\sqrt{n}}$ 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, \dots, Z_k があるとき,

$$\chi^2 = Z_1^2 + Z_2^2 + \dots + Z_k^2$$

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.

$$f(x) = \frac{x^{\frac{k}{2}-1}}{2^{\frac{k}{2}} \Gamma(\frac{k}{2})} \exp\left(-\frac{x}{2}\right)$$

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.

$$t(x) = \frac{\Gamma(\frac{k+1}{2})}{\sqrt{k} \pi \Gamma(\frac{k}{2})} \left(1 + \frac{x^2}{k}\right)^{-\frac{k+1}{2}}$$

⁴⁷ 「A.2.3 χ^2 分布」を参照のこと.

⁴⁸ 「A.2.4 t 分布」を参照のこと.

A.3 仮説検定

仮説検定⁴⁹ (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”ということを経験調査を元にして、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⁵⁰

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 18 に示す。これを元に検定の作業を進める。

表 18: $Bi(20, 1/2)$ 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 18 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）

⁴⁹より厳密に**統計的仮説検定**ともいう。

⁵⁰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代わりに 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。⁵¹

まず、 $\mu = 15$ とする t 統計量を求めると、

$$t = \frac{13.7 - 15}{2.3/\sqrt{25}} = -2.826 \dots$$

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、

$$t_{\frac{0.05}{2}}(24) = 2.06389856 \dots$$

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 9 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することができる。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。

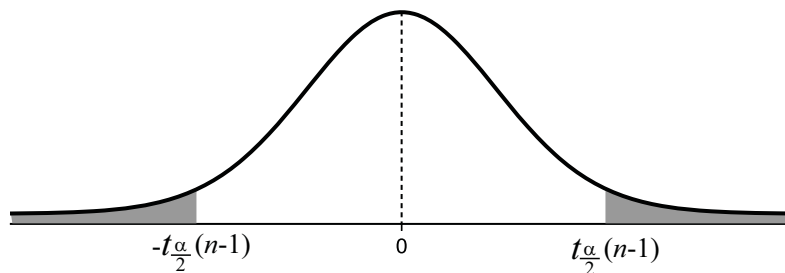


図 9: 両側検定の棄却域（ t 検定）

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 10 の (a) の領域となる。この場合のパーセント点を求めると、

$$t_{0.05}(24) = 1.710882 \dots$$

⁵¹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。

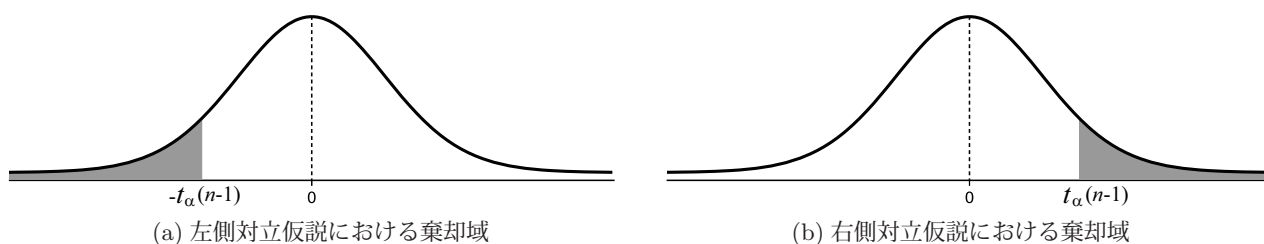


図 10: 片側検定の棄却域

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 10 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 $\chi_{1-\frac{\alpha}{2}}^2(n-1)$ と $\chi_{\frac{\alpha}{2}}^2(n-1)$ の 2 つのパーセント点の値を求め、

$$\chi_{1-\frac{\alpha}{2}}^2(n-1) < \chi^2 < \chi_{\frac{\alpha}{2}}^2(n-1)$$

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 $\chi_{\alpha}^2(n-1) \leq \chi^2$ 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 $\chi^2 \leq \chi_{1-\alpha}^2(n-1)$ 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、

$$\chi^2_{0.05}(24) = 13.8484 \dots$$

$$\chi^2_{0.95}(24) = 36.4150 \dots$$

となる。(図 11 参照)

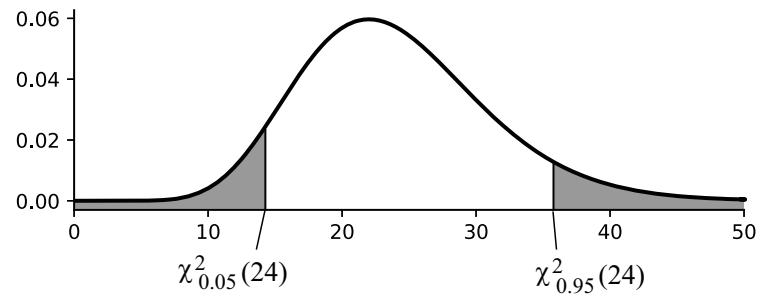


図 11: 両側検定の棄却域 (χ^2 検定)

