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

OpenCV / Pillow / pygame / ffmpeg-python / NumPy / matplotlib / SciPy / SymPy / gmpy2 / hashlib, passlib / Cython / Numba / ctypes / PyInstaller / curses / tqdm / JupyterLab / json / psutil / urllib / jaconv

中村勝則

2026年3月2日

免責事項

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

本書におけるサンプルプログラムの実行方法

本書では様々な形で Python スクリプトの実行例を示す．‘〜.py’ の名前を持つファイルとして掲載するサンプルプログラム（Python スクリプト）は次のようにして Python 処理系と共に起動する．

● **Windows のコマンドウィンドウで PSF 版 Python を使用する場合**

`py　スクリプト名.py` Enter

● **macOS，Linux 環境で PSF 版 Python を使用する場合**

`python3　スクリプト名.py` Enter

● **Anaconda Prompt 環境の Python を使用する場合**

`python　スクリプト名.py` Enter

【Python 対話モードでの実行】

Python 処理系（インタプリタ）に直接 Python の式や文を与える場合は，Python 処理系のプロンプト「>>>」の後に入力する．本書では，対話モードの実行例を示す場合は基本的にこのプロンプト「>>>」も表示する．このような表示の実行例を実際に試みる場合は「>>>」の部分は入力しないこと．

目 次

1　画像の入出力と処理

1.1　OpenCV

OpenCV ライブラリは米インテル社によって開発された．現在は OpenCV.org を中心としたオープンソース・コミュニティによって維持，管理され，BSD ライセンスで配布されるオープンソースソフトウェアである．このライブラリは静止画像，動画像の入出力から画像処理，画像認識，機械学習のための機能を提供する．また，クロスプラットフォームのライブラリであり，インターネットサイト https://opencv.org/ から関連の情報を入手することができる．

OpenCV は独自の UI を実現する機能を備えており，画像の表示や，キーボード，マウスからの入力を受け付けるための簡便な機能も提供する．本書では OpenCV に関して導入的な内容について説明するので，更に詳しい事柄に関しては上記の情報サイトなどを参照すること．

このライブラリの使用に先立って，必要なソフトウェアを Python 処理系に予めインストールしておく必要がある．

例. Windows 版 PSF の Python 環境でのインストール作業

```
py -m pip install opencv-python
```

OpenCV の機能を Python で利用するためのライブラリとして cv2 があり，次のようにして Python 処理系に読み込む．

```
import cv2
```

Python における OpenCV モジュールの利用は，別のモジュール NumPy を前提としており，画像のフレームは NumPy の ndarray クラス（多次元配列）のオブジェクトとして扱われる．

【サンプルプログラム】

システムに接続されたカメラから画像フレームを入力して，それ（動画）をディスプレイに表示するサンプルプログラム opencv01.py を次に示す．

プログラム：opencv01.py

```
import cv2

# 動画像入力の開始
cap0 = cv2.VideoCapture(0)
# フレームレートの取得
fps = cap0.get(cv2.CAP_PROP_FPS)
# フレームサイズの取得
w = cap0.get(cv2.CAP_PROP_FRAME_WIDTH)
h = cap0.get(cv2.CAP_PROP_FRAME_HEIGHT)
print(fps,'(fps)'); print(w,'*',h)

# キャプチャと表示
while True:
    # フレームの読取り
    (ret,frame) = cap0.read()
    if ret:
        # フレームの表示
        cv2.imshow('Camera: 0',frame)
        # キーボードの読取り
        k = cv2.waitKey(1)
        if k == 27:     # ESCなら終了
            break
        elif k == 67 or k == 99:    # 'C' 'c' なら静止画保存
            # JPEGのQualityは 0 - 100 : 大きいほど高画質
            # cv2.imwrite('capture.jpg',frame,[cv2.IMWRITE_JPEG_QUALITY,100])
            # PNGのQualityは 0 - 9 : 小さい程高画質でファイルサイズ大
            cv2.imwrite('capture.png',frame,[cv2.IMWRITE_PNG_COMPRESSION,3])
    else:
        print('Camera is not ready.')
        break

# 終了処理
cap0.release()          # 動画像入力の開放
cv2.destroyAllWindows() # ウィンドウの消去
```

このプログラムは，カメラから画像フレームを取得し，それをウィンドウに表示する処理の繰り返しで動画像をリアルタイムに表示している．また，毎回の繰り返し処理の中でキーボードの値を取り込み，エスケープボタンが押されたら繰り返し処理を中断する形となっている．動画再生中に「C」のキーを押すと，その瞬間のフレームを静止画として画像ファイルに保存する．

1.1.1 使用中の cv2 ライブラリに関する情報

使用中の cv2 ライブラリのバージョン情報は `cv2.__version__` から参照できる．

例. 使用中の cv2 のバージョンを調べる

```
>>> cv2.__version__   Enter
'4.12.0'          ←現行のバージョン
```

このように，バージョン情報が文字列で得られる．

cv2 の詳細情報（ビルド情報など）は getBuildInformation 関数で調べることができる．

例. 使用中の cv2 のビルド情報を調べる（先の例の続き）

```
>>> print( cv2.getBuildInformation() )   Enter
General configuration for OpenCV 4.12.0 =====================================
  Version control:                4.12.0

  Platform:
    Timestamp:                    2025-07-04T16:40:32
    Host:                         Windows 10.0.26100 AMD64
              (途中省略)
  Install to:                     D:/a/opencv-python/opencv-python/_skbuild/
                                            win-amd64-3.9/cmake-install
```

このように，ビルド情報が複数行のテキストとして得られる．

1.1.2 動画像の入力

動画像の入力源はシステムに接続されたカメラもしくは動画ファイルであり，画像フレームは VideoCapture クラスのオブジェクトを介して取得する．動画像の入力処理に先立って，入力元を指定して VideoCapture オブジェクトを生成しておく．

《VideoCapture のコンストラクタ》

1) VideoCapture(カメラ番号)

カメラ番号は 0 から開始する整数であり，システムで最初に認識されるカメラは 0 である．

2) VideoCapture(動画ファイルのパス)

引数には動画ファイルのパスを文字列として与える．

VideoCapture オブジェクトに対して get メソッドを使用して各種の情報を取得することができる．書き方は

VideoCapture オブジェクト.get(**属性番号**)

属性番号は cv2 のプロパティとして表 1 のように定義されている．

表 1: VideoCapture オブジェクトから得られる値（一部）

属性番号	値
cv2.CAP_PROP_FPS	フレームレート (fps)
cv2.CAP_PROP_FRAME_WIDTH	フレーム幅
cv2.CAP_PROP_FRAME_HEIGHT	フレーム高さ

動画像の入力処理を終了する場合は，VideoCapture オブジェクトに対して release メソッドを使用する．

動画像の入力は VideoCapture オブジェクトから read メソッドを使用して 1 フレームずつ取り出す．

《フレームのキャプチャ》
書き方： VideoCapture オブジェクト.read()
メソッド実行後は **(実行結果, フレーム)** のタプルが返される．実行結果は真理値であり，キャプチャ成功の場合は True，失敗の場合は False となる．得られるフレームは NumPy の ndarray オブジェクトである．

1.1.3 cv2 独自のユーザインターフェース

cv2 ライブラリは独自のユーザインターフェースである HighGUI を提供しており，画像の表示やキーボート，マウスからの入力などを行うことができる．

cv2 のメソッド imshow を使用して，read メソッドで読み取った画像フレームを表示することができる．

《**imshow メソッドによるフレームの表示**》
書き方： **cv2.imshow(ウィンドウタイトル, フレーム)**

cv2 のメソッド waitKey を使用して，その時点でのキーボードの値を取得することができる．

《**waitKey メソッドによるキーボードの値の取得**》
書き方： **cv2.waitKey(待ち時間)**
待ち時間の単位は ms である．その時点で押されているキーの値（コード）が返される．何も押されていない場合は -1 が返される．（システム環境によっては 255 が返される）

ユーザインターフェースの使用を終了する場合は，cv2 の destroyAllWindows メソッドを使用する．

1.1.3.1 動画ファイルからの入力

動画ファイルからの画像フレームの入力を繰り返すことで，動画の再生が実現できる．次のサンプルプログラム opencvMovie01.py は，VideoCapture オブジェクトから read メソッドで画像フレームを読み取り，それを imshow メソッドでディスプレイに表示する処理を繰り返すことで動画像の再生を実現している．

プログラム：opencvMovie01.py

```
1  import cv2
2
3  # 映像の入力源の取得
4  cap = cv2.VideoCapture('Boy-21827.mp4') # ビデオファイルから入力
5  print('size:',                          # フレームサイズの表示
6      (cap.get(cv2.CAP_PROP_FRAME_WIDTH), cap.get(cv2.CAP_PROP_FRAME_HEIGHT)))
7
8  while True:
9      (ret, frame) = cap.read()   # retは画像を取得成功フラグ
10     if ret:     # フレームが得られていれば表示する
11         # フレームのリサイズ
12         frame = cv2.resize(frame, (960,540))
13         # フレームを表示する
14         cv2.imshow('Movie Capture', frame)
15         k = cv2.waitKey(30) # キー入力を30msec待つ
16         if k == 27:         # ESCキーで終了
17             break
18     else:   # フレームが得られていなければ終了する
19         break
20
21 # 映像の入力源の開放
22 cap.release()
23 cv2.destroyAllWindows()
```

1.1.4 フレームのファイルへの保存

cv2 の imwrite メソッドを使用することでフレームを静止画としてファイルに保存することができる．

《**imwrite メソッドによるフレームのファイル保存**》

書き方：　cv2.imwrite(ファイル名, フレーム, 圧縮指定)

ファイル名は拡張子を付けた文字列で指定する．特に拡張子が .jpg, .png の場合は，それぞれ JPEG,PNG フォーマットとして圧縮保存ができる．圧縮指定は次の通り．

JPEG:　[cv2.IMWRITE_JPEG_QUALITY, 値]

値は 0〜100 までの整数値で，値が大きい方が高画質（ファイルサイズ大）である．

圧縮指定を省略すると 95 が暗黙値となる．

PNG:　[cv2.IMWRITE_PNG_COMPRESSION, 値]

値は 0〜9 までの整数値で，値が小さい方が高画質（ファイルサイズ大）である．

圧縮指定を省略すると 3 が暗黙値となる．

1.1.5　静止画像の読込み

cv2 の imread メソッドを使用することで，画像ファイルを読み込むことができる．

《**imread メソッドによる画像ファイルの読込み**》

書き方：　cv2.imread(ファイル名, 読込みフラグ)

ファイル名は拡張子を付けた文字列で指定する．読み込みフラグの意味は次の通り．

cv2.IMREAD_UNCHANGED　：　画像データをそのまま読み込む（変更なし）

cv2.IMREAD_COLOR　：　アルファチャンネル（不透明度の指定）を無視する（デフォルト）

cv2.IMREAD_GRAYSCALE　：　グレースケールに変換して読み込む

読み取った画像をフレームオブジェクト (ndarray) として返す．

静止画像を読み込んで表示するプログラム opencv02.py を次に示す．

プログラム：opencv02.py

```
1  import cv2
2
3  # 画像ファイルの読込み
4  frame = cv2.imread('Earth.jpg',cv2.IMREAD_UNCHANGED)    # そのまま
5  #frame = cv2.imread('Earth.jpg',cv2.IMREAD_COLOR )      # αチャンネル無視
6  #frame = cv2.imread('Earth.jpg',cv2.IMREAD_GRAYSCALE )  # グレースケールに変換
7
8  # リサイズ
9  fr2 = cv2.resize(frame,(640,640))
10
11 # 表示
12 cv2.imshow('Image',fr2)
13
14 # 待ち
15 while True:
16         # キーボードの読取り
17         k = cv2.waitKey(1)
18         if k == 27:     # ESCなら終了
19             break
20
21 # 終了処理
22 cv2.destroyAllWindows() # ウィンドウの消去
```

このプログラムの 9 行目にあるように，cv2 の resize メソッドを使用することで画像サイズの拡縮ができる．

《**resize メソッドによる画像の拡縮**》

書き方：　cv2.resize(フレーム, (幅, 高さ))

引数に与えたフレームを（幅, 高さ）にリサイズしたフレームを返す．

■ **フレームのサイズの取得**

フレームオブジェクトのプロパティ shape の第 0 番目の要素にはフレームの高さが，第 1 番目の要素にはフレームの横幅が格納されている．

例. フレーム im のサイズの取得

```
>>> im.shape[1], im.shape[0]   Enter      ←サイズの取得
(199, 67)      ← 199 × 67 のサイズが得られた
```

1.1.6 色空間とチャネル

色は決まった種類の成分から構成され，それら成分を（色の）**チャネル**と呼ぶ．色の成分として最も基本的なものに **RGB**（赤，緑，青）があり，これに基づく色の合成を**加法混色**（加色混合）[1] と呼ぶ．RGB 以外にも色の成分のとり方には各種のものがあり，**CMY** （シアン，マゼンタ，イエロー）の組み合わせによる色の合成を**減法混色**（減色混合）[2] という．

色の成分（チャネル）と，それらの組み合わせで得られる色の種類の対応を**色空間**という．

OpenCV を用いてコンピュータビジョンに関する処理を行う場合に多く用いられる色空間を表 2 に示す．

表 2: OpenCV でよく使用される色空間

色空間	解説
RGB	R（赤），G（緑），B（青）のチャネルで構成される最も標準的な色空間． $0 \leq \mathrm{R} \leq 255$，$0 \leq \mathrm{G} \leq 255$，$0 \leq \mathrm{B} \leq 255$ matplotlib や Pillow などで画像を扱う際にはこの色空間を採用する．
BGR	B（青），G（緑），R（赤）のチャネルで構成される． OpenCV における暗黙の色空間で，RGB の逆．
GRAY	モノクロの階調
HSV	H（色相），S（彩度），V（明度）のチャネルで構成される． $0 \leq \mathrm{H} \leq 180$，$0 \leq \mathrm{S} \leq 255$，$0 \leq \mathrm{V} \leq 255$　（値の範囲は OpenCV 独自のもの）
Lab	明度 L と，a，b の補色次元のチャネルで構成される．(CIE1976 $L^*a^*b^*$ の変種) $0 \leq \mathrm{L} \leq 255$，$0 \leq \mathrm{a} \leq 255$，$0 \leq \mathrm{b} \leq 255$　（値の範囲は OpenCV 独自のもの）
YCrCb	輝度 Y と赤の輝度差 Cr，青の輝度差 Cb のチャネルで構成される．Y，Cr，Cb の値の範囲は 0〜255 である．YCbCr と呼ばれるもの（JPEG，MPEG）と本質的に同じであるが，チャネルの順序が異なる．主にデジタル画像用の色空間である．
YUV	上記（YCrCb）と同様であるが，主にアナログ映像用（PAL，NTSC）の色空間である．各チャネルの値の範囲は 0〜255 である．

* OpenCV の色空間変換は内部で 8 ビット整数（0〜255）にスケーリングされるため，数学的に定義された値域とは異なる場合がある．例えば，HSV の H は 0〜360 ° ではなく 0〜180 に縮小され，Lab の a,b は本来の範囲 $[-128, 127]$ を 0〜255 にシフトして表現している．

* RGB と BGR は末尾に α チャネル（不透明度）を取ることができる．

* この表に挙げたもの以外にも多くの色空間が OpenCV では扱える．

* 各チャネルの値の範囲は 8 ビットの場合を想定している

1.1.6.1 画像フレームのチャネルの順序

OpenCV の画像フレームでは，画素の色成分の順序（チャネルの順序）が BGR α であり，標準的な RGB α の順序と異なっている．従って，OpenCV の画像フレームを matplotlib [3] などの作図用ライブラリで扱う際には注意が必要である．これに関しては「3.1.27.3 OpenCV における画素の配列」（p.171）でも解説する．

[1] 多くの画像ファイル（画像フォーマット）で採用されている．

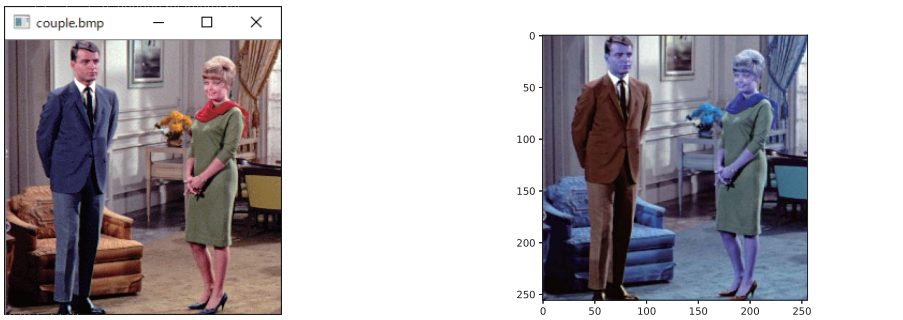
[2] プリンタのインクやトナーのための基本的な色の構成．

[3] 「3.1 数値計算と可視化のためのライブラリ：NumPy / matplotlib」（p.51〜）で解説する．

例. チャネルの順序に関する確認

```
>>> import cv2  Enter      ← OpenCV ライブラリの読込み
>>> im = cv2.imread( 'couple.bmp' )  Enter      ←画像ファイル[4] の読込み
>>> cv2.imshow( 'couple.bmp', im )  Enter      ← OpenCV の機能を用いて画像を表示
>>> while True:  Enter      ←キー入力受付サイクル
...       k = cv2.waitKey(10)  Enter
...       if k == 27:  break  Enter
...  Enter      ←キー入力受付サイクルの記述の終了
>>> cv2.destroyAllWindows()  Enter      ←ウィンドウの終了処理
```

これを実行すると図 1 の (a) のように正常に表示される.



(a) OpenCV の imshow 関数による表示　　(b) matplotlib の imshow 関数による表示

図 1: 異なるライブラリによる表示の比較

しかし，次の例のような操作で同じオブジェクト im を matplotlib で表示すると，チャネルの順序が OpenCV と異なることが原因となって図 1 の (b) のように異常な発色となる.

例. matplotlib による画像フレームの表示（先の例の続き）

```
>>> import matplotlib.pyplot as plt  Enter      ← matplotlib ライブラリの読込み
>>> g = plt.imshow( im )  Enter      ← matplotlib の機能を用いて画像を表示
>>> plt.show()  Enter      ←作図の実行
```

OpenCV の画像フレームを matplotlib で正しく扱うためには，例えば次のように変換処理を施す必要がある.

例. BGR から RGB への変換（先の例の続き）

```
>>> im2 = im[:,:,[2,1,0]]  Enter      ←チャネルの順序を変更
>>> g = plt.imshow( im2 )  Enter      ← matplotlib の機能を用いて画像を表示
>>> plt.show()  Enter      ←作図の実行
```

この処理の結果，図 2 のように（正常に）表示される.

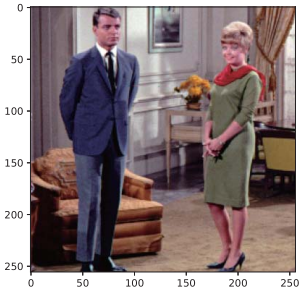


図 2: チャネルの順序を修正後，matplotlib で表示

BGR ⇔ RGB 間の変換は，次に説明する cvtColor でも可能である.

1.1.6.2　色空間の変換

cvtColor を使用することで画像データの色空間を変換することができる.

書き方：　cvtColor(画像フレーム, 変換指定)

[4]標準画像データベース SIDBA から引用.

引数に与えた「画像フレーム」の色空間を「変換指定」に従って変換したものを返す．例えば，画像の色空間を BGR から RGB に変換するには次のようにする．

例. BGR から RGB への変換（先の例の続き）

```
>>> imRGB = cv2.cvtColor( im, cv2.COLOR_BGR2RGB )  Enter    ← BGR から RGB に変換
>>> plt.imshow( imRGB )  Enter    ← matplotlib の機能を用いて画像を表示
<matplotlib.image.AxesImage object at 0x0000019E75E42F48>
>>> plt.show()  Enter    ←作図の実行
```

この結果，図 2 と同じ結果が表示される．

cvtColor の第 2 引数に与える「変換指定」の一部を表 3 に示す．

表 3: cvtColor の引数に与える「変換指定」の値（使用頻度の高いもの）

変換指定（cv2 内での定義）	解　説	変換指定（cv2 内での定義）	解　説
COLOR_BGR2RGB	BGR → RGB	COLOR_BGR2GRAY	BGR →グレースケール
COLOR_RGB2BGR	RGB → BGR	COLOR_RGB2GRAY	RGB →グレースケール
COLOR_GRAY2BGR	グレースケール→ BGR	COLOR_BGR2HSV	BGR → HSV
COLOR_GRAY2RGB	グレースケール→ RGB	COLOR_RGB2HSV	RGB → HSV
COLOR_BGR2LAB	BGR → Lab	COLOR_HSV2BGR	HSV → BGR
COLOR_RGB2LAB	RGB → Lab	COLOR_HSV2RGB	HSV → RGB
COLOR_LAB2BGR	Lab → BGR	COLOR_BGR2YCrCb	BGR → YCrCb
COLOR_LAB2RGB	Lab → RGB	COLOR_YCrCb2BGR	YCrCb → BGR
COLOR_BGR2YUV	BGR → YUV		
COLOR_YUV2BGR	YUV → BGR		

注意）グレースケールからカラーへの変換は着色を意味しない，形式上の変換である．

▲▲ 注意 ▲▲

OpenCV の利用においては画像フレームの色空間を常に意識すること．
OpenCV における**基本的な色空間は BGR である**．他のライブラリを併用して画像を取り扱う場合は RGB に変換することも多いので特に注意すること．

1.1.6.3　チャネルの分解と合成

画像フレームのチャネルの分解と合成には split, merge を使用する．

《チャネルの分解と合成》

分解：　cv2.split(**画像フレーム**)

引数に与えた「画像フレーム」のチャネルを分解し，**(第 1 チャネル, 第 2 チャネル, …)** のタプルを返す．

合成：　cv2.merge(**チャネルのタプル**)

与えた「チャネルのタプル」の順に合成した画像フレームを返す．

画像を読み込んで，分解と再合成をするプログラム opencv03.py を次に示す．

プログラム：opencv03.py

```
1  import cv2
2
3  # 画像ファイルの読込み（BGR）
4  frame = cv2.imread('Earth.jpg',cv2.IMREAD_UNCHANGED)
5
6  # リサイズ
7  f = cv2.resize(frame,(320,320))
8
```

```
# 色分解（BGR）
(f_b, f_g, f_r) = cv2.split(f)

# 表示
cv2.imshow('Red',f_r)        # 赤の成分
cv2.imshow('Green',f_g)      # 緑の成分
cv2.imshow('Blue',f_b)       # 青の成分

# 再度合成（BGR）
f_bgr = cv2.merge( (f_b, f_g, f_r) )
# 表示
cv2.imshow("All",f_bgr)

# 待ち
while True:
        # キーボードの読取り
        k = cv2.waitKey(1)
        if k == 27:      # ESCなら終了
            break

# 終了処理
cv2.destroyAllWindows() # ウィンドウの消去
```

このプログラムを実行すると，画像ファイル Earth.jpg を読み込み，青，緑，赤の各チャネルに分解したものをグレースケールで表示する．さらに，それらを再度合成したカラー画像を表示する．

参考）各色のチャネルの分解と合成は，NumPy の配列操作としても実行でき，高速である．詳しくは後の「3.1 数値計算と可視化のためのライブラリ：NumPy / matplotlib」（p.51）を参照のこと．

1.1.6.4 色空間の選択に関すること

コンピュータビジョンの分野では，処理の目的に応じて画像の色空間を選択する．ここでは，RGB の画像を HSV，Lab に変換し，それらの色空間で各チャネルがどのようなものになるかを例示する．

例. サンプル画像の読込みと表示

```
>>> import cv2   Enter      ← OpenCV ライブラリの読込み
>>> import matplotlib.pyplot as plt   Enter      ← matplotlib ライブラリの読込み
>>> imBGR = cv2.imread('Pepper.bmp')   Enter      ←サンプル画像[5] の読込み
>>> imRGB = cv2.cvtColor( imBGR, cv2.COLOR_BGR2RGB )   Enter      ← RGB に変換
>>> g = plt.imshow( imRGB )   Enter      ← matplotlib で画像を表示
>>> plt.show()   Enter      ←作図を実行
```

これにより図 3 の様にサンプル画像が表示される．

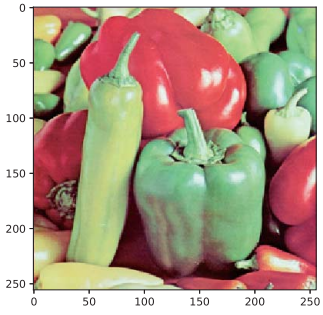


図 3: サンプル画像 'Pepper.bmp'

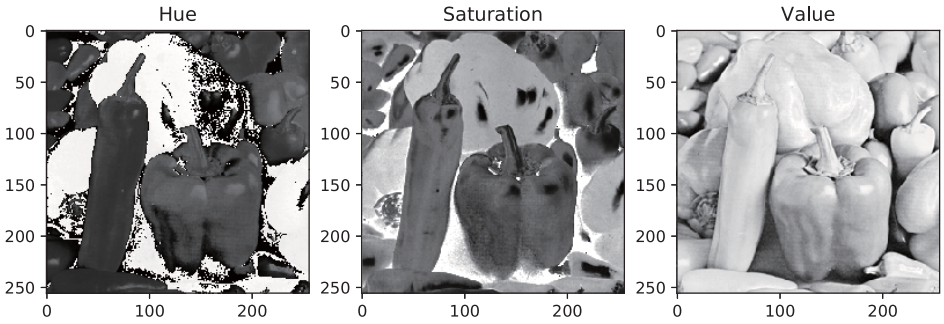
次に，この画像の色空間を HSV に変換する．

[5]標準画像データベース SIDBA から引用.

例. HSV に変換し，各チャネルをグレースケールで表示する（先の例の続き）

```
>>> imHSV = cv2.cvtColor( imBGR, cv2.COLOR_BGR2HSV )  Enter    ← HSV に変換
>>> (imH,imS,imV) = cv2.split( imHSV )  Enter    ←各チャネルの取り出し
>>> (fig,ax) = plt.subplots( 1,3, figsize=(10,4) )  Enter    ←複数の作図のための設定
>>> a1 = ax[0].imshow( imH, cmap=plt.cm.gray )  Enter    ← H チャネルの作図
>>> t1 = ax[0].set_title('Hue')  Enter
>>> a2 = ax[1].imshow( imS, cmap=plt.cm.gray )  Enter    ← S チャネルの作図
>>> t2 = ax[1].set_title('Saturation')  Enter
>>> a3 = ax[2].imshow( imV, cmap=plt.cm.gray )  Enter    ← V チャネルの作図
>>> t3 = ax[2].set_title('Value')  Enter
>>> f = plt.show()  Enter    ←作図を実行
```

これにより図 4 の様に HSV の各チャネルの画像が表示される．



H チャネル（色相）　S チャネル（彩度）　V チャネル（明度）

図 4: HSV に変換した後の各チャネル

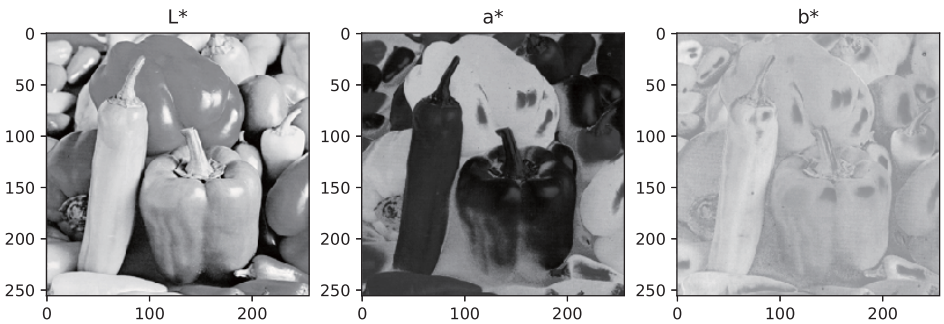
補足）HSV の H 値は 0–179 に制限され，OpenCV 内部で 8bit に収められているため，表示すると粗く見えることがある．

次に，色空間を Lab に変換する．

例. Lab に変換し，各チャネルをグレースケールで表示する（先の例の続き）

```
>>> imLab = cv2.cvtColor( imBGR, cv2.COLOR_BGR2LAB )  Enter    ← Lab に変換
>>> (imL,ima,imb) = cv2.split( imLab )  Enter    ←各チャネルの取り出し
>>> (fig,ax) = plt.subplots( 1,3, figsize=(10,4) )  Enter    ←複数の作図のための設定
>>> a1 = ax[0].imshow( imL, cmap=plt.cm.gray )  Enter    ← L チャネルの作図
>>> t1 = ax[0].set_title('L*')  Enter
>>> a2 = ax[1].imshow( ima, cmap=plt.cm.gray )  Enter    ← a チャネルの作図
>>> t2 = ax[1].set_title('a*')  Enter
>>> a3 = ax[2].imshow( imb, cmap=plt.cm.gray )  Enter    ← b チャネルの作図
>>> t3 = ax[2].set_title('b*')  Enter
>>> plt.show()  Enter    ←作図を実行
```

これにより図 5 の様に Lab の各チャネルの画像が表示される．



L チャネル（明度）　a チャネル（補色次元 1）　b チャネル（補色次元 2）

図 5: Lab に変換した後の各チャネル

補足）Lab では L:0–100，a/b:− 128〜127 を 0–255 にマッピングして表現している．

ここで示した様に，色空間の各チャネルによって際立つものが異なる．このことを応用して，画像から物体を認識する処理などにおいては，色空間と色チャネルを適宜選択する．

1.1.6.5　参考）HSV の色相に関すること

HSV 色空間の色相（H チャネル）を応用すると，画像を構成する画素の「色の種類」を判別することができる．色相は 0 ° 〜 360 ° の円環構造となるが，OpenCV では 0〜179 の範囲の円環構造（図 6）で表現する．

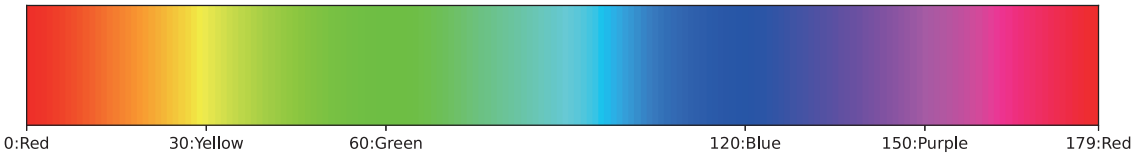


図 6: OpenCV における HSV 空間の色相（H チャネル）

図 6 を描画するためのプログラムを opencv05.py に示す．

プログラム：opencv05.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import cv2
4
5  # 0〜180の色相を図示するための画像
6  H = np.array([list(range(0,180))]*20).astype('uint8')    # 0 <= H < 180
7  S = np.asarray( [[255]*180]*20 ).astype('uint8')         # 0〜255
8  V = np.asarray( [[255]*180]*20 ).astype('uint8')         # 0〜255
9
10 imHSV = cv2.merge( (H,S,V) )                  # 上で作成した各チャネルを合成して
11 imRGB = cv2.cvtColor( imHSV, cv2.COLOR_HSV2RGB )     # 表示用にRGBに変換
12
13 plt.figure( figsize=(12,2) )
14 plt.imshow(imRGB)
15 plt.xlim(0,179)
16 plt.xticks([0,30,60,120,150,179],
17         ['0:Red','30:Yellow','60:Green','120:Blue','150:Purple','179:Red'])
18 plt.yticks(ticks=[])
19 plt.show()
```

1.1.7　カラー画像からモノクロ画像への変換

cvtColor を用いて，カラー画像をモノクロ画像（グレースケール）に変換することができる．

例. カラー画像からモノクロ画像への変換

```
>>> import cv2   Enter      ← OpenCV ライブラリの読込み
>>> import matplotlib.pyplot as plt   Enter      ← matplotlib ライブラリの読込み
>>> im = cv2.imread( 'couple.bmp' )   Enter      ←画像ファイル [-2pt] [6] の読込み
>>> imGr = cv2.cvtColor( im, cv2.COLOR_BGR2GRAY )   Enter      ←グレースケールへの変換
>>> plt.imshow( imGr, cmap=plt.cm.gray )   Enter      ←画像の表示
>>> plt.show()   Enter      ←作図の実行
```

この処理の結果，図 7 のような画像が表示される．

1.1.7.1　白黒 2 値化 (1)：一定の閾値で判定

threshold 関数を使用すると，画像を「白」(255) と「黒」(0) の 2 階調に変換することができる．

書き方 (1)：　threshold(画像フレーム, 閾値, 設定する値, cv2.THRESH_BINARY)

「閾値」以上の値の画素を「設定する値」に置き換える．また，閾値を自動設定する次のような方法もある．

書き方 (2)：　threshold(画像フレーム, 0, 設定する値,
cv2.THRESH_BINARY | cv2.THRESH_OTSU)

threshold 関数は，タプル（**閾値, 変換後の画像フレーム**）を返す．

[6]標準画像データベース SIDBA から引用.

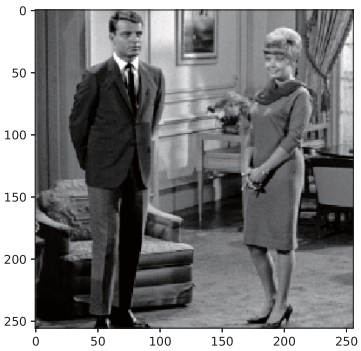


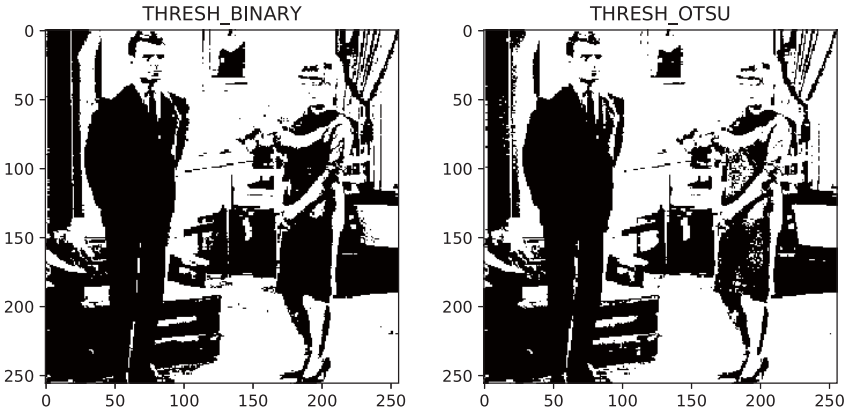
図 7: カラー画像からモノクロ画像（グレースケール）への変換

この関数を用いてグレースケール画像を 2 値化する例を示す.

例. グレースケール画像の 2 値化（先の例の続き）

```
>>> (r1, imBW1) = cv2.threshold( imGr, 105, 255, cv2.THRESH_BINARY )  Enter    ← 2 値化 (1)
>>> (r2, imBW2) = cv2.threshold( imGr, 0, 255,  Enter
...                           cv2.THRESH_BINARY | cv2.THRESH_OTSU )  Enter    ← 2 値化 (2)
>>> (fig,ax) = plt.subplots( 1,2, figsize=(9,4) )  Enter    ←描画処理の開始
>>> a1 = ax[0].imshow( imBW1, cmap=plt.cm.gray )  Enter    ←表示処理 (1)
>>> t1 = ax[0].set_title('THRESH_BINARY')  Enter
>>> a2 = ax[1].imshow( imBW2, cmap=plt.cm.gray )  Enter    ←表示処理 (2)
>>> t2 = ax[1].set_title('THRESH_OTSU')  Enter
>>> plt.show()  Enter    ←描画の実行
```

この処理の結果図 8 のような画像が表示される.



(a) cv2.THRESH_BINARY による 2 値化　(b) cv2.THRESH_OTSU による 2 値化

図 8: グレースケールを 2 値化した画像

1.1.7.2　白黒 2 値化 (2)：閾値を適応的に判定

またこれらとは別に，adaptiveThreshold 関数を用いる 2 値化の方法もある.

書き方：　adaptiveThreshold(画像フレーム, 設定する値, 閾値の算出方法,
cv2.THRESH_BINARY, 近傍の画素のサイズ, 閾値の調整)

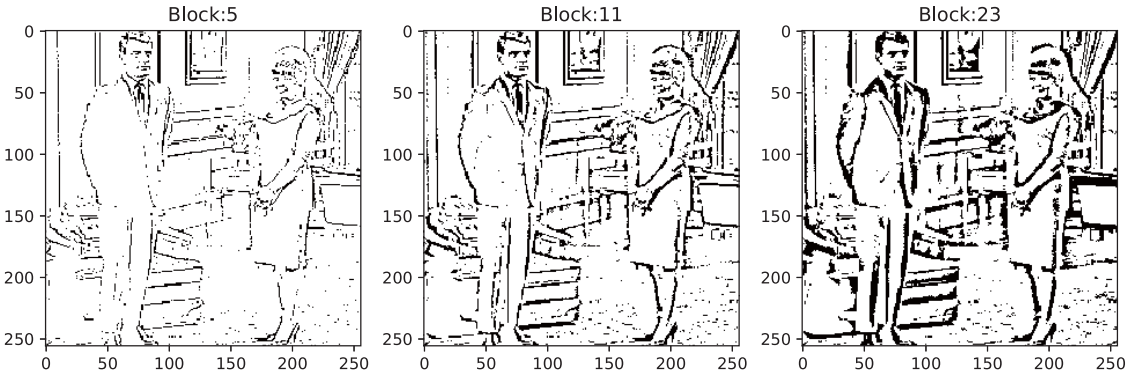
この関数は 2 値化処理の対象の画素の近傍の画素（「近傍の画素のサイズ」の四方）から閾値を自動的に算出し，変換後の画素を 0 にするか「設定する値」にするかを判断する.「閾値の調整」には，自動的に算出した閾値から更に調整するための値を与える.「閾値の算出方法」には cv2.ADAPTIVE_THRESH_GAUSSIAN_C や cv2.ADAPTIVE_THRESH_MEAN_C などを与える．この関数は変換後の画像フレームを返す.

「近傍の画素のサイズ」を [5, 11, 23] と変えながら画像を 2 値化する例を次に示す.

例. adaptiveThreshold による 2 値化（先の例の続き）

```
>>> (fig,ax) = plt.subplots( 1,3, figsize=(12,4) )  Enter    ←描画処理の開始
>>> for i,n in enumerate([5,11,23]):  Enter    ← for 文による繰り返しの開始
...     imAdpt = cv2.adaptiveThreshold( imGr, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,  Enter
...                 cv2.THRESH_BINARY, n, 15 )  Enter
...     a = ax[i].imshow( imAdpt, cmap=plt.cm.gray )  Enter    ←表示処理
...     t = ax[i].set_title('Block:'+str(n) )  Enter
...  Enter    ← for 文による繰り返しの終了
>>> plt.show()  Enter    ←描画の実行
```

この処理の結果，図 9 のような画像が表示される．



(a) 近傍画素数 5　(b) 近傍画素数 11　(b) 近傍画素数 23

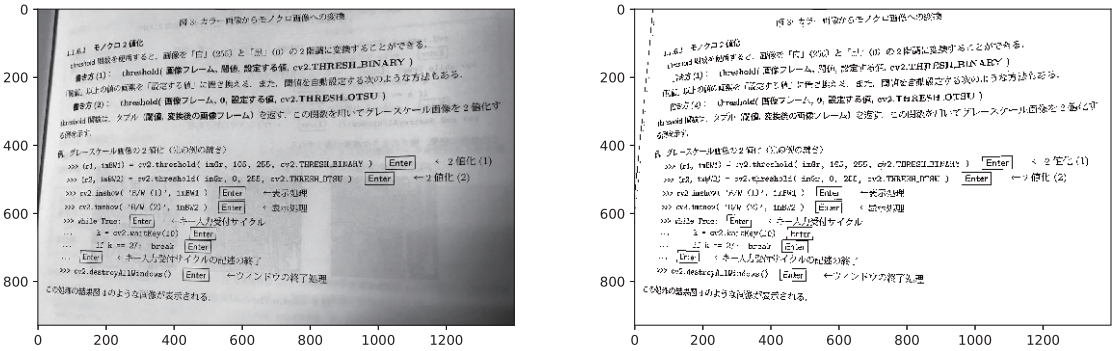
図 9: adaptiveThreshold による 2 値化

影の写り込んだ文書の画像を 2 値化する例を次に示す．

例. 影の写り込んだ文書画像の読込み

```
>>> import cv2  Enter    ← OpenCV の読込み
>>> import matplotlib.pyplot as plt  Enter    ← matplotlib の読込み
>>> imGr = cv2.imread( 'grscale01.jpg', cv2.IMREAD_GRAYSCALE )  Enter    ←画像の読込み
>>> g = plt.imshow( imGr, cmap=plt.cm.gray )  Enter    ←画像の表示処理
>>> plt.show()  Enter    ←描画の実行
```

これを実行して画像を表示した例を図 10 の (a) に示す．



(a) 元の画像　(b) 2 値化した画像

図 10: 影の写り込んだ文書画像の 2 値化

図 10 の (a) の画像を 2 値化する例を次に示す．

例. 閾値の算出方法に cv2.ADAPTIVE_THRESH_MEAN_C を指定する（先の例の続き）

```
>>> imAdpt = cv2.adaptiveThreshold( imGr, 255, cv2.ADAPTIVE_THRESH_MEAN_C,  Enter
...                              cv2.THRESH_BINARY, 5, 19 )  Enter    ← 2 値化の処理
>>> g = plt.imshow( imAdpt, cmap=plt.cm.gray )  Enter    ←画像の表示処理
>>> plt.show()  Enter    ←描画の実行
```

この例では近傍のサイズを 5 に（比較的細かく判定），閾値の調整に 19（大雑把に切り落とす）を設定している．この処理の結果，図 10 の (b) のような画像が表示される．

1.1.8　画像フレームに対する描画

OpenCV の画像フレームは NumPy の配列オブジェクトであり，配列の要素の書き換えによって描画ができる．これを応用することで，物体認識の処理の結果を元の画像の上に輪郭線や矩形を描く形でデモンストレーションすることができる．あるいは，画像の上に各種の注釈を表示したり，様々な形のマーキングを行うこともできる．

ここでは，cv2 の描画関数を用いて NumPy の配列に対して描画する形で解説する．その関係上，基本的には RGB の色空間に基づいて解説する．cv2 の HighGUI で描画結果を表示（cv2.imshow 関数など）する際は，適宜色空間を変換（cv2 の基本は BGR である）すること．また cv2 の画像オブジェクトでは，個々の画素の成分は uint8（符号なし 8 ビット整数）であるので，NumPy の配列として画像を作成する際もこの形式の配列にすること．

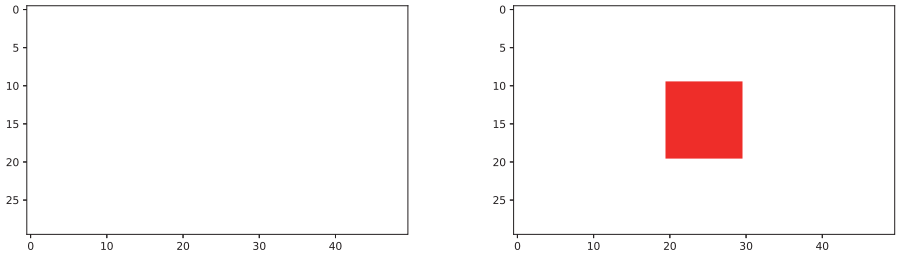
1.1.8.1　基本的な考え方

画像フレームの配列の画素を直接設定することで描画する．これに関して段階を踏んで例示する．

例. 白地の画像フレームを作成する

```
>>> import numpy as np   Enter      ← NumPy の読込み
>>> import matplotlib.pyplot as plt   Enter      ← matplotlib の読込み
>>> im = np.full( (30,50,3), 255, dtype=np.uint8 )   Enter      ←全画素が白の画像フレーム
>>> g = plt.imshow( im )   Enter      ←画像の表示
>>> plt.show()   Enter      ←描画の実行
```

これは，NumPy の配列 im を画像フレームとして作成し，matplotlib を用いて表示する例である．この結果，図 11 の (a) ような画像が表示される．



(a) 全ての画素が白の画像　　(b) 赤い矩形の描画（RGB）

図 11: 画素を直接設定して描画

次に，作成した白い画像フレームの上に赤い矩形を描く処理を示す．

例. 画像フレームに直接 RGB = [255,0,0] の画素を設定する（先の例の続き）

```
>>> im[10:20,20:30] = [255,0,0]   Enter      ←配列の要素に値を設定
>>> g = plt.imshow( im )   Enter      ←画像の表示
>>> plt.show()   Enter      ←描画の実行
```

この例では配列の 10〜19 の行範囲，20〜29 の列範囲の画素に直接「赤」を意味する値を設定している．この結果，図 11 の (b) ような画像が表示される．この処理は，画像フレームに対する破壊的な変更であるので，実際の処理では，適宜複製を作成するなどの配慮が必要となる．

注意）上に示した例は RGB 色空間によるものである．ただし，後に説明する cv2 の描画用の API では，色成分の指定は ‘**(B,G,R)**’ の順序を前提としているので，HighGUI での画像表示においては，matplotlib の色チャネルとの順序の違いを常に意識すること．また，NumPy 配列のインデックスによるアクセスは ‘**[行, 列]**’ の順序であり，cv2 では ‘**(横位置, 縦位置)**’ の順序であることに注意すること．

NumPy 配列に対する各種図形の描画や文字列の描画のための機能が OpenCV には備わっている．次に，それらに関して解説するが，必要なライブラリ（NumPy，matplotlib，cv2）は次のようにして読込み済みであることを前提として実行例を示す．

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
```

1.1.8.2　線分，矢印

line 関数で画像フレームに線分を書き込むことができる．

書き方：　line(画像フレーム, 始点, 終点, 画素, thickness=太さ, lineType=描画手法)

描画対象の「画像フレーム」に「太さ」の線分を描く．「始点」「終点」は **(横位置, 縦位置)** の形で，「画素」には線分を構成する画素の色成分を **(チャネル 1, チャネル 2,…)** の形で与える．この場合，各チャネルは，matplotlib では R, G, B に，cv2 では B, G, R に対応する．「描画手法」には表 4 に挙げた定数（cv2.LINE_AA が推奨されている）を与える．この関数は与えた「画像フレーム」を直接変更し，それを返す．

表 4: lineType に指定する値

定数表現	値	解説
LINE_4	4	4 連結線．古い表示装置や特殊用途向け．（推奨されない）
LINE_8	8	8 連結線．アンチエイリアスなしで描画が早い．（デフォルト）
LINE_AA	16	アンチエイリアスの描画．滑らかな線になるが描画が少し遅い．

arrowedLine 関数で画像フレームに矢印を書き込むことができる．

書き方：　arrowedLine(画像フレーム, 始点, 終点, 画素, thickness=太さ, tipLength=矢の長さ)

描画対象の「画像フレーム」に「太さ」の矢印を描く．「始点」「終点」「画素」に関しては line 関数の場合と同様である．矢印の矢の部分は終点に現れ，その長さは「矢の長さ」で与える．また「矢の長さ」は矢印全体の長さに対する比率で与える．

例. 線分と矢印の描画

```
>>> imRGB = np.full( (360,640,3), 255, dtype=np.uint8 )  Enter    ←白の画素配列を作成
>>> im = cv2.line( imRGB, (30,20), (610,20), (255,0,0),  Enter
...                       thickness=2, lineType=cv2.LINE_AA )  Enter    ←赤い線分の描画
>>> im = cv2.line( imRGB, (30,45), (610,45), (0,255,0),  Enter
...                       thickness=8, lineType=cv2.LINE_AA )  Enter    ←緑の線分の描画
>>> im = cv2.line( imRGB, (30,90), (610,90), (0,0,255),  Enter
...                       thickness=32, lineType=cv2.LINE_AA )  Enter    ←青の線分の描画
>>> im = cv2.arrowedLine( imRGB, (30,135), (610,135), (0,255,255),  Enter
...                       thickness=2, tipLength=0.03 )  Enter    ←シアンの矢印の描画
>>> im = cv2.arrowedLine( imRGB, (30,200), (610,200), (255,0,255),  Enter
...                       thickness=8, tipLength=0.07 )  Enter    ←マゼンタの矢印の描画
>>> im = cv2.arrowedLine( imRGB, (30,300), (610,300), (255,255,0),  Enter
...                       thickness=32, tipLength=0.1 )  Enter    ←黄色の矢印の描画
>>> f1 = plt.imshow( imRGB )  Enter    ←画像の表示
>>> plt.show()  Enter    ←描画の実行
```

この結果，図 12 のような画像フレームが表示される．

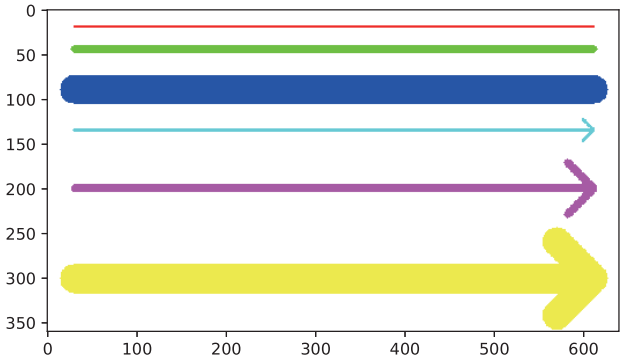


図 12: 線分と矢印の描画

1.1.8.3　矩形，円，楕円，円弧

rectangle 関数で画像フレームに矩形を書き込むことができる．

書き方：　rectangle(画像フレーム, 始点, 終点, 画素, thickness=太さ, lineType=描画手法)

「始点」「終点」を対角線とする「太さ」の矩形を描く．「太さ」に負の値を与えると，矩形を塗りつぶす．「画素」「描画手法」に関しては line 関数の場合と同様である．

例. 矩形の描画

```
>>> imRGB = np.full( (150,410,3), 255, dtype=np.uint8 )  Enter    ←白の画素の配列を作成
>>> im = cv2.rectangle( imRGB, (10,10), (50,140), (255,0,0),  Enter
...                     thickness=2, lineType=cv2.LINE_AA )  Enter    ←赤い矩形の描画
>>> im = cv2.rectangle( imRGB, (60,10), (120,140), (0,255,0),  Enter
...                     thickness=8, lineType=cv2.LINE_AA )  Enter    ←緑の矩形の描画
>>> im = cv2.rectangle( imRGB, (150,25), (250,125), (0,0,255),  Enter
...                     thickness=32, lineType=cv2.LINE_AA )  Enter    ←青の矩形の描画
>>> im = cv2.rectangle( imRGB, (280,10), (400,140), (255,255,0),  Enter
...          thickness=-1, lineType=cv2.LINE_AA )  Enter ←黄色の矩形（塗りつぶし）の描画
>>> f1 = plt.imshow( imRGB )  Enter    ←画像の表示
>>> plt.show()  Enter    ←描画の実行
```

この結果，図 13 のような画像フレームが表示される．

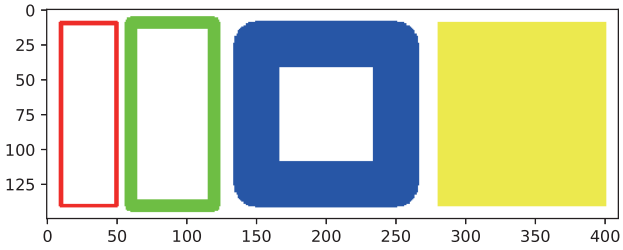


図 13: 矩形の描画

circle 関数で画像フレームに円を書き込むことができる．

書き方：　circle(画像フレーム, 中心, 半径, 画素, thickness=太さ, lineType=描画手法)

「中心」の位置に「半径」の円を「太さ」で描く．「太さ」に負の値を与えると，円を塗りつぶす．「画素」「描画手法」に関しては line 関数などの場合と同様である．

例. 円の描画

```
>>> imRGB = np.full( (100,200,3), 255, dtype=np.uint8 )  Enter    ←白の画素の配列を作成
>>> im = cv2.circle( imRGB, (50,50), 45, (255,0,0),  Enter
...                     thickness=5, lineType=cv2.LINE_AA )  Enter    ←赤い円の描画
>>> im = cv2.circle( imRGB, (150,50), 48, (0,0,255),  Enter
...                     thickness=-1, lineType=cv2.LINE_AA )  Enter    ←青い円の描画
>>> f1 = plt.imshow( imRGB )  Enter    ←画像の表示
>>> plt.show()  Enter    ←描画の実行
```

この結果，図 14 のような画像フレームが表示される．

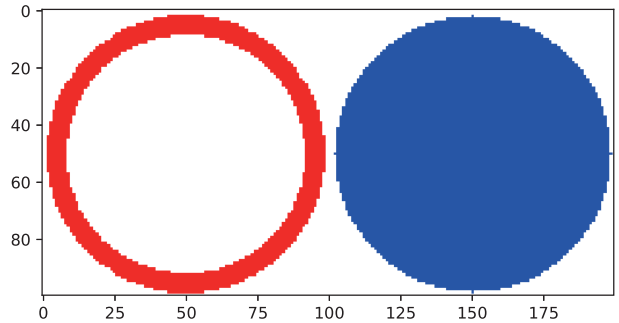


図 14: 円の描画

ellipse 関数で画像フレームに楕円を書き込むことができる.

書き方 (1)：　ellipse(画像フレーム, (中心, (横幅, 高さ), 傾斜角度), 画素,
thickness=太さ, lineType=描画手法)

「中心」の位置に「横幅」「高さ」の楕円を「太さ」で描く.「傾斜角度」には時計回りの角度（°）を与える.「太さ」に負の値を与えると，楕円を塗りつぶす.「画素」「描画手法」に関しては line 関数などの場合と同様である.

例. 楕円の描画

```
>>> imRGB = np.full( (100,200,3), 255, dtype=np.uint8 )  Enter    ←白の画素の配列を作成
>>> im = cv2.ellipse( imRGB, ((50,50), (90,50), 0), (255,0,0),  Enter
...                       thickness=4, lineType=cv2.LINE_AA )  Enter    ←赤い楕円の描画
>>> im = cv2.ellipse( imRGB, ((100,50), (90,50), 60), (0,255,0),  Enter
...                       thickness=4, lineType=cv2.LINE_AA )  Enter    ←緑の楕円の描画
>>> im = cv2.ellipse( imRGB, ((150,50), (90,50), 0), (0,0,255),  Enter
...                       thickness=-1, lineType=cv2.LINE_AA )  Enter    ←青い楕円の描画
>>> f1 = plt.imshow( imRGB )  Enter    ←画像の表示
>>> plt.show()  Enter    ←描画の実行
```

この結果，図 15 のような画像フレームが表示される.

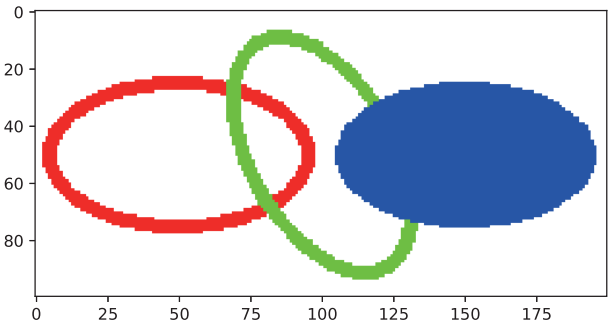


図 15: 楕円の描画

ellipse 関数は画像フレームに円弧（楕円弧）を書き込むこともできる.（先の場合とは引数の与え方が異なる）

書き方 (2)：　ellipse(画像フレーム, 中心, (横幅, 高さ), 傾斜角度, 開始角度, 終了角度, 画素,
thickness=太さ, lineType=描画手法)

「中心」の位置に，円弧（楕円弧）の元になる楕円を「横幅」「高さ」「太さ」で想定し,「開始角度」から「終了角度」までの範囲の弧を描く．角度は時計回りで単位は「°」である．他の引数に関しては先の書き方 (1) に準ずる.

例. 楕円弧の描画

```
>>> imRGB = np.full( (80,270,3), 255, dtype=np.uint8 )  Enter    ←白の画素の配列を作成
>>> im = cv2.ellipse( imRGB, (55,40), (50,30), 0,  Enter
...                       30, 240, (255,0,0),  Enter
...                       thickness=4, lineType=cv2.LINE_AA )  Enter    ←赤い楕円弧の描画
>>> im = cv2.ellipse( imRGB, (150,40), (50,30), 0,  Enter
...                       120, 330, (0,255,0),  Enter
...                       thickness=4, lineType=cv2.LINE_AA )  Enter    ←緑の楕円弧の描画
>>> im = cv2.ellipse( imRGB, (232,40), (50,30), -30,  Enter
...                       120, 330, (0,0,255),  Enter
...                       thickness=4, lineType=cv2.LINE_AA )  Enter    ←青い楕円弧の描画
>>> f1 = plt.imshow( imRGB )  Enter    ←画像の表示
>>> plt.show()  Enter    ←描画の実行
```

この結果，図 16 のような画像フレームが表示される.

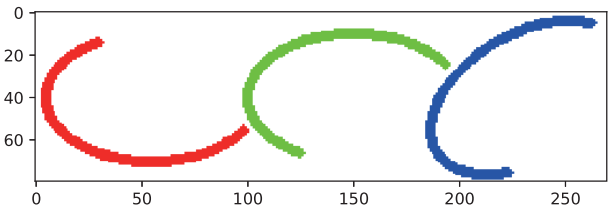


図 16: 楕円弧の描画

1.1.8.4 折れ線，多角形

polylines 関数で画像フレームに折れ線を書き込むことができる．

書き方： **polylines(画像フレーム, [座標データ], クローズ選択, 画素, thickness=太さ, lineType=描画手法)**

「座標データ」は [[x1,y1], [x2,y2],…, [xn,yn]] の形の配列（np.int32 型の配列）であり，描く折れ線の始点，コーナー，終点の座標を保持する．「クローズ選択」に True を与えると始点と終点を結び，False を与えると始点と終点が開いた折れ線となる．「画素」「太さ」「描画手法」に関しては line 関数などの場合と同様である．

fillPoly 関数で画像フレームに塗りつぶしの多角形を書き込むことができる．

書き方： **fillPoly(画像フレーム, [座標データ], 画素)**

「座標データ」については polylines に準ずる．

例. 各種の折れ線の描画

```
>>> imRGB = np.full( (90,150,3), 255, dtype=np.uint8 )  Enter    ←白の画素の配列を作成
>>> pl1 = np.array([[10,40],[10,10],[40,40],[40,10],[70,40]],  Enter
...                                          dtype=np.int32)  Enter ←折れ線 1 の座標データ
>>> im = cv2.polylines( imRGB, [pl1], False, (255,0,0),  Enter
...                            thickness=4, lineType=cv2.LINE_AA )  Enter    ←折れ線 1 の描画
>>> pl2 = np.array([[10,50],[10,80],[40,50],[40,80],[70,50]],  Enter
...                                          dtype=np.int32)  Enter ←折れ線 2 の座標データ
>>> im = cv2.polylines( imRGB, [pl2], True, (0,0,255),  Enter
...                            thickness=4, lineType=cv2.LINE_AA )  Enter    ←折れ線 2 の描画
>>> pl3 = np.array([[80,20],[110,40],[110,20],[140,45],  Enter
...              [110,70],[110,50],[80,70]], dtype=np.int32)  Enter    ←折れ線 3 の座標データ
>>> im = cv2.fillPoly( imRGB, [pl3], (0,255,0) )  Enter    ←折れ線 2 の描画
>>> f1 = plt.imshow( imRGB )  Enter    ←画像の表示
>>> plt.show()  Enter    ←描画の実行
```

この結果，図 17 のような画像フレームが表示される．

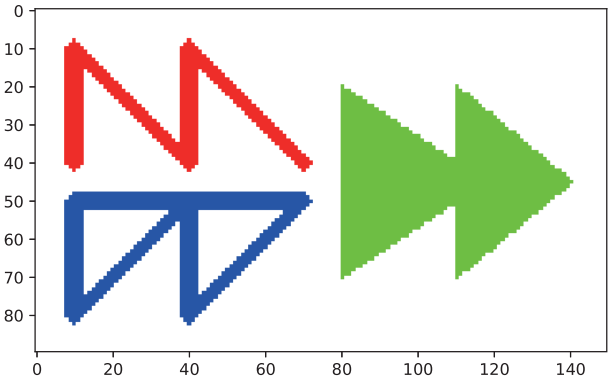


図 17: 折れ線の描画

fillConvexPoly 関数で画像フレームに塗りつぶしの多角形（凸）を書き込むことができる．

書き方： **fillConvexPoly(画像フレーム, 座標データ, 画素)**

引数については fillPoly に準ずる．

例. 多角形の描画

```
>>> imRGB = np.full( (100,120,3), 255, dtype=np.uint8 )  Enter   ←白の画素の配列を作成
>>> pl1 = np.array([[60,10],[110,35],[110,65],  Enter
...              [60,90],[10,65],[10,35]], dtype=np.int32)  Enter ←多角形の頂点の座標データ
>>> im = cv2.fillConvexPoly( imRGB, pl1, (255,0,255) )  Enter   ←多角形の描画
>>> f1 = plt.imshow( imRGB )  Enter   ←画像の表示
>>> plt.show()  Enter   ←描画の実行
```

この結果，図 18 のような画像フレームが表示される．

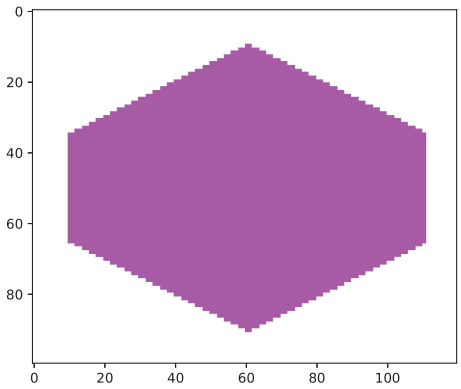


図 18: 多角形（凸）の描画

1.1.8.5 文字列

putText 関数で画像フレームに文字列を書き込むことができる．

書き方：　putText(画像フレーム, 文字列, 位置, フォント, 比率, 画素,
thickness=太さ, lineType=描画手法)

「位置」に「文字列」を表示する．この場合の「位置」は文字列の左下を基準とする．「比率」には文字の標準の大きさに対する比率を与える．「画素」「太さ」「描画手法」に関しては line 関数などの場合と同様である．「フォント」に指定できるものを表 5 に挙げる．

表 5: putText 関数に与えるフォント（一部）

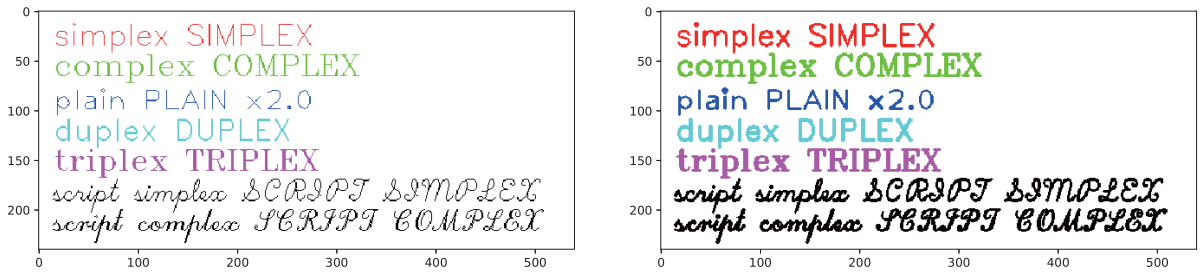
FONT_HERSHEY_SIMPLEX	FONT_HERSHEY_COMPLEX	FONT_HERSHEY_PLAIN
FONT_HERSHEY_DUPLEX	FONT_HERSHEY_TRIPLEX	FONT_HERSHEY_SCRIPT_SIMPLEX
FONT_HERSHEY_SCRIPT_COMPLEX		

表 5 に挙げたフォントで文字列を描画する例を示す．

例. 文字列の描画

```
>>> imRGB = np.full( (240,540,3), 255, dtype=np.uint8 )  Enter   ←白の画素の配列を作成
>>> im = cv2.putText( imRGB, 'simplex SIMPLEX', (15, 35),  Enter   ←文字列 1 の描画
...                   cv2.FONT_HERSHEY_SIMPLEX, 1.0, (255,0,0), thickness=1)  Enter
>>> im = cv2.putText( imRGB, 'complex COMPLEX', (15, 65),  Enter   ←文字列 2 の描画
...                   cv2.FONT_HERSHEY_COMPLEX, 1.0, (0,255,0), thickness=1)  Enter
>>> im = cv2.putText( imRGB, 'plain PLAIN x2.0', (15, 100),  Enter   ←文字列 3 の描画
...                   cv2.FONT_HERSHEY_PLAIN, 2.0, (0,0,255), thickness=1)  Enter
>>> im = cv2.putText( imRGB, 'duplex DUPLEX', (15, 130),  Enter   ←文字列 4 の描画
...                   cv2.FONT_HERSHEY_DUPLEX, 1.0, (0,255,255), thickness=1)  Enter
>>> im = cv2.putText( imRGB, 'triplex TRIPLEX', (15, 160),  Enter   ←文字列 5 の描画
...                   cv2.FONT_HERSHEY_TRIPLEX, 1.0, (255,0,255), thickness=1)  Enter
>>> im = cv2.putText( imRGB, 'script simplex SCRIPT SIMPLEX', (15, 190), Enter ←文字列 6 の描画
...                   cv2.FONT_HERSHEY_SCRIPT_SIMPLEX, 1.0, (0,0,0), thickness=1)  Enter
>>> im = cv2.putText( imRGB, 'script complex SCRIPT COMPLEX', (15, 220), Enter ←文字列 7 の描画
...                   cv2.FONT_HERSHEY_SCRIPT_COMPLEX, 1.0, (0,0,0), thickness=1)  Enter
>>> f1 = plt.figure( figsize=(8,4) )  Enter   ←画像サイズの設定
>>> f2 = plt.imshow( imRGB )  Enter   ←画像の表示
>>> plt.show()  Enter   ←描画の実行
```

この結果，図 19 のような画像フレームが表示される．



(a) thickness=1 の場合　　(b) thickness=2 の場合

図 19: 文字列の描画（thickness の値で太さを調整できる）

1.1.8.6　マーカー

drawMarker 関数で画像フレームにマーカーを書き込むことができる．

書き方：　drawMarker(画像フレーム, 位置, 画素, マーカーの種類,
thickness=太さ, line_type=描画手法)

「位置」に「マーカーの種類」で指定したマーカーを表示する．「画素」「太さ」「描画手法」に関しては line 関数などの場合と同様であるが，引数名が 'line_type=' であることに注意すること．「マーカーの種類」に指定できるものを表 6 に挙げる．

表 6: drawMarker 関数で表示できるマーカー

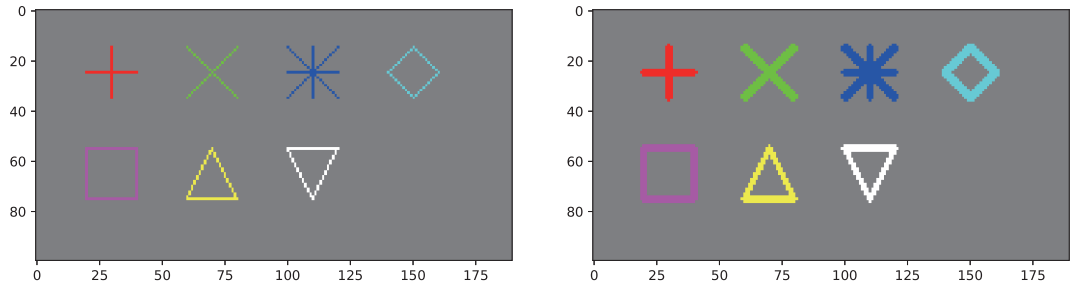
マーカーの種類	表示	マーカーの種類	表示
MARKER_CROSS	＋	MARKER_TILTED_CROSS	×
MARKER_STAR	＊	MARKER_DIAMOND	◇
MARKER_SQUARE	□	MARKER_TRIANGLE_UP	△
MARKER_TRIANGLE_DOWN	▽		

表 6 に挙げたマーカーを描画する例を示す．

例. マーカーの描画（cv2,plt は読込み済みであるとする）

```
>>> imRGB = np.full( (100,190,3), 127, dtype=np.uint8 )  Enter    ←グレーの画素の配列を作成
>>> im = cv2.drawMarker( imRGB, (30,25), (255,0,0),  Enter
...         markerType=cv2.MARKER_CROSS, markerSize=20,  Enter    ←「＋」の描画
...         thickness=1, line_type=cv2.LINE_AA )  Enter
>>> im = cv2.drawMarker( imRGB, (70,25), (0,255,0),  Enter
...         markerType=cv2.MARKER_TILTED_CROSS, markerSize=20,  Enter    ←「×」の描画
...         thickness=1, line_type=cv2.LINE_AA )  Enter
>>> im = cv2.drawMarker( imRGB, (110,25), (0,0,255),  Enter
...         markerType=cv2.MARKER_STAR, markerSize=20,  Enter    ←「＊」の描画
...         thickness=1, line_type=cv2.LINE_AA )
>>> im = cv2.drawMarker( imRGB, (150,25), (0,255,255),  Enter
...         markerType=cv2.MARKER_DIAMOND, markerSize=20,  Enter    ←「◇」の描画
...         thickness=1, line_type=cv2.LINE_AA )  Enter
>>> im = cv2.drawMarker( imRGB, (30,65), (255,0,255),  Enter
...         markerType=cv2.MARKER_SQUARE, markerSize=20,  Enter    ←「□」の描画
...         thickness=1, line_type=cv2.LINE_AA )  Enter
>>> im = cv2.drawMarker( imRGB, (70,65), (255,255,0),  Enter
...         markerType=cv2.MARKER_TRIANGLE_UP, markerSize=20,  Enter    ←「△」の描画
...         thickness=1, line_type=cv2.LINE_AA )  Enter
>>> im = cv2.drawMarker( imRGB, (110,65), (255,255,255),  Enter
...         markerType=cv2.MARKER_TRIANGLE_DOWN, markerSize=20,  Enter    ←「▽」の描画
...         thickness=1, line_type=cv2.LINE_AA )  Enter
>>> f1 = plt.imshow( imRGB )  Enter    ←画像の表示
>>> plt.show()  Enter    ←描画の実行
```

この結果，図 20 のような画像フレームが表示される．



(a) thickness=1 の場合　　(b) thickness=2 の場合

図 20: マーカーの描画（thickness の値で太さを調整できる）

1.2 Pillow

Pillow は Python で静止画像を処理するためのモジュールである．本書では Pillow に関する導入的な内容について説明する．更に詳しい事柄に関しては，インターネットサイト https://pillow.readthedocs.io/ をはじめとする情報源を参照すること．

このモジュールの使用に先立って，必要なソフトウェアを Python 処理系にインストールしておく必要がある．実際に使用する場合は，次のようにして必要なサブモジュールを Python 処理系に読み込む．

`from PIL import` **Pillow のサブモジュール**

システムにインストールされている Pillow のバージョンを確認するには `__version__` 属性を参照する．

例. Pillow のバージョン確認

```
>>> import PIL  Enter      ← PIL の読込み
>>> PIL.__version__  Enter      ←バージョンの確認
'11.3.0'              ←バージョン番号
```

1.2.1 画像ファイルの読込みと保存

画像ファイルを読み込むには，Image モジュールのメソッド open を使用する．

例. 画像ファイル test01.jpg の読込み．

```
>>> from PIL import Image  Enter          ← Image モジュールの読込み
>>> im = Image.open('test01.jpg')  Enter  ←画像ファイルの読み込み
```

《画像ファイルの読込み》

書き方： `Image.open(` **ファイルのパス** `)`

このメソッドの実行の結果，画像データが Image オブジェクトとして返される．ファイルのパスは文字列で与えるが，ファイルオブジェクトを与えることもできる．

Pillow で読み書きがサポートされている画像フォーマットの代表的なものは次の通りである．

BMP	EPS	GIF	ICNS	ICO
JPEG	JPEG 2000	MSP	PNG	PPM
SGI	TIFF	WebP	XBM	

注意）open メソッドの第 2 引数としてモードを与えることも可能であるが，推奨されない．

Image オブジェクトに対して各種の編集や処理を行うことができる．ファイルから読み込んだ Image オブジェクトには size，format，mode ， といったプロパティがあり，それぞれ画素サイズ（横縦各成分のタプル），画像フォーマット，画像モードの情報が保持されている．また info プロパティには辞書型オブジェクトとして各種情報が保持されている．

例. 画像の各種プロパティの調査

```
>>> from PIL import Image  Enter          ←パッケージの読込み
>>> im = Image.open('Earth.jpg')  Enter  ←画像ファイルの読込み
>>> type( im )  Enter  ←取得したデータの型を調べる
<class 'PIL.JpegImagePlugin.JpegImageFile'>  ← Pillow 独自のデータ型
>>> im.size  Enter          ←画素サイズの調査
(2048, 2048)              ←画素サイズ
>>> im.format  Enter          ←画像フォーマットの調査
'JPEG'                    ←画像フォーマット
>>> im.mode  Enter          ←画像モードの調査
'RGB'                     ←画像モード
>>> im.info['dpi']  Enter          ←画像解像度の調査
(300.0, 300.0)            ←画像解像度
```

画像モードは色成分の構成を意味するもので，表 7 のような種類がある．

表 7: 画像モード（一部）

画像モード	色成分の構成	画像モード	色成分の構成
'1'（数字）	白黒 2 値	'L'	8 ビットグレースケール
'RGB'	24 ビットカラー	'RGBA'	24 ビットカラー＋ α 値（8 ビット）
'CMYK'	減色混合カラー（32 ビット）	'HSV'	HSV 色空間表現によるカラー（24 ビット）

※ 各色の成分のことを**バンド**（band）と呼ぶ

info プロパティの中には 'dpi' というキーワードがあり，対象画像の解像度の値が保持されている．

1.2.1.1 EPS を読み込む際の解像度

EPS（Encapsulated PostScript）はベクトル図形[7] であり，Image オブジェクトとして EPS を読み込んだ後で画像処理を行うには，**ラスタライズ**[8] する必要がある．実際に open メソッドで EPS ファイルを読み込むと，自動的にラスタライズの処理が行われ，画素に展開される．ただし，デフォルトでは 72dpi の解像度なので，必要に応じて，得られた Image オブジェクトに対して load メソッドを実行する．この際に，キーワード引数 'scale=**解像度比**' を与える．（次の例を参照のこと）

例. EPS ファイル 'pic01.eps' を 4 倍の解像度でラスタライズする

```
>>> im = Image.open('pic01.eps')      ←'pic01.eps' を暗黙の解像度で一旦読み込む
>>> im.load(scale=4)                  ← 72 × 4dpi の解像度で再度ラスタライズ
```

注意） Pillow は EPS のラスタライズに Ghostscript を使用するので，予めこれをインストールしておく必要がある．Ghostscript 本体と関連情報は公式インターネットサイト

https://www.ghostscript.com/

から入手できる．Ghostscript は OS のコマンドサーチパスに正しく登録されている必要があるので注意すること．

1.2.1.2 画像ファイルの保存

Image オブジェクトは save メソッドを使用することでファイルに保存することができる．

《画像のファイルへの保存》

書き方：　Image オブジェクト.save(保存先のパス, オプション)

この結果 Image オブジェクトがファイルに保存される．「保存先のパス」は拡張子を伴う文字列で与える．拡張子によって，保存の際のフォーマットが決まる．

保存時の画像解像度はオプションとして引数 'dpi=(x,y)' で指定（x：水平解像度，y：垂直解像度）する．スカラー値を与えると，水平垂直とも同じ解像度が設定される．

本書では特に圧縮形式のフォーマットとして JPEG, PNG の利用頻度が高いと考え，これらのフォーマットで保存する場合のオプションについて説明する．

表 8: 画像ファイルを保存する際のオプション（一部）

フォーマット	キーワード引数	意味
JPEG	quality=**整数値**	画質の指定．1〜95 の値で値が大きいほど高画質．デフォルトは 75
PNG	compress_level=**整数値**	圧縮率の指定．0〜9 の値で値が小さいほど高画質．デフォルトは 6

[7] 線画図形，ドロー系画像などと呼ばれる．

[8] **ラスタライズ**：画素に展開すること．

1.2.2 Image オブジェクトの新規作成

Image モジュールの new メソッドを使用することで，Image オブジェクトを新規に作成することができる．

《Image オブジェクトの作成》

書き方： Image.new(**モード, (横幅, 高さ), 初期ピクセル値**)

初期ピクセル値： 生成した直後に全ての画素に与える初期値
モードが '1' の場合，初期ピクセル値は 0（黒）か 1（白），'L' の場合は 0（黒）〜255 （白），その他のモードでは各成分 0〜255 のタプル．

例. 100 × 30 ピクセルの赤い画像

```
imR = Image.new( 'RGB', (100,30), (255,0,0) )
```

この結果，imR に  のような Image オブジェクトが得られる．

新規に作成した Image オブジェクト上に描画したり，他の Image オブジェクトを配置（複写）することができる．

1.2.3 画像の閲覧

Image オブジェクトに対して show メソッドを使用すると，OS に設定されている画像ビューワが起動して当該オブジェクトの内容を表示することができる．

例. Image オブジェクト im の表示

```
>>> im.show()
```

この結果，画像ビューワが起動して im の内容が表示される．

このメソッドを使用するには，Python 処理系を実行する OS において，使用する画像ビューワの設定をしておく必要がある．

1.2.4 画像の編集

ここでは，Image オブジェクトを加工する方法について説明する．

1.2.4.1 画像の拡大と縮小

resize メソッドを使用することで Image オブジェクトの画素サイズを変更することができる．

例. Image オブジェクトの画素サイズ変更

```
im2 = im1.resize( (300,300) )
```

これは Image オブジェクト im1 を 300 × 300 の画素サイズにして，それを im2 としている例である．resize メソッドは非破壊であり，元の Image オブジェクトを変更せず，処理結果を新規に作成して返す．

画像の画素サイズを変更すると，画素が乱れて画質が劣化することが多いが resize メソッドを実行する際に，画質の劣化を軽減するための各種の**フィルタ**（表 9）を指定することができる．

例. リサイズにおけるフィルタ指定

```
im2 = im1.resize( (300,300), resample=Image.LANCZOS )
```

フィルタ選択の判断は扱う画像によって異なるので実際に実行して目視確認するのが良い．

参考）画素サイズ変更には thumbnail メソッドも使用できる．thumbnail メソッドは対象オブジェクトそのものを変更する．

1.2.4.2 画像の部分の取り出し

crop メソッドを使用すると，Image オブジェクトから矩形領域を取り出すことができる．Image オブジェクト im から部分を取り出す場合，取り出したい矩形領域の**左上の座標**を (U_x, U_y)，**右下の座標**を (L_x, L_y) とするとき次のようにする．

表 9: 各種フィルタ

フィルタ	効　果
Image.NEAREST	画素の補間に近傍の画素を使用する.（デフォルト）
Image.BOX	画素の補間に**ボックス近似**を使用する.
Image.BILINEAR	画素の補間に**線形近似**を使用する.
Image.HAMMING	線形近似より若干鮮明な補間処理.
Image.BICUBIC	画素の補間に **3 次補間**を使用する.（無難な補間）
Image.LANCZOS	Lanczos フィルタによる補間処理.（縮小時に高品質）

```
im2 = im.crop( (Ux,Uy,Lx,Ly) )
```

この結果，指定した矩形領域が Image オブジェクト im2 として得られる.

1.2.4.3　画像の複製

Image オブジェクトの複製を作成するには copy メソッドを使用する.

例. Image オブジェクト im の複製 im2 を作成する

```
im2 = im.copy()
```

1.2.4.4　画像の貼り付け

Image オブジェクトの上に別の Image オブジェクトを貼り付ける（複写する）には paste メソッドを使用する．貼り付けられる Image オブジェクトを im1, 貼り付ける Image オブジェクトを im2 とする場合，im1 上の貼り付ける位置を (P_x, P_y) とすると，次のように記述して実行する.

書き方：　`im1.paste(im2,(Px,Py))`

この結果，im1 上の (P_x, P_y) の位置に im2 の内容が貼り付けられる.（im1 自体が変更される）

1.2.4.5　画像の回転

Image オブジェクトを回転させたものを得るには rotate メソッドを使用する.

書き方：　**Image オブジェクト**.rotate(**角度**)

引数に与える角度の単位は「度」である．この処理によって，回転された結果の Image オブジェクトが返される.

回転の結果，元の画像が Image オブジェクトの画素サイズに収まらずに切れてしまうことがある．その場合は rotate メソッドの引数にキーワード引数 expand=True を与えると，回転後に画像が収まるように結果の Image オブジェクトの画素サイズが拡大される．また，回転処理によって画素が乱れることがあるが，キーワード引数 resample=**フィルタ** を与えることで乱れを軽減することができる．フィルタは基本的には表 9 に挙げたものが指定できる.（注：使用できないものもある）

rotate メソッドの他に，Image オブジェクトの 90 度単位の回転や，上下左右の反転を実行する transpose メソッドがある.

書き方：　**Image オブジェクト**.transpose(**手法**)

引数に指定した手法に従って，Image オブジェクトを回転もしくは反転したオブジェクトを返す．指定できる手法（method）を表 10 に示す.

表 10: 回転・反転の手法

method	処理
Image.FLIP_LEFT_RIGHT	左右反転
Image.FLIP_TOP_BOTTOM	上下反転
Image.ROTATE_90	反時計回り 90 度回転
Image.ROTATE_180	180 度回転
Image.ROTATE_270	時計回り 90 度回転

1.2.5 画像処理

ここでは，Image オブジェクトのピクセル値を処理（画像補正をはじめとする処理）する方法について説明する．

1.2.5.1 色の分解と合成

Image オブジェクトの色成分（バンド）を分解して，別々の Image オブジェクトとして取り出すには split メソッドを使用する．

書き方： **Image オブジェクト**.split()

これにより，各色成分に分けられた Image オブジェクトのタプルが返される．例えばモードが 'RGB' である Image オブジェクト im があるとき，

```
(r,g,b) = im.split()
```

とすると，r，g，b にそれぞれ赤，緑，青に分解された Image オブジェクトが得られる．これら Image オブジェクトの画像モードはグレースケールすなわち 'L' である．

split メソッドとは逆に，グレースケールの Image オブジェクトからカラーの Image オブジェクトを合成するには merge メソッドを使用する．

書き方： Image.merge(**画像モード, 画像のタプル**)

画像モードは表 7 のものを指定する．画像のタプルは画像モードの各色成分（バンド）とするグレースケールの Image オブジェクトを並べたのもである．

例. グレースケールの Image オブジェクト r, g, b の合成

```
im2 = Image.merge( 'RGB', (r,g,b) )
```

これにより，カラーのイメージオブジェクト im2 が得られる．

1.2.5.2 カラー画像からモノクロ画像への変換

Image オブジェクトに対して convert メソッドを使用すると，画像モード（p.23 の表 7）を変換することができる．これを応用することでカラー画像をグレースケール画像に変換することができる．このことを，以下に例を挙げて示す．

例. カラー画像の読込みと表示

```
>>> from PIL import Image  Enter      ← Pillow ライブラリの読込み
>>> im = Image.open('couple.bmp')  Enter      ←画像ファイルの読込み
>>> import numpy as np  Enter      ← NumPy ライブラリの読込み
>>> imA = np.asarray(im)  Enter      ← Image オブジェクトを NumPy の配列に変換
>>> import matplotlib.pyplot as plt  Enter      ← matplotlib ライブラリの読込み
>>> g = plt.imshow(imA)  Enter      ←配列に変換した画像を表示
>>> plt.show()  Enter      ←描画の実行
```

この処理の結果，図 21 のような画像が表示される．

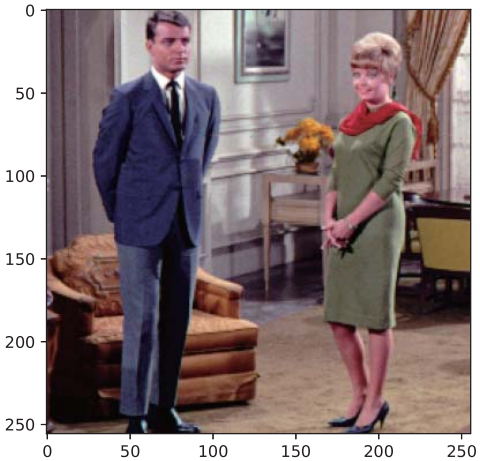


図 21: カラー画像

※ NumPy，matplotlib に関しては「3.1 数値計算と可視化のためのライブラリ：NumPy / matplotlib」（p.51〜）で解説する．

次に，この画像を convert メソッドでグレースケールに変換する．

例. グレースケール画像への変換（先の例の続き）

```
>>> imGr = im.convert('L')  Enter    ←画像モード 'L'（グレースケール）に変換
>>> imGrA = np.asarray(imGr)  Enter    ← NumPy の配列に変換
>>> plt.imshow(imGrA,cmap=plt.cm.gray)  Enter    ←配列に変換した画像を表示
>>> plt.show()  Enter    ←描画の実行
```

このように convert メソッドの引数には画像モードを与える．
この処理の結果，図 22 のような画像が表示される．

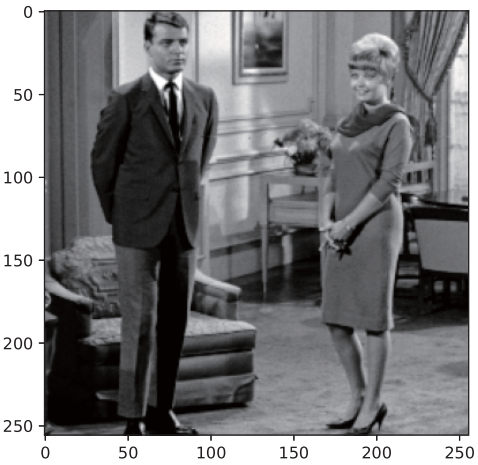


図 22: グレースケールに変換した画像

1.2.6 描画

Pillow の ImageDraw サブパッケージを使用すると，Image オブジェクトの上に図形や文字を描画することができる．描画機能を使用するには次のようにしてパッケージを読み込む．

```
from PIL import ImageDraw
```

■ 描画の考え方

Image オブジェクトに描画するには，対象の Image オブジェクトから取得した Draw オブジェクトに対して各種の描画メソッドを実行する．

例. Image オブジェクト im から Draw オブジェクト drw を取得する

```
from PIL import ImageDraw
drw = ImageDraw.Draw(im)
```

以後，この drw オブジェクトに対して各種描画メソッドを実行すると im 上に描画される．

■ 描画処理の例：直線の描画

新規に作成した Image オブジェクトに直線を描画するプログラム pillow01.py を示す．

プログラム：pillow01.py

```
1  from PIL import Image, ImageDraw
2
3  # 白いImageオブジェクトの生成
4  im = Image.new( 'RGB', (640,480), (255,255,255) )
5  # Drawオブジェクトの取得
6  drw = ImageDraw.Draw(im)
7
8  # 楕円の描画
9  drw.ellipse( [100,100,539,379], fill=(0,0,255) )
10 # 直線の描画
11 drw.line( [0,0,639,479], fill=(255,0,0), width=32 )
12
13 im.show()   # ビューワによる表示
```

解説：

7 行目で，白に初期化された Image オブジェクト im を作成し，9 行目で im から Draw オブジェクト drw を取得している．12 行目で drw に対して楕円を，14 行目で直線を描画している．楕円の描画には ellipse メソッド，直線の描画には line メソッドを使用しており，引数に座標リストと，各種のキーワード引数を与えている．キーワード引数の 'fill=' には色の成分をタプルで与える．また 'width=' には線の太さをピクセル数で与える．

これにより，im 上に楕円と直線が描かれる．このプログラムを実行すると図 23 のような画像が表示される．

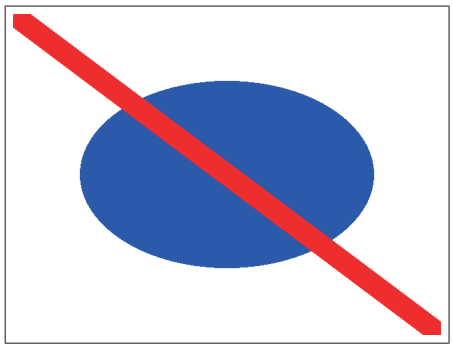


図 23: 実行結果

■ 描画メソッド

Pillow で使用できる描画メソッドの一部を紹介（表 11）する．

表 11: Draw オブジェクトに対する描画メソッド（一部）

メソッド	働き
point(座標リスト, fill=色)	点（複数）を描画する
line(座標リスト, fill=色, width=太さ)	折れ線を描画する
ellipse(座標リスト, fill=色)	楕円を描画する
arc(座標リスト, 開始角, 終了角, fill=色)	円弧を描画する
pieslice(座標リスト, 開始角, 終了角, fill=色)	パイの形を描画する
rectangle(座標リスト, fill=色)	長方形を描画する
polygon(座標リスト, fill=色)	ポリゴン（多角形）を描画する

1.2.7 アニメーション GIF の作成

Image オブジェクトに対する save メソッドに，アニメーション GIF として保存する機能がある．save メソッドの第 1 引数にファイル名を指定する際，拡張子として '.gif' を付けると GIF 形式で保存される．更に save メソッドのキーワード引数として

save_all=True, append_images=**Image オブジェクトのリスト**, duration=**フレームの表示時間**

といったものを指定するとアニメーション GIF として保存される．

例. Image オブジェクト im0, im1, im2, … , imN をアニメーション GIf として保存する手順

```
imglist = [ im1, im2, … , imN ]      ← Image オブジェクト群 im1, im2, … , imN のリストを作成
im0.save('anim.gif', save_all=True, append_images=imglist, duration=1000 )   ←保存
```

これで，アニメーション GIF が 'anim.gif' として保存される．**フレームの表示時間**にはミリ秒単位の整数値を指定する．繰り返し表示するアニメーションを作成するには，save メソッドにキーワード引数 loop=True を指定する．

1.2.8 Image オブジェクトから画素の数値配列への変換

NumPy ライブラリ[9] を使用すると，Image オブジェクトを構成する画素の配列を数値の配列に変換することができる．これに関しては「3.1.27 画像データの扱い」（p.169）で解説する．

[9]「3.1 数値計算と可視化のためのライブラリ：NumPy / matplotlib」（p.51）を参照のこと．

2 GUIとマルチメディア

2.1 pygame

pygame はウィンドウ上の描画機能や UI デバイス（キーボード，ゲーム用コントロールパッドなど）からの入力をハンドリングする機能，マルチメディア再生の機能を提供するモジュールであり，SDL [10] を用いて構築されたライブラリである．pygame はゲームプログラムを構築するために便利な機能を提供するが，ウィンドウ描画と UI デバイスのハンドリングを実現するための高速でコンパクトな汎用ライブラリと見るべきである．本書では pygame の基本的な使用方法について解説する．pygame に関する情報はインターネットサイト https://www.pygame.org/ を参照のこと．

pygame の使用に先立って，次のようにして必要なモジュールを読み込んでおく必要がある．

```
import pygame
```

2.1.1 基礎事項

pygame の機能を使用するには，最初に init 関数を呼び出して初期化処理をする必要がある．

例. pygame の初期化

```
pygame.init()
```

2.1.1.1 Surface オブジェクト

pygame では，画像データなどのビットマップを **Surface オブジェクト**として扱う．アプリケーションウィンドウも Surface オブジェクトとして生成し，その上に図形や文字を描いたり，画像などの Surface オブジェクトを貼り込む形で描画を実現する．

アプリケーションウィンドウの Surface オブジェクトは次のようにして生成する．

例. Surface オブジェクトの生成

```
sf = pygame.display.set_mode( (400,300) )
```

この例では横 400 ドット，縦 300 ドットのサイズのウィンドウが生成され，Surface オブジェクト sf として扱われる．この後，この sf 上に描画したい別の Surface オブジェクトを貼り付けたり，各種の描画メソッドで図形や文字を描く．

pygame の座標系は，多くの GUI ライブラリと共通である．すなわち，左上を原点として，横方向を X 軸（右に行くほど座標値は大），縦方向を Y 軸（下に行くほど座標値は大）とする．

2.1.1.2 アプリケーションの実行ループ

pygame を用いたアプリケーションの基本的な動作は，

1) 描画処理
アプリケーションウィンドウの Surface オブジェクトに対する描画処理である．

2) イベントハンドリング
イベントキュー[11] からイベントを取り出し，対応する処理を実行する．

3) ディスプレイの更新

を繰り返すループである．上記 3 つの処理はこの順で固定されたものではなく，実装するプログラムの仕様に合わせてユーザが自由に記述して良い．

pygame は SDL を用いて構築されているため，描画とイベント処理の実行速度が大きい．このため，上記ループにおいてアプリケーション実行のタイミングを適切に制御しなければ，システムの CPU タイムの大きな部分を（不必要に）占有することになる．pygame には，アプリケーション実行のタイミングを制御するための Clock クラスが用意されており，このクラスのオブジェクトを用いて，アプリケーションウィンドウのフレームレートを制御して CPU タイムの不必要な要求を緩和することができる．具体的には次のようにして，フレームレート制御用のオブジェクトを生成する．

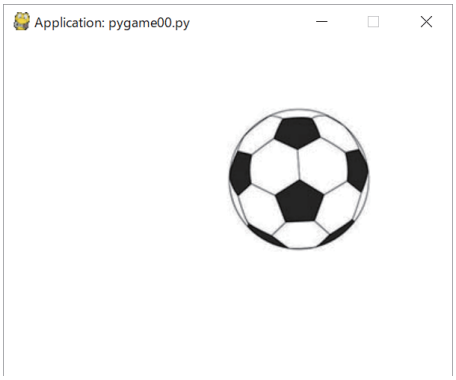
[10] SDL (Simple DirectMedia Layer) はクロスプラットフォームのマルチメディア用 API であり，グラフィックの描画やサウンドの再生などの機能を提供する．SDL は Windows(Microsoft), macOS(Apple), Linux, iOS(Apple), Android(Google) といった OS で利用できる．