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。

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

`==`, 39

`&`, 39

`~`, 39

`.`, 4, 29

1 群の t 検定, 78

`alpha`, 63

`apply`, 48

`at`, 3, 23

`bins`, 60, 63

`both`, 62

`CDF`, 103

`cdf`, 103

`class value`, 108

`close`, 87

`closed`, 61

`comb`, 93

`concat`, 16, 36

`connect`, 85

`Connection`, 85, 88

`copy`, 15

`corr`, 72

`cov`, 73

`CREATE DATABASE`, 88

`CREATE TABLE`, 88

`create_engine`, 82

`CSV`, 50

`DataFrame`, 2, 21

`DataFrame` の連結, 36

`date_range`, 46

`DatetimeIndex`, 42, 46

`DBMS`, 80

`ddof`, 57

`del`, 15, 36

`DELETE`, 86

`describe`, 40, 54

`display`, 25

`drop`, 13, 34

`Engine`, 82

`erf`, 114

`estimation`, 113

`estimator`, 113

`execute`, 85

`factorial`, 92

`figsize`, 63

`figure`, 62

`fillna`, 24

`FROM`, 83

`from_tuples`, 18, 20

`get_dummies`, 70

`get_loc`, 10, 33

`groupby`, 68

`head`, 20, 41

`hist`, 63

`iat`, 5, 24

`iloc`, 5, 26, 30

`Index`, 9, 32, 33

`index`, 9

`info`, 40

`INNER JOIN`, 90

`inplace`, 12, 13, 34

`INSERT`, 86

`INSERT INTO`, 89

`INTEGER`, 89

`Interval`, 61

`IPython`, 1

`Jupyter`, 1

`kurt`, 57

`kurtosis`, 110

`left`, 61, 62

`LEFT OUTER JOIN`, 91

`legend`, 66

`loc`, 3, 25, 30

`lognorm.rvs`, 105

`matplotlib`, 1, 53, 61

`max`, 55, 58

`mean`, 56, 58, 108

measures of central tendency, 108

`median`, 55, 58, 108

`min`, 55, 58

`mode`, 59, 108

`moment`, 111

`MultiIndex`, 18

NaN, 23, 37, 91
 nan, 24
 ndarray, 2, 9, 22, 31, 54, 92
 neither, 62
 NONE, 89
 norm.cdf, 103
 norm.ppf, 104
 now, 44
 NUMERIC, 89
 NumPy, 1, 22, 31

 One-Hot encoding, 70

 pandas, 1
 parameter, 112
 PDF, 94
 pdf, 94
 percentile, 58
 perm, 93
 pivot_table, 68
 plot, 65
 PMF, 99
 pmf, 99
 polyfit, 74
 population, 108
 PPF, 104
 ppf, 104
 PRIMARY KEY, 88

 quantile, 55, 58

 random_state, 106
 range, 63
 RDB, 80
 read_csv, 51
 read_sql, 83
 REAL, 89
 reset_index, 17
 ResultProxy, 85, 87, 90
 right, 61, 62
 RowProxy, 87
 rvs, 105

 sample, 68, 71, 108
 savefig, 63
 SciPy, 1, 53, 92
 scipy.special, 92
 scipy.stats, 53
 seed, 107
 SELECT, 83, 87

 Series, 2
 SET, 85
 show, 62
 skew, 57
 skewness, 111
 sort_index, 13, 19, 34
 sort_values, 12, 33
 SQL, 87
 SQLAlchemy, 1, 81
 SQLite, 1, 81
 SQL クエリを表示, 82
 statistic, 113
 stats.binom, 99
 stats.chi2, 96
 stats.expon, 97
 stats.geom, 100
 stats.hypergeom, 101
 stats.lognorm, 98
 stats.norm, 94
 stats.poisson, 102
 stats.t, 95
 std, 57, 58
 sum, 56, 58
 surmise, 113