[11] イベントは短時間に多くのものが発生する．アプリケーションが受け取ったイベントは**イベントキュー**に蓄積される．

```
fps = pygame.time.Clock()
```

この例では fps に Clock オブジェクトが得られており，これに対してフレームレートの設定を行う．

サンプルプログラム pygame00.py を示しながら，pygame のアプリケーションの基本的な動作について説明する．このプログラムは，ボールの画像をウィンドウに表示して，一定のフレームレートでボールを移動するものである．ボールはウィンドウの端に衝突すると反射（バウンド）する．(図 24)



ボールがウィンドウ内でバウンドする．

図 24: pygame00.py を実行したところ

プログラム：pygame00.py

```
1  import sys
2  import pygame
3
4  pygame.init()       # pygameの初期化
5
6  w = 400; h = 300    # ウィンドウサイズ
7  sf = pygame.display.set_mode( (w,h) )   # アプリケーションウィンドウ
8  pygame.display.set_caption('Application: pygame00.py')  # ウィンドウタイトル
9
10 fps = pygame.time.Clock()   # フレームレート制御のための Clock オブジェクト
11
12 im1 = pygame.image.load('ball01.jpg')   # 画像の読み込み
13 im1_w = im1.get_width()                 # 画像の横幅の取得
14 im1_h = im1.get_height()                # 画像の高さの取得
15
16 # メインループ
17 x = 0; y = 0    # ボールの初期位置
18 dx = 3; dy = 2  # 移動量
19 st = True       # 実行フラグ
20 while st:
21     sf.fill( (255,255,255) )    # 背景の色
22     sf.blit(im1, (x,y) )        # ボールの描画
23     # イベントキューを処理するループ
24     for ev in pygame.event.get():
25         if ev.type == pygame.QUIT:      # 「終了」イベント
26             st = False
27     # ディスプレイの更新
28     pygame.display.update()
29     # フレームレートの設定
30     fps.tick(30)    # 30FPSに設定
31     # ボール移動（位置変更）の処理
32     x += dx; y += dy    # 移動
33     if x + im1_w > w:   # ボールが右端に衝突した場合の処理
34         x = w - im1_w - 1
35         dx *= -1
36     elif x < 0:         # ボールが左端に衝突した場合の処理
37         x = 0
38         dx *= -1
39     if y + im1_h > h:   # ボールが床に衝突した場合の処理
40         y = h - im1_h - 1
41         dy *= -1
42     elif y < 0:         # ボールが天井に衝突した場合の処理
43         y = 0
44         dy *= -1
```

```
45
46 # 終了処理
47 print('quitting...')     # 終了メッセージ
48 pygame.quit()           # pygameの終了
49 sys.exit()              # プログラムの終了
```

プログラムの 4 行目で pygame の初期化処理を行っている．また 7 行目で，ウィンドウの描画面としての Surface オブジェクト sf を作成している．8 行目では，ウィンドウタイトルを設定している．

書き方：　pygame.display.set_caption(ウィンドウタイトル)

プログラムの 10 行目では，画面表示のフレームレート調整のための Clock オブジェクト fps を作成している．

書き方：　pygame.time.Clock()

これは後のイベント処理のループ内で使用する．

プログラムの 12 行目でボールの画像 ball01.jpg を load メソッドで読み込んでいる．

書き方：　pygame.image.load(画像ファイルのパス)

この関数は読み込んだ画像を Surface オブジェクトとして返す．

13〜14 行目では画像の幅と高さをそれぞれ get_width メソッド，get_height メソッドで取得している．

書き方：　画像オブジェクト.get_width()
画像オブジェクト.get_height()

画面描画と更新，イベント処理のループ（20〜44 行目）を開始する前に，17〜18 行目でボールの初期位置と移動量を設定している．このループの実行条件として，変数 st に True を設定（19 行目）しており，これが False になるとループが終了する．

画面描画と更新のループ内では毎回，背景の再設定とボールの描画（21〜22 行目）を行っている．背景の設定には fill メソッドを使用している．

書き方：　Surface オブジェクト.fill((赤, 緑, 青))

描画対象の Surface オブジェクトを指定した色の画素で満たす．各色は 0〜255 の整数値で与える．

Surface オブジェクトに別の Surface オブジェクトを上書きするには blit メソッドを使用する．

書き方：　対象.blit(画像, (x,y))

「対象」の Surface オブジェクト上の (x,y) の位置に「画像」の Surface オブジェクトを上書きする．「画像」の基準位置はその画像の左上である．

システムに発生した**イベント**（マウスやキーボードの操作，ウィンドウの位置やサイズの変更，スプライトの衝突など）の処理は，24〜26 行目の for ループで行っている．

システムに発生するイベントは随時**イベントキュー**に蓄積され，それらを 24〜26 行目の for ループで全て処理している．イベントキューの内容は

pygame.event.get()

で取得することができる．このプログラムでは，イベントキューの要素として，その type 属性が pygame.QUIT であるものを探し，そのようなイベントを検出すると st に False を設定して処理（while ループ）を終了させる．

Windows 環境では，当該アプリケーションの「閉じるボタン」（×ボタン）のクリック，あるいは Alt + F4 のキー操作（Apple 社の macOS では Command + Q ）で，type 属性が pygame.QUIT であるイベントが発生する．

28 行目では pygame.display.update() でウィンドウの更新[12] を行い，30 行目では tick メソッドを用いてアプリケーションウィンドウのフレームレートを設定している．

書き方：　Clock オブジェクト.tick(フレームレート)

上のプログラムでは，while ループの 1 秒あたりの実行回数を 30 回程度としており，画面の描画と更新を毎秒 30 回（30FPS）にしている．

[12] pygame.display.flip() でも同様に画面更新ができる．pygame.display.update() は特定の Surface オブジェクトの更新も可能である．

2.1.2 描画機能

Surface オブジェクトに各種の図形や文字などを表示する方法を説明する．

2.1.2.1 基本的な図形

■ 四角形

塗り潰し： pygame.draw.rect(Sf, Color, Rect)
外周　　： pygame.draw.rect(Sf, Color, Rect, Width)

Surface オブジェクト Sf に対して四角形を描画する．引数に与えるものは次の通り．

Color - (R,G,B) のタプル．値の範囲は 0〜255 の整数値
Rect - 描画位置 (X,Y) とサイズ W × H の値から成るタプル (X,Y,W,H)
Width - 外周を描く場合の線の太さ

■ 楕円

塗り潰し： pygame.draw.ellipse(Sf, Color, Rect)
外周　　： pygame.draw.ellipse(Sf, Color, Rect, Width)

Surface オブジェクト Sf に対して楕円を描画する．楕円の位置とサイズは，その楕円に外接する四角形を元に考える．引数に与えるものは先の pygame.draw.rect に準じる．

■ 円

塗り潰し： pygame.draw.circle(Sf, Color, (X,Y), R)
外周　　： pygame.draw.circle(Sf, Color, (X,Y), R, Width)

Surface オブジェクト Sf に対して円を描画する．引数に与えるものは次の通り．

Color - (R,G,B) のタプル．値の範囲は 0〜255 の整数値
(X,Y) - 円の中心の座標
R - 円の半径
Width - 外周を描く場合の線の太さ

■ 線分

書き方： pygame.draw.line(Sf, Color, (X1,Y1),(X2,Y2), Width)

Surface オブジェクト Sf に対して線分を描画する．引数に与えるものは次の通り．

Color - (R,G,B) のタプル．値の範囲は 0〜255 の整数値
(X1,Y1),(X2,Y2) - 始点と終点の座標
Width - 線の太さ

■ 折れ線

書き方： pygame.draw.lines(Sf, Color, Closing, Plist, Width)

Surface オブジェクト Sf に対して折れ線を描画する．引数に与えるものは次の通り．

Color - (R,G,B) のタプル．値の範囲は 0〜255 の整数値
Closing - 始点と終点を結ぶか（True）否か（False）
Plist - 始点から終点までの座標のリスト．要素は各座標のタプル
Width - 線の太さ

■ 多角形

塗り潰し： pygame.draw.polygon(Sf, Color, Plist)
外周　　： pygame.draw.polygon(Sf, Color, Plist, Width)

Surface オブジェクト Sf に対して多角形を描画する．引数に与えるものは次の通り．

Color - (R,G,B) のタプル．値の範囲は 0〜255 の整数値
Plist - 頂点の座標のリスト．要素は各座標のタプル
Width - 外周を描く場合の線の太さ

2.1.2.2　画像

ファイルから読込み　：　pygame.image.load(Fname)

画像ファイルをパス Fname から読み込んで Surface オブジェクトとして返す.

ファイルに保存　　：　pygame.image.save(S, Fname)

Surface オブジェクト S を画像ファイル Fname（パス名）に保存する.

Surface オブジェクトを別の Surface オブジェクトに貼り付けるには blit メソッドを使用する.

例. Surface オブジェクト s1 を別の Surface オブジェクト s2 に貼り付ける

```
s2.blit(s1,(20,10))
```

この例では，Surface オブジェクト s1 を，s2 上の (20,10) の位置に貼り付けている.
アプリケーションウィンドウも Surface オブジェクトであり，画像を表示するには同様の方法を用いる.

2.1.2.3　文字列

pygame では文字列は Surface オブジェクトとして表示する．この際，フォントとサイズ，スタイルなどの情報を保持する Font オブジェクトを生成し，これを用いて文字列データを Surface オブジェクトに変換する．Font オブジェクトを生成するには SysFont メソッドを使用する.

Font オブジェクトの生成：　pygame.font.SysFont(**フォント名, サイズ**)

注）フォント名に None を指定すると自動的にデフォルトのフォントが採用されるが，その場合は日本語が使用できないことが多い.

Font オブジェクトを用いて文字列を Surface オブジェクトに変換するには render メソッドを用いる.

文字列から Surface を生成：　Font **オブジェクト**.render(**文字列, アンチエイリアス指定, 色**)

「アンチエイリアス指定」は True か False で,「色」は RGB 値のタプルで与える.

例. 文字列から Surface オブジェクトを生成

```
fnt = pygame.font.SysFont('ipa ゴシック',32)
txt = fnt.render('日本語のメッセージ', True, (255,255,255) )
```

この結果，文字列をビットマップとして保持する Surface オブジェクト txt が生成される.

■ フォント名の取得

pygame で利用できるフォント名は get_fonts メソッドを使用することで調べることができる．このメソッドを実行すると利用可能なフォント名のリストが返される．次に示すプログラム pygame03.py を実行すると，利用可能なフォント名の一覧が表示される.

プログラム：pygame03.py

```
1 import pygame
2
3 pygame.init()   # pygameの初期化
4
5 # フォントリストの取得と表示
6 fl = pygame.font.get_fonts()
7 for f in fl:
8     print(f)
```

このプログラムを実行すると，次の例のようにフォント名が表示される.

```
arial
arialblack
calibri
    （途中省略）
ipap 明朝
ipaex ゴシック
ipaex 明朝
```

■ 描画のサンプルプログラム

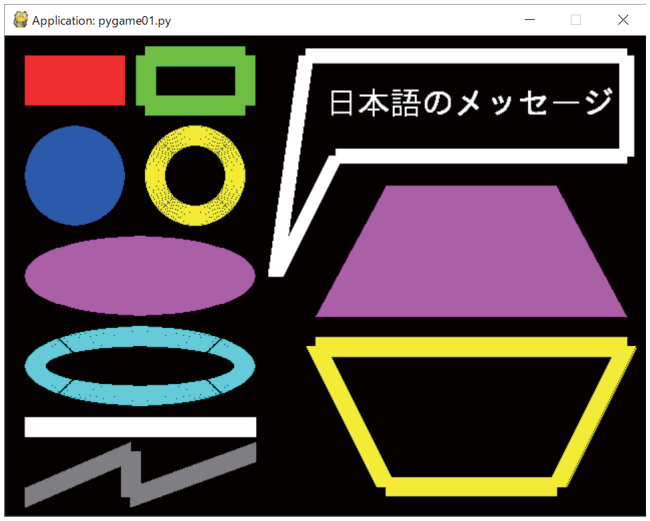
先に解説した描画機能を使用したサンプルプログラム pygame01.py を示す.

プログラム：pygame01.py

```
import sys
import pygame

pygame.init()       # pygameの初期化

sf = pygame.display.set_mode( (640,480) )   # アプリケーションウィンドウ
pygame.display.set_caption('Application: pygame01.py')

fps = pygame.time.Clock()   # フレームレート制御のための Clock オブジェクト

# メインループ
st = True
while st:
    # 四角形（塗りつぶし）の描画
    pygame.draw.rect(sf, (255,0,0), (20,20,100,50))
    # 四角形（枠）の描画
    pygame.draw.rect(sf, (0,255,0), (140,20,100,50), 20 )

    # 円（塗りつぶし）の描画
    pygame.draw.circle(sf, (0,0,255), (70,140), 50 )
    # 円（線による円周）の描画
    pygame.draw.circle(sf, (255,255,0), (190,140), 50, 20 )

    # 楕円（塗りつぶし）の描画
    pygame.draw.ellipse(sf, (255,0,255), (20,200,230,80) )
    # 楕円（線による周）の描画
    pygame.draw.ellipse(sf, (0,255,255), (20,290,230,80), 20 )

    # 線分の描画
    pygame.draw.line(sf, (255,255,255), (20,390),(250,390), 20 )

    # 折れ線の描画（開放端）
    plst = [(20,460), (125,415), (125,460), (250,415)]
    pygame.draw.lines(sf, (127,127,127), False, plst, 20 )
    # 折れ線の描画（始点と終点を結ぶ）
    plst = [(270,240), (300,20), (620,20), (620,120), (330,120)]
    pygame.draw.lines(sf, (255,255,255), True, plst, 15 )

    # 文字の描画（システムフォント）
    fnt = pygame.font.SysFont('ipaゴシック',32)     # システムフォント
    txt = fnt.render('日本語のメッセージ', True, (255,255,255) )
    sf.blit(txt,(320,50))

    # ポリゴンの描画（塗りつぶし）
    plst = [(310,280), (380,150), (550,150), (620,280)]
    pygame.draw.polygon(sf, (255,0,255), plst )

    # ポリゴンの描画（線による周）
    buf = pygame.Surface( (330,160) )   # バッファ
    plst = [(10,10), (80,150), (250,150), (320,10)] # バッファ上の座標
    pygame.draw.polygon(buf, (255,255,0), plst, 20 )    # バッファに描画
    sf.blit( buf, (300,300) )   # バッファをウィンドウへ

    # イベントキューを処理するループ
    for ev in pygame.event.get():
        if ev.type == pygame.QUIT:      # 「終了」イベント
            st = False

    # ディスプレイの更新
    pygame.display.update()
    # フレームレートの設定
    fps.tick(4) # 4FPSに設定

# 終了処理
print('quitting...')
pygame.quit()
```

```
67 sys.exit()
```

このプログラムを実行すると図 25 のようなウィンドウが表示される．



各種の描画機能を実行したサンプル

図 25: pygame01.py を実行したところ

2.1.2.4 回転，拡縮など

画像に対して回転，拡大，縮小をするには，元の画像（Surface オブジェクト）に対してそれらの処理を施したものを生成して，それを別の Surface オブジェクトに貼り付けるという手順を踏む．

■ 回転，拡縮のためのメソッド

回転　　： pygame.transform.rotate(Sur,Angle)
サイズ変更： pygame.transform.scale(Sur,NewSize)

これらメソッドは Surface オブジェクト Sur を回転，拡縮し，その結果として得られる Surface オブジェクトを返す．元の Sur は変更されない．回転処理において Angle には回転角を反時計回りで指定する．単位は 360 進法の「度」である．サイズ変更において，NewSize には幅と高さのタプル (W,H) で与える．要素は整数で与えること．

■ 回転，拡縮のサンプルプログラム

画像の回転，拡縮を応用したアニメーションを表示するサンプルプログラムを示す．図 26 に示す画像を回転するアニメーションを表示するプログラムを pygame02.py に，拡縮するアニメーションを表示するプログラムを pygame04.py に示す．

pygame_logo.png

図 26: この画像を回転，拡縮する

プログラム：pygame02.py

```
1  import sys
2  import pygame
3
4  pygame.init()    # pygameの初期化
5
6  sf = pygame.display.set_mode( (320,240) )    # アプリケーションウィンドウ
7  pygame.display.set_caption('Application: pygame02.py')
8
9  fps = pygame.time.Clock()    # フレームレート制御のための Clock オブジェクト
10
11 # 画像の読込み
12 im1 = pygame.image.load('pygame_logo.png')
```

```
agl = 0.0   # 初期角度

# メインループ
st = True
while st:
    sf.fill( (0,0,0) )  # 毎フレームクリアする
    # 画像の回転と表示
    im2 = pygame.transform.rotate(im1,agl%360)  # 回転処理
    sf.blit(im2, (50,10) )  # ウィンドウに転送
    agl += 2.4  # 次回の角度

    # イベントキューを処理するループ
    for ev in pygame.event.get():
        if ev.type == pygame.QUIT:      # 「終了」イベント
            st = False

    # ディスプレイの更新
    pygame.display.update()
    # フレームレートの設定
    fps.tick(30)    # 30FPSに設定

# 終了処理
print('quitting...')
pygame.quit()
sys.exit()
```

プログラム：pygame04.py

```
import sys
import pygame

pygame.init()   # pygameの初期化

sf = pygame.display.set_mode( (320,120) )   # アプリケーションウィンドウ
pygame.display.set_caption('Application: pygame04.py')

fps = pygame.time.Clock()   # フレームレート制御のための Clock オブジェクト

# 画像の読込み
im1 = pygame.image.load('pygame_logo.png')
im1_w = im1.get_width()
im1_h = im1.get_height()
rat = 0.02  # 初期比率
dr = 0.02

# メインループ
st = True
while st:
    sf.fill( (0,0,0) )  # 毎フレームクリアする
    # 画像の拡縮と表示
    im2 = pygame.transform.scale(im1,(int(rat*im1_w),int(rat*im1_h)) )  # 拡縮処理
    sf.blit(im2, (10,10) )  # ウィンドウに転送
    rat += dr   # 次回の比率
    if rat > 1.5:
        rat = 1.5
        dr *= -1
    elif rat < 0.0:
        rat = 0.02
        dr *= -1

    # イベントキューを処理するループ
    for ev in pygame.event.get():
        if ev.type == pygame.QUIT:      # 「終了」イベント
            st = False

    # ディスプレイの更新
    pygame.display.update()
    # フレームレートの設定
    fps.tick(30)    # 30FPSに設定

# 終了処理
```

```
print('quitting...')
pygame.quit()
sys.exit()
```

■ **Surface オブジェクトのサイズ**

Surface オブジェクトに対して get_width，get_height メソッドを引数なしで使用することで幅と高さが得られる．

2.1.3 キーボードとマウスのハンドリング

キーボードやマウスのイベントとして代表的なもの（イベント種別）を表 12 に挙げる．これらイベントは pygame の定数として定義されており，それらを読み込んで使用することができる．

表 12: マウスとキーボードの代表的なイベント

イベント定数	イベントの種類	利用できるイベント属性（プロパティ）の一部
MOUSEBUTTONDOWN	マウスのボタンが押された	pos,button：位置とボタン
MOUSEBUTTONUP	マウスのボタンが放された	pos,button：位置とボタン
MOUSEMOTION	マウスが動いた	pos,buttons：位置とボタンタプル
KEYDOWN	キーボードが押された	key,mod,unicode： キー番号，モディファイア番号，対応する文字
KEYUP	キーボードが放された	key,mod：キー番号，モディファイア番号

キーボードとマウスのイベントをハンドリングするサンプルプログラム pygame05.py を示す．

プログラム：pygame05.py

```
import sys
import pygame

pygame.init()   # pygameの初期化

sf = pygame.display.set_mode( (320,240) )   # アプリケーションウィンドウ
pygame.display.set_caption('Application: pygame05.py')

fps = pygame.time.Clock()   # フレームレート制御のための Clock オブジェクト

# メインループ
st = True
while st:
    # イベントキューを処理するループ
    for ev in pygame.event.get():
        if ev.type == pygame.QUIT:      # 「終了」イベント
            st = False
        elif ev.type == pygame.MOUSEBUTTONDOWN:
            print('Mouse button was pressed:\t',ev.pos,ev.button)
        elif ev.type == pygame.MOUSEBUTTONUP:
            print('Mouse button was released:\t',ev.pos,ev.button)
        elif ev.type == pygame.MOUSEMOTION:
            print('Mouse is moving:\t\t',ev.pos,ev.buttons)
        elif ev.type == pygame.KEYDOWN:
            print('A key was pressed:\t\t',ev.key,ev.mod,ev.unicode)
        elif ev.type == pygame.KEYUP:
            print('The key was released:\t\t',ev.key,ev.mod)
    # ディスプレイの更新
    pygame.display.update()
    # フレームレートの設定
    fps.tick(12)    # 12FPSに設定

# 終了処理
print('quitting...')
pygame.quit()
sys.exit()
```

このプログラムを実行すると小さなウィンドウが表示され，キーボード，マウスからのイベントを受けて，それらイベントの各種プロパティの値が表示されることが確認できる．

2.1.4 音声の再生

pygame は WAV 形式音声データに加えて，MP3 [13] や Ogg Vorbis [14] といったフォーマットの音声データを再生することができる．pygame の音声再生機能は pygame.mixer.music パッケージにあり，例えば音声データ 'dat1.mp3' を読込んで再生するには次のように記述する．

```
pygame.mixer.music.load('dat1.mp3')
pygame.mixer.music.play()
```

pygame.mixer.music パッケージの主要な機能を表 13 に挙げる．

表 13: サウンド再生のための主な機能

機能（関数）	説明
load(Fname)	Fname のパスから音声データを読み込む．
play( )	読み込まれた音声データを再生する．'loops=n' 引数で n 回再生を繰り返す．(n=-1 の場合は無限ループ) 'start=t' 引数で再生開始位置（t）を指定できる．(ただし，start 引数に未対応のフォーマットあり) 'fade_ms=m' 引数で開始時に m ミリ秒のフェードイン効果を与える．
stop( )	再生中の音声を停止する．
pause( )	再生中の音声を一時停止する．
unpause( )	一時停止した音声の再生を再開する．
rewind( )	再生中の音声を巻き戻す．(先頭に戻す)
fadeout(t)	再生中の音声をフェードアウトして停止する．フェードアウトタイムは t で指定する．(単位：ミリ秒)
get_pos( )	再生中の位置（時刻）を返す．(単位：ミリ秒)
set_pos(t)	再生中の位置（時刻）t を指定する．(単位：ミリ秒)
get_volume( )	音量の設定値（0〜1.0）を取得する．
set_volume(v)	音量を設定（$0.0 \leq v \leq 1.0$）する．
get_busy( )	音声が再生中であれば True を，そうでなければ False を返す．

注）再生位置の制御や精度に関しては，扱う音声フォーマットによって扱いが異なる．

音声データを再生するプログラムの例 pygame06.py をに示す．

プログラム：pygame06.py

```
1  import sys
2  import pygame
3
4  pygame.init()   # pygameの初期化
5
6  sf = pygame.display.set_mode( (320,240) )   # アプリケーションウィンドウ
7  pygame.display.set_caption('Application: pygame06.py')
8
9  fps = pygame.time.Clock()   # フレームレート制御のための Clock オブジェクト
10
11 # サウンドデータの読込みと再生
12 pygame.mixer.music.load('sound02.ogg')
13 pygame.mixer.music.play()
14
15 # メインループ
16 st = True
17 while st:
18     # イベントキューを処理するループ
19     for ev in pygame.event.get():
20         if ev.type == pygame.QUIT:      # 「終了」イベント
21             st = False
22
23     # 再生が終了したときの処理
24     if pygame.mixer.music.get_busy():
25         pass    # 再生中である
26     else:       # 終了時
27         print('music has finished.')
28         st = False
```

[13] **MP3**（MPEG-1 Audio Layer-3）：最も広く普及している音声圧縮フォーマットである．2017 年までは利用ライセンスによる利用制限があったが，現在では自由に使える．

[14] **Ogg Vorbis**：Vorbis コーデックで圧縮伸張し，Ogg コンテナにサウンドを格納する音声フォーマットである．オープンソースであり自由に利用できる．他の圧縮フォーマットと比べても音質は劣らない．

```
29
30      print('Playing:',pygame.mixer.music.get_pos(),'msec' )
31
32      # ディスプレイの更新
33      pygame.display.update()
34      # フレームレートの設定
35      fps.tick(2) # 2FPSに設定
36
37  # 終了処理
38  print('quitting...')
39  pygame.quit()
40  sys.exit()
```

このプログラムを実行すると，小さなウィンドウを表示した後，音声ファイル'sound02.ogg' を読み込んで再生を開始する．再生中は再生の経過時間が標準出力にミリ秒単位で表示される．

注意）pygame.mixer.music の機能によって同時に再生できるサウンドは 1 つであり，これは，長時間再生される背景音楽（BGM）に適している．これに対して次に解説する Sound オブジェクトは複数同時再生が可能であり，短い効果音を同時に複数再生することに適している．

2.1.4.1 Sound オブジェクトを用いる方法

Sound オブジェクトを用いて音声を再生する方法を解説する．このクラスのコンストラクタの書き方は次の通り．

書き方： pygame.mixer.Sound(音声ファイルのパス)

「音声ファイルのパス」からサウンドデータを読込み，それを保持する Sound オブジェクトが返される．このオブジェクトには表 14 に示すようなメソッドが使用できる．

表 14: Sound オブジェクトに対する主なメソッド

メソッド	解説	メソッド	解説
play()	再生開始	stop()	再生終了
get_volume()	音量の取得	set_volume()	音量の設定
get_length()	再生時間の取得		

Sound オブジェクトには，再生位置（時刻）の取得や設定のためのメソッドは無い．Sound オブジェクトを用いて先に示したサンプルプログラム pygame06.py と同等の機能を実現するプログラム pygame07.py を示す．

プログラム：pygame07.py

```
1   import sys
2   import pygame
3
4   pygame.init()   # pygameの初期化
5
6   sf = pygame.display.set_mode( (320,240) )   # アプリケーションウィンドウ
7   pygame.display.set_caption('Application: pygame07.py')
8
9   fps = pygame.time.Clock()   # フレームレート制御のための Clock オブジェクト
10
11  # サウンドデータの読込みと再生
12  snd = pygame.mixer.Sound('sound02.ogg')
13  print('長さ:',snd.get_length())
14  print('音量:',snd.get_volume())
15  snd.play()
16
17  # メインループ
18  st = True
19  while st:
20      # イベントキューを処理するループ
21      for ev in pygame.event.get():
22          if ev.type == pygame.QUIT:  # 「終了」イベント
23              st = False
24
25      # 再生が終了したときの処理
26      if pygame.mixer.get_busy():
```

```
27          pass
28      else:
29          print('music has finished.')
30          st = False
31
32      # ディスプレイの更新
33      pygame.display.update()
34      # フレームレートの設定
35      fps.tick(2) # 2FPSに設定
36
37  # 終了処理
38  print('quitting...')
39  pygame.quit()
40  sys.exit()
```

このプログラムの 26 行目にあるように，Sound オブジェクトの再生においては，その終了を pygame.mixer.get_busy() で検出する．

2.1.4.2 Sound オブジェクトから NumPy の配列への変換

音声データを NumPy ライブラリを用いて解析するには，Sound オブジェクトが持つ音声のデータ列を NumPy [15] の配列データに変換する必要がある．

NumPy 配列への変換： pygame.sndarray.array(Sound オブジェクト)

この処理によって Sound オブジェクトの音声データ列が NumPy の配列として得られる．得られる配列は，元のサウンドファイルの波形データではなく，後に説明する pygame.mixer.pre_init の設定に従って複製したものである．

ファイルから読み込んだ音声データを NumPy の配列にして，波形グラフを表示するプログラム pygame08.py を示す．

プログラム： pygame08.py

```
1   import pygame
2   import numpy as np  # 数値演算ライブラリ
3   import matplotlib.pyplot as plt     # 可視化ライブラリ
4
5   # pygameの初期化
6   sr = 44100      # サンプリング周波数
7   pygame.mixer.pre_init(frequency=sr, size=-16, channels=2, buffer=4096)
8   pygame.init()
9
10  # サウンドデータの読込み
11  snd = pygame.mixer.Sound('aaa.wav')
12  print('長さ:',snd.get_length())
13  print('音量:',snd.get_volume())
14
15  # NumPy配列への変換
16  ar = pygame.sndarray.array( snd )
17  print('配列の形状:',ar.shape)
18  print('配列の型:',ar.dtype)
19
20  # 左右の分離
21  arL = ar[:,0]   # 左チャンネル
22  arR = ar[:,1]   # 右チャンネル
23  t = np.arange(len(arL)) / sr    # 時間軸
24
25  # プロット
26  (fig,ax) = plt.subplots( 2, 1, figsize=(10,5) )
27  plt.subplots_adjust(hspace=0.5)
28  ax[0].plot(t,arL)
29  ax[0].set_title('Left')
30  ax[0].set_xlabel('time(sec)')
31  ax[0].set_ylabel('level')
32  ax[1].plot(t,arR)
33  ax[1].set_title('Right')
34  ax[1].set_xlabel('time(sec)')
35  ax[1].set_ylabel('level')
```

[15]「3.1 数値計算と可視化のためのライブラリ：NumPy / matplotlib」（p.51）で解説する．

```
36 plt.show()
```

このプログラムの 7 行目で pygame で再生する音声に関する設定が pygame.mixer.pre_init によって行われる．これは pygame.init() に先立って実行する．pygame.mixer.pre_init のキーワード引数は次のようなものである．

frequency=サンプリング周波数	サンプリング周波数を設定する（単位：Hz）
size=量子化ビット数	量子化ビット数（ビット長）を設定する． 負の値を与えると，符号付き整数の形でサンプリングが行われる．
channels=チャンネル数	ステレオの場合は 2 を，モノラルの場合は 1 を設定する．
buffer=バッファサイズ	通常は 4096 を設定しておく．この値を小さくすると遅延は 減るが，音切れリスクが高まる．

このプログラムを実行するとターミナルウィンドウに次のように表示される．

```
pygame 2.6.1 (SDL 2.28.4, Python 3.13.5)
Hello from the pygame community.  https://www.pygame.org/contribute.html
長さ:  0.401995450258255
音量:  1.0
配列の形状:  (17728, 2)
配列の型:  int16
```

同時に図 27 のような波形がプロットされる．

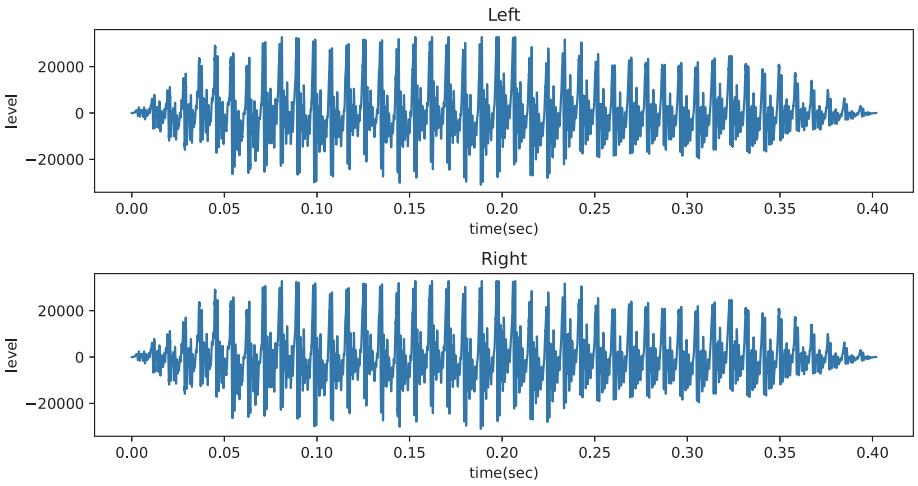


図 27: pygame08.py の実行で表示される波形のグラフ

2.1.5 スプライトの利用

先に挙げたプログラム pygame00.py では，1 つのボールがウィンドウ内で移動する様子を示した．プログラムの基本的な流れとしては概ね，

1. ウィンドウ内の消去
2. ゲームフィールド 1 場面の状態（画像の位置など）の生成
3. ゲームフィールド 1 場面の表示とウィンドウの更新

というものであり，これらを全て「メインループ」内で記述している．

実際のゲームでは，ゲームフィールド内で複数のキャラクタ（画像部品など）がそれぞれ独自の動きをすることが多く，全ての画像部品の動きを表現するための変数などを個別に用意して，メインループ内で全ての変数の変化を記述するのは非常に煩雑な作業となる．

ここで紹介する**スプライト**を使用すると，ゲームフィールド内にある画像部品を別々のオブジェクトとして管理でき，ゲーム開発が大幅に簡素化できる．具体的には，ゲームフィールドに登場するそれぞれのキャラクタを個別のオブジェクト（Sprite クラス）として生成して扱い，個々のキャラクタの動きを，そのオブジェクトに対するメソッドとして実装するというスタイルを取る．

2.1.5.1 Sprite クラス

Sprite クラスは，ゲームキャラクタの画像を image 属性として，その位置とサイズを rect 属性として保持する．ゲームキャラクタはゲーム展開の個々のフレームで位置を始めとする状態を更新することで動きを実現する．Sprite オブジェクトの状態を変化させる（更新する）ためのメソッドとして update，それを Surface に描画するためのメソッドとして draw があり，Sprite クラスの拡張クラスの定義でそれらをオーバーライドして，各種スプライトに独自の動きを実現する．Sprite クラスは pygame.sprite.Sprite としてアクセスできる．

■ サンプルプログラム 1

次に示すサンプルプログラム pygame_sprt1.py は，先に挙げたプログラム pygame00.py と類似の動作（ボールがバウンドするアニメーション）をするものである．これに沿ってスプライトの最も基本的な取り扱いについて説明する．

プログラム：pygame_sprt1.py

```
1  import sys
2  import pygame
3
4  pygame.init()   # pygameの初期化
5  ########################################################
6  # スプライト用クラスの定義                              #
7  # pygame.sprite.Sprite を継承して拡張する               #
8  # 基本プロパティ：                                      #
9  #   image   -   スプライト用画像                       #
10 #   rect    -   スプライトの位置とサイズ（矩形）       #
11 #   vx, vy  -   移動時の差分（移動量）                 #
12 ########################################################
13 class MySprite( pygame.sprite.Sprite ):
14     def __init__(self,imgFname,x,y,vx,vy):
15         pygame.sprite.Sprite.__init__(self)
16         self.image = pygame.image.load(imgFname).convert_alpha()
17         width = self.image.get_width()
18         height = self.image.get_height()
19         self.rect = pygame.Rect(x,y,width,height)
20         self.vx = vx
21         self.vy = vy
22     def update(self):
23         global w, h     # ウィンドウサイズ（大域変数）
24         if self.rect.left < 0:
25             self.vx = -self.vx
26             self.rect.left = 0
27         if self.rect.right > w:
28             self.vx = -self.vx
29             self.rect.right = w
30         if self.rect.top < 0:
31             self.vy = -self.vy
32             self.rect.top = 0
33         if self.rect.bottom > h:
34             self.vy = -self.vy
35             self.rect.bottom = h
36         self.rect.move_ip(self.vx,self.vy)
37     def draw(self,screen):
38         screen.blit(self.image,self.rect)
39
40 ########################################################
41 # ゲームフィールドの準備                                #
42 ########################################################
43 w = 400;    h = 300     # ウィンドウサイズ
44
45 # ウィンドウ（Surface）の生成
46 sf = pygame.display.set_mode( (w,h) )
47 pygame.display.set_caption('Sprite Test (1)')
48
49 # Clockオブジェクトの生成（フレームレート制御関連）
50 fps = pygame.time.Clock()
51
52 ########################################################
53 # スプライトの生成                                      #
54 ########################################################
55 spr1 = MySprite('ball.png', 0,0, 5,3 )
```

```
56
57 ########################################################
58 # メインループ                                         #
59 ########################################################
60 st = True
61 while st:
62     # フレームレート設定（毎回）
63     fps.tick(30)
64     # ウィンドウ消去（毎回）
65     sf.fill( (0,0,0) )       # 毎回，真っ暗にする
66     # スプライト描画
67     spr1.update()        # スプライトの状態の更新
68     spr1.draw(sf)        # スプライトの表示
69     # イベント検出部
70     for ev in pygame.event.get():
71         if ev.type == pygame.QUIT:       # 終了処理
72             st = False
73     # ウィンドウの更新
74     pygame.display.update()
75
76 # 終了処理
77 print('終了します．')
78 pygame.quit()
79 sys.exit()
```

このプログラムでは，Sprite クラスの拡張クラスとして MySprite クラスを定義して，このクラスのインスタンスとしてゲームキャラクタを取り扱う．MySprite クラスはゲームキャラクタ用の画像（image 属性）と，その位置とサイズの情報（rect 属性）を保持するだけでなく，フレーム更新時の位置の変化量も vx, vy プロパティとして保持するように実装されている．MySprite クラスのインスタンスを生成する際，コンストラクタの引数に，画像のファイル名，それに初期の位置と変化量を与える．

メインループ内では，ウィンドウの消去とスプライトの状態の更新，表示を繰り返し行なっている．ボールの絵として表示されるスプライト spr1 は，フレームの更新の度に，

- 移動量プロパティ vx, vy に基づく位置の変更
- ウィンドウの端に衝突したかどうかの判定と，それに基づく移動方向の変更

を update メソッドで行っている．実際の表示は draw メソッドで行っている．

rect プロパティは配下に top, bottom, left, right プロパティを持っており，それぞれ上下左右の座標値に対応している．これらプロパティの値を参照して，ウィンドウの端に衝突したことを判定している．

このプログラムを実行した様子を図 28 に示す．

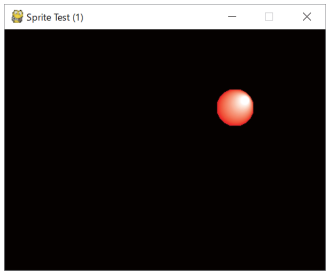


図 28: 1 個のボールがバウンドするアニメーション

■ サンプルプログラム 2

複数のスプライトを保持する Group クラスを使用すると，複数のゲームキャラクタを同時に扱うことが容易になる．Group クラスは pygame.sprite.Group としてアクセスできる．

先のプログラム pygame_sprt1.py を変更して，複数のスプライトを同時に扱う形にしたプログラム pygame_sprt2.py を次に示す．

プログラム：pygame_sprt2.py

```
import sys
import pygame

pygame.init()   # pygameの初期化
#########################################################
# スプライト用クラスの定義                                #
# pygame.sprite.Sprite を継承して拡張する                 #
# 基本プロパティ：                                        #
#   image   -   スプライト用画像                          #
#   rect    -   スプライトの位置とサイズ（矩形）          #
#   vx, vy  -   移動時の差分（移動量）                    #
#########################################################
class MySprite( pygame.sprite.Sprite ):
    def __init__(self,imgFname,x,y,vx,vy):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(imgFname).convert_alpha()
        width = self.image.get_width()
        height = self.image.get_height()
        self.rect = pygame.Rect(x,y,width,height)
        self.vx = vx
        self.vy = vy
    def update(self):
        global w, h     # ウィンドウサイズ（大域変数）
        if self.rect.left < 0:
            self.vx = -self.vx
            self.rect.left = 0
        if self.rect.right > w:
            self.vx = -self.vx
            self.rect.right = w
        if self.rect.top < 0:
            self.vy = -self.vy
            self.rect.top = 0
        if self.rect.bottom > h:
            self.vy = -self.vy
            self.rect.bottom = h
        self.rect.move_ip(self.vx,self.vy)

#########################################################
# ゲームフィールドの準備                                  #
#########################################################
w = 400;    h = 300     # ウィンドウサイズ

# ウィンドウ（Surface）の生成
sf = pygame.display.set_mode( (w,h) )
pygame.display.set_caption('Sprite Test (2)')

# Clockオブジェクトの生成（フレームレート制御関連）
fps = pygame.time.Clock()

#########################################################
# スプライトの生成                                        #
#########################################################
sg = pygame.sprite.Group()      # スプライトグループの生成
for i in range(5):
    spr = MySprite('ball.png', i*80,i*60, 5,3 )
    sg.add(spr)

#########################################################
# メインループ                                            #
#########################################################
st = True
while st:
    # フレームレート設定（毎回）
    fps.tick(30)
    # ウィンドウ消去（毎回）
    sf.fill( (0,0,0) )      # 毎回，真っ暗にする
    # スプライト描画
    sg.update()
    sg.draw(sf)
    # イベント検出部
```

```
71      for ev in pygame.event.get():
72          if ev.type == pygame.QUIT:       # 終了処理
73              st = False
74      # ウィンドウの更新
75      pygame.display.update()
76
77  # 終了処理
78  print('終了します.')
79  pygame.quit()
80  sys.exit()
```

このプログラムの前半部は pygame_sprt1.py と同じであるが，複数のスプライトをまとめる Group である sg を生成し，そこに 5 個の同じスプライトを初期位置を変えて登録するものである．メインループ内でも，スプライトグループである sg に対して update し，draw している．

このプログラムを実行した様子を図 29 に示す．

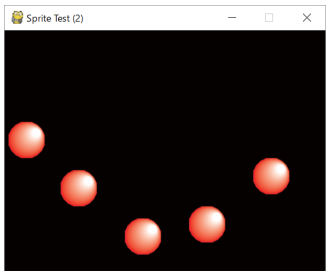


図 29: 5 個のボールがバウンドするアニメーション

2.2 メディアデータの変換： ffmpeg-python

ffmpeg-python は，広く普及しているマルチメディア処理ソフトウェアである FFmpeg を Python から操作するためのライブラリ[16] である．画像，音声，動画の形式変換やトリミング，結合，フィルタ処理などを Python の関数呼び出しとして記述できるのが特徴である．

ffmpeg-python はメディアデータの処理を FFmpeg に依頼するので，利用に先立って FFmpeg [17] をインストールして，ffmpeg の関連ツールのディレクトリをコマンドサーチパスに登録しておく必要がある．Windows 環境下の PSF 版 Python に，ffmpeg-python を pip で導入する作業の例を次に示す．

例. `py -m pip install ffmpeg-python`

ffmpeg-python を Python 処理系に読み込むには次のようにする．

```
import ffmpeg
```

本書では，メディアデータの変換処理をはじめとする，ffmpeg-python の最も基本的な使用方法について解説する．

2.2.1 基本的な使用方法

ffmpeg-python が提供する機能は，本来 FFmpeg コマンドをターミナル環境で行う処理を Python 処理系で行うものである．FFmpeg コマンドでは，入力，フィルタ処理，出力などの手順を複数のコマンドオプション（スイッチ）として記述するが，ffmpeg-python では，それらの処理を順につなげた**操作のパイプライン**として表現する．これにより，FFmpeg の複雑なコマンド構文を Python の関数呼び出しで直感的に記述することができる．

2.2.1.1 ファイルの読込み処理

input 関数で，ファイル読込み処理のためのストリームオブジェクトを作成することができる．

書き方： input(入力ファイルのパス, format=形式名, ss=開始時刻, t=長さ)

「入力ファイルのパス」からファイルを読み込むためのストリームオブジェクトを返す．「形式名」には入力ファイルのメディアデータのフォーマット（表 15〜17）を指定する．format 引数を省略すると，入力ファイル名からフォーマットを推測する．開始時刻，長さの単位は秒である．引数 ss，t は省略可能で，その場合はメディアデータの先頭から全体を読み込む処理を意味する．

表 15: 音声データの形式名（一部）

形式名	拡張子	解　　説	形式名	拡張子	解　　説
wav	.wav	WAV 形式（PCM）	mp3	.mp3	広く普及した圧縮形式
aac	.aac	高品質な圧縮形式	ogg	.ogg	オープンな圧縮形式
flac	.flac	高音質アーカイブ（可逆圧縮）	opus	.opus	低ビットレート向き

表 16: 動画データの形式名（一部）

形式名	拡張子	解　　説	形式名	拡張子	解　　説
mp4	.mp4	広く普及した形式	mkv	.mkv	Matroska コンテナ
avi	.avi	古典的なコンテナ	mov	.mov	QuickTime 形式
flv	.flv	Flash Video	webm	.webm	オープンな Web 動画形式
mpeg	.mpg	MPEG コンテナ			

表 17: 静止画像データの形式名（一部）

形式名	拡張子	解　　説	形式名	拡張子	解　　説
png	.png	PNG 画像	jpeg	.jpg	JPEG 画像
bmp	.bmp	Windows 標準ビットマップ	gif	.gif	GIF 画像

[16] https://github.com/kkroening/ffmpeg-python

[17] FFmpeg：メディアデータの変換のためのソフトウェア（https://ffmpeg.org/）

例えば，OS のコマンドとして ffmpeg を使用して aaa.wav という WAV 形式音声ファイルを aaa.mp3 という MP3 形式音声ファイルに変換する場合は次のような操作を行う．

```
ffmpeg -i aaa.wav -b:a 128k aaa.mp3
```

この「-i aaa.wav」の部分を表すストリームオブジェクト生成の例を次に示す．

例. ファイル aaa.wav を読み込む処理を記述する

```
>>> import ffmpeg  Enter      ←モジュールの読込み
>>> s1 = ffmpeg.input('aaa.wav')  Enter      ←ファイル読込み処理の記述
```

この処理によって，ファイル aaa.wav を読み込む処理の手続きが**ストリームオブジェクト** s1 に得られる．ffmpeg-python におけるストリームオブジェクトとは，いわゆるデータの入出力を意味する「ストリーム」ではなく，FFmpeg コマンドの操作の流れを扱うものであり，謂わば「コマンド操作のパイプライン」と解釈されるものである．実際に，上の例の処理の結果としては FFmpeg は何の処理も行っておらず，ストリームオブジェクトとしての s1 が得られただけである．

例. ストリームオブジェクトの型を調べる（先の例の続き）

```
>>> type(s1)  Enter      ←型を調べる
<class 'ffmpeg.nodes.FilterableStream'>      ←このような型
```

s1 は FilterableStream クラスのオブジェクトであることがわかる，これは「ファイルを読み込む」という作業を表現するものである．

2.2.1.2 ファイルの出力処理

output メソッド／関数で，ファイル出力処理のためのストリームオブジェクトを作成することができる．

書き方：　output(出力ファイルのパス, format=形式名,
ar=音声サンプリング周波数, audio_bitrate=音声ビットレート, ac=音声チャンネル数,
acodec=音声コーデック,
r=動画フレームレート, video_bitrate=動画ビットレート, s=動画の画素構成,
vcodec=動画コーデック)

format 引数に与える「形式名」には概ね input の場合と共通のものが指定できる．「音声チャンネル数」，「音声サンプリング周波数」，「動画フレームレート」は整数で指定する．「音声ビットレート」，「動画ビットレート」は文字列の形式で '128k'，'2M' などの記述を与える．「動画の画素構成」は文字列の形式で '640x480' などと，'**横幅 x 高さ**' の形で記述する．「音声コーデック」，「動画コーデック」には表 18〜19 に挙げるようなものが指定できる．

表 18: 使用できる音声コーデック（一部）

コーデック	解　　説
aac	広く普及した非可逆圧縮音声 (1)
mp3	広く普及した非可逆圧縮音声 (2)
ac3 / eac3	Dolby Digital 系
libopus	Opus 音声コーデック（ストリーミング向き）
libvorbis	Vorbis 音声（オープンな非可逆圧縮）
flac	可逆圧縮音声（劣化なし保存用）
pcm_s16le	非圧縮 PCM（16bit リトルエンディアン）
alac	Apple の可逆圧縮音声

読み込んだ aaa.wav を output メソッドによって MP3 形式に変換して aaa.mp3 というファイルにするためのストリームオブジェクトを作成する処理を次に示す．

例. MP3 に変換して aaa.mp3 として出力する処理を記述する（先の例の続き）

```
>>> s2 = s1.output('aaa.mp3', audio_bitrate='128k')  Enter      ←ファイル出力処理の記述
>>> type(s2)  Enter      ←型を調べる
<class 'ffmpeg.nodes.OutputStream'>      ←このような型
```

s2 は OutputStream クラスのオブジェクトであることがわかる，これは「s1 の処理に続いてファイル出力を行う」と

表 19: 使用できる動画コーデック（一部）

コーデック	解　　説
libx264	H.264/AVC エンコード（最も広く使われる）
libx265	H.265/HEVC エンコード（高圧縮だが時間がかかる）
libvpx-vp8	VP8 エンコード（WebM 向け）
libvpx-vp9	VP9 エンコード（WebM 高圧縮用途）
libaom-av1	AV1 エンコード（新しい高圧縮形式）
libxvid	古典的な MPEG-4 Part2 エンコード（Xvid）
mpeg4	MPEG-4 Part 2 標準エンコーダ
prores / prores_ks	Apple ProRes 出力
libtheora	Theora 映像（Ogg 向け）

いう作業を表現するものである．

この処理でも具体的な処理は何も実行されないが，s2 に対して run メソッド（後に解説する）を実行することで，s2 で表現される処理が実際に行われる．

参考）動画の 2 パスエンコーディングの指定

H.264/AVC をはじめとする高画質の動画エンコーダでは，圧縮処理を 2 段階に分けて実行（2 パスエンコーディング）して動画の品質を高めることができる．これを有効にするには，output メソッドに引数 'pass_=2' を与える．

2.2.1.3　ストリームの実行

ストリームオブジェクトに対して run メソッドを実行する，あるいは run 関数を実行することで，そのストリームオブジェクトが実際に実行される．

書き方 (1)：　ストリームオブジェクト.run(quiet=True, overwrite_output=True)
書き方 (2)：　run(ストリームオブジェクト,quiet=True, overwrite_output=True)

引数 quiet=True は FFmpeg システムからのターミナル出力を抑制する．これを省略すると，FFmpeg の処理中のメッセージがターミナルウィンドウに表示される．また，引数 overwrite_output=True を省略すると，出力ファイルが既存の場合に上書き保存されない．run は実行結果として，標準出力（stdout），標準エラー出力（stderr）の内容をそれぞれバイト列の形式にしたものを要素として持つタプルを返す．

例. ストリームオブジェクトを実際に実行する（先の例の続き）

```
>>> s3 = s2.run(quiet=True,overwrite_output=True)  Enter      ← s2 までの処理を実行する
```

この結果，ファイルの変換処理が実行され，FFmpeg からの端末出力がタプル s3 に得られる．

例. run の実行結果の確認（先の例の続き）

```
>>> type(s3)  Enter      ←戻り値の型を調べる
<class 'tuple'>       ←タプルである
>>> len(s3)  Enter      ←要素の個数を調べる
2                ← 2 個
>>> s3[0].decode('utf-8')  Enter      ←先頭要素を文字列に変換して確認
''                ←なし
>>> s3[1].decode('utf-8')  Enter      ←次の要素を文字列に変換して確認
"ffmpeg version 2025-10-01-git-1a02412170-full_build-www.gyan.dev Copyright (c) 2000-2025
the FFmpeg developers\r\n built with gcc 15.2.0 (Rev8, Built by MSYS2 project)\r\n
configuration: --enable-gpl --enable-version3 --enable-static
                :
            (途中省略)
                :
[out#0/mp3 @ 00000271cb297f00] video:0KiB audio:7KiB subtitle:0KiB other streams:0KiB
global headers:0KiB muxing overhead: 6.488388%\r\n size=       7KiB time=00:00:00.40
bitrate= 150.6kbits/s speed=51.3x elapsed=0:00:00.00    \r\n"
```

2.2.1.4　メソッドチェーンによる簡便な実行方法

先に解説した処理を**メソッドチェーン**の形式で，より簡単に実行することができる．先に示した処理例をより簡単に実行する方法を示す．

例. メソッドチェーンを応用した実行方法

```
>>> import ffmpeg   Enter
>>> so,se = (ffmpeg   Enter
...     .input('aaa.wav')   Enter
...     .output('aaa.mp3', audio_bitrate='128k')   Enter
...     .run(quiet=True,overwrite_output=True))   Enter
```

これは，ストリームオブジェクトに対するメソッドが更に次のストリームオブジェクトを返すことを応用したもので，1 つの代入文で入力から出力までの実行を完了している．代入文の右辺は複数の行に渡って記述していることから，右辺全体を括弧 '( )' で括っている．

2.2.2　音声データを NumPy の配列に変換する方法

input 関数で読み込んだ音声データを，Python プログラムで扱うためのオブジェクトに変換するには，**パイプ**を介した方法を応用する．output メソッドは出力先にパイプを指定することができ，ファイルに出力するものをそのままプログラムで受け取ることができる．出力先をパイプにするには，ファイルのパスの代りに 'pipe:' を指定する．

音声ファイルを読み込んで NumPy 配列に変換し，それをプロットするプログラム ffmpegPipeWav01.py を示す．

プログラム：ffmpegPipeWav01.py

```
1  import ffmpeg
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  sr = 44100          # サンプリング周波数
6  # WAVサウンドファイルの読込み
7  so,se = (ffmpeg
8      .input('aaa.wav')
9      .output('pipe:', format='f32le', acodec='pcm_f32le', ac=2, ar=sr)
10     .run(quiet=True))
11
12 # パイプからの内容をNumPy配列に変換
13 a = np.frombuffer(so, np.float32).reshape([-1, 2])
14 # 左右チャンネルの分離
15 aLeft, aRight = a[:,0], a[:,1]
16
17 # 波形のプロット
18 t = np.arange(len(aLeft)) / sr  # 時間軸
19 fig,ax = plt.subplots( 2,1, figsize=(10,4) )
20 plt.subplots_adjust(hspace=0.5)
21 ax[0].plot(t,aLeft)
22 ax[0].set_ylabel('level')
23 ax[0].set_title('Left')
24 ax[1].plot(t,aRight)
25 ax[1].set_xlabel('time(sec)')
26 ax[1].set_ylabel('level')
27 ax[1].set_title('Right')
28 plt.show()
```

解説：

プログラムの 7〜10 行目で音声ファイル aaa.wav を読み込み，内容をパイプにしている．13 行目で，パイプの内容を NumPy の配列に変換し，15 行目でステレオサウンドの左右を分離している．得られた数値配列を 18 行目移行でグラフにプロット（matplotlib を使用）している．

NumPy と matplotlib に関しては「3.1 数値計算と可視化のためのライブラリ：NumPy / matplotlib」（p.51）で解説しているので参照のこと．

ffmpegPipeWav01.py を実行すると図 30 のようなグラフが表示される．

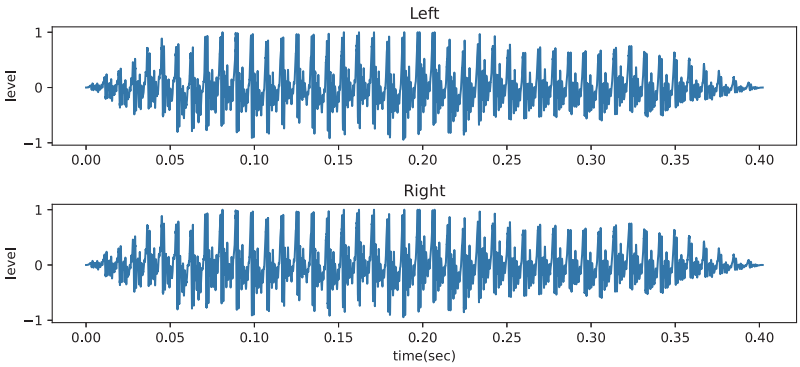


図 30: ffmpegPipeWav01.py の実行結果

3　科学技術系

3.1　数値計算と可視化のためのライブラリ：NumPy / matplotlib

ここでは，数値計算のためのパッケージである NumPy と データを可視化するためのパッケージである matplotlib に関して導入的に解説する．

NumPy は LAPACK（http://www.netlib.org/lapack/）や BLAS （http://www.openblas.net/）といった数値演算ライブラリを使用して構築されており，大規模な配列データを処理するための機能を提供する．NumPy はオープンソースのソフトウェアであり，ソフトウェア本体やドキュメントなどが公式のインターネットサイト https://numpy.org/ で公開されている．

NumPy を使用するには次のようにしてモジュールを読み込む必要がある．

```
import numpy
```

あるいは，次のようにしてパッケージ名の別名を指定して読み込むことも慣例となっている．

```
import numpy as np
```

こうすることで，NumPy の各種関数やクラス，プロパティを ‘numpy.〜’ として記述する代わりに ‘np.〜’ として記述することができる．（以後の説明ではこの慣例に従う）

■ NumPy のバージョン確認

NumPy の変数 `__version__` から，導入されている NumPy のバージョン番号が参照できる．

例. バージョン確認

```
>>> import numpy as np   Enter      ← NumPy の読込み
>>> np.__version__   Enter      ←バージョン確認
'2.2.6'          ←使用している NumPy のバージョン番号
```

3.1.1　NumPy で扱うデータ型

Python は動的な型付けの言語処理系であり，リストをはじめとするデータ構造の要素の型に制限はない．しかし NumPy の処理は C 言語や FORTRAN などで実装された演算機能を含んでおり，扱うデータ配列の要素の型は主として表 20 に挙げるようなもの[18] である．

表 20: NumPy の代表的なデータ型

タイプ	意味	タイプ	意味
`int8`	符号付き 8 ビット整数	`uint8`	符号無し 8 ビット整数
`int16`	符号付き 16 ビット整数	`uint16`	符号無し 16 ビット整数
`int32`	符号付き 32 ビット整数	`uint32`	符号無し 32 ビット整数
`int64`	符号付き 64 ビット整数	`uint64`	符号無し 64 ビット整数
`float16`	16 ビット浮動小数点数	`complex64`	64 ビット複素数
`float32`	32 ビット浮動小数点数	`complex128`	128 ビット複素数
`float64`	64 ビット浮動小数点数	`complex192`	192 ビット複素数 *
`float96`	96 ビット浮動小数点数 *	`complex256`	256 ビット複素数 *
`float128`	128 ビット浮動小数点数 *	`bool`	真理値（True/False）

※ 処理系によっては使えない型もあるので確認すること．* の型は特殊な環境でのみ使用可能である．
※ bool は内部では uint8 と同じサイズである．

NumPy では，1 つの配列オブジェクトの全要素は，原則として表 20 のどれか 1 種類に限定される．表 20 に挙げたデータ型の配列に対する処理は，Python 元来のリストなどに対する処理と比較して非常に高速である．また NumPy では ‘object’ という型[19] も扱うことができ，異なる型の要素が混在する配列も作ることができるが，その場合は表 20 に挙げたデータ型の配列に比べると処理が遅く[20] なる．

[18] 他にも unicode（文字列型）や object （Python オブジェクト）といった型がある．

[19] NumPy の新しい版では ’object’ は表示の際に ’O’ となることがある．

[20] これに関しては「?? 要素の型によって異なる計算速度」（p.??）で具体例を示す．

NumPy の複素数型（complex）の実部と虚部は共に浮動小数点数である．

表 20 に挙げたデータ型は NumPy 独自のものであり，Python 元来の int，float，complex，bool とは異なる．NumPy のスカラー値は

np. 型名 (初期値)

と記述して生成することができる．「初期値」には Python 元来の数値などを与える．

例. NumPy 独自の int64 の生成

```
>>> import numpy as np  Enter    ← NumPy の読込み
>>> a = np.int64( 2 )  Enter    ← int64 の整数を生成
>>> print( a )  Enter    ←値の確認
2
```

int64 型の整数 2 が変数 a に得られている．また，print 関数で出力すると，Python 元来の数値型と同様の形式で出力されることがわかる．

この値の型を確認する例を次に示す．

例. データ型の確認（先の例の続き）

```
>>> a  Enter    ← print 関数を使わずに値を確認
np.int64(2)          ← Python 元来の int ではないことがわかる
>>> type( a )  Enter    ←データ型を確認
<class 'numpy.int64'>      ←このようなクラス
```

真理値型（bool）も，NumPy が独自の型を導入しており，Python 元来の bool 型とは別のものである．

例. NumPy の真理値

```
>>> np.bool(True)  Enter    ← NumPy での True は
np.True_            ←このような表記
>>> np.bool(False)  Enter    ← NumPy での False は
np.False_           ←このような表記
```

このように `np.True_`，`np.False_` と表記される．

3.1.1.1 NumPy 独自の型のデータの表示形式

NumPy は数値や真理値などにおいても独自のデータ型を採用しており，Python 言語処理系の元来のものとは異なる．このため，対話環境（REPL）での値の表示形式が独特なものになることがある．例えば，先に示したように整数型においても，`np.int64(2)` のように，真理値においても `np.True_` のように独特の表示形式となる．また，NumPy のバージョンによっても，対話環境下での値の表示形式が異なることがあり，NumPy の古いバージョンでは，数値や真理値が Python 元来の数値型や真理値型と同様に表示されていた．

読者の対話環境（REPL）における NumPy のデータの表示形式と本書の表示が異なることがあるが，適宜読み替えていただきたい．データの型を明確に調べる場合は type 関数を使用するか，dtype 属性[21] を参照すること．

3.1.2 配列オブジェクトの基本的な扱い方

NumPy では配列データ（データ列，行列，多次元配列）を独自の ndarray オブジェクト（本書では以降，ndarray オブジェクトを **array オブジェクト**と呼ぶことがある）として扱う．ndarray オブジェクトを生成する array 関数の引数に，配列データをリストなどの形式で与えることができる．

書き方： array(配列の初期値のデータ)

例. リストから NumPy の配列を生成する例

```
>>> import numpy as np  Enter    ←パッケージを'np' として読み込んでいる
>>> lst = [1,2,3,4,5]  Enter    ←通常のリストの形でデータ列を生成
>>> ar = np.array(lst)  Enter    ← NumPy の配列に変換
>>> ar  Enter    ←内容の確認
array([1, 2, 3, 4, 5])    ← NumPy の配列になっている
```

[21]「3.1.2.1 配列の要素の型について」（p.53）で解説する．

ndarray オブジェクトが ar に得られていることがわかる．上の例のように print 関数を使わずに直接オブジェクトを参照すると array(…) と表示される．ただし，type 関数でその型を調べると ndarray であることがわかる．(次の例)

例. array オブジェクトの型の確認（先の例の続き）

```
>>> type( ar )  Enter      ←型を調べると
<class 'numpy.ndarray'>     ← ndarray 型
```

array 関数による方法以外にも，様々な方法で配列を生成することができる．

■ NumPy の配列の実記憶上での扱い

NumPy の配列は，基本的には C 言語の配列と互換性があり，C/C++のプログラムとそのまま受け渡しできるケース[22] が多い．従ってその性質上，一度作成した ndarray オブジェクトの実メモリ上でのサイズと配置は基本的には変更できない（要素の値は自由に変更できる）．これは，Python に組み込みのデータ型（リストやセット，辞書など）と明確に異なる性質である．

ndarray オブジェクトに対する要素の挿入や削除に関する様々な機能[23] が提供されているが，それらは基本的には元の配列を変更せず，処理結果を新たなオブジェクトとして実メモリ上に作成して返す．

3.1.2.1 配列の要素の型について

array オブジェクトのプロパティ dtype には，当該 array オブジェクトの要素の型が保持されている．

例. array の要素の型（先の例の続き）

```
>>> ar.dtype  Enter      ←型の調査
dtype('int64')      ←要素の型は 'int64' であることがわかる
```

もちろん，個々の要素も型の情報を持っており，dtype 属性から型名が参照できる．

例. 要素の値と型の確認（先の例の続き）

```
>>> ar[0]  Enter      ←先頭要素の値の確認
np.int64(1)
>>> ar[0].dtype  Enter      ←先頭要素の型の確認
dtype('int64')          ←型
```

array オブジェクトを生成する際，コンストラクタにキーワード引数　dtype=**型**　を与えることで要素の型を指定することができる．このときの型は表 20 のもの（パッケージ名. を先頭に付けたもの）を指定する．

例. 型を指定した配列の生成

```
>>> lst = [1,2,3,4,5]  Enter      ←通常のリストの形でデータ列を生成
>>> ar = np.array(lst,dtype=np.float64)  Enter      ← NumPy の配列に変換（float64）
>>> ar  Enter      ←内容の確認
array([ 1., 2., 3., 4., 5.])      ←浮動小数点数（float64）の要素を持つ NumPy の配列になっている
>>> ar.dtype  Enter      ←型の調査
dtype('float64')      ←要素の型は 'float64' であることがわかる
```

'dtype=' には型名を文字列で与えても良い．すなわち，

```
>>> ar = np.array(lst,dtype='float64')  Enter      ← NumPy の配列に変換（float64）
```

としても，同様の結果となる．

NumPy のオブジェクトの型については，後の「3.1.2.3 型の表記に関する事柄」(p.54) で更に詳しく解説する．

※ NumPy は複素数を扱うことができる．これに関しては「3.1.22 複素数の計算」(p.150) で解説する．

3.1.2.2 真理値の配列

配列の生成の際にデータ型として 'bool' を指定すると要素は真理値となる．

[22] 標準ライブラリの ctype，サードパーティが提供する pybind11 などによるプログラミングインターフェースを介したデータ交換処理.

[23] 後に解説する append，insert，delete など.

例. 真理値の配列

```
>>> lst = [True,False,False,True,False]  Enter    ←真理値のリストを生成
>>> ar = np.array(lst,dtype='bool')  Enter    ← NumPy の配列に変換
>>> ar  Enter    ←内容の確認
array([ True, False, False, True, False])    ←結果表示
```

この例における lst に数値のリストを与えることもできる．その場合は 0 を False，0 以外を True と解釈する．すなわち，

```
>>> lst = [2,0,0,-5,0]  Enter    ←数値のリスト
```

としても，同様の結果となる．

真理値の配列は，配列同士を比較する際にも現れる．これに関しては後の「3.1.26 行列の検査」(p.167) でも説明する．

3.1.2.3 型の表記に関する事柄

先に解説したように，NumPy で扱うデータの型は表 20 に挙げられており，更に，実際のデータの dtype 属性を参照することで個別に調べることができる．ただし，それらは**型の別名**（alias）であり，便宜的に使用するための表記である．それに対して**正式な内部表現**（canonical form）としての型名もあり（単に内部表現と呼ぶこともある），それは dtype 属性の更に str 属性から得られる．(次の例)

例. 型の別名と内部表現

```
>>> ar = np.array([1,2,3])  Enter    ←整数の配列
>>> ar.dtype  Enter    ← dtype 属性からは
dtype('int64')    ←型の別名が得られる
>>> ar.dtype.str  Enter    ←正式な内部表現を調べる
'<i8'    ←'int64' の正式な内部表現
```

NumPy を利用する場合，別名による型の指定で十分であることが多いが，正式な内部表現によって，より詳細に型を取り扱うことができる．

例. 内部表現による型の指定

```
>>> np.array([1,2,3],dtype='<i4')  Enter    ←このような型指定で
array([1, 2, 3], dtype=int32)    ← 4 バイト符号付き整数の配列が得られる
```

型名の内部表現は，バイトオーダー，値の種類，データの大きさの順に並んだ文字列で，その内訳を表 21 に示す．

表 21: NumPy の型の内部表現（一部）

順序	意味するもの	解説
1)	バイトオーダー	'<'：リトルエンディアン，'>'：ビッグエンディアン， '='：実行環境ネイティブのバイトオーダー，'|'：バイトオーダーなし
2)	値の種類	'i'：符号付き整数，'u'：符号なし整数，'f'：浮動小数点数， 'c'：複素数，'b'：真理値 (bool)，'U'：Unicode 文字列， 'S'：バイト列（raw 文字列など），'V'：任意のバイト列 (void)， 'M'：日付時刻（datetime64），'m'：経過時間（timedelta64）
3)	データの大きさ	Unicode 文字列の場合は文字数，それ以外の場合はバイト長
4)	時間の単位	'[時間の単位]'：このフィールドは日付時刻と経過時間の場合のみ現れる． 例. '[Y]'：年，'[M]'：月，'[W]'：週，'[D]'：日，'[h]'：時， '[m]'：分，'[s]'：秒，'[ms]'：ミリ秒，'[us]'：マイクロ秒， '[ns]'：ナノ秒 ※「3.1.30 日付，時刻の扱い」(p.177) で解説する．

別名を持たない型は内部表現のみで表される．

特殊な型のデータを保持する配列を作成する例を次に示す．

例. 特殊な型のデータ（1）

```
>>> np.array([b'x',b'y',b'z'])  Enter      ←純粋な ASCII 文字の配列
array([b'x', b'y', b'z'], dtype='|S1')
>>> np.array(['あ','い','う'])  Enter      ← Unicode 文字の配列
array(['あ', 'い', 'う'], dtype='<U1')
```

例. 特殊な型のデータ（2）

```
>>> obj = np.array([3,'abc',{10,11,12},True],dtype='object')  Enter  ← Python のオブジェクト
>>> obj  Enter      ←確認
array([3, 'abc', {10, 11, 12}, True], dtype=object)
>>> obj.dtype  Enter      ←型を調べると
dtype('O')                  ← 'O' となっている
```

例. 特殊な型のデータ（3）

```
>>> dt = np.datetime64('2025-08-29')  Enter      ←日付のデータ[24]
>>> dt  Enter      ←確認
np.datetime64('2025-08-29')
>>> dt.dtype  Enter      ←型を調べると
dtype('<M8[D]')
```

重要）NumPy の重要な目的の 1 つに数値の高速演算がある．ここで挙げたような特殊なデータはその目的には適していない．数値の高速演算を実行するには，配列の要素は整数，浮動小数点数，複素数，真理値に（1 つの配列内の要素は全て同じ型に）統一するべきである．

3.1.2.4 基本的な計算処理

配列には算術演算（+，-，*，/ ）やべき乗（**）が実行できる．

例. 配列同士の算術演算（和と差）

```
>>> a1 = np.array([1,3])  Enter      ←配列 a1 を用意
>>> a2 = np.array([2,4])  Enter      ←配列 a2 を用意
>>> a1 + a2  Enter      ←和
array([3, 7])          ←計算結果
>>> a1 - a2  Enter      ←差
array([-1, -1])          ←計算結果
```

例. 配列同士の算術演算（積，除算，べき乗：先の例の続き）

```
>>> a1 * a2  Enter      ←積
array([ 2, 12])          ←計算結果
>>> a1 / a2  Enter      ←除算
array([0.5 , 0.75])      ←計算結果
>>> a1 ** a2  Enter      ←べき乗
array([ 1, 81])  ←計算結果
```

この例からわかるように，配列同士の算術演算では，対応する要素の間の演算が行われ，それぞれの結果を要素とする配列が得られる．

次に，配列と数値（スカラー）との間の演算について説明する．

例. 配列とスカラーとの間の演算（和と差：先の例の続き）

```
>>> a1 + 4  Enter      ←配列 a1 に 4 を加える
array([5, 7])          ←計算結果
>>> a1 - 4  Enter      ←配列 a1 から 4 を引く
array([-3, -1])          ←計算結果
```

[24]「3.1.30 日付，時刻の扱い」（p.177）で解説する．

例. 配列とスカラーとの間の演算（積，除算，べき乗：先の例の続き）

```
>>> a1 * 4  Enter      ←配列 a1 を 4 倍する
array([ 4, 12])          ←計算結果
>>> a1 / 4  Enter      ←配列 a1 を 4 で割る
array([0.25, 0.75])      ←計算結果
>>> a2 ** 2  Enter     ←配列 a2 を 2 乗する
array([ 4, 16]   ←計算結果
```

このように，配列の各要素に対してスカラー値の演算を施す[25] 結果となる．

線形代数の演算に関しては後の「3.1.23 行列，ベクトルの計算（線形代数のための計算）」（p.151）で解説する．

NumPy は，Python 標準の math モジュールが提供する関数や定数と同じ名前のものを多く提供[26] しており，

np. 関数名 (引数並び)
np. 定数名

の形式で呼び出すことができる．

例. pi，sin の値

```
>>> np.pi  Enter      ← π（円周率）
3.141592653589793        ←値
>>> np.sin( np.pi / 2 )  Enter      ← sin(π/2)
np.float64(1.0)          ←値
```

NumPy が提供する数学関数は基本的に配列を扱うことができる．これを応用するとデータ集合に対する写像を簡単に実現することができる．

例. データ集合に対する写像

```
>>> X = np.array([0, np.pi/4, np.pi/2, np.pi*3/4, np.pi])  Enter  ← [0, π/4, π/2, 3/4 * π, π]
>>> np.sin( X )  Enter      ←一度に sin の写像を作成
array([0.00000000e+00, 7.07106781e-01, 1.00000000e+00, 7.07106781e-01,      ←得られた写像
       1.22464680e-16])
>>> X = np.array([0, 1, 4, 9, 16])  Enter      ←データ集合
>>> np.sqrt( X )  Enter      ←一度に平方根の写像を作成
array([0., 1., 2., 3., 4.])      ←得られた写像
```

この方法は，関数の可視化（プロット）の際にも応用できる．詳しくは「3.1.12 配列に対する演算：1 次元から 1 次元」（p.85）を参照のこと．

NumPy が提供する数学関数の多くは複素数を扱うことができる．

例. 平方根の計算

```
>>> np.sqrt( -2 )  Enter      ← √−2 を求める試み
<stdin>-243:1:  RuntimeWarning:  invalid value encountered in sqrt  ←警告メッセージ「不正な値」
np.float64(nan)      ←戻り値は「非数」（後で説明する）
>>> np.sqrt( -2+0j )  Enter      ←引数を複素数にして √−2 を求めると
np.complex128(1.4142135623730951j)      ←複素数で計算結果が得られる
```

平方根を求める関数 np.sqrt は，引数に複素数が与えられると複素数の戻り値を返す．

NumPy が提供する emath サブモジュールも数学関数を提供しており，複素数の扱いをはじめ，より柔軟な処理を行う．

例. emath が提供する sqrt 関数

```
>>> np.emath.sqrt( -2 )  Enter      ← √−2 を求める
np.complex128(1.4142135623730951j)      ←複素数で得られる
>>> np.emath.sqrt( -2+0j )  Enter      ← √−2 + 0i を求める
np.complex128(1.4142135623730951j)      ←上と同じ結果
```

[25] この機能は**ブロードキャスト**の一部．

[26] 詳しくは NumPy の公式インターネットサイト https://numpy.org/ を参照のこと．

NumPy での複素数の扱いについては後の「3.1.22 複素数の計算」(p.150) で更に詳しく解説する.

3.1.2.5 角度の変換

NumPy は，弧度法による角度（1 周が 2π ラジアン）と度数法（1 周が 360 ° ）による角度を変換するための関数を提供している.

■ 弧度法から度数法への変換

書き方： rad2deg(角度データ)

弧度法による「角度データ」を度数法に変換したものを返す．スカラー，配列のどちらの形でも「角度データ」を与えることができる．この関数は別名 degrees を持ち，この名前で実行することもできる.

■ 度数法から弧度法への変換

書き方： deg2rad(角度データ)

度数法による「角度データ」を弧度法に変換したものを返す．スカラー，配列のどちらの形でも「角度データ」を与えることができる．この関数は別名 radians を持ち，この名前で実行することもできる.

例. スカラー値の角度を変換

```
>>> np.rad2deg( np.pi )  Enter     ← π ラジアンは
np.float64(180.0)                  ← 180 °
>>> np.deg2rad( 180 )  Enter       ← 180 ° は
np.float64(3.141592653589793)      ← π ラジアン
```

例. 弧度法の角度の配列を度数法に変換

```
>>> r = np.linspace( 0, np.pi, 7 )  Enter     ← 0 ～ π ラジアンを 7 個の配列にする
>>> r  Enter     ←確認
array([0.        , 0.52359878, 1.04719755, 1.57079633, 2.0943951 , 2.61799388, 3.14159265])
>>> d = np.rad2deg(r)  Enter     ←弧度法の角度の配列を度数法に変換
>>> d  Enter     ←確認
array([  0.,  30.,  60.,  90., 120., 150., 180.])     ←度数法になっている
>>> np.degrees(r)  Enter     ←別名の関数としてで実行
array([  0.,  30.,  60.,  90., 120., 150., 180.])     ←同上
```

例. 度数法の角度の配列を弧度法に変換（先の例の続き）

```
>>> np.deg2rad(d)  Enter     ←度数法の角度の配列を弧度法に変換
array([0.        , 0.52359878, 1.04719755, 1.57079633, 2.0943951 , 2.61799388, 3.14159265])
>>> np.radians(d)  Enter     ←別名の関数としてで実行
array([0.        , 0.52359878, 1.04719755, 1.57079633, 2.0943951 , 2.61799388, 3.14159265])
```

←同上

3.1.2.6 扱える値の範囲

iinfo，finfo 関数を用いると配列の要素として扱える値の範囲を調べることができる．前者は整数値に関するもの，後者は浮動小数点数に関するものである.

具体的には，これら関数の戻り値の min，max プロパティを参照する.（表 22）

表 22: 扱える値の範囲を調べる方法

調査対象の型	最小値	最大値
整数	np.iinfo(型名).min	np.iinfo(型名).max
浮動小数点数	np.finfo(型名).min	np.finfo(型名).max

例. 扱える整数の範囲

```
>>> print( 'int16:  ', np.iinfo( 'int16' ).min, '～', np.iinfo( 'int16' ).max )  Enter
int16:  -32768 ～ 32767     ←符号付き 16 ビット整数の範囲
>>> print( 'uint16:', np.iinfo( 'uint16' ).min,'～', np.iinfo( 'uint16' ).max )  Enter
uint16:  0 ～ 65535     ←符号無し 16 ビット整数の範囲
```

例. 扱える浮動小数点数の範囲

```
>>> print('float64:  ',np.finfo( 'float64' ).min, '〜',np.finfo( 'float64' ).max ) Enter
float64:  -1.7976931348623157e+308 〜 1.7976931348623157e+308  ←64 ビット浮動小数点数の範囲
>>> print('complex128:',np.finfo('complex128').min,'〜',np.finfo('complex128').max)  Enter
complex128:  -1.7976931348623157e+308 〜 1.7976931348623157e+308  ←128 ビット複素数の範囲
```

複素数に関しては，実部と虚部の値の範囲が得られる．

3.1.2.7　特殊な値：無限大と非数

NumPy では表 23 にあるような無限大や非数を表す記号が定義されている．

表 23: 特殊な数

記号	意　味	記号	意　味
inf	正の無限大	nan	非数

これらの特殊な値は IEEE 754 にも規定されており，NumPy もそれに準拠している[27] .

例. 無限大と非数

```
>>> np.inf > 10**10000  Enter     ← np.inf と非常に大きな数 10^10000 の比較
True     ← np.inf の方が大きい
>>> np.inf + np.inf  Enter     ← np.inf 同士の和
inf     ←正の無限大
>>> np.inf / np.inf  Enter     ←無限大同士の除算
nan     ←非数
```

注意）inf，nan はあくまで形式的なものであり，厳密な意味では数学的な値ではない．そのため，それらを値として数値計算に用いるべきではない．

■ 特殊な値同士の「==」,「!=」の判定

無限大や非数同士の「==」,「!=」の判定には注意すること．(次の例)

例. 特殊な値同士の「==」,「!=」の判定

```
>>> np.inf == np.inf  Enter     ←無限大同士は
True          ←等しい
>>> np.nan == np.nan  Enter     ←非数同士は
False          ←等しくない
>>> np.nan != np.nan  Enter     ←非数同士は
True          ←等しくない
```

■ 特殊な値かどうかの判定

NumPy は inf や nan といった特殊な値を判定する関数や有限の数値を判定する関数（下記）を提供している．

- **isinf(値)**　：　値が正もしくは負の無限大なら True，それ以外なら False
- **isnan(値)**　：　値が nan なら True，それ以外なら False
- **isfinite(値)**　：　値が有限の数値なら True，それ以外なら False

例. 様々な判定処理

```
>>> print( np.isinf( np.inf ) ) Enter
True
>>> print( np.isinf( 2 ) ) Enter
False
>>> print( np.isfinite( 2 ) ) Enter
True
>>> print( np.isfinite( np.nan ) ) Enter
False
```

```
>>> print( np.isnan( np.nan ) ) Enter
True
>>> print( np.isnan( 2 ) ) Enter
False
>>> print( np.isnan( np.inf ) ) Enter
False
```

[27] C 言語の double 型に準拠している．

3.1.2.8　配列の「等しさ」の判定

2 つの配列の「等しさ」を判定する array_equal 関数がある．これは，比較する 2 つの配列の形状が同じで，対応する要素に「==」が成立する場合に True を，そうでない場合に False を返す．

例. 精度が異なる配列の比較

```
>>> af64 = np.array([1,2,3,4],dtype=np.float64)  Enter      ← 64 ビット浮動小数点数
>>> af32 = np.array([1,2,3,4],dtype=np.float32)  Enter      ← 32 ビット浮動小数点数
>>> np.array_equal(af64,af32)  Enter      ←これらは
True          ←等しい
```

例. 型が異なる配列の比較（先の例の続き）

```
>>> ai32 = np.array([1,2,3,4],dtype=np.int32)  Enter      ← 32 ビット整数
>>> np.array_equal(af64,ai32)  Enter      ←これらは
True          ←等しい
```

例. 形状が異なる配列の比較（先の例の続き）

```
>>> af64x2 = np.array([[1,2],[3,4]],dtype=np.float64)  Enter      ← 2 次元
>>> np.array_equal(af64,af64x2)  Enter      ←次元が異なると
False         ←等しくない
```

例. 無限大を含む配列の比較

```
>>> aInf1 = np.array([1,2,np.inf,4],dtype=np.float64)  Enter      ←同じ位置に
>>> aInf2 = np.array([1,2,np.inf,4],dtype=np.float64)  Enter      ←無限大を含む
>>> np.array_equal(aInf1,aInf2)  Enter      ←これらは
True          ←等しい
```

例. 非数を含む配列の比較

```
>>> aNan1 = np.array([1,2,np.nan,4],dtype=np.float64)  Enter      ←同じ位置に
>>> aNan2 = np.array([1,2,np.nan,4],dtype=np.float64)  Enter      ←非数を含む
>>> np.array_equal(aNan1,aNan2)  Enter      ←これらは
False         ←等しくない
```

ただし，オプション引数「equal_nan=True」を与えると非数を等しいとみなす．

例. 非数を含む配列の比較（先の例の続き）

```
>>> np.array_equal(aNan1,aNan2,equal_nan=True)  Enter      ←このオプションでは
True          ←等しくなる
```

重要）比較する配列の，対応する要素に少しでも差があれば array_equal の判定は False となる．

例. 若干の差がある 2 つの配列の比較

```
>>> b = np.array([1,2,3,4,5])  Enter
>>> a = np.array([1,2,3,4,5.00001])  Enter      ←少しだけ差がある配列
>>> np.array_equal(b,a)  Enter      ←これらは
False         ←等しくない
```

配列間の若干の差を無視して「近い」ことを判定するには，次に解説する allclose を使う．

3.1.2.9　配列の「近さ」の判定

ある基準となる数値配列 b に対して，別の数値配列 a が「十分に近い」（ほぼ同じ）かどうかを判定するための関数 allclose がある．これは，次の 2 つの**許容誤差の指標**に基づく判定である．

指標	解　　説	デフォルト
rtol	小さくない値を比較するときの許容誤差の指標	10^{-5}
atol	小さな値を比較する場合の許容誤差の指標	10^{-8}

この指標に基づいて，次の条件を満たす場合，配列 a は基準の配列 b に十分に近いとする．

$$|a - b| \leq atol + rtol \times |b|$$

注意）この条件は a，b で非対称である．すなわち，b は基準の配列であり，これに a が近いかどうかの判定という解釈である．a，b が複素数の場合はノルムの考えを適用する．

例. a が b に近いかどうかの判定

```
>>> b = np.array([1,2,3,4,5])  Enter    ←基準の配列
>>> a = np.array([1,2,3,4,5.00001])  Enter    ←調査対象の配列
>>> np.allclose(a,b)  Enter    ← a は b に近いかを判定
True              ←判定結果（近い）
>>> a = np.array([1,2,3,4,5.0001])  Enter    ←この a は
>>> np.allclose(a,b)  Enter    ←判定すると
False             ←近くない
```

非数（np.nan）同士は近いとは判定されない.（次の例）

例. nan 同士は近いかどうかの判定

```
>>> n = np.nan  Enter    ←非数
>>> b = np.array([n,n,n,n])  Enter    ←非数ばかりの配列 (1)
>>> a = np.array([n,n,n,n])  Enter    ←非数ばかりの配列 (2)
>>> np.allclose(a,b)  Enter    ←それらは
False             ←近くない
```

非数同士を「近い」とみなすには，オプション引数「equal_nan=True」を与える．

例. 非数同士を「近い」とみなすオプション（先の例の続き）

```
>>> np.allclose(a,b,equal_nan=True)  Enter    ←このオプションでは
True              ← a は b に近い
```

rtol，atol の値も引数で指定することができる．各引数の与え方を次に示す.（デフォルト値付きの表記）

書き方：　np.allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)

3.1.2.10　データ列の生成（数列の生成）

■ 等差数列

range 関数に似た方法でデータ列を生成する関数として arange がある．

書き方：　arange(始点, 上限, 増分)

「始点」から「上限」までの範囲で「増分」の刻みの数列の配列を作成して返す．ただし，作成される配列に「上限」の値は含まない．

例. 数列の生成

```
>>> a = np.arange(0.0, 2.0, 0.1)  Enter    ← 0 以上 2 未満の範囲の数列を 0.1 刻みで生成
>>> a  Enter    ←内容の確認
array([ 0.  , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.  ,
       1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])          ←得られた数列
```

これとは別に，指定した区間を等分して n 個の要素から成る数列を得るには linspace 関数を使用する．

書き方：　linspace(始点, 終点, 個数)

「始点」から「終点」までの範囲を「個数」に等分した数列の配列を作成して返す．作成される配列は「始点」と「終点」を含む．

例.　区間 $[n_1, n_2]$（n_1 以上 n_2 以下）を等分して 10 個のデータを得る

```
>>> np.linspace( 0, 1.0, 10 )  Enter    ← 0〜1 までを等分（最後の 1 を含む）
array([ 0.        ,  0.11111111,  0.22222222,  0.33333333,  0.44444444,  ←得られた数列
        0.55555556,  0.66666667,  0.77777778,  0.88888889,  1.        ])
```

このように最後の 1 を含んで 10 個のデータを得るため，結果的に 9 等分したものとなる．linspace 関数にキーワード引数 'endpoint=False' を与えると最後の 1 を含まず，結果として 10 等分したデータを得ることができる．

例. 10 等分した形でデータを得る

```
>>> np.linspace( 0, 1.0, 10, endpoint=False )  Enter     ←0〜1 までを 10 等分
array([ 0.  , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])     ←得られた数列
```

linspace に引数 'retstep=True' を与えると，数列と公差を同時に返す.（次の例）

例. linspace で公差を取得する

```
>>> ar, stp = np.linspace( 0, 1.0, 10, retstep=True ) Enter  ←公差取得オプションを与える
>>> ar  Enter     ←生成された数列の確認
array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
>>> stp  Enter     ←得られた公差の確認
np.float64(0.1111111111111111)     ←公差
```

arange や linspace を使用して生成したデータ列は，NumPy の各種関数の定義域として与えることができ，値域のデータ列を得る目的に使用することができる．また，グラフ作成のための定義域データを作成する場合は linspace の方が好ましい[28] .

■ 等比数列（1）（対数スケールのデータ列）

対数スケールで関数の定義域のデータ列（等比数列）を生成するには logspace を使用するのが良い．これは，対数目盛り上で等間隔（指数を等差）に点を取り，その値を線形スケールに戻すと等比数列になることによる.

書き方：　logspace(e_s**,** e_e**,** n**, base=**b **)**

初項 b^{e_s} から末項 b^{e_e} までの n 項の等比数列を生成して返す.

例. 2^0〜2^{10} の数列を対数スケールで 6 個の要素として生成

```
>>> np.logspace( 0, 10, 6, base=2 )  Enter     ←対数スケール列生成
array([ 1.00000000e+00, 4.00000000e+00, 1.60000000e+01,     ←得られた数列
        6.40000000e+01, 2.56000000e+02, 1.02400000e+03])
```

キーワード引数 'base=' を省略すると 10 が基数となる.

例. 10^0〜10^{10} の数列を対数スケールで 6 個の要素として生成

```
>>> np.logspace( 0, 10, 6 )  Enter     ←対数スケール列生成
array([ 1.00000000e+00, 1.00000000e+02, 1.00000000e+04,     ←得られた数列
        1.00000000e+06, 1.00000000e+08, 1.00000000e+10])
```

参考） logspace(e_s, e_e, n, base=b) の公比 r は次の計算によって求めると良い.

$$r = b^{(e_e - e_s)/(n-1)} \qquad (\text{ただし } n > 1)$$

例. logspace で得られる数列の公比

```
>>> es = 2; ee = 13; n = 7; b = np.e  Enter     ←引数に与える値の設定
>>> a = np.logspace( es, ee, n, base=b )  Enter     ←等比数列の作成
>>> print(a)  Enter     ←確認
[7.38905610e+00 4.62163362e+01 2.89069362e+02 1.80804241e+03     ←数列が得られている
 1.13087646e+04 7.07329408e+04 4.42413392e+05]
>>> r1 = b**((ee-es)/(n-1))  Enter     ←公比の算出
>>> print(r1)  Enter     ←確認
6.254700951936328     ←得られた公比
```

【考察】　公比は，得られた数列 a の隣接する要素の比から直接的に算出することもできるが，その方法では誤差が大きくなることがある.

例. a[1]/a[0] による公比の算出（先の例の続き）

```
>>> r2 = a[1]/a[0]  Enter     ←公比の算出
>>> print(r2)  Enter     ←確認
6.254700951936326     ←得られた公比：先の方法で得られたものと異なる
```

2 つの方法で，若干異なる公比が 2 つ得られたが，これを元に数列の末項を算出し，a の末項と比較する.

[28] arange では数列生成の際に生じる誤差の関係上，求める数列の長さが意図したものにならない場合があるため.

例. 数列の末項の比較（先の例の続き）

```
>>> print( a[n-1] )  Enter      ←数列の末項の確認
442413.3920089202          ← (1)
>>> x1 = b**es * r1**(n-1); print(x1)  Enter      ←公比 r1 から末項を算出
442413.39200892            ← (2)
>>> x2 = b**es * r2**(n-1); print(x2)  Enter      ←公比 r2 から末項を算出
442413.39200891927         ← (3)
```

(2)，(3) の値は共に (1) の値と異なる．(2)-(1) の誤差と (3)-(1) の誤差を比較する．(次の例)

例. 誤差の比較（先の例の続き）

```
>>> err1 = x1 - a[n-1]; print( err1 )  Enter      ← (2)-(1)
-1.7462298274040222e-10
>>> err2 = x2 - a[n-1]; print( err2 )  Enter      ← (3)-(1)
-9.313225746154785e-10
>>> print( err2 / err1 )  Enter      ← 2 つの誤差の比は
5.333333333333333                    ← 5 倍以上
```

以上のことから，はじめに示した公比の算出方法の方が好ましいことがわかる．

■ 等比数列（2）（初項，末項，データ数から公比を決定する方法）

等比数列を作成する関数 geomspace が使用できる．

書き方：　geomspace(初項, 末項, データ数)

「初項」から始まり「末項」で終わる「データ数」の長さの等比数列を生成して返す．

例. 等比数列の作成

```
>>> st = 1; ed = 256; n = 9  Enter      ←初項 st，末項 ed，データ個数 n の設定
>>> a = np.geomspace( st, ed, n )  Enter      ←等比数列の生成
>>> a  Enter      ←確認
array([ 1., 2., 4., 8., 16., 32., 64., 128., 256.])
```

公比は　$r = \sqrt[n-1]{\dfrac{ed}{st}}$　として求めると良い．(ただし $n > 1$)

例. 公比を求める方法（先の例の続き）

```
>>> r = (ed/st)**(1/(n-1))  Enter      ←公比の算出
>>> print(r)  Enter      ←確認
2.0        ←公比
```

3.1.2.11　多次元配列の生成

多次元の配列もリストから生成することができる．

例. 2 次元配列の生成

```
>>> a2 = np.array([[1,2,3,4],[5,6,7,8]])  Enter      ← 2 次元配列（行列）の生成
>>> a2  Enter      ←内容確認
array([[1, 2, 3, 4],        ←生成結果
       [5, 6, 7, 8]])
```

2 行 4 列の配列が得られている．

例. 3 次元配列の生成（先の例の続き）

```
>>> a3 = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])  Enter      ← 3 次元配列の生成
>>> a3  Enter          ←内容確認
array([[[1, 2],         ←生成結果
        [3, 4]],
       [[5, 6],
        [7, 8]]])
```

3.1.2.12　配列の形状の調査

配列オブジェクトの shape プロパティに配列の各次元のサイズがタプルとして保持されている.

例. 配列のサイズ（先の例の続き）

```
>>> a2.shape  Enter      ← a2 のサイズを求める
(2, 4)          ← 2 行 4 列である
>>> a3.shape  Enter      ← a3 のサイズを求める
(2, 2, 2)          ← 3 次元配列（2 × 2 × 2）である
```

1 次元のデータ列からも shape は取得できる.

例. 1 次元のデータ列のサイズ（先の例の続き）

```
>>> a1 = np.array([1,2,3])  Enter      ←データ列の生成
>>> a1.shape  Enter      ← a1 のサイズを求める
(3,)          ←タプルの形で得られる
>>> a1.shape[0]  Enter      ←タプルの要素としてサイズを求める
3          ←長さが得られる（len(a1) としても同様）
```

配列オブジェクトの次元は ndim プロパティに保持されている.

例. 配列の次元（先の例の続き）

```
>>> a1.ndim  Enter      ← a1 の次元を求める
1                ← 1 次元
>>> a2.ndim  Enter      ← a2 の次元を求める
2                ← 2 次元
>>> a3.ndim  Enter      ← a3 の次元を求める
3                ← 3 次元
```

3.1.2.13　配列の要素へのアクセス

NumPy の配列の要素へのアクセスには，リストなどの場合と同様に**スライス** '[ ]' が使用できる.

例. スライスによる要素の参照

```
>>> import numpy as np  Enter      ← NumPy の読込み
>>> a = np.array([10,20,30,40])  Enter      ←配列（1 次元）の作成
>>> a[1:3]  Enter      ←スライスを指定した参照
array([20, 30])          ←参照結果
```

また，スライスオブジェクトを用いることもできる.（次の例）

例. スライスオブジェクトの使用（先の例の続き）

```
>>> s = slice(1,3)  Enter      ←スライスオブジェクトの作成
>>> a[s]  Enter      ←スライスオブジェクトで配列の要素を参照
array([20, 30])        ←参照結果
```

多次元の配列の要素にアクセスする場合には，スライス内にコンマ ',' を記述して，各次元の要素の格納位置を指定する.