 T, 41
 table, 80
 tail, 20, 41
 TEXT, 89
 Timedelta, 44, 45
 Timestamp, 42
 title, 66
 to_csv, 50
 to_datetime, 43
 to_numpy, 9, 10, 32, 33
 to_sql, 82
 t 検定, 118
 t 分布, 95
 t 分布, 116
 t 検定, 78
 t 統計量, 78, 115, 116

 uniform.rvs, 105
 UPDATE, 85
 UTC, 43

 value_counts, 59
 VALUES, 86, 89
 values, 9, 10, 31, 32, 58

var, 56, 58
 weekday, 43
 WHERE, 85, 90
 x, 65
 xlabel, 66
 xlim, 66
 y, 65
 ylabel, 66
 ylim, 66
 Z_α , 114
 z 検定, 76
 z スコア, 77
 浅いコピー, 16
 誤り, 118
 アルファ値, 63
 一様乱数, 105
 一様性, 112
 インデックス, 2, 3
 上側信頼限界, 114
 回帰, 72
 階級, 108
 階乗, 92
 χ^2 検定, 119
 χ^2 分布, 96, 116
 確率質量関数, 99, 108, 109
 確率分布, 108
 確率変数, 108
 確率密度, 108
 確率密度関数, 94, 108, 109
 仮説検定, 117
 片側検定, 118
 片側対立仮説, 119
 カラム, 21
 カラム名, 21
 関係データベース, 80
 関係モデル, 80
 外部キー, 80
 ガウスの誤差積分, 114
 幾何分布, 100, 109
 幾何平均, 108
 棄却, 117
 棄却域, 118
 期待値, 110, 111
 基本統計量, 108
 帰無仮説, 117
 協定世界時, 43
 共分散, 73
 クエリ, 80
 区間, 61
 組合せ, 93
 クロス集計, 68
 グループ集計, 67
 欠損値, 23, 37, 91
 結合, 80, 88, 90
 合計, 56
 誤差, 92–94, 112
 誤差関数, 114
 最小値, 55, 108
 採択, 117
 採択域, 118
 最大値, 55, 108
 最頻値, 59, 108
 最頻の区間, 61
 最尤原理, 113
 最尤法, 113
 指数分布, 97, 110
 下側信頼限界, 114
 四分位数, 54, 55, 108
 射影, 80
 シャッフル, 71
 集計処理, 67
 主キー, 80
 信頼区間, 114
 信頼係数, 114
 真理値, 38
 辞書, 2, 22
 順列, 93
 条件式, 38
 推測, 113
 推定, 108, 113
 推定量, 113
 スライス, 5
 スライスオブジェクト, 10
 正規化, 80
 正規分布, 53, 76, 94, 110, 112
 正規母集団, 114
 整列, 12
 セット, 2
 選言, 39
 選択, 80
 尖度, 57, 110
 相関係数, 72

ソート, 12
対数正規分布, 53, 98, 110, 111
対立仮説, 117
多項式回帰, 74
単純無作為抽出, 112
第一種の誤り, 118
第二種の誤り, 118
代表値, 108
ダミー変数, 70
中央値, 54, 55, 108
抽出, 108
中心極限定理, 112
超幾何分布, 101, 109
重複するインデックス, 8
調和平均, 108
点推定, 113
転置, 41
データ構造, 1
データベース, 80
データベース管理システム, 80
統計量, 113
トランザクション, 81, 85
度数, 108
度数分布, 108
度数分布図, 62, 108
ドット, 4, 29
内部結合, 91
二項分布, 99, 109
ノン・パラメトリックな推定, 113
箱ひげ図, 64
パラメトリックな推定, 113
パーセンタイル, 55
パーセント点, 55, 114, 116
パーセント点関数, 104
非数, 91
ヒストグラム, 60, 62, 108
左片側検定, 119
左外部結合, 91
否定, 39
非復元抽出, 109
表, 80
標準化変数, 114
標準誤差, 77, 116
標準正規分布, 114
標準偏差, 57, 108, 110
標本, 108
標本標準偏差, 57

標本分散, 57, 112, 113
標本平均, 112, 113
ピボットテーブル, 68
深いコピー, 16
不偏性, 112
不偏標準偏差, 57
不偏分散, 57, 112, 115
分散, 56, 108
平均, 56, 110
平均値, 108, 111
ベルヌーイ試行, 109, 113
母集団, 108
母数, 112
母数空間, 113
母分散, 112
母分散に関する検定, 119
母平均, 112
母平均に関する検定, 118
ポアソン分布, 102, 109
マルチインデックス, 18
右片側検定, 119
ミドルウェア, 81
無作為抽出, 71
モーメント, 111
有意水準, 117
要約統計量, 40, 54, 108
乱数, 105
ランダムサンプリング, 71
リスト, 1, 2
両側検定, 118
両側対立仮説, 118
リレーショナル・データベース, 80
累積分布関数, 103
レコード, 80
連言, 39
歪度, 57, 111
ワンホットエンコーディング, 70
タイムゾーン, 43

「Python3 によるデータ処理の基礎」

著者：中村勝則

発行：2021 年 09 月 10 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

本書はフリーソフトウェアです，著作権は保持していますが，印刷と再配布は自由にさせていただいて結構です．（内容を改変せずをお願いします） 内容に関して不備な点がありましたら，是非ご連絡ください．ご意見，ご要望も受け付けています．

● 連絡先

nkatsu2012@gmail.com

中村勝則