例. 3 × 3 の配列の各要素へのアクセス

```
>>> import numpy as np  Enter          ← NumPy モジュールの読込み
>>> a = np.zeros( (3,3,3) )  Enter      ← 3 × 3 の配列の生成[29]
>>> for i in range(3):  Enter      ←配列の各要素への値の設定（ここから）
...     for j in range(3):  Enter
...         for k in range(3):  Enter
...             a[i,j,k] = 100*(i+1)+10*(j+1)+k+1  Enter  ←スライス内をコンマで区切っている
...  Enter      ←（ここまで）
```

[29] p.152「3.1.23.2 単位行列，ゼロ行列，他」で説明する.

例. 配列の内容確認（先の例の続き）

```
>>> a  Enter          ←配列の内容の確認
array([[[111., 112., 113.],
        [121., 122., 123.],
        [131., 132., 133.]],
       [[211., 212., 213.],
        [221., 222., 223.],
        [231., 232., 233.]],
       [[311., 312., 313.],
        [321., 322., 323.],
        [331., 332., 333.]]])
```

多次元配列における特定の要素を指定するには上の例のように `a[i,j,k]` のように記述するが，タプルの形式で `a[(i,j,k)]` のように各次元の位置を与えても良い.（次の例）

例. 多次元配列のインデックス（先の例の続き）

```
>>> a[1,0,2]  Enter      ←通常の形式のインデックス
np.float64(213.0)        ←値が参照できている
>>> s = (1,0,2)  Enter   ←タプル形式のインデックスを
>>> a[ s ]  Enter        ←与えることもできる
np.float64(213.0)        ←上と同じ値が参照できている
```

このように，タプルをインデックスとして与えることができるが，リストや配列をインデックスとして与えた場合は全く異なる動作をする[30] ので注意すること.

参考）NumPy 独自の方法として，`np.s_[ ... ]` という記述でインデックスを作成することができる.

例. NumPy 独自のスライス（先の例の続き）

```
>>> s = np.s_[1,0,2]  Enter   ← NumPy 独自のインデックス
>>> a[ s ]  Enter     ←それを使用して
np.float64(213.0)     ←値を参照している
>>> s  Enter          ←実体は
(1, 0, 2)             ←タプルである
```

■ 基本インデックスと高機能インデックス

NumPy の配列の要素にアクセスするためのスライスには，整数や**スライスオブジェクト**を使用することができる. このようなアクセス方法は**基本インデックス**（basic indexing）[31] と呼ばれる．それら以外にも，NumPy の配列ではスライス '`[ ]`' に様々なものを与えることができ，更に便利なアクセス方法を提供する．そのような高度なアクセス方法は**高機能インデックス**（advanced indexing）[32] と呼ばれる.

3.1.2.14 スライスにデータ列を与えた場合の動作

NumPy の配列のスライスにデータ列を与えると，そのデータ列をインデックスの並びと見て元の配列の対応する要素を取り出し（高機能インデックス），それを配列として返す.

例. スライスにデータ列を与える例

```
>>> a = np.array( [0,2,4,8,16,32,64,128,256,512,1024] )  Enter   ← 2^n の配列
>>> idx = [0,2,4,6,8,10]  Enter   ←スライスに与えるインデックス n の並びを作成
>>> a[idx]  Enter     ←要素を参照する
array([ 0,  4, 16, 64, 256, 1024])   ←結果
```

3.1.2.15 複数の要素への一斉代入

配列の複数の要素にスカラー値（単一の値）を代入すると，対象の複数の要素に一斉に同じ値が代入される.

---

[30] 後の「3.1.2.14 スライスにデータ列を与えた場合の動作」（p.64）で解説する.

[31] 整数，スライスオブジェクトによる方法以外にも，newaxis による方法があり，これも**基本インデックス**の範疇に含まれる. newaxis については後の「3.1.6.1 newaxis オブジェクトによる方法」（p.75）で解説する.

[32] 公式な呼称ではないが**ファンシーインデックス**と呼ばれることもある.

例. サンプル配列の作成

```
>>> a = np.array(range(10))  Enter
>>> a  Enter      ←内容確認
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

例. 一斉代入（先の例の続き）

```
>>> a[:]  = 99   Enter      ←全要素に 99 を代入
>>> a  Enter      ←内容確認
array([99, 99, 99, 99, 99, 99, 99, 99, 99, 99])      ←全て同じ値
>>> a[2:5] = 1   Enter      ←指定した部分に 1 を代入
>>> a  Enter      ←内容確認
array([99, 99, 1, 1, 1, 99, 99, 99, 99, 99])      ←指定部分のみ代入されている
```

3.1.2.16　指定した行，列へのアクセス

リストのスライスに記述するように ‘:’ を用いると，指定した 1 つの行や列にアクセスできる.（次の例参照）

例. 先頭の行の取り出し

```
>>> a = np.array( [[11,12,13],[21,22,23],[31,32,33]] )  Enter      ← 2 次元の配列を用意
>>> a  Enter      ←内容確認
array([[11, 12, 13],      ←結果表示
       [21, 22, 23],
       [31, 32, 33]])
>>> a[0,:]  Enter      ←先頭行（第 0 番目の行）の取り出し
array([11, 12, 13])      ←先頭行が 1 次元配列として得られている
```

同様に，1 つの列を取り出す例を示す.

例. 先頭の列の取り出し（先の例の続き）

```
>>> a[:,0]  Enter      ←最初の列（第 0 番目の列）の取り出し
array([11, 21, 31])      ←最初の列が 1 次元配列として得られている
```

■ 元の配列の次元を意識した行，列の取り出し

先の例では，a[0,:] や a[:,0] として行や列を取り出したが，どちらも単純な 1 次元配列として得られている．元の配列の次元の解釈を維持した形で行，列を取り出す方法を次の例に示す.

例. 元の配列の次元を意識した行の取り出し（先の例の続き）

```
>>> a[[0],:]  Enter      ←先頭行（第 0 番目の行）の取り出し
array([[11, 12, 13]])      ← 2 次元配列の中の 1 つの行として得られている
```

例. 元の配列の次元を意識した列の取り出し（先の例の続き）

```
>>> a[:,[0]]  Enter      ←最初の列（第 0 番目の列）の取り出し
array([[11],          ← 2 次元配列の中の 1 つの列として得られている
       [21],
       [31]])
```

このことは，スライスに与えるインデックスの構造[33] から理解できる．もう 1 つ例を示す.

例. 0 列目と 2 列目を取り出す（先の例の続き）

```
>>> a[:,[0,2]]  Enter      ←最初の列と最後の列（第 0，2 番目の列）の取り出し
array([[11, 13],
       [21, 23],
       [31, 33]])
```

特定の行や列に対して 1 度に値を与えることができる.

[33]「3.1.2.14 スライスにデータ列を与えた場合の動作」（p.64）で解説したことによる.

例. 指定した行に 1 度で値を設定する（先の例の続き）

```
>>> a[1,:] = [101,102,103]  Enter     ←インデックス 1 番目の行にまとめて値を与える
>>> a  Enter     ←内容確認
array([[ 11, 12, 13],     ←結果表示
       [101, 102, 103],
       [ 31, 32, 33]])
```

例. 指定した列に 1 度で値を設定する（先の例の続き）

```
>>> a[:,1] = [-1,-2,-3]  Enter     ←インデックス 1 番目の列にまとめて値を与える
>>> a  Enter     ←内容確認
array([[ 11, -1, 13],     ←結果表示
       [101, -2, 103],
       [ 31, -3, 33]])
```

この例では，行や列にリストで値を与えているが，配列の形で与えても良い．

3.1.2.17 配列形状の変形

配列を指定した次元の構造に変形するには reshape メソッドを使用する．

例. 1 次元を 2 次元に変形

```
>>> a = np.arange( 8 )  Enter     ← 1 次元配列
>>> a  Enter         ←内容確認
array([0, 1, 2, 3, 4, 5, 6, 7])     ←結果表示
>>> a2 = a.reshape( (2,4) )  Enter     ← 2 次元配列（2 行 4 列）に変形
>>> a2  Enter         ←内容確認
array([[0, 1, 2, 3],     ←結果表示
       [4, 5, 6, 7]])
```

例. 1 次元を 3 次元に変形（先の例の続き）

```
>>> a3 = a.reshape( (2,2,2) )  Enter     ← 3 次元配列（2 × 2 × 2）に変形
>>> a3  Enter         ←内容確認
array([[[0, 1],     ←結果表示
        [2, 3]],
       [[4, 5],
        [6, 7]]])
```

reshape の引数に与えるサイズ指定において負の値を与えると，行や列のサイズを NumPy が自動的に設定する．

例. 行，列のサイズの自動設定（先の例の続き）

```
>>> a.reshape( (-1,4) )  Enter     ←行数に-1 を与えて自動設定（4 列になるよう調整される）
array([[0, 1, 2, 3],     ←結果表示
       [4, 5, 6, 7]])
>>> a.reshape( (2,-1) )  Enter     ←列数に-1 を与えて自動設定（2 行になるよう調整される）
array([[0, 1, 2, 3],     ←結果表示
       [4, 5, 6, 7]])
```

多次元の配列を 1 次元に変形（平坦化）するには flatten メソッドを使用する．

例. 3 次元を 1 次元に変形（先の例の続き）

```
>>> a3.flatten()  Enter     ← 1 次元に変形
array([0, 1, 2, 3, 4, 5, 6, 7])     ←結果表示
```

flatten とは別に ravel，reshape といったメソッドでも配列を平坦化することができ，処理に要する時間は flatten より短い．それぞれのメソッド毎に，配列の平坦化に要する時間を比較するプログラム flatten01.py を示す．

プログラム：flatten01.py

```
1 import numpy as np
2 import time
3
4 sz = 10000
5 # 2次元配列の作成
6 m0 = np.arange(0,sz**2)
```

```
m  = m0.reshape((sz,-1))
print('2次元配列の形状:',m.shape)

#--- flattenによる平坦化 ---
t1 = time.time()
m1 = m.flatten()
t2 = time.time()
b = (m0==m1).all()
print('flattenの所要時間:\t',t2-t1,'\t検証結果:',b)

#--- ravelによる平坦化 ---
t1 = time.time()
m2 = m.ravel()
t2 = time.time()
b = (m0==m2).all()
print('ravelの所要時間:\t',t2-t1,'\t検証結果:',b)

#--- reshapeによる平坦化 ---
t1 = time.time()
m3 = m.reshape(-1)
t2 = time.time()
b = (m0==m3).all()
print('reshapeの所要時間:\t',t2-t1,'\t検証結果:',b)
```

このプログラムにあるように reshape(-1)，ravel() を多次元の配列に対して実行する．このプログラムを実行した様子を次に示す．(計算機環境：CPU Intel Core i7-6770HQ 2.6GHz，RAM 16GB，Windows 10)

```
2 次元配列の形状:  (10000, 10000)
flatten の所要時間:         0.08709502220153809     検証結果:  True
ravel の所要時間:           0.0     検証結果:  True
reshape の所要時間:         0.0     検証結果:  True
```

ravel，reshape による平坦化の処理にはほとんど時間がかからないことがわかる．

3.1.2.18　行，列の転置

配列の行と列を転置するには T プロパティを参照する．

例. 行，列の転置

```
>>> a = np.array( [[1,2,3],[4,5,6]])   Enter      ←配列の作成
>>> a   Enter      ←内容確認
array([[1, 2, 3],      ←結果表示
       [4, 5, 6]])
>>> a.T   Enter      ←行と列の転置
array([[1, 4],      ←結果表示
       [2, 5],
       [3, 6]])
```

この処理は元の配列を一切変更しない．

3.1.2.19　行，列の反転と回転

flip 関数を使用すると行や列の反転ができる．

書き方：　flip(配列, axis=[1 か 0])

'axis=0' とすると縦方向（上下）の反転，'axis=1' とすると横方向（左右）の反転となる．

例. 行，列の反転

```
>>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])  Enter    ←配列の作成
>>> a  Enter    ←内容確認
array([[1, 2, 3],    ←結果表示
       [4, 5, 6],
       [7, 8, 9]])
>>> np.flip(a,axis=0)  Enter    ←縦方向（上下）の反転
array([[7, 8, 9],    ←結果表示
       [4, 5, 6],
       [1, 2, 3]])
>>> np.flip(a,axis=1)  Enter    ←横方向（左右）の反転
array([[3, 2, 1],    ←結果表示
       [6, 5, 4],
       [9, 8, 7]])
```

行，列を回転するには roll 関数を使用する．

書き方：　roll(配列, shift=回転量, axis=[1 か 0])

axis=0’ とすると縦方向（上下）の回転，‘axis=1’ とすると横方向（左右）の回転となる．「回転量」に負の値を与えると，回転方向が逆になる．

例. 行，列の回転

```
>>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])  Enter    ←配列の作成
>>> a  Enter    ←内容確認
array([[1, 2, 3],    ←結果表示
       [4, 5, 6],
       [7, 8, 9]])
>>> np.roll(a,shift=1,axis=0)  Enter    ←縦方向（上下）の回転
array([[7, 8, 9],    ←結果表示
       [1, 2, 3],
       [4, 5, 6]])
>>> np.roll(a,shift=1,axis=1)  Enter    ←横方向（左右）の回転
array([[3, 1, 2],    ←結果表示
       [6, 4, 5],
       [9, 7, 8]])
```

3.1.2.20　型の変換

配列の要素の型を変換するには astype メソッドを使用する．

例. 要素の型の変換

```
>>> a = np.arange( 0, 10, dtype='float64' )  Enter    ←'float64' 型の要素の配列
>>> a  Enter    ←内容確認
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])    ←結果表示（小数点が表示されている）
>>> a.astype(np.int16)  Enter    ←要素の型を 'int16' に変換
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int16)    ←整数として表示されている
```

参考） astype の引数には，「3.1.2.3 型の表記に関する事柄」（p.54）で解説した内部表現を与えることができる．
この方法により，型と**バイトオーダー**の両方を変換すること[34] ができる．

スカラー値の型の変換は　**np. 型名 (値)**　とすることもできる．

型を変換する際は，数値の精度に注意すること．

例. 精度低下で値が損なわれる例

```
>>> a = np.arange(100000,160001,20000,dtype='float64')  Enter    ←'float64' 型の要素の配列
>>> a  Enter    ←内容確認
array([100000., 120000., 140000., 160000.])    ←正しく値が保持されている
>>> a.astype(np.int16)  Enter    ←低い精度の整数に変換
array([-31072, -11072, 8928, 28928], dtype=int16)    ←値が損なわれている
```

[34] スカラー値はバイトオーダーの変更ができないことに注意すること．

これは顕著な例であり，浮動小数点数を精度の低い整数の型に変換する場合は特に注意が必要である．

3.1.2.21　配列の複製（コピー）

配列を複製するには copy 関数（あるいは copy メソッド）を使用する．これに関して段階的に例を挙げて説明する．

例. 配列を変数記号に割り当てる

```
>>> a = np.array([[1,2],[3,4]])  Enter     ←配列を作成して変数 a に割り当てる
>>> a   Enter     ←変数 a の値の確認
array([[1, 2],     ←変数 a の内容
       [3, 4]])
>>> b = a   Enter     ←変数 a が指し示す配列を変数 b に割り当てる
>>> b   Enter     ←変数 b の値の確認
array([[1, 2],     ←変数 b の内容
       [3, 4]])
```

次に，a の配列の変更を試みる．

例. 変数 a に割り当てられた配列の内容を変更する（先の例の続き）

```
>>> a[1,0] = 5; a[1,1] = 6   Enter     ←変数 a の配列の内容を変更
>>> a   Enter     ←変数 a の内容を確認
array([[1, 2],     ←変数 a の内容
       [5, 6]])
>>> b   Enter     ←変数 b の内容を確認すると…
array([[1, 2],     ←こちらも変更されている
       [5, 6]])
```

この例からもわかる通り，ある変数に割り当てられた配列を別の変数に ‘=’ で割り当てた場合，初めの変数に割り当てた配列と，後の変数に割り当てた配列は同一のものである．

既存の配列の複製（コピー）を別の配列として作成するには copy 関数を使用する．

例. 配列の複製（先の例の続き）

```
>>> c = np.copy( a )   Enter     ←変数 a の配列の複製を変数 c に与える
>>> c   Enter     ←変数 c の内容を確認
array([[1, 2],     ←変数 a と同じ内容
       [5, 6]])
>>> a[1,0] = 3; a[1,1] = 4   Enter     ←変数 a の配列の内容を変更
>>> a   Enter     ←変数 a の値の確認
array([[1, 2],     ←変数 a の内容
       [3, 4]])
>>> c   Enter     ←変数 c の内容は…
array([[1, 2],   Enter     ←変数 a に対する編集の影響がない
       [5, 6]])
```

この例の最初の部分を

```
c = a.copy()
```

として，メソッドの形で copy を実行することもできる．

3.1.3　配列の連結と繰り返し

3.1.3.1　append，concatenate による連結

例. append による配列の連結

```
>>> a = np.array([1,2])   Enter     ←配列の生成 (1)
>>> b = np.array([3,4])   Enter     ←配列の生成 (2)
>>> np.append(a,b)   Enter     ←上記 2 つの配列の連結
array([1, 2, 3, 4])     ←連結結果
>>> np.append(a,[5,6])   Enter     ←配列にリストを連結
array([1, 2, 5, 6])     ←連結結果は配列として得られる
```

append 関数は連結結果を新たな配列として返す．また多次元配列の連結もできる．

例. 2 次元の配列同士の連結

```
>>> a = np.array([[1,2],[3,4]])  Enter     ← 2 次元配列の生成 (1)
>>> b = np.array([[5,6],[7,8]])  Enter     ← 2 次元配列の生成 (2)
>>> np.append(a,b,axis=0)  Enter     ←新たな行として連結
array([[1, 2],     ←連結結果（行の追加）
       [3, 4],
       [5, 6],
       [7, 8]])
>>> np.append(a,b,axis=1)  Enter     ←新たな列として連結
array([[1, 2, 5, 6],     ←連結結果（列の追加）
       [3, 4, 7, 8]])
```

この例のように，行として連結する場合は append 関数にキーワード引数 ‘axis=0’ を，列として連結する場合は ‘axis=1’ を与える．

複数の配列を連結する場合は concatenate 関数を用いる．

例. 複数の配列の連結 (1 次元同士)

```
>>> a1 = np.array([0,1]); a2 = np.array([2,3]); a3 = np.array([4,5])  Enter
>>> ac = np.concatenate( [a1,a2,a3] ) Enter  ←全て連結     ↑ 3 つの 1 次元配列を用意
>>> ac  Enter                ←内容確認
array([0, 1, 2, 3, 4, 5])                ←結果表示
```

例. 複数の配列の連結 (2 次元同士)

```
>>> a = np.array([[0,1],[2,3]])  Enter     ← 2 次元配列を 3 つ用意
>>> b = np.array([[4,5],[6,7]])  Enter
>>> c = np.array([[8,9],[10,11]])  Enter
>>> np.concatenate( [a,b,c], axis=0 )  Enter     ←行方向に連結
array([[ 0, 1],     ←連結結果
       [ 2, 3],
       [ 4, 5],
       [ 6, 7],
       [ 8, 9],
       [10, 11]])
>>> np.concatenate( [a,b,c], axis=1 )  Enter     ←列方向に連結
array([[ 0, 1, 4, 5, 8, 9],     ←連結結果
       [ 2, 3, 6, 7, 10, 11]])
```

参考）append はその内部で concatenate を使用している．従って append 処理を多数実行する処理においては concatenate を用いると実行時間が少し短縮できる場合がある．後に述べる hstack，vstack，stack，r_，c_ なども内部で concatenate を使用している．

3.1.3.2 hstack，vstack による連結

水平方向（列方向）に配列を連結する hstack 関数と，垂直方向（行方向）に配列を連結する vstack 関数がある．

例. hstack,vstack による配列の連結（2 次元）

```
>>> a = np.array([[0,1],[2,3]])  Enter     ←配列を用意
>>> b = np.array([[4,5],[6,7]])  Enter     ←配列を用意
>>> np.hstack( (a,b) )  Enter     ← hstack で連結
array([[0, 1, 4, 5],     ←連結結果
       [2, 3, 6, 7]])
>>> np.vstack( (a,b) )  Enter     ← vstack で連結
array([[0, 1],     ←連結結果
       [2, 3],
       [4, 5],
       [6, 7]])
```

これら関数によって 1 次元配列同士を連結すると次のようになる．

例. hstack,vstack による 1 次元配列同士の連結

```
>>> a = np.array([0,1])   Enter      ← 1 次元配列を用意
>>> b = np.array([2,3])   Enter      ← 1 次元配列を用意
>>> np.hstack( (a,b) )    Enter      ← hstack で連結
array([0, 1, 2, 3])      ←連結結果（単純な連結）
>>> np.vstack( (a,b) )    Enter      ← vstack で連結
array([[0, 1],      ←連結結果（2 次元になる）
       [2, 3]])
```

3.1.3.3 stack による連結

複数の配列を要素として連結し，1 つ次元の高い配列を作る stack 関数がある．

例. 1 次元の配列を連結して 2 次元の配列を作る

```
>>> a = np.array([0,1])   Enter      ← 1 次元配列を用意
>>> b = np.array([2,3])   Enter      ← 1 次元配列を用意
>>> np.stack( (a,b) )     Enter      ← stack で連結
array([[0, 1],      ←連結結果（2 次元になる）
       [2, 3]])
```

これは，a，b を行の成分とみなして連結し，2 次元の配列を作る例である．この処理は先の vstack の処理と変わらないが，引数 ‘axis=’ で連結方法を制御できる．デフォルト（上の例）では ‘axis=0’ である．‘axis=0’ の場合の実行例を次に示す．

例. 連結方法を変える（先の例の続き）

```
>>> np.stack( (a,b), axis=1 )   Enter      ← stack で連結
array([[0, 2],      ←連結結果（2 次元になる）
       [1, 3]])
```

これは，a，b を列の成分とみなして連結し，2 次元の配列を作る例である．同様の考えに従って，n 次元配列を連結して n+1 次元配列を作ることができ，引数 ‘axis=’ で連結方法を制御できる．

例. 2 次元の配列を連結して 3 次元の配列を作る

```
>>> a = np.array([[0,1],[2,3]])   Enter      ← 2 次元配列を用意
>>> b = np.array([[4,5],[6,7]])   Enter      ← 2 次元配列を用意
>>> np.stack( (a,b) )     Enter      ← stack で連結
array([[[0, 1],      ←連結結果（3 次元になる）
        [2, 3]],
       [[4, 5],
        [6, 7]]])
```

例. 連結方法を変える（先の例の続き）

```
>>> np.stack( (a,b), axis=1 )   Enter      ← stack で連結
array([[[0, 1],      ←連結結果（3 次元になる）
        [4, 5]],
       [[2, 3],
        [6, 7]]])
```

3.1.3.4 r_，c_による連結

先に解説したものとは別に，r_，c_ による連結機能が使用できる．

例. 1 次元の配列を 2 つ連結

```
>>> a = np.array([0,1])   Enter      ← 1 次元配列を用意
>>> b = np.array([2,3])   Enter      ← 1 次元配列を用意
>>> np.r_[ a, b ]   Enter      ← r_ で連結
array([0, 1, 2, 3])      ←単純な連結（結果は 1 次元）
>>> np.c_[ a, b ]   Enter      ← c_ で連結
array([[0, 2],           ← a，b を列要素とみなして連結（結果は 2 次元）
       [1, 3]])
```

3 つ以上の配列を連結することもできる.

例. 1 次元の配列を 3 つ連結（先の例の続き）

```
>>> c = np.array([4,5])  Enter      ←もう 1 つ 1 次元配列を用意
>>> np.r_[ a, b, c ]  Enter      ← r_ で連結
array([0, 1, 2, 3, 4, 5])      ←単純な連結（結果は 1 次元）
>>> np.c_[ a, b, c ]  Enter      ← c_ で連結
array([[0, 2, 4],            ← a，b，c を列要素とみなして連結（結果は 2 次元）
       [1, 3, 5]])
```

2 次元配列の連結処理を次に示す.

例. 2 次元の配列を 2 つ連結

```
>>> a = np.array([[0,1],[2,3]])  Enter      ← 2 次元配列を用意
>>> b = np.array([[4,5],[6,7]])  Enter      ← 2 次元配列を用意
>>> np.r_[ a, b ]  Enter      ← r_ で連結
array([[0, 1],            ← vstack と同様の処理
       [2, 3],
       [4, 5],
       [6, 7]])
>>> np.c_[ a, b ]  Enter      ← c_ で連結
array([[0, 1, 4, 5],      ← hstack と同様の処理
       [2, 3, 6, 7]])
```

3.1.3.5　tile による配列の繰り返し

tile 関数を使用すると，与えた配列を繰り返した形の配列を作成することができる.

書き方：　tile(配列, (縦の繰り返し回数, 横の繰り返し回数))

例. 配列の繰り返し

```
>>> a = np.array([[1,2],[3,4]])  Enter      ←配列を用意
>>> a  Enter      ←内容確認
array([[1, 2],      ←結果表示
       [3, 4]])
>>> np.tile( a, (2,3) )  Enter      ←配列 a を縦に 2 回，横に 3 回繰り返す
array([[1, 2, 1, 2, 1, 2],      ←結果表示
       [3, 4, 3, 4, 3, 4],
       [1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4]])
```

3.1.4　配列への要素の挿入

insert を使用することで，配列の指定した位置に要素を挿入することができる．insert の第 1 引数には対象となる配列を，第 2 引数には挿入位置（インデックス）を，第 3 引数には挿入する値を与える.

例. 1 次元配列への要素の挿入

```
>>> a = np.array([0,1,2,3,4,5])  Enter      ←配列を用意
>>> a2 = np.insert( a, 2, 9 )  Enter      ←インデックスが 2 の位置に 9 を挿入
>>> a2  Enter      ←内容確認
array([0, 1, 9, 2, 3, 4, 5])      ←結果表示
```

このように insert は挿入処理の結果の配列を返す．元の配列は変化しない．挿入位置として指定できるのは 0 から「配列の末尾のインデックス+1」までである．それより大きな値を指定するとエラーとなる.

参考）insert の第 3 引数には配列（ndarray）を与えても良い.

insert は複数の値を挿入することができる.

例. 複数の要素の挿入（先の例の続き）

```
>>> np.insert( a, 3, [7,8,9] )  Enter      ←インデックスが 3 の位置に [7,8,9] を挿入
array([0, 1, 2, 7, 8, 9, 3, 4, 5])      ←挿入結果
```

これは，インデックスが 3 の位置に複数の値を挿入する例である．これとは別に，挿入位置を複数指定して，個別に値を挿入することもできる．(次の例参照)

例. 複数の挿入位置に個別に要素を挿入（先の例の続き）

```
>>> np.insert( a, [3,4,5], [7,8,9] )  Enter     ←挿入位置を複数指定
array([0, 1, 2, 7, 3, 8, 4, 9, 5])     ←挿入結果
```

この例からわかるように，挿入位置は元の配列を基準にした形で指定する．

3.1.4.1 行，列の挿入

insert の第 4 引数に ‘axis=0’ を与えると，2 次元配列の指定した位置に行を挿入することができる．

例. 行の挿入

```
>>> a = np.array( [[11,12,13,14],[21,22,23,24],  Enter     ←配列を用意
...                [31,32,33,34],[41,42,43,44]] )
>>> a  Enter     ←内容確認
array([[11, 12, 13, 14],     ←内容表示
       [21, 22, 23, 24],
       [31, 32, 33, 34],
       [41, 42, 43, 44]])
>>> np.insert( a, 1, [91,92,93,94], axis=0 )  Enter     ←インデックス位置 1 に行を挿入
array([[11, 12, 13, 14],     ←挿入結果
       [91, 92, 93, 94],
       [21, 22, 23, 24],
       [31, 32, 33, 34],
       [41, 42, 43, 44]])
```

複数の行を挿入することもできる．

例. 複数行の挿入（先の例の続き）

```
>>> i1 = [[71,72,73,74],[81,82,83,84]]  Enter     ←挿入する複数の行を用意
>>> np.insert( a, 1, i1, axis=0 )  Enter     ←インデックス位置 1 に行を挿入
array([[11, 12, 13, 14],     ←挿入結果
       [71, 72, 73, 74],
       [81, 82, 83, 84],
       [21, 22, 23, 24],
       [31, 32, 33, 34],
       [41, 42, 43, 44]])
```

挿入位置を複数指定することもできる．

例. 挿入位置を複数指定（先の例の続き）

```
>>> np.insert( a, [1,3], i1, axis=0 )  Enter     ←インデックス 1,3 の位置に挿入
array([[11, 12, 13, 14],     ←挿入結果
       [71, 72, 73, 74],
       [21, 22, 23, 24],
       [31, 32, 33, 34],
       [81, 82, 83, 84],
       [41, 42, 43, 44]])
```

insert の第 4 引数に ‘axis=1’ を与えると，2 次元配列の指定した位置に列を挿入することができる．

例. 列の挿入（先の例の続き）

```
>>> np.insert( a, 2, [63,73,83,93], axis=1 )  Enter     ←インデックス位置 2 に列を挿入
array([[11, 12, 63, 13, 14],     ←挿入結果
       [21, 22, 73, 23, 24],
       [31, 32, 83, 33, 34],
       [41, 42, 93, 43, 44]])
```

複数の列を挿入することもできる．

例. 複数の列の挿入（先の例の続き）

```
>>> i2 = [[63,73,83,93],[64,74,84,94]]  Enter    ←挿入する複数の列を用意（行形式）
>>> np.insert( a, 1, i2, axis=1 )  Enter    ←インデックス位置 1 に列を挿入
array([[11, 63, 64, 12, 13, 14],    ←挿入結果
       [21, 73, 74, 22, 23, 24],
       [31, 83, 84, 32, 33, 34],
       [41, 93, 94, 42, 43, 44]])
```

これは，insert の第 2 引数にスカラー値（整数値）を与える場合である．次に複数の列位置に列を挿入する場合について説明する．

例. 列の挿入位置を複数指定する（先の例の続き）

```
>>> np.insert( a, [1,3], np.array(i2).T, axis=1 ) Enter  ←インデックス 1,3 の列位置に挿入
array([[11, 63, 12, 13, 64, 14],    ←挿入結果
       [21, 73, 22, 23, 74, 24],
       [31, 83, 32, 33, 84, 34],
       [41, 93, 42, 43, 94, 44]])
```

この例では insert の第 3 引数に挿入するデータを与える際に，**転置処理**して「2 列の配列」に変換して与えている．このように，挿入位置として非スカラーのもの（リストなど）を与える際は，挿入するデータも元の配列と同じ行数の形式に変換する必要がある．

3.1.5 配列要素の部分的削除

delete を使用することで，配列要素を部分的に削除することができる．delete の第 1 引数には対象となる配列を，第 2 引数には削除位置（インデックス）を与える．

例. 1 次元配列の要素の削除

```
>>> a = np.array([0,1,2,3,4,5])  Enter    ←配列を用意
>>> a2 = np.delete( a, 3 )  Enter    ←インデックス位置 3 の要素を削除
>>> a2  Enter    ←内容確認
array([0, 1, 2, 4, 5])    ←結果表示
```

このように delete は削除結果の配列を返す．元の配列は変化しない．要素が存在しないインデックス位置を指定するとエラー（IndexError）となる．

削除対象のインデックスは複数与えることができる．

例. 複数の要素の削除（先の例の続き）

```
>>> np.delete( a, [0,2,4] )  Enter    ←インデックス位置 0,2,4 の要素を削除
array([1, 3, 5])    ←削除結果
```

delete の第 2 要素に削除対象インデックスをリストにして与えている．

3.1.5.1 区間を指定した削除

delete の第 2 引数に**スライスオブジェクト**[35] を与えることで，指定した区間の要素を全て削除することができる．

書き方：　delete(配列, slice(n_1, n_2))

このように記述することで $n_1 \sim n_2 - 1$ の区間の要素が削除される．

例. 区間の削除（先の例の続き）

```
>>> np.delete( a, slice(1,5) )  Enter    ←インデックス位置 1〜(5-1) の区間を削除
array([0, 5])    ←削除結果
```

これは a[1:5] に相当する部分を削除したことになる．

3.1.5.2 行，列の削除

delete の第 3 引数に ‘axis=0’ を与えると，2 次元配列の指定した行を削除することができる．

[35] データ列の位置を表現するための特殊なオブジェクト．slice(…) のように記述する．

例. サンプル配列

```
>>> a = np.array( [[11,12,13,14],[21,22,23,24],   Enter      ←配列を用意
...                [31,32,33,34],[41,42,43,44]] )
>>> a   Enter      ←内容確認
array([[11, 12, 13, 14],      ←内容表示
       [21, 22, 23, 24],
       [31, 32, 33, 34],
       [41, 42, 43, 44]])
```

例. 行の削除（先の例の続き）

```
>>> np.delete( a, 1, axis=0 )   Enter      ←インデックス位置 1 の行を削除
array([[11, 12, 13, 14],      ←削除結果
       [31, 32, 33, 34],
       [41, 42, 43, 44]])
```

delete の第 2 引数にはリストやスライスオブジェクトを与えて，複数の行を削除することができる．

delete の第 3 引数に ‘axis=1’ を与えると，2 次元配列の指定した列を削除することができる．

例. 列の削除（先の例の続き）

```
>>> np.delete( a, 1, axis=1 )   Enter      ←インデックス位置 1 の列を削除
array([[11, 13, 14],      ←削除結果
       [21, 23, 24],
       [31, 33, 34],
       [41, 43, 44]])
```

delete の第 2 引数にはリストやスライスオブジェクトを与えて，複数の列を削除することができる．

3.1.6 配列の次元の拡大

3.1.6.1 newaxis オブジェクトによる方法

newaxis オブジェクト（接頭辞を付けて np.newaxis）を用いて配列のスライス（添字）を追加することができる．その結果として配列の次元が拡大される．

例. 1 次元の配列を 2 次元に拡大（その 1）

```
>>> a = np.array([0,1,2])   Enter      ← 1 次元配列の作成
>>> a.shape   Enter          ←形状の確認
(3,)          ← 3 つの要素を持つ 1 次元配列である
>>> a2 = a[ np.newaxis, :  ]   Enter      ← a のスライス（添字）を 1 つ増やすことで次元を拡大
>>> a2   Enter          ←内容確認
array([[0, 1, 2]])          ←結果表示
>>> a2.shape   Enter          ←形状の確認
(1, 3)          ← 1 行 3 列のサイズの 2 次元配列である
```

この例で最初に作成した配列 a は，スライス付きの表現で a[:] と記述することでその全体を表すことができる．この表記に則って

```
a[ np.newaxis, :  ]
```

と記述することでスライスが新たに 1 つ追加されることになる．このようにして拡大された配列 a2 は 2 次元配列となる．(次の例参照)

例. 2 次元配列 a2 に別の 2 次元配列を行の方向に連結（先の例の続き）

```
>>> np.append(a2,[[3,4,5]],axis=0)   Enter      ←行の方向に追加
array([[0, 1, 2],          ←連結処理の結果
       [3, 4, 5]])
```

np.newaxis はスライスの任意の位置に挿入できる．

例. 1 次元の配列を 2 次元に拡大（その 2：先の例の続き）

```
>>> a3 = a[ :, np.newaxis ]  Enter     ← a のスライス（添字）を 1 つ増やすことで次元を拡大
>>> a3.shape  Enter          ←形状の確認
(3, 1)          ← 3 行 1 列のサイズの 2 次元配列である
>>> a3  Enter          ←内容確認
array([[0],          ←結果表示
       [1],
       [2]])
```

例. 2 次元配列 a3 に別の 2 次元配列を列の方向に連結（先の例の続き）

```
>>> np.append(a3,[[3],[4],[5]],axis=1)  Enter     ←列の方向に追加
array([[0, 3],          ←連結処理の結果
       [1, 4],
       [2, 5]])
```

3.1.6.2 expand_dims による方法

newaxis オブジェクトによる方法とは別に，expand_dims 関数を用いて配列の次元を拡大することもできる．

例. 1 次元の配列を 2 次元に拡大（その 1）

```
>>> a = np.array([0,1,2])  Enter     ← 1 次元配列の作成
>>> a2 = np.expand_dims( a, axis=0 )  Enter     ← a の先頭の次元を 1 つ増やす
>>> a2  Enter          ←内容確認
array([[0, 1, 2]])          ←結果表示
```

例. 1 次元の配列を 2 次元に拡大（その 2：先の例の続き）

```
>>> a3 = np.expand_dims( a, axis=1 )  Enter     ← a の末尾の次元を 1 つ増やす
>>> a3  Enter          ←内容確認
array([[0],          ←結果表示
       [1],
       [2]])
```

※ newaxis オブジェクトによる方法は expand_dims による方法に比べて，より柔軟である．

3.1.7 データの抽出

データ列の中から指定した要素を抽出する方法について説明する．

3.1.7.1 真理値列によるマスキング

配列のスライスに真理値の配列を与えること[36] で要素の抽出ができる．この場合，与えた真理値列の要素が True である位置に対応する要素を抽出する．

例. 真理値列によるマスキング

```
>>> import numpy as np  Enter     ←モジュールの読込み
>>> a = np.arange(10)  Enter     ← 0〜9 の整数列の生成
>>> a  Enter     ←内容確認
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])     ←結果表示
>>> msk = [True,False,True,False,True,False,False,False,False,False] Enter ←真理値列の作成
>>> a[ msk ]  Enter     ← msk の要素が True である位置に対応する要素の抽出（マスキング）
array([0, 2, 4])     ←抽出結果
```

マスキングに用いる真理値列は ndarray でも良い．その場合は，dtype='bool' として 1 と 0 を要素として配列を作成すると便利である．(次の例)

例. マスキング用配列の作成（先の例の続き）

```
>>> np.array([1,0,1,0,1,0,0,0,0,0],dtype='bool')  Enter     ←内容確認
array([ True, False,  True, False,  True, False, False, False, False, False])
```

先の msk と同等の内容の ndarray ができていることがわかる．

[36] NumPy の高機能インデックス（ファンシーインデックス）である．

3.1.7.2　条件式による要素の抽出

条件式から真理値列を生成することもできる．これを応用して配列の要素を抽出する例を示す．

例. 条件式から真理値列を生成（先の例の続き）

```
>>> a%2==0  Enter      ←条件から真理値列を生成
array([ True, False, True, False, True, False, True, False, True, False ])  ←結果
>>> a[ a%2==0 ]  Enter      ←スライスに条件式を与える（真理値列を与えたことになる）
array([0, 2, 4, 6, 8])      ←抽出結果
```

3.1.7.3　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要素の抽出ができる．条件式の結合や否定は表 24 のような記述による．（注：and，or，not は使えない）

表 24: 条件式の結合や否定のための演算子

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

例. 条件式から真理値列を生成（先の例の続き）

```
>>> a[ (a%2==0) & (a>5) ]  Enter      ←連言（and）による条件式の結合
array([6, 8])      ←結果
>>> a[ ~(a%2==0) & (a>5) ]  Enter      ←否定と連言（and）による結合
array([7, 9])      ←結果
>>> a[ (a%2==0) | (a>5) ]  Enter      ←選言（or）による条件式の結合
array([0, 2, 4, 6, 7, 8, 9])      ←結果
```

指定した条件を満たす要素を取り出すには，次に説明する where が便利である．

3.1.7.4　where による要素の抽出と置換

配列を用いて記述した条件式を where 関数に与えると，その条件を満たす要素のインデックスの配列を束ねたタプルを返す．（注：要素の配列を束ねたタプルではないことに注意すること）

書き方 1：　where(条件式)

例. where による要素の抽出

```
>>> a = np.array( range(10,101,10) ) Enter  ← 10〜100 の 10 刻みの配列を作成
>>> a  Enter      ←内容確認
array([ 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])      ←結果
>>> idx = np.where( a > 50 )  Enter      ← 50 より大きな要素のインデックスを取得する
>>> idx  Enter      ←内容確認
(array([5, 6, 7, 8, 9]),)      ←インデックスの配列のタプル
>>> a[ idx[0] ]  Enter      ←条件を満たす要素（50 より大きな要素）の抽出
array([ 60, 70, 80, 90, 100])      ←結果（配列）
```

指定した条件を満たす要素，満たさない要素を戻り値に使用することも可能である．

書き方 2：　where(条件式, 真の場合の値, 偽の場合の値)

この書き方を用いると，条件に基づく配列の**要素の置換**ができる．

例. 条件に基づく要素の置換　（先の例の続き）

```
>>> np.where( a<50, a, 10*a )  Enter      ← 50 以上の要素のみ 10 倍する処理
array([ 10, 20, 30, 40, 500, 600, 700, 800, 900, 1000])      ←結果（配列）
```

この処理は多次元の配列に対しても実行できる．

例. 多次元配列に対する where

```
>>> a = np.identity( 3 )  Enter      ←3 × 3 の単位行列[37]
>>> a  Enter      ←内容確認
array([[1., 0., 0.],      ←結果
       [0., 1., 0.],
       [0., 0., 1.]])
>>> np.where( a==1, 4, 2 )  Enter      ←1 の要素を 4 に，それ以外を 2 に置き換える処理
array([[4, 2, 2],      ←結果
       [2, 4, 2],
       [2, 2, 4]])
```

先の「書き方 1」の場合に where 関数がタプルを返すのは，多次元配列のインデックス取得に対応するためである．これを次の例で示す．

例. サンプル配列の作成

```
>>> a = np.arange(6).reshape( (2,3) )  Enter      ←2 行 3 列の配列の作成
>>> a  Enter      ←内容確認
array([[0, 1, 2],
       [3, 4, 5]])
```

得られた配列 a の偶数要素は「0 行 0 列目」,「0 行 2 列目」,「1 行 1 列目」である．この配列 a の偶数要素のインデックスを求める例を次に示す．

例. 偶数要素の探索（先の例の続き）

```
>>> idx = np.where( a%2==0 )  Enter      ←偶数要素のインデックスを取得
>>> idx  Enter      ←内容確認
(array([0, 0, 1]), array([0, 2, 1]))      ←2 要素のタプル
```

得られた idx はタプルであり，先頭要素が「行位置」，次の要素が「列位置」のインデックスを意味する配列である．これらを対応させると,「0 行 0 列目」,「0 行 2 列目」,「1 行 1 列目」となり，配列 a の偶数要素の位置を示していることがわかる．

複数の条件に対応する処理に関しては，後の「3.1.8 複数条件による一括置換」（p.80）で解説する．

3.1.7.5 非ゼロ要素の位置と個数の調査

nonzero 関数を使用すると，配列中の非ゼロ要素の位置のインデックスを配列の形で得られる．

例. 非ゼロ要素の位置を調べる

```
>>> q = np.array([0]*3+[1]+[0]*2+[2]+[0,3])  Enter      ←サンプルデータ（1 次元）の作成
>>> q  Enter      ←内容確認
array([0, 0, 0, 1, 0, 0, 2, 0, 3])      ←サンプルデータ
>>> np.nonzero(q)  Enter      ←非ゼロ要素の位置の確認
(array([3, 6, 8]),)      ←検出位置のインデックス配列のタプル
```

タプルの形式で結果が得られているが，その理由は，多次元配列の処理への対応である．（次の例）

例. 2 次元配列中の非ゼロ要素の位置を調べる

```
>>> a = np.array([[0,6,0],[7,0,8]])  Enter      ←サンプルデータ（2 次元）の作成
>>> a  Enter      ←内容確認
array([[0, 6, 0],      ←サンプルデータ
       [7, 0, 8]])
>>> np.nonzero(a)  Enter      ←非ゼロ要素の位置の確認
(array([0, 1, 1]), array([1, 0, 2]))      ←検出位置データ
```

このように，非ゼロ要素の検出位置を，

(行位置インデックス配列, 列位置インデックス配列)

の形で得ている．

count_nonzero 関数を使用すると，配列中のゼロでない要素の数を調べることができる．

[37]「3.1.23.2 単位行列，ゼロ行列，他」（p.152）で解説する．

例. 非ゼロ要素の個数を調べる（先の例の続き）

```
>>> np.count_nonzero(a)   Enter
3
```

3.1.7.6 最大値，最小値，その位置の探索

配列の要素の**最大値**，**最小値**はそれぞれ max，min メソッドで求めることができる．

例. 最大値，最小値

```
>>> a = np.array([2,4,6,8,10,8,6,4,2,4,6,8,10])   Enter    ←サンプルデータ
>>> a.max()   Enter    ←最大値を求める
np.int64(10)    ←結果
>>> a.min()   Enter    ←最小値を求める
np.int64(2)    ←結果
```

最大値，最小値が最初に現れる位置（インデックス値）を argmax ，argmin で調べることができる．

例. 最大値，最小値の位置　（先の例の続き）

```
>>> a.argmax()   Enter    ←最初の最大値の位置（インデックス値）を求める
np.int64(4)    ←結果（インデックスの 4 番目）
>>> a.argmin()   Enter    ←最初の最小値の位置（インデックス値）を求める
np.int64(0)    ←結果（インデックスの 0 番目）
```

■ 2 次元配列に対する max，min，argmax，argmin

2 次元配列に対する max，min，argmax，argmin の動作について例を挙げて説明する．

例. 2 次元配列に対する max，argmax

```
>>> a0 = np.array([[1,5,9],[2,6,7],[3,4,8]])   Enter    ← 2 次元のサンプルデータ
>>> a0.max()   Enter    ←最初の最大値
np.int64(9)          ←結果
>>> np.argmax( a0 )   Enter    ←最初の最大値の位置（インデックス値）を求める
np.int64(2)          ←結果
```

このように，与えられた配列を 1 次元に平坦化した場合の最大値，そのインデックス位置を返す．また max，argmax の引数にキーワード引数 ‘axis=’ を与えると「各列の最大値」,「各行の最大値」に関する値を返す．

例. キーワード引数 ‘axis=0’ を与えた場合（先の例の続き）

```
>>> print( a0 )   Enter    ← 2 次元表示で確認
[[1 5 9]
 [2 6 7]
 [3 4 8]]
>>> a0.max( axis=0 )   Enter    ←各列の最大値を求める
array([3, 6, 9])        ←結果
>>> np.argmax( a0, axis=0 )   Enter    ←各列の最大値の位置を求める
array([2, 1, 0])        ←結果
```

この例の動作の解釈を図 31 に示す．

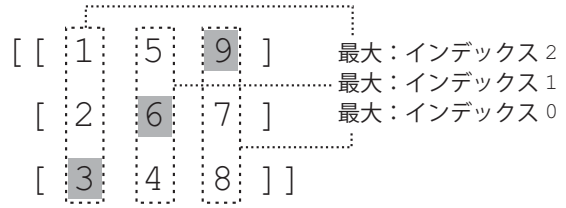


図 31: 各列の最大値の考え方

例. キーワード引数 'axis=1' を与えた場合

```
>>> a1 = np.array([[3,2,1],[4,6,5],[8,7,9]])  Enter      ← 2 次元のサンプルデータ
>>> print( a1 )  Enter      ← 2 次元表示で確認
[[3 2 1]
  [4 6 5]
  [8 7 9]]
>>> a1.max( axis=1 )  Enter      ←各行の最大値を求める
array([3, 6, 9])          ←結果
>>> np.argmax( a1, axis=1 )  Enter      ←各行の最大値の位置を求める
array([0, 1, 2])          ←結果
```

この例の動作の解釈を図 32 に示す.

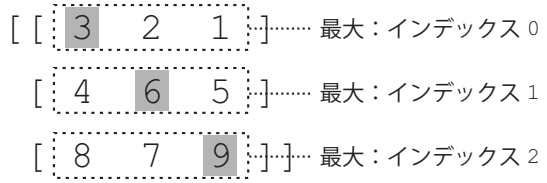


図 32: 各行の最大値の考え方

min，argmin に関しても同様に，キーワード引数 'axis=' によって「列ごと」「行ごと」の処理ができる.

3.1.8 複数条件による一括置換

select 関数を使用すると，複数の条件を配列の各要素に対して判定して値の置換処理を行うことができる．例を挙げて select 関数に関して解説する.

まず次のようなサンプルデータを用意する.

例. サンプルデータ

```
>>> x = np.arange( -2, 7 )  Enter
>>> x  Enter
array([-2, -1, 0, 1, 2, 3, 4, 5, 6])      ← -2〜6 の配列
```

このデータ x の内容を次のような規則で置換することを考える.

1) 0 未満なら 'minus'
2) 2 未満なら 'small'
3) 4 未満なら 'medium'
4) 6 未満なら 'large'
5) 上記以外なら 'very large'

この条件と値の並びを次のようにして作成する.

例. 条件と値の並びの作成（先の例の続き）

```
>>> conds =    [ x<0,      x<2,      x<4,        x<6 ]  Enter        ←条件の並び
>>> choices = [ 'minus', 'small', 'medium', 'large' ]  Enter      ←値の並び
```

これを用いて select 関数を実行する.（次の例）

例. select 関数による一括置換（先の例の続き）

```
>>> r = np.select( conds, choices, default='very large' )  Enter      ←置換処理の実行
>>> r  Enter        ←結果の確認
array(['minus', 'minus', 'small', 'small', 'medium', 'medium', 'large',
       'large', 'very large'], dtype='<U10')
```

条件に対応する形で値が置換されている様子がわかる．select 関数の記述は次の通り.

書き方： select(条件並び, 値の並び, default=デフォルト値)

「条件並び」と「値の並び」を対応させて，各要素の値を決定した新しい配列を返す．どの条件にも合致しない要素の部分は「デフォルト値」となる．また,「条件並び」の中の各条件は互いに排他的である必要はないが，複数の条件

に合致する場合には「条件並び」の前にある条件が優先される．

上と同等の処理を where 関数[38] を用いて実現する方法を次に示す．

例. 同じ処理を where 関数で実現する（先の例の続き）

```
>>> np.where( x<0, 'minus',  Enter
...      np.where( x<2, 'small',  Enter
...          np.where( x<4, 'medium',  Enter
...              np.where( x<6, 'large', 'very large'))))  Enter
array(['minus', 'minus', 'small', 'small', 'medium', 'medium', 'large',
       'large', 'very large'], dtype='<U10')
```

先の例と比較して見ると，select 関数を用いる方が可読性が高いことがわかる．

3.1.9 データの整列（ソート）

sort 関数（あるいは sort メソッド）を使用すると配列の要素を整列（昇順）することができる．

sort 関数：　sort(整列対象の配列)

「整列対象の配列」を整列した結果の配列を返す．元の配列は変更しない．

sort メソッド：　整列対象の配列.sort()

「整列対象の配列」そのものに整列処理を施す．値は返さない．

例. 1 次元配列の整列

```
>>> a = np.array( [3,5,1,4,2,6] )  Enter    ←乱雑な順序の配列
>>> np.sort(a)  Enter    ← sort 関数による整列の実行
array([1, 2, 3, 4, 5, 6])    ←整列結果
>>> a  Enter    ←元の配列の内容確認
array([3, 5, 1, 4, 2, 6])    ←変化無し
>>> a.sort()  Enter    ← sort メソッドによる整列の実行
>>> a  Enter    ←元の配列の内容確認
array([1, 2, 3, 4, 5, 6])    ←整列されている
```

例. 降順に整列

```
>>> a = np.array( [3,5,1,4,2,6] )  Enter    ←乱雑な順序の配列（先の例と同じ）
>>> np.sort(a)[::-1]  Enter    ← sort 関数による整列の後，スライスの工夫で逆順にしている
array([6, 5, 4, 3, 2, 1])    ←整列結果（降順）
```

3.1.9.1 2 次元配列の整列

2 次元配列に対して行方向，あるいは列方向を指定して整列ができる．サンプルとして次のような 2 次元配列を用意する．

例. 2 次元配列のサンプル

```
>>> a = np.array([[1,2,10],[300,3,100],[30,200,20]])  Enter    ← 2 次元配列の作成
>>> a  Enter    ←内容確認
array([[  1,   2,   10],    ← 2 次元になっている
       [300,   3,  100],
       [ 30, 200,   20]])
```

この配列 a に対して sort 関数を実行する．

[38]「3.1.7.4 where による要素の抽出と置換」（p.77）で解説．

例. 2 次元配列の整列

```
>>> np.sort(a,axis=0)  Enter      ←縦方向の整列（列毎の整列）
array([[  1,   2,   10],     ←縦方向に整列されている
       [ 30,   3,   20],
       [300, 200, 100]])
>>> np.sort(a,axis=1)  Enter      ←横方向の整列（行毎の整列）
array([[  1,   2,   10],     ←横方向に整列されている
       [  3, 100, 300],
       [ 20,   30, 200]])
```

この例のように sort 関数にキーワード引数 ‘axis=’ を与えることで整列の方向を制御できる．‘axis=0’ が縦方向，‘axis=1’（暗黙値）が横方向の整列を意味する．sort メソッドの場合も同様のキーワード引数を与えることができる．

3.1.9.2 整列結果のインデックスを取得する方法

argsort 関数（あるいは argsort メソッド）を使用すると配列の要素を整列（昇順）した際のインデックスの並びを返す．結果として得られるインデックスの配列の要素は元の配列に対する位置を意味する．

例. 整列結果のインデックスの並びを取得する

```
>>> a = np.array( [3,5,1,4,2,6] )  Enter      ←乱雑な順序の配列
>>> np.argsort(a)  Enter      ← argsort 関数の実行
array([2, 4, 0, 3, 1, 5]      ←整列結果の要素のインデックス並びが得られている
>>> a.argsort()  Enter      ← argsort メソッドの実行
array([2, 4, 0, 3, 1, 5]      ←同様の結果となる
>>> a  Enter      ←元の配列の内容確認
array([3, 5, 1, 4, 2, 6])      ←変化無し
```

この処理は元の配列を変更しない．この処理の解釈を図 33 に示す．

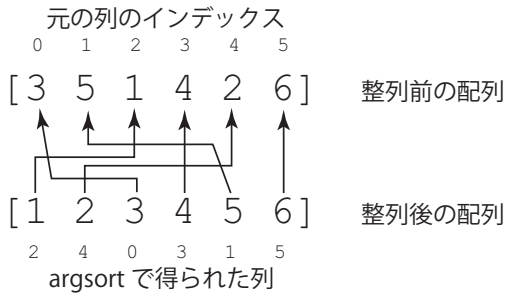


図 33: argsort の処理

argsort は 2 次元の配列に対しても使用できる．

例. 2 次元配列に対する argsort

```
>>> a = np.array([[1,2,10],[300,3,100],[30,200,20]])  Enter      ← 2 次元配列の作成
>>> a  Enter      ←内容確認
array([[  1,   2,   10],     ← 2 次元になっている
       [300,   3,   100],
       [ 30, 200,   20]])
>>> a.argsort(axis=0)  Enter      ←縦方向の整列
array([[0, 0, 0],     ←インデックスの配列が得られている
       [2, 1, 2],
       [1, 2, 1]])
>>> a.argsort(axis=1)  Enter      ←横方向の整列
array([[0, 1, 2],     ←インデックスの配列が得られている
       [1, 2, 0],
       [2, 0, 1]])
```

3.1.10 配列要素の差分の配列

diff 関数を使用すると配列要素間の**差分**を配列として取得できる．

書き方： diff(配列, 差分の階数)

第 2 引数を省略すると 1 階の差分が得られる．例を挙げてこの関数について解説する．

例. サンプルデータ

```
>>> a = np.arange(0,8,1)  Enter
>>> a3 = a**3  Enter
>>> a3  Enter
array([ 0, 1, 8, 27, 64, 125, 216, 343])     ←これをサンプルデータとする
```

このようにして得られた配列 a3 の差分を取る処理を次に示す．

例. 配列の差分（1〜2 階）の配列を得る（先の例の続き）

```
>>> np.diff(a3)  Enter              ← 1 階の差分を求める
array([ 1, 7, 19, 37, 61, 91, 127])     ←結果
>>> np.diff(a3,2)  Enter            ← 2 階の差分を求める
array([ 6, 12, 18, 24, 30, 36])     ←結果
```

例. 配列の差分（3〜4 階）の配列を得る（先の例の続き）

```
>>> np.diff(a3,3)  Enter            ← 3 階の差分を求める
array([6, 6, 6, 6, 6])     ←結果
>>> np.diff(a3,4)  Enter            ← 4 階の差分を求める
array([0, 0, 0, 0])     ←結果
```

この例で行った処理を図 34 に示す．

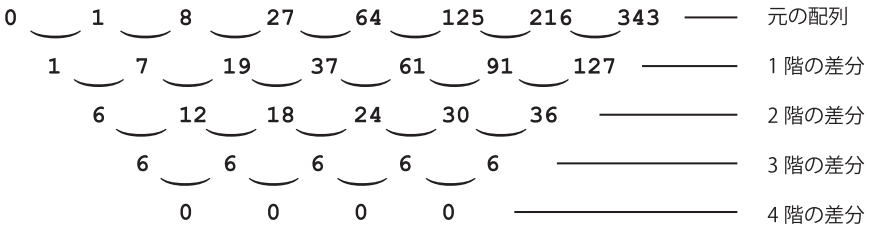


図 34: 配列の差分

3.1.11 配列に対する様々な処理

3.1.11.1 重複する要素の排除

unique 関数を使用すると，配列から重複する要素を排除することができる．

例. 配列の中の重複する要素を排除する

```
>>> a = np.array([2,6,7,5,2,7,5,7,3,8,1,9,1,4,8,4,6,9,3])  Enter  ←重複する要素を持つ配列
>>> a  Enter     ←内容確認
array([2, 6, 7, 5, 2, 7, 5, 7, 3, 8, 1, 9, 1, 4, 8, 4, 6, 9, 3])
>>> np.unique(a)  Enter     ←重複要素の排除
array([1, 2, 3, 4, 5, 6, 7, 8, 9])     ←処理結果
```

重複を排除した**唯一要素**の配列（整列済み）が返される．また，元の配列の内容は変更されない．

3.1.11.2 要素の個数の集計

unique にキーワード引数 ‘return_counts=True’ を与えると，要素毎の出現回数を求めることができる．

例. 要素の個数の集計（先の例の続き）

```
>>> (u,c) = np.unique(a,return_counts=True)  Enter      ←要素数の集計
>>> u  Enter          ←戻り値の第 1 要素の内容確認
array([1, 2, 3, 4, 5, 6, 7, 8, 9])       ←要素の配列
>>> c  Enter          ←戻り値の第 2 要素の内容確認
array([2, 2, 2, 2, 2, 2, 3, 2, 2])  ←上記の要素に対応する出現回数の配列
```

この例の場合は unique はタプルを返す．戻り値のタプルの第 1 要素は「唯一要素」の配列（先の例と同じ），第 2 要素は要素毎の出現回数の配列である．

この手法は，数値以外の要素を持つ配列に対しても用いることができるので，統計学で言う**質的データ（カテゴリデータ）**の集計を実現することができる．

例. 数値以外の要素を持つ配列の集計

```
>>> a = np.array(['a','b','a','c','b'])  Enter          ←非数値要素の配列
>>> np.unique(a,return_counts=True)  Enter     ←要素数の集計
(array(['a', 'b', 'c'], dtype='<U1'), array([2, 2, 1]))  ←集計できている
```

参考）上の例の unique の戻り値のタプルを辞書オブジェクトにしておくと便利なことがある.

例. unique の集計結果を辞書にする（先の例の続き）

```
>>> d = dict( zip(u,c) ) Enter  ←集計結果を zip オブエジェクトにした後で辞書オブジェクトにする
>>> d   Enter                          ←内容確認
{'a':  2, 'b':  2, 'c':  1}        ←辞書オブジェクト
>>> d['b']   Enter          ←要素 'b' の個数を求める
np.int64(2)                   ← 'b' の個数
```

得られた辞書オブジェクトを変数に割り当てておくと，要素の集計表として利用できる.

unique に引数 '`return_index=True`' を与えると，元の配列の中で，唯一要素が最初に現れる位置のインデックスが配列の形で得られる.

例. 唯一要素のインデックス配列を取得する（先の例の続き）

```
>>> np.unique(a,return_index=True)  Enter
(array(['a', 'b', 'c'], dtype='<U1'), array([0, 1, 3]))
```

これは，元の配列の中で 'a' が最初に現れる位置が 0，'b' が最初に現れる位置が 1，'c' が最初に現れる位置が 3 であることを意味する.

unique に引数 '`return_inverse=True`' を与えると，元の配列における唯一要素の位置を，唯一要素の配列内のインデックスで表現したものが配列の形で得られる.

例. 元の配列を唯一要素のインデックスで表現したものを取得する（先の例の続き）

```
>>> np.unique(a,return_inverse=True)  Enter
(array(['a', 'b', 'c'], dtype='<U1'), array([0, 1, 0, 2, 1]))
```

得られた唯一要素の配列の中で 'a' のインデックスは 0，'b' のインデックスは 1，'c' のインデックスは 2 であり，これらインデックスで元の配列を表現すると

```
array([0, 1, 0, 2, 1])
```

となることを意味する.

3.1.11.3　整数要素の集計

負でない整数要素（0 以上の整数の要素）の個数を集計するための関数 bincount がある.

例. 整数要素の集計

```
>>> a = np.array([1,2,2,3,3,3,4,4,4,4,5,5,5,5,5]) Enter  ←サンプルデータの配列
>>> np.bincount(a) Enter  ←集計
array([0, 1, 2, 3, 4, 5])  ←集計結果
```

集計の結果「0 が 0 個，1 が 1 個，2 が 2 個，3 が 3 個，4 が 4 個，5 が 5 個」存在することがわかる．すなわち，結果の配列のインデックスが元の配列の要素に対応する．集計対象のデータに負（マイナス）の値が含まれる場合はエラー（ValueError）となる.

3.1.11.4　指定した条件を満たす要素の集計

`count_nonzero` 関数[39] を応用すると，指定した条件を満たす要素の個数を得ることができる.

[39]「3.1.7.5 非ゼロ要素の位置と個数の調査」（p.78）で解説.

例. 条件を満たす要素の個数を求める

```
>>> q = np.arange( 10 )  Enter          ←サンプルデータの作成
>>> print( q )  Enter          ←内容確認
[0 1 2 3 4 5 6 7 8 9]          ← 0〜9 の配列（要素数は 10）
>>> np.count_nonzero( q < 4 )  Enter          ← 4 未満の要素の個数を求める
4          ←個数
>>> np.count_nonzero( q >= 4 )  Enter          ← 4 以上の要素の個数を求める
6          ←個数
```

これは，真理値配列の False を 0 とみなして True の個数をカウントする処理である．

3.1.12 配列に対する演算：1 次元から 1 次元

NumPy に用意されている数学関数は，配列から配列を生成することができる．これを応用すると関数のプロット（2 次元）が実現できる．

例. 正弦関数の配列の生成

```
>>> import numpy as np  Enter      ←パッケージを'np' として読み込んでいる
>>> lx = np.arange(0.0,6.28,0.01)  Enter      ←データ列（定義域）の生成
>>> ly = np.sin(lx)  Enter      ←上記データ列の各要素に対する正弦関数の値の配列の生成
```

これで，定義域 `lx` に対する正弦関数の値域 `ly` が生成された．これらデータ列を matplotlib パッケージでプロットする例を次に示す．

例. 正弦関数の配列のプロット（先の例の続き）

```
>>> import matplotlib.pyplot as plt  Enter  ← matplotlib パッケージを'plt' として読み込んでいる
>>> g = plt.plot(lx,ly)  Enter      ←プロット処理
>>> plt.show()  Enter      ←プロットを表示
```

この結果，図 35 に示すようなプロットが表示される．

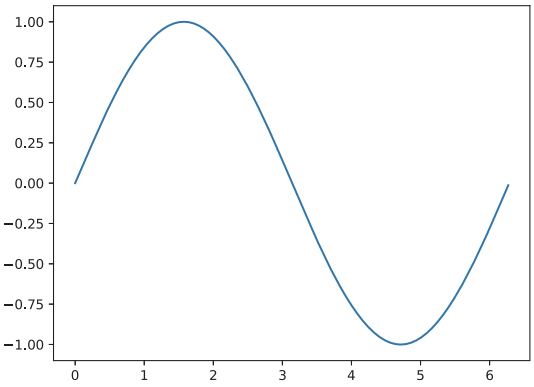


図 35: プロットの表示

あるいは，plt.bar(lx,ly) とすることで**棒グラフ**を描画することもできる．matplotlib に関しては「3.1.13 データの可視化（基本）」以降で解説する．

ここで紹介した正弦関数 sin は NumPy が提供する関数群の中の 1 つであり，この他にもたくさんの関数が提供されている．統計処理において使用頻度が高いものを表 25 に示す[40] ．詳しくは Nympy の公式サイトを参照のこと．

表 25: NumPy が提供する統計処理関連の関数の一部

関数	説明	関数	説明
sum	配列要素の合計を求める	mean	配列要素の平均を求める
var	配列要素の分散を求める	std	配列要素の標準偏差を求める
max	配列要素の最大値を求める	min	配列要素の最小値を求める

[40]使用方法は「3.1.17 統計に関する処理」（p.117）で説明する．

3.1.13　データの可視化（基本）

matplotlib はデータを可視化するためのオープンソースのパッケージであり，関連情報がインターネットサイト https://matplotlib.org/ で公開されている．matplotlib に含まれるデータの可視化の機能は matplotlib.pyplot モジュールにあり，これを読み込むには次のようにする．

```
import matplotlib.pyplot as plt
```

こうすることで，パッケージの別名として plt を指定することができ，可視化のための関数，クラス，プロパティを ‘plt.〜’ として記述することができる．（以後の説明ではこの慣例に従う）

3.1.13.1　作図処理の基本的な手順

上のような形でパッケージを読み込んだ後は，次のような手順で作図処理を行う．

1) 作図の準備

Figure オブジェクトを生成して作図処理に必要となる準備を整える．このとき，グラフの描画サイズを
キーワード引数 ‘figsize=(**横のサイズ**, **縦のサイズ**)’ で指定[41] することができる．

例． `plt.figure( figsize=(6,2) )`　← figure 関数（描画サイズを 6×2 とする）

実行結果として Figure オブジェクトが返される．figure には解像度を引数 ‘dpi=’ で指定（デフォルトは 100）
することもできる．多くの場合においてこの figure 関数の実行は省略できる．

一度に複数の描画面（座標系）を作成する場合は subplots を実行する[42] 必要がある．

例． `plt.subplots( 2,3 )`

これは，小さなグラフを 2 行 3 列に並べて描画する場合の準備の例であり，Figure オブジェクトと
Axes オブジェクトの配列がタプルの形で返される[43] ．subplots には figure と同様の引数を与えることができる．

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

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[44] ．この段階で作図の流れは完了する．

4) 作図環境の終了処理

close() を呼び出して作図処理を終了する．多くの場合においてこの処理は省略できる．

3.1.13.2　2 次元のプロット：折れ線グラフ

正弦関数と余弦関数をプロットするプログラムを例に挙げて，2 次元プロットの基本的な方法について説明する．まずプログラム例 nplot01.py を示す．

プログラム：nplot01.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # データ列の生成
5  lx = np.arange(0.0,6.28,0.01)        # 定義域の生成
6  ly1 = np.sin(lx)                     # 正弦関数の列
7  ly2 = np.cos(lx)                     # 余弦関数の列
8
9  # データ列のプロット
10 plt.figure( figsize=(6,3) )          # 作図処理の開始（省略可）
11 plt.plot(lx,ly1, label='sin(x)')     # プロット(1)
12 plt.plot(lx,ly2, label='cos(x)')     # プロット(2)
13 plt.hlines([-1,0,1],                 # 水平の線
14     0,np.pi*2,ls='--',lw=1.5,color='#b0b0b0')
15 plt.vlines([0,np.pi/2,np.pi,np.pi*3/2,np.pi*2],
16     -1,1,ls='--',lw=1.5,color='#b0b0b0')     # 垂直の線
```

[41] サイズの単位はインチであるが，表示に使用するデバイス（ディスプレイ）によって若干の違いが生じる．デフォルトは (6.4,4.8) である．

[42] これはグラフ作成の柔軟な制御を可能にする方法であり，多くの場合で推奨される．

[43] 「3.1.13.7 複数のグラフの作成」（p.92）で解説する．

[44] 特殊な対話環境（Jupyter Notebook など）では show の実行が必須でないことがある．

```
17 plt.xlabel('x')      # 横軸ラベル
18 plt.ylabel('y')      # 縦軸ラベル
19 plt.legend()         # 凡例の表示
20 plt.title('trigonometric functions: sin, cos')  # タイトルの表示
21 plt.show()       # プロットの表示
22 plt.close()      # 作図処理の終了（省略可）
```

プログラムの説明：

5〜7 行目で定義域の集合 lx とそれに対する，正弦関数，余弦関数の値域の集合 ly1, ly2 を生成している．それらをプロットしているのが 11,12 行目であり，plot 関数を使用している．plot 関数の第 1，第 2 の引数に横軸データと縦軸データをそれぞれ与え，キーワード引数　'label='　にそのデータ列の**ラベル**を与える．これはプロットを表示する際の**凡例**となる．

プログラムの 13〜16 行目では hlines，vlines 関数によって水平と垂直の線を描いている．プログラムの 17,18 行目はグラフの横軸と縦軸のラベルを関数 xlabel，ylabel で与えている．19 行目では関数 legend によってグラフに凡例を付与している．20 行目では関数 title によってグラフにタイトルを付与している．(p.88 の表 28 参照)

最後に関数 show によって実際にプロットを表示している．

このプログラムを実行すると図 36 のようなプロットが表示される．

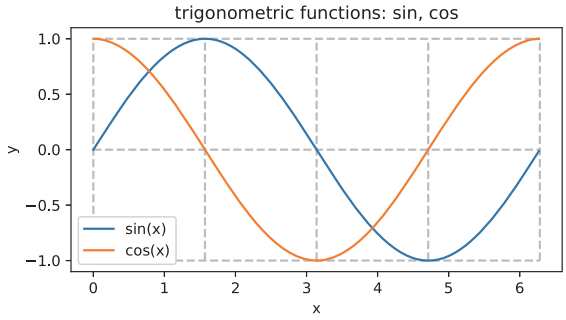


図 36: プロットの表示

3.1.13.3　グラフ描画に関する各種の設定

【plot 関数のキーワード引数】

plot 関数のキーワード引数には先に説明した　'label='　以外にも様々なもの（表 26）がある．

表 26: plot 関数のキーワード引数（一部）

キーワード引数	説明
label=**凡例**	「凡例」を文字列で与える．(デフォルトはラベルなし) legend 関数の実行により，この値が凡例に表示される．
color=**色**	描画色を指定する．次の文字列が指定できる． 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'black' この他にも '#' で始まる 16 進数表現の RGB 指定の文字列 * や **カラーマップ**を用いた方法もある．
lw=**太さ**	描画の線の太さを与える．単位はポイント
ls=**スタイル**	線のスタイルを次のような文字列で与える． '-'（実線：デフォルト），'--'（破線），':'（点線），'-.'（一点鎖線）， 'None'（線なし）　　（他にもあり）
marker= **マーカーの種類**	マーカーの種類を次のような文字列で与える． '.'（ドット），'o'（丸），'s'（■），'*'（星），'+'（十字），'x'（×）， '^'（三角形），'v'（逆三角形），'>'（右向き三角形），'<'（左向き三角形） （その他多数）
markersize= **マーカーのサイズ**	マーカーのサイズを数値で与える．単位はポイント
alpha=**α 値**	透明度を　$0 \leq \alpha \leq 1$　の範囲で指定する．大きいほど濃い．

* HTML，CSS でよく用いられる色表現．

● 色の指定方法の補足

表 26 の 'color=' には **CSS 色名**（CSS color names：いわゆる「Web カラー」）を文字列で与えることができる他，更に様々な形式で色を指定することができる．（下記）

RGBA のタプル ： (R,G,B,A) の形式で各要素は 0〜1.0 の数値．A（α 値）は省略可．
グレースケール ： '0.0'〜'1.0' の文字列表現の数値
サイクラー色 ： 自動着色に使用する色．'C0'〜'C9'（文字列）の 10 色

サイクラー色が意味する色と，それに対応する色コード，別名を表 27 に示す．

表 27: サイクラー色

色	意味	色コード	別名	色	意味	色コード	別名
'C0'	青	'#1f77b4'	'tab:blue'	'C1'	オレンジ	'#ff7f0e'	'tab:orange'
'C2'	緑	'#2ca02c'	'tab:green'	'C3'	赤	'#d62728'	'tab:red'
'C4'	紫	'#9467bd'	'tab:purple'	'C5'	茶	'#8c564b'	'tab:brown'
'C6'	ピンク	'#e377c2'	'tab:pink'	'C7'	灰	'#7f7f7f'	'tab:gray'
'C8'	黄緑	'#bcbd22'	'tab:olive'	'C9'	水色	'#17becf'	'tab:cyan'

※ 同一座標に複数のグラフを重ねて表示する際，色が C0 → C1 →⋯→ C9 → C0 →⋯ と適用される．

表 27 の中の「色」,「色コード」,「別名」の項目を文字列（str）の形で matplotlib の色指定に使用することができる．

参考）matplotlib ではデフォルトの色指定としてサイクラー色を用いるが，これは，（米）Salesforce 社の可視化ツール **Tableau** の色のセットである Tableau Colors（通称 Tab10）に由来する．サイクラー色は色相，彩度，明度の観点で識別し易い色のセットであるとされる．

【水平，垂直の線を描く関数】

● hlines([縦位置のリスト]，左端の位置，右端の位置，オプション⋯)

「左端の位置」〜「右端の位置」の水平の線を描く，第 1 引数に与えた「縦位置のリスト」の要素に対応した縦位置の水平線（複数）を描く．第 1 引数にスカラーを与える（1 つの線の描画）こともできる．

● vlines([横位置のリスト]，下端の位置，上端の位置，オプション⋯)

「下端の位置」〜「上端の位置」の垂直の線を描く，第 1 引数に与えた「横位置のリスト」の要素に対応した横位置の垂直線（複数）を描く．第 1 引数にスカラーを与える（1 つの線の描画）こともできる．

hlines，vlines 関数の「オプション」には plot 関数のキーワード引数に与えるものと同じものがいくつか使用できる．

【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 28 に示す．

表 28: グラフ描画に関する各種の設定を行う関数

関数	説明	関数	説明
xlim(下限, 上限)	横軸の描画範囲の指定	ylim(下限, 上限)	縦軸の描画範囲の指定
xlabel(横軸ラベル)	横軸のラベルの設定	ylabel(縦軸ラベル)	縦軸のラベルの設定
title(タイトル)	グラフのタイトルの設定	legend( )	凡例の表示

【描画内容のレイアウトの調整】

show 関数でグラフを表示する直前に `tight_layout` 関数を実行すると，改めてレイアウトが調整される．この関数は，グラフ描画時に必ず実行すべきものではなく，小さな領域にグラフを作成する際にレイアウトが崩れることがあり，そのような場合にこの関数を実行することで，レイアウトが改善することがある．

線のスタイル，色，太さを設定する例として，次のプログラム nplot02.py を示す．これは，不連続な関数（正接関数：$\tan(x)$）を 3 つの定義域

$$-\frac{\pi}{2} \le x \le \frac{\pi}{2}, \qquad \frac{\pi}{2} \le x \le \frac{3\pi}{2}, \qquad \frac{3\pi}{2} \le x \le \frac{5\pi}{2}$$

に分けてプロットする例であり，それぞれの定義域で線の設定を異なるものにしている．

プログラム：nplot02.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # データ列の生成
5  p1 = -1.0 * np.pi / 2.0
6  p2 = np.pi / 2.0
7  p3 = 3.0 * np.pi / 2.0
8  p4 = 5.0 * np.pi / 2.0
9
10 lx1 = np.arange(p1+0.01, p2, 0.01)  # 定義域の生成(1)
11 ly1 = np.tan(lx1)                   # 正接関数の列(1)
12 lx2 = np.arange(p2+0.01, p3, 0.01)  # 定義域の生成(2)
13 ly2 = np.tan(lx2)                   # 正接関数の列(2)
14 lx3 = np.arange(p3+0.01, p4, 0.01)  # 定義域の生成(3)
15 ly3 = np.tan(lx3)                   # 正接関数の列(3)
16
17 # データ列のプロット
18 plt.figure( figsize=(6,3) )
19 plt.plot(lx1,ly1, color='red',  lw=3, ls=':'  ) # プロット(1)
20 plt.plot(lx2,ly2, color='black',lw=2, ls='--' ) # プロット(2)
21 plt.plot(lx3,ly3, color='blue', lw=1, ls='-.' ) # プロット(3)
22 plt.grid(ls='--',lw=0.8,alpha=1.0)              # グリッド表示
23 plt.ylim(-100,100)
24 plt.xlabel('x')     # 横軸ラベル
25 plt.ylabel('y')     # 縦軸ラベル
26 plt.title('trigonometric functions: tan(x)')  # タイトルの表示
27 plt.tight_layout()  # レイアウトの自動調整
28 plt.show()      # プロットの表示
```

このプログラムを実行すると図 37 のようなプロットが表示される．

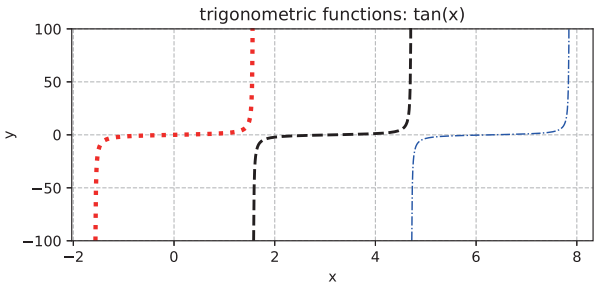


図 37: プロットの表示

プロットに**グリッド線**を描くには grid 関数を実行する．この関数の引数には plot 関数のキーワード引数と共通するものがある．

3.1.13.4 グラフの目盛りの設定

特に指定しない場合はグラフの目盛りは自動的に作成される．

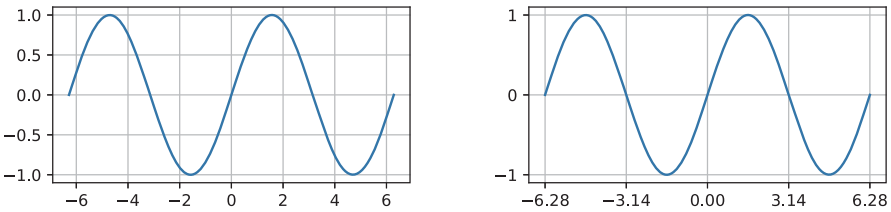
例. 自動的に付けられる目盛り

```
>>> import numpy as np   Enter     ← NumPy の読込み
>>> import matplotlib.pyplot as plt   Enter     ← matplotlib の読込み
>>> x = np.linspace( -2*np.pi, 2*np.pi, 100 )   Enter     ←横軸データの作成
>>> y = np.sin( x )   Enter     ←縦軸データの作成
>>> f = plt.figure( figsize=(4,2) )   Enter     ←描画処理の開始
>>> g = plt.plot( x, y )   Enter     ←グラフのプロット
>>> plt.grid()   Enter     ←グリッド線の表示
>>> plt.show()   Enter     ←描画の実行
```

この処理の結果，図 38 の (a) のような目盛りが表示される．

独自の目盛りを設定するには次のような関数 xticks，yticks を使用する．

横軸の設定： **xticks(ticks=[目盛り位置のリスト])**
縦軸の設定： **yticks(ticks=[目盛り位置のリスト])**



(a) 目盛りの自動設定　　(b) 独自の目盛りの設定

図 38: プロットの表示

これらを使用する例を次に示す.

例. 独自の目盛りを設定（先の例の続き）

```
>>> f = plt.figure( figsize=(4,2) )  Enter
>>> g = plt.plot( x, y )  Enter
>>> t1 = plt.xticks(ticks=[-2*np.pi, -np.pi, 0, np.pi, 2*np.pi])  Enter ←横軸目盛りの設定
>>> t2 = plt.yticks(ticks=[-1,0,1])  Enter  ←縦軸目盛りの設定
>>> plt.grid()  Enter
>>> plt.show()  Enter
```

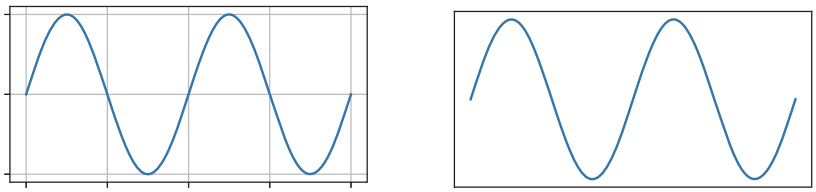
この処理の結果，図 38 の (b) のような目盛りが表示される.

xticks，yticks 関数にキーワード引数 ‘visible=False’ を与えると，図 39 の (a) のように目盛りの数値が表示されない.

例. 目盛りの数値を非表示にする（先の例の続き）

```
>>> f = plt.figure( figsize=(4,2) )  Enter
>>> g = plt.plot( x, y )  Enter
>>> t1 = plt.xticks(ticks=[-2*np.pi, -np.pi, 0, np.pi, 2*np.pi], visible=False )  Enter
>>> t2 = plt.yticks(ticks=[-1,0,1], visible=False )  Enter
>>> plt.grid()  Enter
>>> plt.show()  Enter
```

この例の実行結果として図 39 の (a) のようなグラフが表示される.



(a) 目盛りの数値を非表示に設定　　(b) 目盛りなし

図 39: プロットの表示

目盛りの数値を非表示にするには `tick_params` 関数に引数 `labelbottom=False`, `labelleft=False` を与えて実行することが推奨される.（次の例）

例. 目盛りの数値を非表示にするもう 1 つの方法（先の例の続き）

```
>>> f = plt.figure( figsize=(4,2) )  Enter
>>> g = plt.plot( x, y )  Enter
>>> t1 = plt.xticks(ticks=[-2*np.pi, -np.pi, 0, np.pi, 2*np.pi] )  Enter
>>> t2 = plt.yticks(ticks=[-1,0,1] )  Enter
>>> plt.tick_params( labelbottom=False, labelleft=False )  Enter
>>> plt.grid()  Enter
>>> plt.show()  Enter
```

この例でも図 39 の (a) のようなグラフが表示される.

■ 目盛り自体を非表示にする方法

xticks，yticks 関数のキーワード引数 ‘ticks=’ に空リストを与えると，目盛りが非表示となる.（これによってグリッドも結果的に非表示となる）

例. 目盛りを非表示にする（先の例の続き）

```
>>> f = plt.figure( figsize=(4,2) )   Enter
>>> g = plt.plot( x, y )   Enter
>>> t1 = plt.xticks( ticks=[] )   Enter
>>> t2 = plt.yticks( ticks=[] )   Enter
>>> plt.grid()   Enter      ←これが自動的に無効になる
>>> plt.show()   Enter
```

これを実行すると，図 39 の (b) のようなグラフが表示される．

3.1.13.5　座標の設定に関する簡便な方法

axis 関数を使うと，座標に関する設定を，より簡便な形で行うことができる．この関数で設定できることは，縦横のプロット範囲，座標の表示の有無，アスペクト比を 1:1 にすることなどで，しかも，その内の 1 項目の設定である．

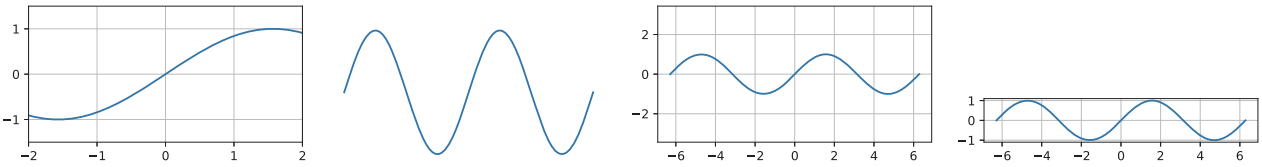
例. 縦横のプロット範囲の設定（先の例の続き）

```
>>> f = plt.figure( figsize=(4,2) )   Enter
>>> g = plt.plot( x, y )   Enter
>>> a = plt.axis([-2,2,-1.5,1.5])   Enter      ←縦横のプロット範囲の設定
>>> plt.grid()   Enter
>>> plt.show()   Enter
```

プロット範囲はリストの形式で

[**横軸の下限, 横軸の上限, 縦軸の下限, 縦軸の上限**]

で設定する．この例では，横軸のプロット範囲を -2 から 2，縦軸のプロット範囲を -1.5 から 1.5 と設定している．この関数はプロット範囲のデータを返す．これを実行すると，図 40 の (a) のように表示される．



(a) `[-2,2,-1.5,1.5]`　(b) `'off'`　(c) `'equal'`　(d) `'scaled'`

図 40: axis 関数による座標の設定：与える引数ごとの表示例

axis 関数の引数に `'off'` を与えると図 40 の (b) のようにグラフのみ（座標の表示なし）の表示，`'equal'` を与えると図 40 の (c) のようにアスペクト比が 1:1 の表示，`'scaled'` を与えると図 40 の (d) のようにアスペクト比が 1:1 で表示され，座標の表示範囲が最低限のもの（縦横のプロット範囲がデータの範囲に設定される）となる．

3.1.13.6　対数軸のグラフの作成

グラフの横軸（x 軸）を**対数軸**にするには xscale 関数に引数'log' を与えて実行する．同様に，縦軸（y 軸）を対数軸にするには yscale 関数を実行する．対数軸の**底**（base）を設定するには，キーワード引数「base=**底**」（デフォルトは 10）を与える．

右の例は 10^0 から 10^5 までの x の値に対する $100\log_{10} x$ の値をプロットするものである．右の例では縦横の座標とも通常の設定（対数軸ではない）であり，図 41 の (a) のようなグラフが作成される．

右の例の処理において plot 関数の後に

```
plt.xscale('log')
```

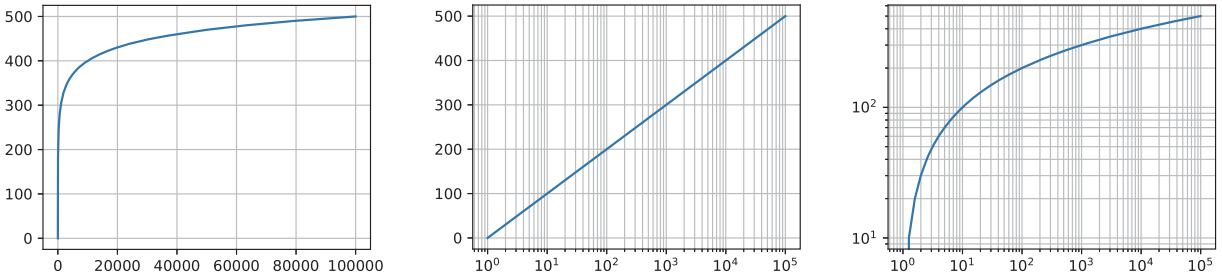
を実行すると横軸が対数軸となり，図 41 の (b) のようなグラフが作成される．更に，これに加えて

```
plt.yscale('log')
```

を実行すると縦横の両方の軸が対数軸となり，図 41 の (c) のようなグラフが作成される．

例. 通常の設定でのプロット

```
>>> import numpy as np   Enter
>>> import matplotlib.pyplot as plt   Enter
>>> x = np.logspace(0,5,51)   Enter
>>> y = 100*np.log10(x)   Enter
>>> fig = plt.figure( figsize=(4,3) )   Enter
>>> g = plt.plot( x, y )   Enter
>>> plt.grid()   Enter
>>> plt.show()   Enter
```



(a) 通常の座標軸　(b) 横軸を対数軸にした例　(c) 縦横の軸を対数軸にした例

図 41: xscale，yscale 関数による軸の設定

3.1.13.7　複数のグラフの作成

1 枚の図に複数のグラフを作成するには subplots 関数を使用する．subplots を呼び出す際にグラフの縦横の並び（行，列の数）を指定すると，それに対応する作図用オブジェクト（Axes オブジェクト）が生成される．実際にプログラムの例を示して説明する．

■ 2 つのグラフを作成する例

正弦関数と余弦関数の 2 つを別のグラフとして作成するプログラム nplot02-2.py を次に示す．

プログラム：nplot02-2.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # データ列の生成
5  x  = np.arange(-6.3, 6.3, 0.01) # 定義域の生成
6  y1 = np.sin(x)                  # 値域の生成(1)
7  y2 = np.cos(x)                  # 値域の生成(2)
8
9  # matplotlibによるプロット
10 (fig, ax) = plt.subplots(2, 1, figsize=(5,3))
11 fig.subplots_adjust(hspace=1.0)
12
13 ax[0].plot(x, y1, linewidth=1, color='red')
14 ax[0].set_title('sin(x)')
15 ax[0].set_ylabel('y')
16 ax[0].set_xlabel('x')
17 ax[0].grid(True)
18
19 ax[1].plot(x, y2, linewidth=1, color='green')
20 ax[1].set_title('cos(x)')
21 ax[1].set_ylabel('y')
22 ax[1].set_xlabel('x')
23 ax[1].grid(True)
24
25 plt.show()
```

このプログラムを実行してグラフを表示した例を図 42 に示す．

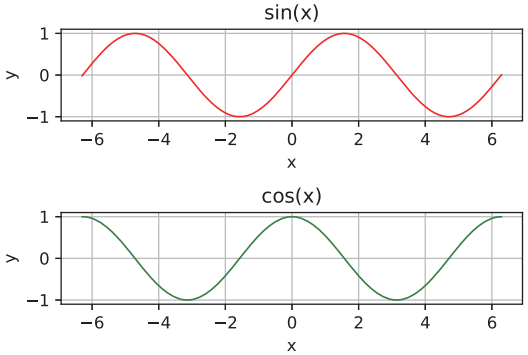


図 42: 2 つのグラフを表示した例（縦に 2 つ）

プログラムの解説：

nplot02-2.py の 5〜7 行目で定義域と値域（正弦関数，余弦関数）を生成している．10 行目で 2 行 1 列の並びでグ

ラフを表示する形で subplots を呼び出している．

書き方： subplots(**行数, 列数**)

また，このときキーワード引数 figsize=(**横のサイズ, 縦のサイズ**) を与えると，グラフ全体のサイズを指定することができる．subplots の実行後，Figure オブジェクトと Axes オブジェクトのタプルが返される．今回のプログラムでは，これらを (fig, ax) に受け取っており，ax[インデックス] に対して描画処理を行っている．

11 行目にあるように subplots_adjust メソッドを呼び出すと，描画するグラフの間隔を設定することができる．縦に並ぶグラフの上下の間隔はキーワード引数 hspace に，横に並ぶグラフの左右の間隔はキーワード引数 wspace に指定する．また，指定する値は，各描画領域の大きさ（の平均）に対する比率である．

プログラム nplot02-2.py の 10〜11 行目を，

```
(fig, ax) = plt.subplots(1, 2, figsize=(7,3))
fig.subplots_adjust(wspace=0.4)
```

と書き換えると，左右にグラフを並べる形の表示となる．(図 43 参照)

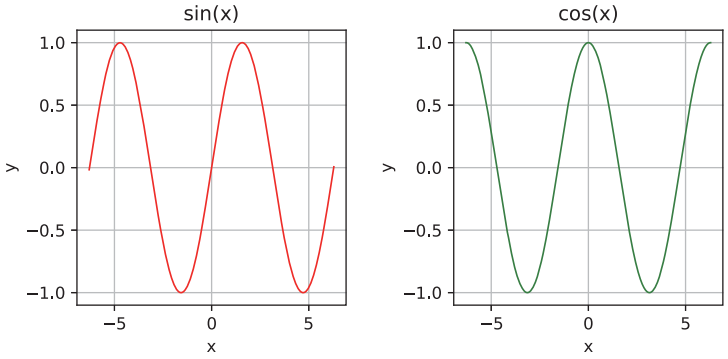


図 43: 2 つのグラフを表示した例（横に 2 つ）

参考） plt.subplots(...) として実行することも可能である．

■ 縦横にグラフを表示する例

正弦関数，余弦関数，指数関数，対数関数の 4 つを別のグラフとして作成するプログラム nplot02-4.py を次に示す．

プログラム： nplot02-4.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # データ列の生成
5  x1 = np.arange(-6.3, 6.3, 0.01) # 定義域の生成(1)
6  y1 = np.sin(x1)                 # 値域の生成(1)
7  y2 = np.cos(x1)                 # 値域の生成(2)
8  y3 = np.exp(x1)                 # 値域の生成(3)
9
10 x2 = np.arange(0.01, 10, 0.01)  # 定義域の生成(2)
11 y4 = np.log(x2)                 # 値域の生成(4)
12
13 # matplotlibによるプロット
14 (fig, ax) = plt.subplots(2, 2, figsize=(8,4))
15 fig.subplots_adjust(wspace=0.3,hspace=0.7)
16
17 ax[0,0].plot(x1, y1, linewidth=1, color='red')
18 ax[0,0].set_title('sin(x)')
19 ax[0,0].set_ylabel('y')
20 ax[0,0].set_xlabel('x')
21 ax[0,0].grid(True)
22
23 ax[1,0].plot(x1, y2, linewidth=1, color='green')
24 ax[1,0].set_title('cos(x)')
25 ax[1,0].set_ylabel('y')
26 ax[1,0].set_xlabel('x')
27 ax[1,0].grid(True)
28
29 ax[0,1].plot(x1, y3, linewidth=1, color='blue')
30 ax[0,1].set_title('exp(x)')
31 ax[0,1].set_ylabel('y')
```

```
ax[0,1].set_xlabel('x')
ax[0,1].grid(True)

ax[1,1].plot(x2, y4, linewidth=1, color='black')
ax[1,1].set_title('log(x)')
ax[1,1].set_ylabel('y')
ax[1,1].set_xlabel('x')
ax[1,1].grid(True)

plt.show()
```

プログラムの解説：

nplot02-4.py の 5〜11 行目でデータ列（定義域，正弦関数，余弦関数，指数関数，対数関数）を生成している．14 行目で 2 行 2 列の並びでグラフを表示する形で subplots を呼び出している．この結果として生成されたオブジェクト ax[行インデックス, 列インデックス] に対して描画処理を行っている．

このプログラムを実行してグラフを表示した例を図 44 に示す．

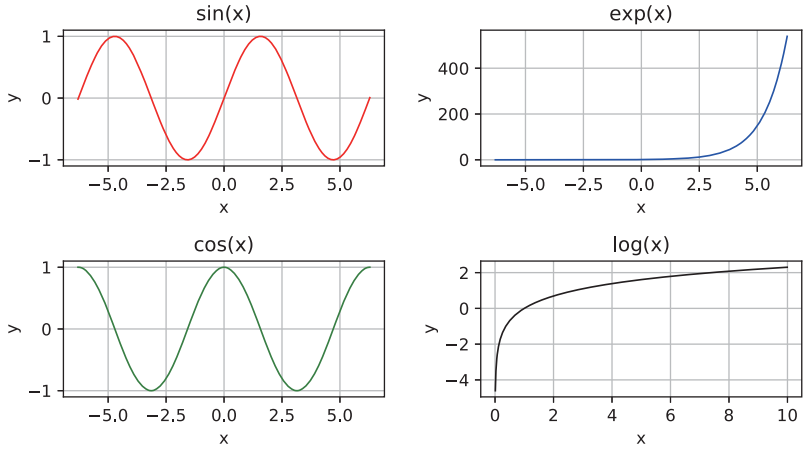


図 44: 縦横に 4 つのグラフを表示した例

3.1.13.8 matplotlib のグラフの構造

1 つのグラフを描画する方法と複数のグラフを描画する方法について，別々のケースとして先に解説したが，ここでは matplotlib が描画するグラフの構造について説明し，両方のケースの違いについての理解を促す．その前に，matplotlib が描くグラフの各部分について改めて説明する．

次のようなプログラム nplot02-6.py が作成するグラフを例に用いて説明する．

プログラム：nplot02-6.py

```
import numpy as np
import matplotlib.pyplot as plt
# データ1
x1 = np.linspace(-2.2,2.2,100)
y1 = x1**5-5*x1**3+4*x1
# データ2
x2 = np.linspace(-2,2,100)
y2 = x2**2-1
############################################################
# 2つの図                                                  #
############################################################
(fig,axs) = plt.subplots(1,2,figsize=(9,4))
fig.subplots_adjust(wspace=0.32)
# 1つ目の図
axs[0].set_xlim(x1.min(),x1.max())
axs[0].set_ylim(y1.min(),y1.max())
axs[0].plot(x1,y1)
axs[0].vlines(0,y1.min(),y1.max(),lw=1.5,color='gray')
axs[0].hlines(0,x1.min(),x1.max(),lw=1.5,color='gray')
axs[0].grid(True)
axs[0].set_title('x**5-5*x**3+4*x')
axs[0].set_xlabel('x')
axs[0].set_ylabel('y')
```

```
24 # 2つ目の図
25 axs[1].set_xlim(x2.min(),x2.max())
26 axs[1].set_ylim(y2.min(),y2.max())
27 axs[1].plot(x2,y2)
28 axs[1].vlines(0,y2.min(),y2.max(),lw=1.5,color='gray')
29 axs[1].hlines(0,x2.min(),x2.max(),lw=1.5,color='gray')
30 axs[1].grid(True)
31 axs[1].set_title('x**2-1')
32 axs[1].set_xlabel('x')
33 axs[1].set_ylabel('y')
34 fig.suptitle('Multiple plots')
35 plt.show()
```

このプログラムは 2 つの関数 $y = x^5 - 5x^3 + 4x$，$y = x^2 - 1$ のグラフを描くものであり，実行すると図 45 のようなグラフが表示される．

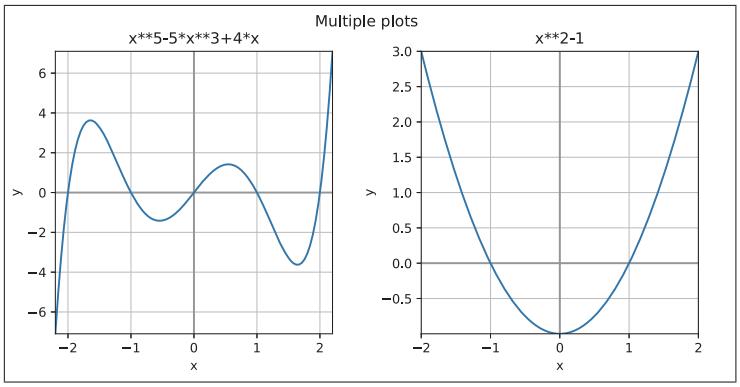


図 45: nplot02-6.py の実行結果

このグラフを図解すると図 46 のような部分から成ることがわかる．

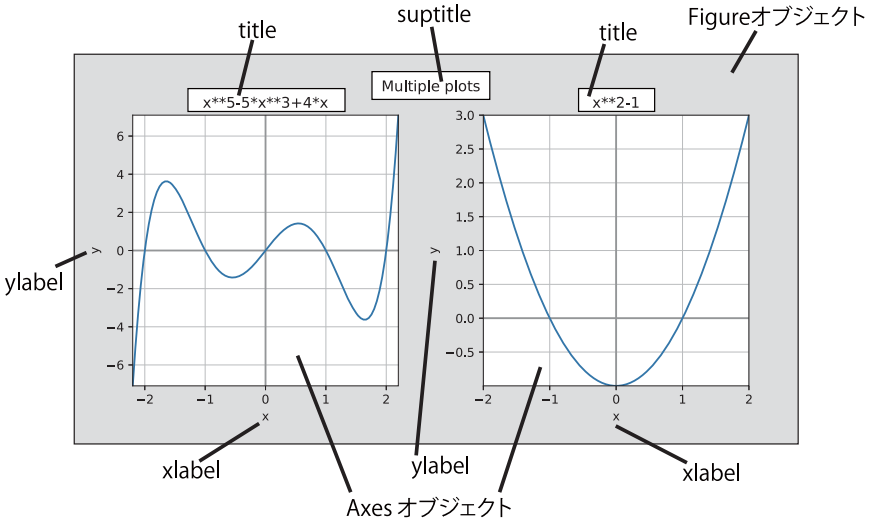


図 46: matplotlib のグラフの構成

【グラフの各部分の説明】

● Figure オブジェクト

これは matplotlib が show メソッドで描画するグラフ全体を意味する．このオブジェクトは，グラフ全体のタイトルを意味する suptitle を持つ．Figure オブジェクトは Axes オブジェクトを持ち，これが実際にグラフを描く描画面（座標系）である．複数のグラフを同時に描画するケースでは，1 つの Figure オブジェクトが複数の Axes を配列の形で保持する．

● Axes オブジェクト　（AxesSubplots と表示される場合あり）

これはプロットされた個々のグラフ平面を意味する．このオブジェクトは下記のような部分を持つ．

xlabel - グラフの横軸ラベル
ylabel - グラフの縦軸ラベル
title - グラフのタイトル

グラフのタイトルや軸ラベル，描画範囲を設定するメソッドなどが，1 つのグラフを描画する場合と複数のグラフを描画する場合で異なることがある．これは，処理の対象とするオブジェクトが異なることに起因する．すなわち，下記のような 2 つの異なる描画形態がある．

1. Figure オブジェクトが唯 1 つの Axes オブジェクトを持つ場合
描画に関する各種のメソッドは「plt.〜」の形（pyplot 関数）で実行する．実際の処理は Figure オブジェクト配下の Axes オブジェクトに対して行われる．

2. Figure オブジェクトが複数の Axes オブジェクト（配列）を持つ場合
描画に関する各種のメソッドは個々の Axes オブジェクトに対して行う．

以上のことを踏まえると，単一のグラフを描画する場合においても，2 つの方法（Figure に対する描画と Axes に対する描画）を取ることができる．次に示す 2 つのプログラム nplot02-5.py，nplot02-7.py は，同じ処理をそれぞれ異なる方法で描画するものである．

プログラム：nplot02-5.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  # データ1
4  x1 = np.linspace(-2.2,2.2,100)
5  y1 = x1**5-5*x1**3+4*x1
6  ###################################
7  # 1つの図
8  plt.figure( figsize=(4,4) )
9  # 描画範囲の指定
10 plt.xlim(x1.min(),x1.max())
11 plt.ylim(y1.min(),y1.max())
12 plt.plot(x1,y1)
13 plt.vlines(0,y1.min(),y1.max(),
14             lw=1.5,color='gray')
15 plt.hlines(0,x1.min(),x1.max(),
16             lw=1.5,color='gray')
17 plt.grid(True)
18 plt.xlabel('x')
19 plt.ylabel('y')
20 plt.title('x**5-5*x**3+4*x')
21 plt.suptitle('Single plot')
22 plt.show()
```

プログラム：nplot02-7.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  # データ1
4  x1 = np.linspace(-2.2,2.2,100)
5  y1 = x1**5-5*x1**3+4*x1
6  ###################################
7  # 1つの図：別の方法
8  fig,ax = plt.subplots(
9              figsize=(4,4))
10 ax.set_xlim(x1.min(),x1.max())
11 ax.set_ylim(y1.min(),y1.max())
12 ax.plot(x1,y1)
13 ax.vlines(0,y1.min(),y1.max(),
14             lw=1.5,color='gray')
15 ax.hlines(0,x1.min(),x1.max(),
16             lw=1.5,color='gray')
17 ax.grid(True)
18 ax.set_xlabel('x')
19 ax.set_ylabel('y')
20 ax.set_title('x**5-5*x**3+4*x')
21 fig.suptitle('Single plot')
22 plt.show()
```

上記 2 つのプログラムが使用しているメソッドの対比を表 29 に示す．

表 29: 各種の設定のための関数の違い

説明	pyplot 関数	Axes 用メソッド
横軸の描画範囲の指定	xlim(下限, 上限)	set_xlim(下限, 上限)
縦軸の描画範囲の指定	ylim(下限, 上限)	set_ylim(下限, 上限)
横軸の設定	xticks(各種の引数)	set_xticks(各種の引数)
縦軸の設定	yticks(各種の引数)	set_yticks(各種の引数)
横軸のラベルの設定	xlabel(横軸ラベル)	set_xlabel(横軸ラベル)
縦軸のラベルの設定	ylabel(縦軸ラベル)	set_ylabel(縦軸ラベル)
グラフのタイトルの設定	title(タイトル)	set_title(タイトル)

先の 2 つのプログラムを実行すると，どちらも図 47 のようなグラフを描画する．

参考）Figure オブジェクト，Axes オブジェクトは gcf()，gca() によって

```
fig = plt.gcf()
ax = plt.gca()
```

として取得することも可能であるが，この方法はあまり推奨されない．

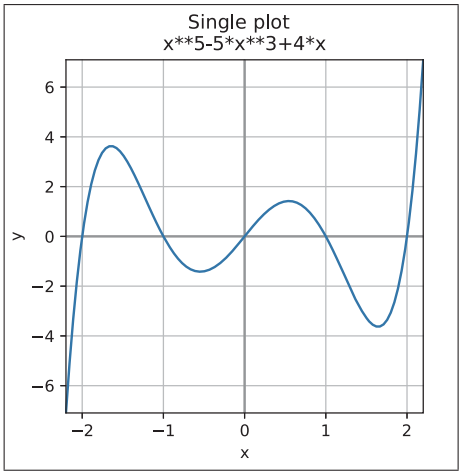


図 47: nplot02-5.py，nplot02-7.py の実行結果

3.1.13.9　グラフの枠を非表示にする方法

グラフの枠は Axes オブジェクトの spines プロパティとしてアクセスできる．このプロパティは辞書オブジェクト（OrderedDict）であり，上下左右を意味する 'top'，'bottom'，'left'，'right' のキーを持つ．例えば Axes オブジェクト a の上の枠は，

```
a.spines['top']
```

という記述でアクセスでき，それは Spine オブジェクトである．

グラフの枠線を非表示にするには，当該 Spine オブジェクトの表示属性を False にする．例えば Axes オブジェクト a の上の枠を非表示にするには，

```
a.spines['top'].set_visible(False)
```

とする．set_visible は Spine オブジェクトの表示属性を設定するメソッドである．

以上のことを応用したサンプルプログラム nplot02-8.py を示す．

プログラム：nplot02-8.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  # サンプルデータ
4  x = np.linspace(0,2*np.pi,360)
5  y = np.sin(3*x)
6
7  # プロット
8  (fig,axs) = plt.subplots(1,5,figsize=(10,2))
9  for ax in axs:      # グラフを5枚描画
10     ax.plot(x,y)
11     ax.set_xticks(ticks=[]);    ax.set_yticks(ticks=[])
12 # 枠の消去
13 axs[0].set_title('no top')
14 axs[0].spines['top'].set_visible(False)      # 上の枠を消去
15 axs[1].set_title('no bottom')
16 axs[1].spines['bottom'].set_visible(False)  # 下の枠を消去
17 axs[2].set_title('no left')
18 axs[2].spines['left'].set_visible(False)    # 左の枠を消去
19 axs[3].set_title('no right')
20 axs[3].spines['right'].set_visible(False)   # 右の枠を消去
21 axs[4].set_title('borderless')
22 axs[4].spines[:].set_visible(False)         # 全ての枠を消去
23 plt.show()
```

このプログラムでは，9〜11 行目で同じグラフを 4 つ描画し，14 行目，16 行目，18 行目，20 行目，22 行目で枠の属性を非表示にしている．このプログラムを実行すると図 48 のように表示される．

参考）枠，目盛りを全て無効にするには次のような記述も使える．

```
Axes オブジェクト.axis('off')
```

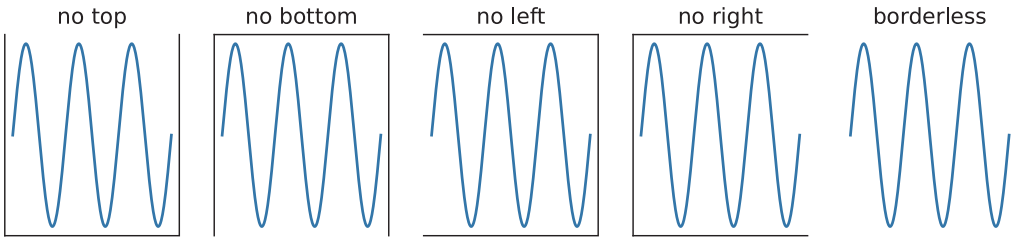


図 48: 枠の非表示

3.1.13.10　アスペクト比の設定

描画するグラフの縦横の比率である**アスペクト比**は，描画対象のデータが描画領域に収まるようにシステムが自動的に調整する．これを変更するには，Axes オブジェクトに対する set_aspect メソッドを使用する．

書き方：　Axes オブジェクト.set_aspect(設定)

「設定」に 'auto'（デフォルト）を指定すると自動調整となる．また 'equal' を指定すると縦横の比率が 1:1 となる.「設定」に数値を与えると，横軸に対する縦軸の比率が設定でき，自由にアスペクト比を設定したい場合にこの方法を取る．

アスペクト比の設定について，次のサンプルプログラム nplot02-9.py を用いて解説する．

プログラム：nplot02-9.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  t = np.linspace(0,2*np.pi,361)
5  x = np.cos(t)
6  y = np.sin(t)
7
8  fig,ax = plt.subplots(figsize=(5,2.5))
9  ax.plot(x,y)
10 ax.set_xlabel('x')
11 ax.set_ylabel('y')
12 ax.set_title('Plot of (x=cos(t),y=sin(t)): 0<=t<=2*pi')
13 #ax.set_aspect( 'equal', adjustable='datalim' )
14 plt.tight_layout()
15 plt.show()
```

これは原点 (0,0)，半径 1 の円を描くプログラムであるが，実行すると図 49 のように横に伸びたようなグラフとなる．

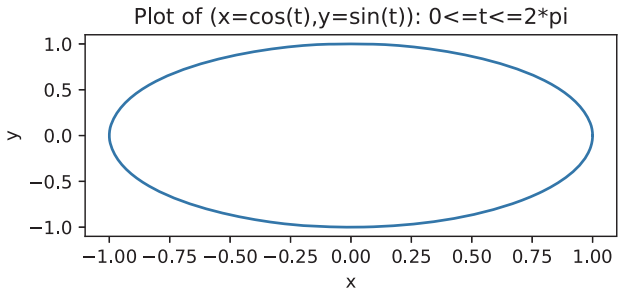
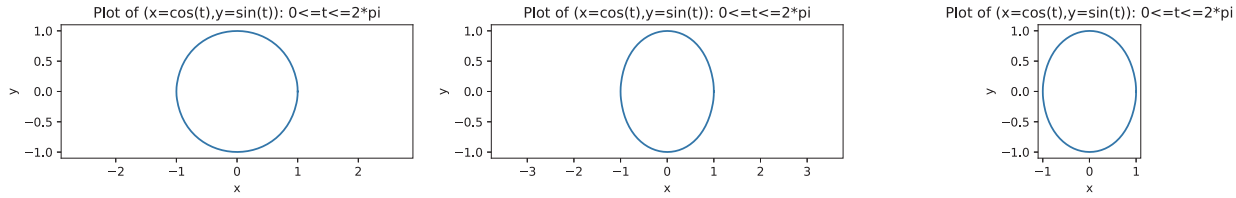


図 49: デフォルトのアスペクト比設定による描画

これは，figsize=(5,2.5) の描画領域に合わせるように自動的にアスペクト比を調整したことが原因である．これを調整するために，nplot02-9.py の 13 行目のコメント記号「#」を外し，set_aspect メソッドを有効にすると，アスペクト比設定が 'equal' となり，図 50 の (a) のようなグラフとなる．

アスペクト比の数値として 1.3 を指定すると図 50 の (b) のようになる．また，オプション引数の adjustable='datalim' を外すと図 50 の (c) のようになり，描画範囲が最小限に限定される．

参考）plt.axis('equal') でも 1:1 のアスペクト比が設定できる．またその場合は強制的に adjustable='datalim' が指定される．



(a) 'equal'　(b) 1.3　(c) 1.3（`adjustable='datalim'` なし）

図 50: アスペクト比の調整

3.1.13.11　極座標プロット

極座標にグラフをプロットするには，描画対象の Axes を極座標形式（PolarAxesSubplot オブジェクト）にする．これに関して例を挙げて説明する．

プログラム：nplotPol01.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  # データの作成
4  x = np.linspace( 0, 6*np.pi, 1080 )
5  y = 2*x
6  # プロット
7  plt.figure( figsize=(5,5) )
8  ax = plt.subplot(projection='polar')
9  ax.plot( x, y )
10 plt.show()
```

プログラム：nplotPol02.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  # データの作成
4  x = np.linspace( 0, 2*np.pi, 360 )
5  y = np.sin(10*x)+1
6  # プロット
7  fig = plt.figure(figsize=(5,5))
8  ax = fig.add_subplot(projection='polar')
9  ax.plot( x, y )
10 plt.show()
```

上のプログラム nplotPol01.py，nplotPol02.py では 8 行目で Axes オブジェクトを取得している．具体的には subplot メソッドや add_subplot メソッドを使用するが，このときにキーワード引数 'projection='polar'' を指定することで，極座標プロット用の PolarAxesSubplot オブジェクトが得られる．そして，figure 関数で作成された Figure オブジェクトにそれらが登録される．

これらプログラムを実行して得られるグラフを図 51 に示す．

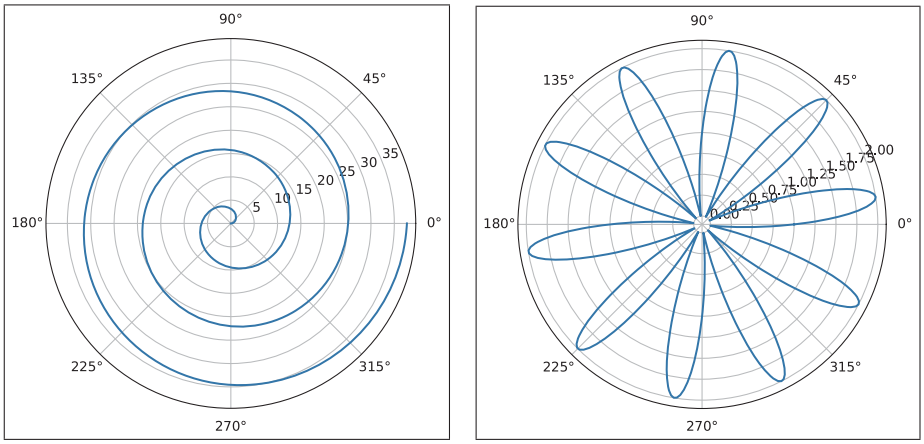


図 51: nplotPol01.py，nplotPol02.py の実行結果

■ 直交座標プロットと極座標プロットを混在させる方法

subplots 関数で複数の描画面（座標系）を作成し，特定の描画面を極座標系にする方法について考える．先のサンプルプログラム nplotPol01.py，nplotPol02.py では plt.figure(figsize=(5,5)) を実行した時点ではまだ Axes オブジェクトも PolarAxesSubplot オブジェクトを存在しておらず，subplot 関数や gca 関数の実行によって PolarAxesSubplot オブジェクトが生成されている．これに対して，subplots 関数で複数の描画面（座標系）を一度に作成するケースでは状況が少し異なる．これについて次のサンプルプログラム nplotPol02_2.py を示して解説する．

プログラム：nplotPol02_2.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  # データの作成
```

```
4  x = np.linspace( 0, 2*np.pi, 360 )
5  y = np.sin(10*x)+1
6  # プロット
7  fig,ax = plt.subplots( 1,2, figsize=(10,5) )
8  # subplotsで作成したインデックス0（1行1列1目の1番目）の直交座標に作図
9  ax[0].plot( x, y )
10 # 極座標を新規作成して「1行2列目の2番目」の描画面として再登録
11 fig.delaxes(ax[1])  # 既存のAxesオブジェクトを削除（必須）
12 ax[1] = fig.add_subplot(1,2,2,projection='polar')   # 新規作成した極座標を登録
13 #ax[1] = fig.add_subplot( 122 ,projection='polar')  #  （もう一つの書き方）
14 ax[1].plot( x, y )
15
16 plt.show()
```

これは同じ関数のプロットを直交座標と極座標で見せるプログラムである．7 行目の subplots 関数を実行した時点で ax[0]，ax[1] の 2 つの Axes オブジェクトが 1 行 2 列の並びで作成される．まず，直交座標 ax[0] にプロットしているが，この時点では ax[1] はまだ直交座標である．これを，11 行目の delaxes 関数によって削除した後，fig に対する add_subplot メソッド（12 行目）で作成した極座標オブジェクトを改めて登録し，それに対してプロットを実行（14 行目）している．

このプログラムを実行すると図 52 のようなグラフが表示される．

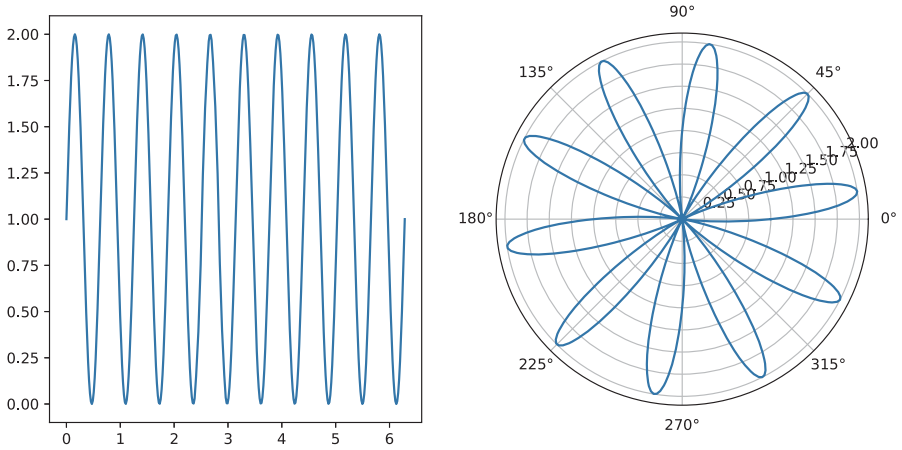


図 52: 1 つの Figure オブジェクト上に直交座標と極座標を混在させた例

nplotPol02_2.py では既存の Axes オブジェクトを delaxes 関数で削除すること（削除しなければ元の Axes オブジェクトの情報が残留する）と，add_subplot で新規に極座標を作成する 2 つのことが重要である．

書き方：　Figure オブジェクト.add_subplot(行, 列, 順序, [オプション])

新規に作成する座標オブジェクトの，Figure オブジェクト内での位置と配置の順序を「行」,「列」,「順序」に整数値（1 から始まる）で指定する．

「行」,「列」,「順序」は**マジックナンバー**の形式で指定することもできる．

書き方：　Figure オブジェクト.add_subplot(マジックナンバー, [オプション])

マジックナンバーは 3 桁の整数値で「n,m,i」と書き,「n 行 m 列目の i 番目の座標オブジェクト」を意味する．先のプログラム nplotPol02_2.py の 13 行目のコメントは，12 行目と同等のメソッド実行の記述例であり,「122」というマジックナンバーが記述されている．これは「1 行 2 列目に配置された 2 番目の座標オブジェクト」を意味する．

参考）matplotlib は MATLAB [45] を意識して設計されている.「1 から始まる整数インデックス」や「マジックナンバー」も MATLAB に由来する.

3.1.13.12　レーダーチャート（極座標の応用）

極座標プロットを応用することで**レーダーチャート**を作成することができる．サンプルプログラム nplotPol03.py を次に示す.

[45]（米）MathWorks 社が開発している数値解析ソフトウェア

プログラム：nplotPol03.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # データの作成
5  lbl = ['A','B','C','D','E','F']                # ラベル
6  x = np.linspace( 0, 2*np.pi, len(lbl)+1 )      # 各ラベルの角度
7  # レーダーチャートにする値
8  # 最初の要素と同じものを最後に加えることでグラフを閉じる
9  d1 = [1, 0.4, 0.2, 0.7, 0.5, 1, 1 ]            # データ1
10 d2 = [0.3, 0.5, 1, 0.2, 0.8, 0.5, 0.3 ]       # データ2
11
12 # プロット
13 plt.figure( figsize=(5,5) )
14 ax = plt.subplot(projection='polar')
15 ax.plot( x, d1, label='d1' )
16 ax.plot( x, d2, label='d2' )
17 ax.set_xticks(x[:-1])
18 ax.set_xticklabels(lbl)
19 #ax.set_thetagrids(x[:-1]*180/np.pi, lbl)     # 上記2行と同等の別の方法
20 ax.spines['polar'].set_visible(False)         # 外周の枠を消去
21 plt.legend()
22 plt.show()
```

このプログラムでは，角度のデータ x に対するレーダーチャートのデータ d1，d2 をプロットするものである．レーダーチャートの形に描画するために，x の範囲は 0 ° ～360 °（0～2 π ラジアン）とし，データ d1，d2 の末尾には先頭と同じ要素を加えてグラフの周を閉じる．

レーダーチャートの各データ点にラベル（lbl の要素）を表示するには 18 行目にあるように `set_xticklabels` メソッドを Axes オブジェクトに対して実行する．

書き方：　set_xticklabels(対応するラベルのデータ列)

参考）これと同等の方法として，`set_thetagrids` メソッドを Axes オブジェクトに対して実行する方法もある．

書き方：　set_thetagrids(角度のデータ配列, 対応するラベルのデータ列)

極座標グラフの外周の枠線を非表示にするには Axes オブジェクトの spines['polar'] プロパティに対して set_visible メソッドで非表示の設定（20 行目）をする．

このプログラムを実行すると図 53 のようなレーダーチャートが表示される．

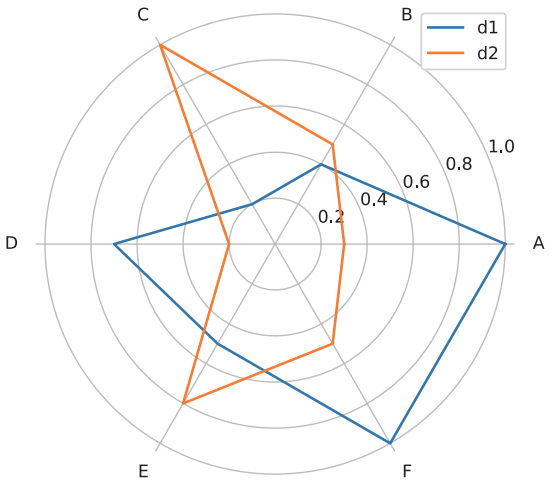


図 53: レーダーチャート

3.1.13.13　ステムプロット

離散的な数値データの系列を視覚化するための手法として**ステムプロット**がある．プロットの形式は，横軸に対する縦軸の値をマーカーで図示するものであり，plot 関数によるマーカープロットと似ているが，基準線からの値の差をステム（茎：stem）で図示するのが特徴である．

離散化された正弦関数をステムプロットで図示するプログラム nplot06-1.py を示す.

プログラム：nplot06-1.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  #=== データ作成 ===
5  x = np.linspace(-2*np.pi,2*np.pi,30)
6  y = np.sin(x)
7
8  #=== ステムプロット ===
9  fig, ax = plt.subplots(figsize=(6,3))
10 ax.stem( x, y, bottom=0 )   # ステムプロット
11 ax.set_xlabel('x')
12 ax.set_ylabel('y')
13 ax.set_title('Stem plot of y = sin(x)')
14 ax.grid()
15 plt.tight_layout()
16 plt.show()
```

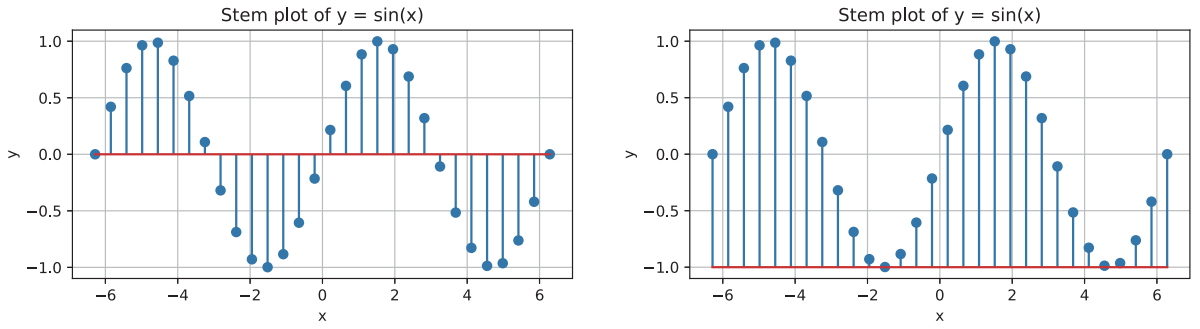
このプログラムの「ax.stem(x, y, bottom=0)」がステムプロットを描画する部分である.

書き方：　Axes オブジェクト.stem(横軸データ, 縦軸データ, bottom=基準値)

「横軸データ」に対する「縦軸データ」をプロットする.「基準値」で指定した高さの位置に水平の**基準線**が描かれる.「基準値」のデフォルトは 0 である.

このメソッドの戻り値は**マーカー**（Line2D オブジェクト），**ステム**（LineCollection オブジェクト），**基準線**（Line2D オブジェクト）の 3 要素から成る StemContainer オブジェクトである.（後で解説する）

このプログラムを実行すると，図 54 の (a) のようなグラフが表示される.



(a) bottom=0　　(b) bottom=-1

図 54: ステムプロット

基準線が赤い水平線として描かれていることがわかる．ステムプロットの部分を「ax.stem(x, y, bottom=-1)」と変更して（基準値を -1 にして）実行すると，図 54 の (b) のようなグラフが表示される.

■ 横向きのステムプロット

stem メソッドにオプションの引数「orientation='horizontal'」を与えると，横向きのステムプロットとなる.次に示すプログラム nplot06-3.py は先の nplot06-1.py と本質的には同じであるが，横向きのステムプロットを作成するものである.

プログラム：nplot06-3.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  #=== データ作成 ===
5  x = np.linspace(-2*np.pi,2*np.pi,30)
6  y = np.sin(x)
7
8  #=== ステムプロット ===
9  fig, ax = plt.subplots(figsize=(6,3))
10 ax.stem( x, y, bottom=0, orientation='horizontal' ) # ステムプロット
11 ax.set_xlabel('y')       # ラベル名の縦横の
12 ax.set_ylabel('x')       # 解釈に注意すること
13 ax.set_title('Stem plot of y = sin(x)')
```

```
ax.grid()
plt.tight_layout()
plt.show()
```

このプログラムを実行すると，図 55 のようなグラフが表示される．

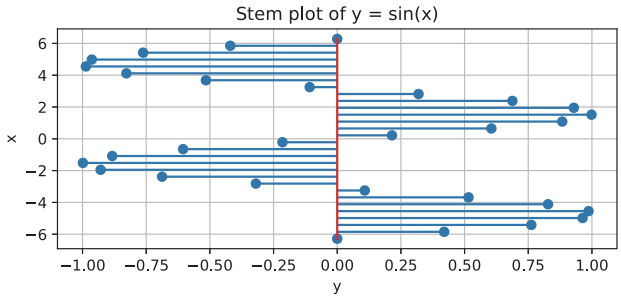


図 55: 横向きのステムプロット

■ ステムプロットの構造

ステムプロットは図 56 に示すように，**マーカー**，**ステム**，**基準線**の 3 つの部分から成る．

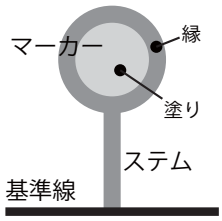


図 56: ステムプロットの構造

stem メソッドは 3 つの要素（マーカー，ステム，基準線）から成る StemContainer オブジェクトを返す．その要素に対して表 30 に示すようなメソッドを実行することで各種の設定ができる．

表 30: マーカー，ステム，基準線に対するメソッド（一部）

マーカーの設定	
メソッド	解　　説
`set_marker`(種類)	マーカーの形状を「種類」に設定する
`set_markersize`(サイズ)	マーカーの大きさを「サイズ」に設定する．(単位：pt)
`set_markeredgewidth`(太さ)	マーカーの縁に「太さ」を設定する．(単位：pt)
`set_markeredgecolor`(色)	マーカーの縁に「色」を設定する．
`set_markerfacecolor`(色)	マーカー内部の塗りに「色」を設定する．
ステム，基準線の設定	
メソッド	解　　説
`set_color`(色)	「色」を設定する．
`set_linewidth`(太さ)	「太さ」を設定する．(単位：pt)
`set_linestyle`(線種)	「線種」を設定する．

※ 色，線種については p.87 の表 26 を参照のこと．

マーカー，ステム，基準線に設定を施す例をプログラム nplot06-2.py で示す．

プログラム：nplot06-2.py

```
import numpy as np
import matplotlib.pyplot as plt

#=== データ作成 ===
x = np.linspace(-2*np.pi,2*np.pi,17)
y = np.sin(x)

#=== ステムプロット ====
```

```
fig, ax = plt.subplots(figsize=(6,3))
mk,st,bs = ax.stem( x, y, bottom=0 )    # ステムプロット

# マーカーの設定
mk.set_marker('o')                    # マーカーの種類
mk.set_markersize(12)                 # マーカーサイズ
mk.set_markeredgewidth(2.0)           # マーカーの縁の太さ
mk.set_markeredgecolor('blue')        # マーカーの縁の色
mk.set_markerfacecolor('yellow')      # マーカー内部の色

# ステム（茎）の設定
st.set_linewidth(2.0)                 # ステムの太さ
st.set_linestyle('--')                # ステムの線種
st.set_color('blue')                  # ステムの色

# 基準線の設定
bs.set_linewidth(3.0)                 # 基準線の太さ
bs.set_color('magenta')               # 基準線の色

# ラベルとタイトル
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('Stem plot of y = sin(x)')
ax.grid()
plt.tight_layout()
plt.show()
```

これを実行すると図 57 のようなグラフが表示される．

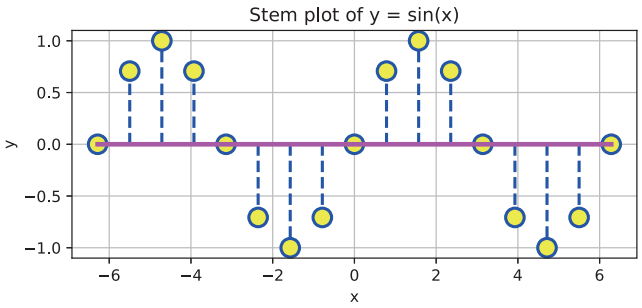


図 57: マーカー，ステム，基準線の設定

3.1.13.14　日本語の見出し・ラベルの表示

グラフ中の見出しやラベルに日本語フォントを使用するには matplotlib の FontProperties クラスを利用する．このクラスは matplotlib.font_manager パッケージにある．

例. 日本語フォントの読み込み（Windows 環境）

```
from matplotlib.font_manager import FontProperties
fp = FontProperties(fname=r'C:¥WINDOWS¥Fonts¥msgothic.ttc', size=11)
```

これは Windows の環境で標準的に利用できる「MS ゴシック（標準）」を読み込んで，11 ポイントのサイズのフォントとして FontProperties クラスの fp オブジェクトを生成している例である．先の例のプログラム nplot01.py を変更してタイトル，軸ラベル，凡例を日本語で表示する形にしたプログラム nplot01j.py を示す．

プログラム：nplot01j.py

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.font_manager import FontProperties

# グラフで使用する日本語フォント
fp = FontProperties(fname=r'C:\WINDOWS\Fonts\msgothic.ttc', size=11)

# データ列の生成
lx = np.arange(-3.15,3.15,0.01) # 定義域の生成
ly1 = np.sin(lx)                # 正弦関数の列
ly2 = np.cos(lx)                # 余弦関数の列

```

```
13 # データ列のプロット
14 plt.figure( figsize=(6,3) )          # 作図処理の開始（省略可）
15 plt.plot(lx,ly1, label='正弦関数')  # プロット(1)
16 plt.plot(lx,ly2, label='余弦関数')  # プロット(2)
17 plt.xlabel('定義域',fontproperties=fp)       # 横軸ラベル
18 plt.ylabel('値域',fontproperties=fp)         # 縦軸ラベル
19 plt.legend(prop=fp)                          # 凡例の表示
20 # タイトルの表示
21 plt.title('正弦関数，余弦関数のプロット',fontproperties=fp)
22 plt.grid(True)
23 # 出力
24 plt.savefig('nplot01_out.png')       # 画像ファイル出力
25 plt.show()      # プロットの表示
```

3 行目で FontProperties クラスを読み込み，6 行目でフォントを読み込んで fp に与えている．17，18 行目にあるように，軸ラベル設定時にキーワード引数 fontproperties=fp を与えると指定したフォントが有効になる．凡例にフォントを設定する場合は 19 行目にあるように legend メソッドのキーワード引数に prop=fp と指定する．

このプログラム実行して作成したグラフの例を図 58 に示す．

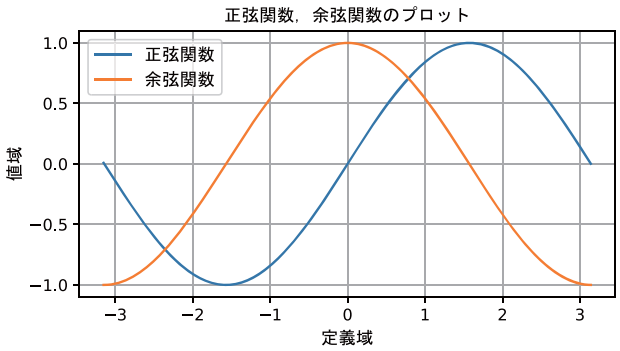


図 58: 見出しとラベルを日本語にした例

実際に使用できるフォントファイルの一覧を取得する方法を後の「使用できるフォントファイル」（p.105）で解説する．また，より自然な形でフォント指定を行う方法を後の「使用できるフォントファミリ」（p.105）と「japanize-matplotlib」（p.107）で解説する．

■ 使用できるフォントファイル

matplotlib の font_manager モジュールを使用すると，当該環境で使用可能な**フォントファイル**の一覧を取得することができる．

例. 使用できるフォントファイルの一覧表示

```
>>> from matplotlib import font_manager  Enter    ←必要なモジュールの読込み
>>> ffiles = font_manager.findSystemFonts()  Enter    ←使用可能なフォントファイル名の取得
>>> for ffile in sorted(ffiles):  Enter    ← 一覧表示ループ
...     print(ffile)  Enter
...  Enter    ←ループの記述の終了
C:\Users\katsu\AppData\Local\Microsoft\Windows\Fonts\NotoSansJP-Black.otf
C:\Users\katsu\AppData\Local\Microsoft\Windows\Fonts\NotoSansJP-Bold.otf
          ⋮
      （途中省略）
          ⋮
C:\Windows\Fonts\yumindb.ttf
C:\Windows\Fonts\yuminl.ttf
```

このように，font_manager の findSystemFonts メソッドによって，フォントファイルの一覧が取得できる．

■ 使用できるフォントファミリ

実際にフォントを使用する場合，**フォントファミリ名**を指定し，更に**フォントサイズ**，**フォントスタイル**，**フォントの重み**を指定する形式を取る方が．直接的にフォントファイルを指定するよりも自然である．

matplotlib のフォント関連の設定は rcParams が保持している．このオブジェクトは一種の辞書であり，フォント関連の主なキーを表 31 に示す．

表 31: フォント関連の rcParams の主なキー

キー	解　説	記 述 例
'font.family'	フォントファミリ名 （優先順位リストで指定可）	'Arial', 'Yu Gothic', ['Yu Gothic','Meiryo','sans-serif']
'font.size'	基準フォントサイズ（単位:pt）	12, 14
'font.style'	フォントスタイル	'normal', 'italic', 'oblique'
'font.weight'	フォントの重み（太さ） （キーワードまたは数値）	'normal', 'bold', 'light' など, もしくは整数値（100〜900）

例. rcParams のフォント関連情報の確認

```
>>> import matplotlib.pyplot as plt  Enter
>>> plt.rcParams['font.family']  Enter    ←フォントファミリ
['sans-serif']
>>> plt.rcParams['font.size']  Enter    ←フォントサイズ
10.0
>>> plt.rcParams['font.style']  Enter    ←フォントスタイル
'normal'
>>> plt.rcParams['font.weight']  Enter    ←フォントの重み
'normal'
```

rcParams の各キーに対して値を代入することでフォント関連の設定を変更することができる. 次のプログラム nplot03j.py は,「HG 創英角ポップ体」フォント（フォントファミリ名は HGSoeiKakupoptai）が使える環境で, プロットのタイトルやラベルにそれを適用する例である.

プログラム：nplot03j.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # フォントファミリの指定
5  plt.rcParams['font.family'] = 'HGSoeiKakupoptai'    # HG創英角ポップ体
6
7  # データ列の生成
8  lx = np.arange(-3.15,3.15,0.01)      # 定義域の生成
9  ly1 = np.sin(lx)                     # 正弦関数の列
10 ly2 = np.cos(lx)                     # 余弦関数の列
11
12 # データ列のプロット
13 plt.figure( figsize=(6,3) )          # 作図処理の開始（省略可）
14 plt.plot(lx,ly1, label='正弦関数')   # プロット(1)
15 plt.plot(lx,ly2, label='余弦関数')   # プロット(2)
16 plt.xlabel('定義域')                 # 横軸ラベル
17 plt.ylabel('値域')                   # 縦軸ラベル
18 plt.legend()                         # 凡例の表示
19 plt.title('正弦関数, 余弦関数のプロット')
20 plt.grid(True)
21 plt.tight_layout()
22 plt.show()       # プロットの表示
```

このプログラムを実行すると, 図 59 のようなグラフが表示される.

当該環境で使用できるフォントファミリ名を調べる方法を次に示す.

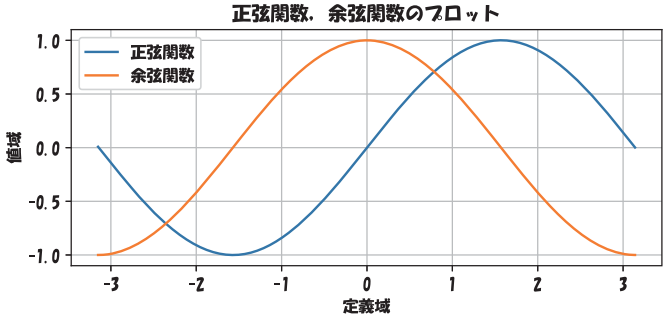


図 59: 日本語フォントを「HG 創英角ポップ体」にした例

例. 使用できるフォントファミリ名の一覧表示

```
>>> from matplotlib import font_manager  Enter    ←必要なモジュールの読込み
>>> fnames = sorted({f.name for f in font_manager.fontManager.ttflist})  Enter
>>> for fname in fnames:  Enter    ← 一覧表示ループ
...     print(fname)  Enter
...  Enter    ←ループの記述の終了
Adobe Arabic          ←フォントファミリの一覧が表示される
Adobe Caslon Pro
Adobe Devanagari
  :
（途中省略）
  :
Yu Gothic
Yu Mincho
ZWAdobeF
```

`font_manager.fontManager.ttflist` には，システムで使用可能なフォントの情報である FontEntry オブジェクトのリストが保持されている．各 FontEntry オブジェクトの name 属性の値が，そのフォントのフォントファミリ名を意味しており，上の例では，それを一覧表示している．

■ japanize-matplotlib

japanize-matplotlib は，手軽に日本語フォントを使用するためのサードパーティ製ライブラリである．これを使用するには pip などのライブラリ管理ツールを用いて予めインストールしておく必要がある．

例. `py -m pip install japanize-matplotlib`　　（Windows 環境の PSF Python の場合）

このライブラリを用いて日本語フォントを使用する例を nplot02j.py に示す．

プログラム：nplot02j.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import japanize_matplotlib          # matplotlib日本語化モジュール
4
5  # データ列の生成
6  lx = np.arange(-3.15,3.15,0.01)     # 定義域の生成
7  ly1 = np.sin(lx)                    # 正弦関数の列
8  ly2 = np.cos(lx)                    # 余弦関数の列
9
10 # データ列のプロット
11 plt.figure( figsize=(6,3) )         # 作図処理の開始（省略可）
12 plt.plot(lx,ly1, label='正弦関数')  # プロット(1)
13 plt.plot(lx,ly2, label='余弦関数')  # プロット(2)
14 plt.xlabel('定義域')                # 横軸ラベル
15 plt.ylabel('値域')                  # 縦軸ラベル
16 plt.legend()                       # 凡例の表示
17 plt.title('正弦関数，余弦関数のプロット')
18 plt.grid(True)
19 plt.show()       # プロットの表示
```

これは先の nplot01j.py と同等の処理を行うもので，より簡潔な記述になっていることがわかる．（ライブラリをインポートする際の名前を `japanize_matplotlib` としていることに注意）

■ 日本語フォントを使用する際の注意事項

日本語フォントを使用すると，負号「-」が正しく表示されない場合がある．そのような場合は，フォントファミリの指定をする際に

```
plt.rcParams['axes.unicode_minus'] = False
```

という記述を追加すると解決することがある．

3.1.13.15　グラフを画像ファイルとして保存する方法

先のプログラム nplot01j.py の 25 行目の記述は，savefig メソッドを用いて作成した**グラフを画像ファイルとして保存する**[46] ためのものである．これは show メソッドに先立って記述しなければならない．

savefig メソッドの第 1 引数には，保存先のファイル名を与える．ファイルの形式（フォーマット）は，与えたファイル名の拡張子によって識別されるが，次のようなものが拡張子として指定できる．

```
png，svg，pdf，ps，eps
```

png のようなビットマップ画像を作成する場合は，キーワード引数 dpi=**解像度**を与えることができる．

■ 出力可能な画像フォーマットの調査

当該環境の Figure オブジェクトの canvas 属性（FigureCanvasQTAgg オブジェクト）に対して get_supported_filetypes メソッドを実行すると，保存可能な画像フォーマットの情報が辞書の形で得られる．

例. 出力可能な画像フォーマットを調べる

```
>>> import matplotlib.pyplot as plt  Enter    ←モジュールの読込み
>>> fmts = plt.gcf().canvas.get_supported_filetypes()  Enter    ←画像フォーマット情報の取得
>>> for fmt in sorted(fmts.keys()):  Enter    ← 一覧出力ループ
...       print( f'{fmt:6s}:  {fmts[fmt]}' )  Enter
...   Enter    ←ループの記述の終了
eps    :  Encapsulated Postscript          ←使用できる画像フォーマットの一覧表示
jpeg   :  Joint Photographic Experts Group
          :
        （途中省略）
          :
tiff   :  Tagged Image File Format
webp   :  WebP Image Format
```

これは，使用可能なフォーマットとその説明を一覧表示するものである．

[46]**注**）日本語の文字を含んだ画像ファイルの出力において，'RuntimeError: TrueType font is missing table' というエラーが発生することがある．これは pdf,ps,eps のようなベクトルグラフィック系の出力において発生することが多い．その場合は svg で出力した後で，別のソフトウェアを介して目的の形式に変換するなどの方法を取るのが良い．

3.1.14 乱数 (1)

NumPy ライブラリの random モジュールは，乱数生成に関する機能を提供する．ここではまず，NumPy の 1.17 版まで推奨され，よく普及した形の API について解説する[47]．ただしこの「3.1.14 乱数 (1)」で解説する内容は後方互換性のためのものであり，新規のプログラミングには後の「3.1.15 乱数 (2)」（p.112）で解説する内容に従うこと．

3.1.14.1 一様乱数の生成

rand 関数は 0 以上 1 未満の乱数を生成する．

書き方：

1) np.random.rand()　　乱数を 1 つ生成する．
2) np.random.rand(**個数**)　　指定した個数の乱数を配列として生成する．
3) np.random.rand(n,m)　　n 行 m 列の乱数の配列を生成する．

例. 一様乱数の生成

```
>>> np.random.rand()  Enter      ←乱数を 1 つ生成
0.9002721968823484            ←生成結果（実行の度に異なる）
>>> np.random.rand(5)  Enter     ←乱数を 5 個生成
array([ 0.32644595, 0.20630809, 0.98017323, 0.09793674, 0.39467418]) ←生成結果（実行の度に異なる）
>>> np.random.rand(5,3)  Enter     ← 5 行 3 列の乱数配列を生成
array([[ 0.41218582, 0.36665746, 0.14565054],     ←生成結果（実行の度に異なる）
       [ 0.2018859 , 0.88586831, 0.88083754],
       [ 0.73630688, 0.78485615, 0.98865664],
       [ 0.58109305, 0.75149191, 0.26337745],
       [ 0.67083326, 0.91048167, 0.9451317 ]])
```

3.1.14.2 整数の乱数の生成

書き方： np.random.randint(L, H, **個数**)

整数 L 以上 H 未満の範囲で乱数を，指定した個数生成する．

例. 整数の乱数

```
>>> np.random.randint(0,10,20)  Enter     ← 0 以上 10 未満の乱数を 20 個生成
array([2, 1, 7, 4, 0, 1, 4, 3, 9, 2, 7, 6, 0, 8, 1, 4, 8, 9, 1, 7], dtype=int32) ←生成結果
```

randint の引数の「個数」の所に (n, m) というタプルを与えると n 行 m 列の配列として乱数を生成する．

3.1.14.3 正規乱数の生成

normal 関数は正規乱数（正規分布となる乱数）を生成する．

書き方： **np.random.normal(平均, 標準偏差, 生成個数)**

生成個数に (n, m) のタプルを与えると n 行 m 列の配列として生成する．また，キーワード引数の形式で値を与えることもでき，引数の省略も可能である．

書き方： **np.random.normal(loc=平均, scale=標準偏差, size=生成個数)**

「loc=平均」,「scale=標準偏差」は省略することができ，デフォルト値はそれぞれ 0，1 である．

例. 正規乱数：1 次元配列として取得

```
>>> np.random.normal(0,1,10)  Enter     ←平均 0, 標準偏差 1 の正規分布データを 10 個生成
array([ 0.01706595, -0.44630863, 0.64802007, -0.86528213, 0.11454459,     ←成績結果
        1.55471322, 0.24815903, 1.16232311, 0.16387414, -0.52767615])
```

例. 正規乱数：2 次元配列として取得

```
>>> np.random.normal(0,1,(5,3))  Enter     ← 5 行 3 列の正規分布データを生成
array([[ 0.79680422, -1.20956605, 0.58653553],
       [-0.49538379, -0.70013524, -1.68294362],
       [-2.03277246, 0.7823404 , -1.4837754 ],
       [ 1.46255008, 0.4963593 , -0.01242249],
       [ 0.49222816, -0.03615764, 1.31829024]])
```

[47] ここで解説する旧 API も多くの実用システムや教材で採用されており，基本的な使用方法を学んでおく必要がある．旧 API による乱数生成は，その方法が簡潔であることから，本書においても，乱数生成を使用する多くの部分で旧 API による方法を採用している．

参考）以上の乱数生成においては，**メルセンヌ・ツイスタ**（MT19937）と呼ばれるアルゴリズムが使用される．

3.1.14.4　乱数の seed について

乱数生成用の関数は実行する度に異なる数値を生成する．（次の例参照）

例. 5 つの正規乱数を 3 回発生させる

```
>>> for i in range(3):  Enter      ←繰り返しの開始
...       print( np.random.normal(0,1,5) )  Enter      ← 5 つの正規乱数を生成
...   Enter      ←繰り返しの終了
[-0.97727788 0.95008842 -0.15135721 -0.10321885 0.4105985 ]      ← 5 つの正規乱数（以下同様）
[0.14404357 1.45427351 0.76103773 0.12167502 0.44386323]
[ 0.33367433 1.49407907 -0.20515826 0.3130677 -0.85409574]
```

乱数生成を for 文で繰り返しているが，毎回異なる乱数列が得られていることがわかる．しかし，random.seed 関数によって乱数生成の状態を初期化することが可能であり，生成する乱数に再現性を持たせることができる．（次の例参照）

例. 再現性のある乱数生成

```
>>> for i in range(3):  Enter      ←繰り返しの開始
...       np.random.seed(0)   Enter      ← seed（種）を 0 として乱数生成状態を初期化
...       print( np.random.normal(0,1,5) )  Enter      ← 5 つの正規乱数を生成
...   Enter      ←繰り返しの終了
[1.76405235 0.40015721 0.97873798 2.2408932 1.86755799]      ← 5 つの正規乱数
[1.76405235 0.40015721 0.97873798 2.2408932 1.86755799]      ← 5 つの正規乱数（同じ乱数列）
[1.76405235 0.40015721 0.97873798 2.2408932 1.86755799]      ← 5 つの正規乱数（同じ乱数列）
```

NumPy が提供する乱数生成関数は，確定的なアルゴリズムを用いている．すなわち，複数の乱数を生成する際，設定された seed（種）に応じて，決められた並びの乱数列を生成する．関数 random.seed は，引数に与えられた種（整数値）[48] に応じて，乱数生成過程の初期状態を設定する．また，この関数を呼び出すことなく乱数を生成する場合は，seed は自動的に設定される．

このように乱数生成過程に再現性を持たせることは，統計処理や機械学習のためのプログラム開発において，確定的なテストデータを作成するのに必要となる．

参考までに，他の乱数生成関数についても同様の実行例を示す．

例. rand による確定的な乱数列の生成

```
>>> for i in range(3):  Enter      ←繰り返しの開始
...       np.random.seed(0)   Enter      ← seed（種）を 0 として乱数生成状態を初期化
...       print( np.random.rand(5) )  Enter      ← 5 つの一様乱数を生成
...   Enter      ←繰り返しの終了
[0.5488135 0.71518937 0.60276338 0.54488318 0.4236548 ]  Enter      ← 5 つの一様乱数
[0.5488135 0.71518937 0.60276338 0.54488318 0.4236548 ]  Enter      ←（同じ乱数列）
[0.5488135 0.71518937 0.60276338 0.54488318 0.4236548 ]  Enter      ←（同じ乱数列）
```

例. randint による確定的な乱数列の生成

```
>>> for i in range(3):  Enter      ←繰り返しの開始
...       np.random.seed(0)   Enter      ← seed（種）を 0 として乱数生成状態を初期化
...       print( np.random.randint(0,100,10) )  Enter      ← 0 以上 100 未満の整数乱数を生成
...   Enter      ←繰り返しの終了
[44 47 64 67 67 9 83 21 36 87]      ← 0 以上 100 未満の整数乱数
[44 47 64 67 67 9 83 21 36 87]      ← 0 以上 100 未満の整数乱数（同じ乱数列）
[44 47 64 67 67 9 83 21 36 87]      ← 0 以上 100 未満の整数乱数（同じ乱数列）
```

3.1.14.5　RandomState オブジェクト

個別に初期条件を指定して乱数を生成する RandomState オブジェクトについて，実行例を示しながら説明する．

[48] $0 \sim 2^{32} - 1$ の値．

例. RandomState オブジェクトを用いた乱数生成

```
>>> for i in range(3):  Enter      ←繰り返しの開始
...       rs = np.random.RandomState(0)   Enter      ← seed（種）を 0 とする乱数発生器 rs を作成
...       print( rs.randint(0,100,10) )   Enter      ← 0 以上 100 未満の整数乱数を生成
...   Enter      ←繰り返しの終了
[44 47 64 67 67 9 83 21 36 87]      ← 0 以上 100 未満の整数乱数
[44 47 64 67 67 9 83 21 36 87]      ← 0 以上 100 未満の整数乱数（同じ乱数列）
[44 47 64 67 67 9 83 21 36 87]      ← 0 以上 100 未満の整数乱数（同じ乱数列）
```

RandomState オブジェクトは乱数発生器と見ることができ，これに対して乱数生成用のメソッドを実行する．この例では seed を 0 とする RandomState オブジェクト rs を for による繰り返しの度に生成し，それに対して randint メソッドを実行している．結果として先に示した例と同じ乱数列を得ている．

RandomState オブジェクトの作成によって，異なる初期条件を持つ複数の乱数発生器を実現できる．

3.1.14.6　三角分布に沿った乱数の生成 (1)

三角分布は，現実の現象から自然に導かれる分布ではなく，仮定を明示するために用いられる便宜的なモデルである．

書き方：　triangular(left=左端, mode=最頻値, right=右端, size=データ個数)

定義域が「左端」から「右端」までの「最頻値」を持つ三角分布（図 60）に従う乱数を，指定した「データ個数」の配列として生成する．

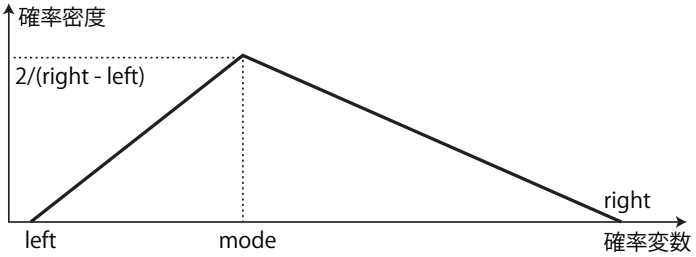


図 60: 三角分布

例. 三角乱数の生成

```
>>> import numpy as np   Enter      ←ライブラリの読み込み
>>> np.random.seed(0)   Enter      ←種の設定
>>> r = np.random.triangular( left=0, mode=0.5, right=1, size=5 )  Enter   ←乱数を 5 個生成
>>> r   Enter      ←内容確認
array([0.52503342, 0.62263371, 0.55433386, 0.52296917, 0.46024711])   ←得られた乱数の配列
```

triangular 関数の引数は，位置引数（順序は先に示した通り）として与えても良い．（次の例）

例. 三角乱数の生成（先の例の続き）

```
>>> np.random.seed(0)   Enter
>>> r = np.random.triangular( 0, 0.5, 1, 5 )  Enter
>>> r   Enter      ←内容確認
array([0.52503342, 0.62263371, 0.55433386, 0.52296917, 0.46024711])
```

先の例と同じ乱数が得られている．

triangular 関数で 10^4 個の乱数を生成して matplotlib の hist 関数でヒストグラムを作成する例を次に示す．

例. 10^4 個の乱数（先の例の続き）

```
>>> np.random.seed(0)   Enter      ←種の設定
>>> r = np.random.triangular(0,0.5,1,10000)   Enter      ←乱数を 10^4 個生成
>>> import matplotlib.pyplot as plt   Enter      ←作図の開始
>>> f = plt.figure( figsize=(7,3.5) )   Enter
>>> g = plt.hist(r,bins=15)   Enter
>>> t1 = plt.xlabel('Random variable')   Enter
>>> t2 = plt.ylabel('Frequency')   Enter
>>> t3 = plt.title('Triangular distribution')   Enter
>>> plt.show()   Enter      ←図の表示
```

この実行結果として図 61 のようなヒストグラムが表示される．

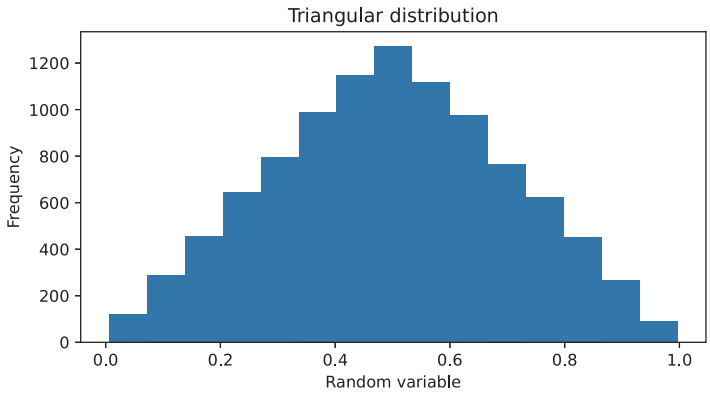


図 61: triangular 関数で生成された乱数の度数分布

3.1.15　乱数 (2)

ここでは，NumPy の新しい API（Generator API）[49] に基づく乱数生成について解説する．Generator API では**乱数発生器**（RNG：random number generator）として表 32 に挙げるものが使用できる．

表 32: Generator API で利用できる RNG

RNG	特　徴	長　所	短　所
PCG64	64 ビットワードの現代的 RNG	・高速かつ高品質 ・並列ストリームの生成が容易	・歴史的な実績は MT ほど多くない ・他環境との互換性が低い
MT19937	古典的で普及度の高い RNG（Mersenne Twister）	・歴史が長く広く普及 ・周期が非常に長い（$2^{19937}-1$） ・十分な実用性	・32 ビットワードで効率が劣る ・並列用途に不向き ・下位ビットに偏りがある
Philox	カウンターベースの RNG	・GPU や並列計算に適する ・独立ストリームを簡単に生成可能 ・統計的品質は良好	・CPU 単独では PCG64 より遅い ・導入事例は少ない
SFC64	軽量・高速な RNG	・非常に高速で実装が簡潔 ・小規模シミュレーションに有効	・導入事例は少ない ・統計的検証の歴史が浅い ・高精度計算には向かない

※ PCG64 が Generator API のデフォルト．改良版の PCG64DXSM も使うことができる．

3.1.15.1　RNG の作成

Generator API による乱数生成を利用するには，まず**乱数生成器**（RNG）を作成する．デフォルトの RNG を作成するには `default_rng` を使用する．

書き方：　np.random.default_rng(種)

「種」を指定して RNG を作成して返す．「種」は int 型（整数）の任意の値を与えることが推奨される[50]．この方法では，デフォルトの PCG64 が採用される．

例. RNG の作成

```
>>> import numpy as np   Enter      ← NumPy の読込み
>>> rng = np.random.default_rng(1)   Enter      ←種「1」を与えて RNG を作成
```

例. 確認（先の例の続き）

```
>>> rng   Enter      ←確認
Generator(PCG64) at 0x245DEC6C040      ← RNG（Generator オブジェクト）
```

この例ではデフォルトの RNG（PCG64）を作成している．これ以外の RNG を作成するには，必要な API をインポートした後，Generator コンストラクタの引数に，表 32 に示した RNG のクラスのオブジェクトを渡す．

書き方：　Generator(RNG クラスのインスタンス)

乱数生成のための RNG（Generator オブジェクト）を作成して返す．「RNG クラスのインスタンス」は，当該クラスのコンストラクタに種を渡して作成する．

[49] NumPy 1.17 から導入され，2.0 以降の推奨 API となっている．

[50] 整数以外にも配列をはじめとするオブジェクトを種として指定することができるが，これらは主に並列計算や高度な用途向けであるため本書では扱わない．

例. 種類を指定した RNG の作成（先の例の続き）

```
>>> from numpy.random import Generator, PCG64, MT19937, Philox, SFC64 Enter   ← API の読込み
>>> rng_mt = Generator(MT19937(1))  Enter      ← MT19937 の RNG
>>> rng_phil = Generator(Philox(1))  Enter      ← Philox の RNG
>>> rng_sfc = Generator(SFC64(1))  Enter       ← SFC64 の RNG
```

この例は，種「1」を与えて各種の RNG を作成するものである．

参考）先に作成したデフォルトの RNG を　`Generator(PCG64(1))`　として作成することもできる．

■ BitGenerator

先に示した「RNG クラスのインスタンス」である `PCG64(種)`，`MT19937(種)`，`Philox(種)`，`SFC64(種)`（Generator コンストラクタの引数に与えるもの）は，より正確には BitGenerator と呼ばれるものである．BitGenerator は低レベルでビット列としての乱数を生成する役割を持つ．一方，Generator は BitGenerator を内部に保持し，それを用いて random，normal，integers などの高レベルな乱数分布を提供する．

このように NumPy の乱数生成は「BitGenerator（乱数エンジン）」と「Generator（利用者が使う窓口）」の二層構造になっている．なお，`default_rng(種)` は内部で Generator(PCG64(種)) を生成して返す糖衣構文である．将来のバージョンにおける既定アルゴリズムの変更の影響を避けたい場合や，再現性を厳密に保証したい場合は，Generator(PCG64(種)) のように BitGenerator を明示して用いることが推奨される．

3.1.15.2　一様乱数の生成

RNG に対して random メソッドを実行することで一様乱数（0 以上 1.0 未満）を生成することができる．

書き方：　Generator オブジェクト.random(個数)

指定した「個数」の乱数を生成して，1 次元配列の形で返す．引数を省略すると 1 つの乱数（スカラー）を返す．引数にタプルを渡すと，タプルの要素で各次元のサイズを指定した多次元配列の形で乱数の配列を返す．

例. 乱数を 1 つ生成（先の例の続き）

```
>>> rng.random()  Enter      ← PCG64 で 乱数を 1 つ生成
0.5118216247002567
>>> rng_phil.random()  Enter      ← Philox で 乱数を 1 つ生成
0.0668020093396563
```

例. 乱数 3 つの 1 次元配列を生成（先の例の続き）

```
>>> rng.random( 3 )  Enter      ← PCG64 で 乱数 3 つの 1 次元配列を生成
array([0.9504637 , 0.14415961, 0.94864945])
>>> rng_phil.random( 3 )  Enter      ← Philox で 乱数 3 つの 1 次元配列を生成
array([0.07901298, 0.00577712, 0.80655669])
```

例. 2 行 3 列の乱数配列を生成（先の例の続き）

```
>>> rng.random( (2,3) )  Enter      ← PCG64 で生成
array([[0.31183145, 0.42332645, 0.82770259],
       [0.40919914, 0.54959369, 0.02755911]])
>>> rng_phil.random( (2,3) )  Enter      ← Philox で生成
array([[0.2451058 , 0.30063499, 0.35955818],
       [0.46409051, 0.55262308, 0.83253762]])
```

■ 整数乱数の生成

RNG で整数乱数を生成するには integers メソッドを使用する．

書き方：　Generator オブジェクト.integers(下限, 上限, 個数)

「下限」以上「上限」未満の乱数を指定した「個数」だけ生成して配列として返す．「個数」の与え方に関しては，先の random メソッドに準ずる．

例. 2 行 3 列の整数の乱数配列を生成（先の例の続き）

```
>>> rng.integers( 10,20,3 )   Enter      ← PCG64 で 乱数 3 つの 1 次元配列を生成
array([18, 17, 18])
>>> rng_phil.integers( 10,20,3 )   Enter      ← Philox で 乱数 3 つの 1 次元配列を生成
array([11, 17, 15])
```

これは 10 以上 20 未満の整数の乱数の配列を生成する例である.

3.1.15.3 正規乱数の生成

RNG に対して normal メソッドを実行することで正規乱数を生成することができる.

書き方： Generator オブジェクト.normal(平均, 標準偏差, 個数)

$N(\mu, \sigma^2)$ の μ に「平均」, σ に「標準偏差」を指定した形で正規乱数を指定した「個数」だけ生成して配列の形で返す.「個数」を省略すると 1 つの乱数（スカラー）を返す. 引数にタプルを渡すと, タプルの要素で各次元のサイズを指定した多次元配列の形で乱数の配列を返す.

キーワード引数の形式で値を与えることもでき, 引数の省略も可能である.

書き方： Generator オブジェクト.normal(loc=平均, scale=標準偏差, size=個数)

「loc=平均」,「scale=標準偏差」は省略することができ, デフォルト値はそれぞれ 0, 1 である.

次に示すプログラム pltNormRndHist01.py は, PCG64, Philox の RNG でそれぞれ 10,000 個の正規乱数（μ=0, σ=1）を生成してヒストグラムを作成するものである.

プログラム：pltNormRndHist01.py

```
1  import numpy as np
2  from numpy.random import Generator, Philox
3  import matplotlib.pyplot as plt
4  # RNG作成
5  rng = np.random.default_rng(1)       # RNG of default（PCG64）
6  rng_phil = Generator(Philox(1))      # RNG of Philox
7  # 乱数生成（正規乱数: μ=0, σ=1）
8  rn1 = rng.normal(size=10000)
9  rn2 = rng_phil.normal(size=10000)
10 # 作図
11 fig,ax = plt.subplots( 1,2, figsize=(10,3) )
12 ax[0].hist(rn1,bins=15)
13 ax[0].set_xlabel('value')
14 ax[0].set_ylabel('frequency')
15 ax[0].set_title('PCG64')
16 ax[1].hist(rn2,bins=15)
17 ax[1].set_xlabel('value')
18 ax[1].set_ylabel('frequency')
19 ax[1].set_title('Philox')
20 fig.suptitle('Histogram of 10,000 random numbers (Normal distribution)')
21 fig.subplots_adjust(wspace=0.25)
22 plt.tight_layout()
23 plt.show()
```

このプログラムを実行すると, 図 62 のようなグラフが表示される.

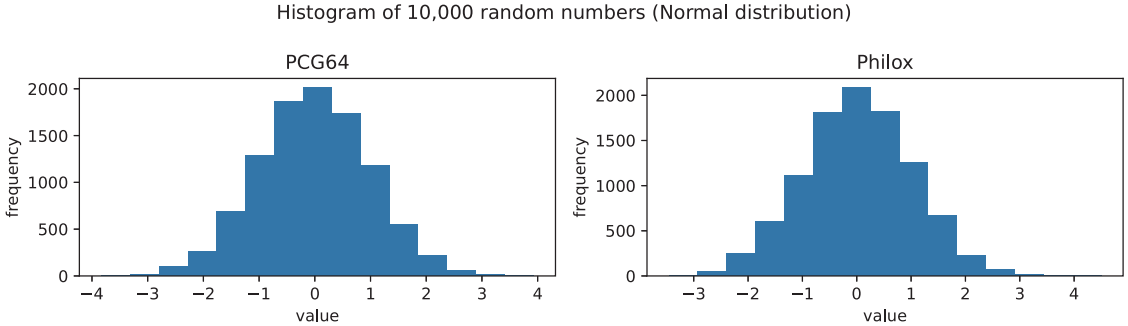


図 62: pltNormRndHist01.py の実行結果

3.1.15.4　三角分布に沿った乱数の生成 (2)

三角分布に従う乱数の生成について，先の「3.1.14.6 三角分布に沿った乱数の生成 (1)」（p.111）で解説したが，Generator オブジェクトで同様の乱数を生成することもできる．

書き方：　Generator オブジェクト.triangular(left=左端, mode=最頻値, right=右端, size=データ個数)

例. 三角分布に沿った乱数を 10^4 個作成してヒストグラムを作成する

```
>>> import numpy as np   Enter
>>> rng = np.random.default_rng(1)   Enter      ← RNG の作成
>>> r = rng.triangular(0,0.5,1,10000)   Enter      ←乱数の生成
>>> import matplotlib.pyplot as plt   Enter
>>> f = plt.figure( figsize=(7,3.5) )   Enter      ←作図の開始
>>> g = plt.hist(r,bins=15)   Enter
>>> t1 = plt.xlabel('Random variable')   Enter
>>> t2 = plt.ylabel('Frequency')   Enter
>>> t3 = plt.title('Triangular distribution')   Enter
>>> plt.show()   Enter      ←図の表示
```

この実行結果として図 61 のようなヒストグラムが表示される．

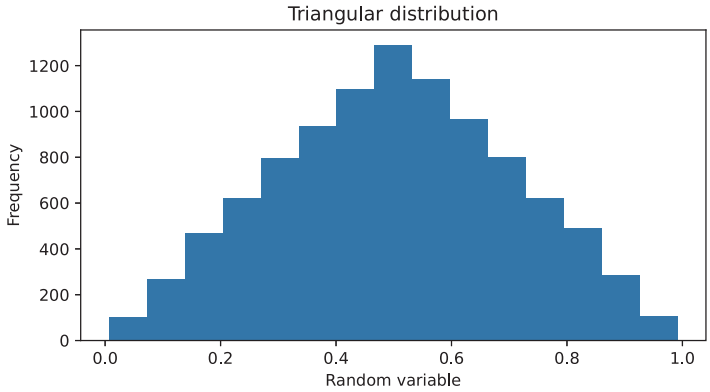


図 63: triangular メソッドで生成された乱数の度数分布

3.1.16　乱数の扱いにおける新旧 API の関係

「3.1.14 乱数 (1)」（p.109〜）,「3.1.15 乱数 (2)」（p.112〜）で解説したように，NumPy には乱数生成のための異なる API 群がある．これらは互いに実装が異なっており，同じアルゴリズム（MT19937）を採用して同じ種から乱数を生成しても，それぞれの API で異なる乱数系列が得られることに注意すること．

次に示すプログラム randomOldNew01.py は，旧 API の RandomState の状態を新 API の RNG の内部状態に移植して，新旧 API で生成される乱数列を比較するものである．random メソッドによる一様乱数の生成のみ，新旧 API で乱数系列が一致することが確認できるが，その他の乱数生成メソッドでは乱数系列が新旧 API で一致しないことが確認できる．(29 行目以降のコメントを切り替えて試されたい)

プログラム：randomOldNew01.py

```
1  import numpy as np
2  from numpy.random import Generator, MT19937
3
4  ############################################################
5  # 旧APIでの「種」とRandomStateの設定                       #
6  ############################################################
7  s = 1       # 旧APIにおける「種」
8  rs = np.random.RandomState(s)
9
10 ############################################################
11 # 乱数生成状態を新APIに移植                                #
12 ############################################################
13 st = rs.get_state()     # RandomStateの状態の取り出し
14 key = st[1].astype(np.uint32)   # key情報の取り出し
15 pos = int(st[2])                # pos情報の取り出し
```

```
16
17 bg = MT19937()       # BitGeneratorを作成（メルセンヌ・ツイスタ）
18 bg.state = {         # 旧APIの乱数生成状態を移植
19     'bit_generator': 'MT19937',
20     'state': {'key': key, 'pos': pos},
21     'has_uint32': 0,
22     'uint32': 0
23 }
24 MT = Generator(bg)       # Generatorを作成
25
26 ############################################################
27 # 新旧APIでの乱数生成の比較                                #
28 ############################################################
29 #print('*** 整数乱数（10以上20未満） ***')
30 #print('旧API:',rs.randint(10,20,4))
31 #print('新API:',MT.integers(10,20,4),'\n')
32
33 print('*** 一様乱数 ***')
34 print('旧API:',rs.random(4))
35 print('新API:',MT.random(4),'\n')
36
37 #print('*** 正規乱数（μ=0, σ=1） ***')
38 #print('旧API:',rs.normal(size=4))
39 #print('新API:',MT.normal(size=4))
```

このプログラムを実行例を次に示す.

実行例. 新旧 API で乱数系列が一致するケース

```
*** 一様乱数 ***
旧API: [4.17022005e-01 7.20324493e-01 1.14374817e-04 3.02332573e-01]
新API: [4.17022005e-01 7.20324493e-01 1.14374817e-04 3.02332573e-01]
```

3.1.17　統計に関する処理

統計処理に必要となる基本的な計算処理について解説する．まず，処理の例示に用いるためのサンプルデータを作成する．

例. 新 API による乱数生成

```
>>> import numpy as np  Enter
>>> rng = np.random.default_rng(1)  Enter
>>> r = rng.normal(4,2,100000)  Enter
```

参考. 旧 API による乱数生成

```
>>> import numpy as np  Enter
>>> np.random.seed(1)  Enter
>>> r = np.random.normal(4,2,100000) Enter
```

$\mu = 4, \sigma = 2$ の正規乱数を 10^5 個生成し，変数 r に保持している．以後，これ（新 API で生成したデータ）を使用して解説する．旧 API で生成したデータ（上の右の例）を用いる場合は，以降の計算結果に若干の差異が出るので注意すること．

3.1.17.1　合計

配列の要素の**合計**を求めるには sum メソッドを用いる．

例. 統計量の算出：合計

```
>>> r.sum()  Enter    ←合計を求める
np.float64(399081.88559142477)    ←計算結果
```

3.1.17.2　最大値，最小値

配列の要素の**最大値**，**最小値**を求めるには max，min メソッドをそれぞれ用いる．

例. 統計量の算出：最大値，最小値　（先の例の続き）

```
>>> r.max()  Enter    ←最大値を求める
np.float64(12.812707045045007)    ←計算結果
>>> r.min()  Enter    ←最小値を求める
np.float64(-4.060808497391202)    ←計算結果
```

3.1.17.3　平均，分散，標準偏差

配列の要素の**平均**，**分散**，**標準偏差**を求めるには mean，var，std メソッドをそれぞれ用いる．

例. 統計量の算出：平均，標本分散，標本標準偏差　（先の例の続き）

```
>>> r.mean()  Enter    ←平均値を求める
np.float64(3.990818855914248)    ←計算結果
>>> r.var()  Enter    ←標本分散を求める
np.float64(3.972313580616153)    ←計算結果
>>> r.std()  Enter    ←標本標準偏差を求める
np.float64(1.9930663763698773)    ←計算結果
```

この例のように，var, std メソッドに引数を与えずに，あるいはキーワード引数 'ddof=0' を与えて実行すると，**標本分散** σ^2，**標本標準偏差** σ をそれぞれ求める[51] ．

$$\text{標本分散：}\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 \qquad \text{不偏分散：}\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

不偏分散 $\hat{\sigma}^2$，**不偏標準偏差** $\hat{\sigma}$ を求めるにはこれらメソッドにキーワード引数 'ddof=1' を与えて実行する．

例. 不偏分散，不偏標準偏差　（先の例の続き）

```
>>> r.var(ddof=1)  Enter    ←不偏分散を求める
np.float64(3.9723533041491947)    ←計算結果
>>> r.std(ddof=1)  Enter    ←不偏標準偏差を求める
np.float64(1.9930763417764996)    ←計算結果
```

3.1.17.4　分位点，パーセント点

配列の要素の**分位点**を求めるには quantile 関数を用いる．

[51] **標本分散**，**標本標準偏差**という呼称は文献によって異なることがあるので，適宜読み替えていただきたい．

例. 分位点　（先の例の続き）

```
>>> np.quantile( r, 0 )  Enter      ←最小値を求める
np.float64(-4.060808497391202)      ←計算結果
>>> np.quantile( r, 0.25 )  Enter      ←四分位数 25%の点の値を求める
np.float64(2.647739948553598)      ←計算結果
>>> np.quantile( r, 0.5 )  Enter      ←四分位数 50%の点の値（中央値）を求める
np.float64(3.9890966725127353)      ←計算結果
>>> np.quantile( r, 0.75 )  Enter      ←四分位数 75%の点の値を求める
np.float64(5.326570079713106)      ←計算結果
>>> np.quantile( r, 1 )  Enter      ←最大値を求める
np.float64(12.812707045045007)      ←計算結果
```

この例のように，quantile 関数の第一引数に対象の配列を，第 2 引数にはデータ数の累積比率を 0〜1.0 の範囲の値で与える．この関数と同様の機能を持った percentile 関数もあり，第 2 引数には累積要素数の百分率を 0〜100 の範囲の値で与える．

3.1.17.5　区間と集計（階級と度数調査）

データ列を，指定した区間で集計する方法について，例を挙げて説明する．まず，与えられたデータ列が，指定した区間列のどこに（どの階級に）位置するかを調べるために digitize を用いる．

書き方：　digitize(集計対象のデータ列, bins=区間のデータ列)

digitize によってデータ列 -2.5，-0.5，0.5，2.5 を区間に区切って集計する例を次に示す．

例. digitize の使用例

```
>>> a = np.array([-2.5, -0.5, 0.5, 2.5])  Enter      ←データ配列の作成
>>> b = np.linspace( a.min(), a.max(), 4 )  Enter      ←最小値〜最大値を 3 等分して 4 つの区間データにする
>>> c = np.digitize( a, bins=b )  Enter      ←元のデータ a の各要素を区間のインデックスでラベル付け
>>> print(b)  Enter      ←区間の区切りの確認
[-2.5         -0.83333333   0.83333333   2.5          ]      ←区間の区切り
>>> print(c)  Enter      ←元のデータの各要素のラベルを確認
[1 2 2 4]      ←元のデータ a をラベル化したデータ
```

この例で示した集計処理を図 64 で図解する．

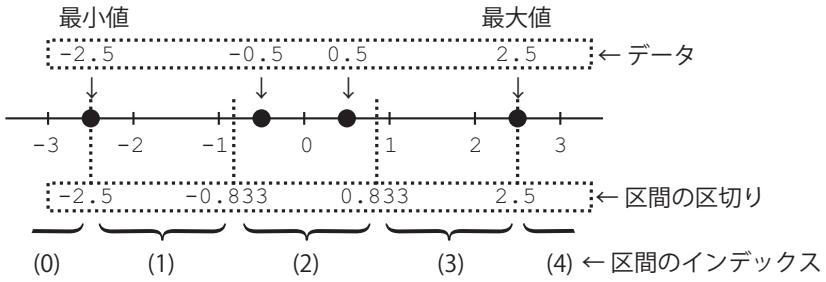


図 64: digitize による区間集計のイメージ
最小値未満の区間のインデックスが 0 となっている

上の例では集計結果が変数 c に [1 2 2 4] として得られているが，これは，元のデータ配列 a の各要素が属する区間（表 33）のインデックスを意味する．また，インデックスが 0 の区間はデータの最小値未満を意味するため，この区間に属するデータの個数は 0 である．インデックスが 4 の区間はデータの最大値以上を意味する．

先の例で得られた c（元のデータ a の各要素をラベル化したもの）を要素ごとに集計すると，元のデータ a の区間データ b に沿った集計ができる．この処理には bincount 関数を使用する．

例. 区間ごとの集計（先の例の続き）

```
>>> np.bincount(c)  Enter      ←データ配列の作成
array([0, 1, 2, 0, 1])      ←区間ごとの集計結果
```

表 33: bins=[-2.5,-0.83333333,0.83333333,2.5] で構成される区間

区間のインデックス	範囲	備考
(0)	$x < -2.5$	最小値未満の区間
(1)	$-2.5 \leq x < -0.83333333$	
(2)	$-0.83333333 \leq x < 0.83333333$	
(3)	$0.83333333 \leq x < 2.5$	
(4)	$2.5 \leq x$	最大値のみを含む区間

次に，先に示した区間集計の考え方に沿って，正規乱数のデータ列の各要素がどの階級に属するかを調べる例を示す．まず，サンプルデータを作成する．

例. 新 API による乱数生成

```
>>> import numpy as np   Enter
>>> rng = np.random.default_rng(1)  Enter
>>> a = rng.normal(50,10,100000)   Enter
```

参考. 旧 API による乱数生成

```
>>> import numpy as np   Enter
>>> np.random.seed(1)   Enter
>>> a = np.random.normal(50,10,100000)  Enter
```

$\mu = 50, \sigma = 10$ の正規乱数を 10^5 個生成し，変数 a に保持している．以後，これ（新 API で生成したデータ）を使用して解説する．旧 API で生成したデータ（上の右の例）を用いる場合は，以降の計算結果に若干の差異が出るので注意すること．

上で作成したデータ a の階級ごとの度数調査を行うために，次に示すような階級の配列 b を用意する．

例. 階級の配列（先の例の続き）

```
>>> b = np.linspace( a.min(), a.max(), 40 )   Enter     ←集計用の区間データ列
```

この例では，正規乱数列 a の最小値から最大値までの範囲を 40 個のデータとして（39 等分して）配列 b に得ている．次に，この階級データ b によって，乱数データ a の各要素をラベル付けする．

例. 正規乱数のデータ列を区間毎にラベル付け（先の例の続き）

```
>>> c = np.digitize(a,bins=b)   Enter     ←調査の実行
>>> print( a[:5], '¥n', c[:5] )   Enter     ←元のデータと集計結果の要素を先頭から 5 つ表示
[53.45584192 58.21618144 53.30437076 36.96842768 59.05355867]     ←元のデータ
[21 23 21 13 23]     ←集計結果：各要素が属する区間のインデックス
```

digitize によって，a の各要素が b のどの区間に属するかを示すインデックスのデータ列 c を得ている．区間の考え方であるが，区間データ列 b の隣接する要素 b_{n-1}, b_n の間が 1 つの区間であり，b_{n-1} 以上 b_n 未満を意味する．

この例において特に注意すべきこととして，b の最初の要素 b[0] は，データ配列 a の最小値であることが挙げられる．このことにより，digitize の結果として得られた配列 c の要素に含まれる 0 は「b[0] 未満の区間」を意味する．また，b の最終要素はデータ配列 a の最大値である．

得られた c に対して bincount [52] を用いると，a の要素の度数分布を得ることができる．

例. bincount による度数調査（先の例の続き）

```
>>> s = np.bincount( c )   Enter     ←度数分布を取得
>>> s.shape   Enter     ←配列の形状を調べる
(41,)
```

結果として配列 s に度数分布が得られる．ここで注意しなければならないこととして，s のデータ個数が 41 となっており，区間を意味する配列 b よりも要素の数が 1 つ多いことがある．これは，上で述べた注意点（最小値未満を意味する区間の存在）によるものである．この例で得られた配列 s の最初の要素 s[0] は a の最小値未満の度数を意味しており，当然のことではあるが，0 となっている．（次の例参照）

[52]「3.1.11.3 整数要素の集計」（p.84）参照のこと．

例. s の内容（先の例の続き）

```
>>> print( s )  Enter      ← s の内容表示
[    0     9    11    15    40    80   144   288   477   732 1227 1810 2660 3665      ← s の内容：
  4797 5739 6913 7814 8322 8650 8524 8018 7101 5963 4957 3781 2785 2015          先頭要素が 0
  1317   894   567   332   147    90    62    31    12     7     3     0     1]        になっている.
```

これを matplotlib を用いて棒グラフとしてプロットする例を次に示す.

例. 先の配列 s を棒グラフとしてプロットする

```
>>> import matplotlib.pyplot as plt  Enter
>>> f = plt.figure(figsize=(6,2.4))  Enter
>>> g = plt.bar( b,s[1:], width=2.0)  Enter
>>> t1 = plt.xlabel('value'); t2 = plt.ylabel('frequency')  Enter
>>> plt.tight_layout()  Enter
>>> plt.show()  Enter
```

この例では，プロットするデータを `s[1:]` としているが，これは区間データの配列 b に合せるためである.
これを実行すると，図 65 に示すようなグラフが表示される.

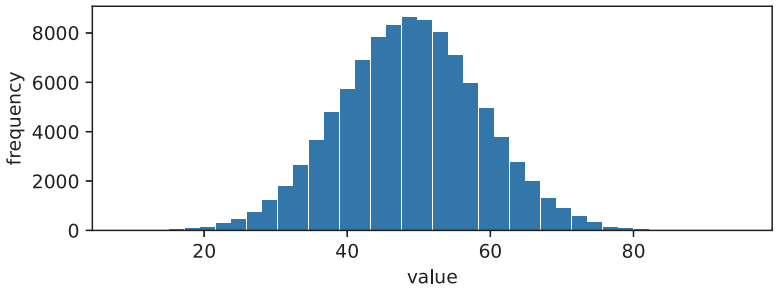


図 65: 度数分布のプロット

【参考：指定した区間で度数を集計する方法】

先の例で作成したデータ配列 a の最小値と最大値は次の通りである.

例. 配列 a の最小値と最大値（先の例の続き）

```
>>> a.min(),a.max()  Enter      ←最小値と最大値の確認
(np.float64(9.695957513043993), np.float64(94.06353522522504))      ←最小値と最大値
```

これらの値から判断し，5〜90 までの 5 刻みの区間で度数を集計することを考える.

例. 5〜90 までの区間を 5 刻みで集計（先の例の続き）

```
>>> b2 = np.linspace( 5, 90, 18 )  Enter      ← 5〜90 までを 17 等分して 18 個のデータ配列を作成
>>> print( b2 )  Enter      ←内容確認
[ 5.  10.  15.  20.  25.  30.  35.  40.  45.
  50.  55.  60.  65.  70.  75.  80.  85.  90.]      ←区間のデータ列
>>> c2 = np.digitize(a,bins=b2)  Enter      ←区間のインデキシング
>>> s2 = np.bincount(c2)  Enter      ←度数の集計
```

集計結果として得られた s2 について調べる.

例. 集計結果の確認（先の例の続き）

```
>>> s2.shape  Enter      ←配列形状の調査
(19,)      ←長さ 19 の一次元配列
>>> print( s2 )  Enter      ←内容確認
[    0     1    26   108   479  1592  4410  9374 14980 19256
 19305 14869  8993  4313  1686   451   130    25     2]      ←集計結果
```

得られた s2 をプロットする処理を次に示す.

例. プロット処理（先の例の続き）

```
>>> f = plt.figure(figsize=(4.5,2))  Enter    ←描画サイズの設定
>>> g = plt.bar( b2,s2[1:], width=4.0)  Enter    ←棒グラフの描画
>>> t1 = plt.xlabel('class'); t2 = plt.ylabel('frequency')  Enter    ←軸ラベルの設定
>>> plt.tight_layout()  Enter    ←レイアウト調整
>>> plt.show()  Enter    ←描画実行
```

これによって図 66 のようなグラフが表示される．

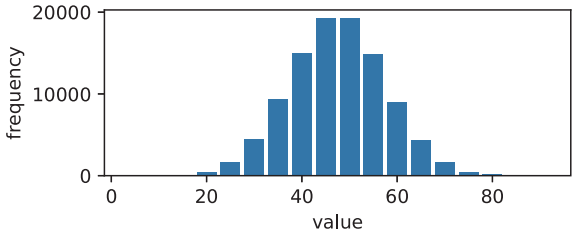


図 66: 度数分布のプロット（5 刻みに区間調整後）

■ 度数分布図（ヒストグラム）の描画に関しては更に簡便な方法があり，それについては後で説明する．
また，棒グラフの描画に関しても後に述べる．

■ bar 関数による棒グラフでヒストグラムを作図する場合，厳密には横軸は区間を表すものではない．
そのため，グラフ全体が左方向に若干（区間の幅の半分ほど）ずれた形になるので注意すること．

3.1.17.6 最頻値を求める方法

区間の集計処理の結果からデータ集合の最頻値（mode）を求める方法について例を挙げて説明する．まず，サンプルデータとして**対数正規分布**（$\mu = 0, \sigma = 0.5$）に従う乱数を作成する．そのための例を次に示す．

例. 新 API による乱数生成

```
>>> import numpy as np  Enter
>>> rng = np.random.default_rng(0)  Enter
>>> dat = rng.lognormal(0,0.5,1000000)  Enter
```

参考. 旧 API による乱数生成

```
>>> import numpy as np  Enter
>>> np.random.seed(0)  Enter
>>> dat = np.random.lognormal(0,0.5,1000000)  Enter
```

以後，このデータ dat（新 API で生成したデータ）を使用して解説する．旧 API で生成したデータ（上の右の例）を用いる場合は，以降の計算結果に若干の差異が出るので注意すること．次に，このデータのヒストグラムを作成して度数分布を確認する

例. 対数正規分布に従う乱数のヒストグラムの描画（先の例の続き）

```
>>> import matplotlib.pyplot as plt  Enter
>>> f = plt.figure( figsize=(6,2) )  Enter    ←作図の開始
>>> g = plt.hist(dat,bins=80)  Enter    ←階級数 80 でヒストグラムを作成
>>> r = plt.xlim(0,4)
>>> plt.show()  Enter    ←作図の実行
```

※ plt.hist 関数については，後の「3.1.18.1 ヒストグラム」（p.126）で詳しく解説する．

この結果，図 67 のようなヒストグラムが表示される．

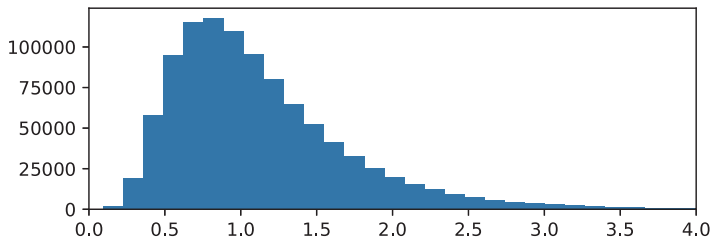


図 67: 対数正規分布に従う乱数のヒストグラム

上の例の実行の結果，区間の境界値の配列が `g[1]` に，区間毎の度数集計の結果の配列が `g[0]` に得られる．

例. 集計区間の確認：開始部分と終了部分（先の例の続き）

```
>>> print( '区間の先頭:',g[1][:3] ); print( '区間の末尾:',g[1][-3:]  )  Enter
区間の先頭:  [0.09633546 0.22831205 0.36028864]
区間の末尾:  [10.39050959 10.52248618 10.65446278]
```

例. 最も大きなデータの度数を持つ区間のインデックスを調べる（先の例の続き）

```
>>> np.argmax( g[0] )  Enter    ← argmax 関数で最大の要素のインデックス位置を調べる
np.int64(5)    ←インデックス位置が 5 の区間の度数が最大
```

この結果，g[1][5] と g[1][6] の中間を最頻値であると結論する[53] .

例. 最頻値を求める（先の例の続き）

```
>>> md = (g[1][5] + g[1][6]) / 2  Enter  ←中間値（最頻値）の算出
>>> print( '最頻値:', md )  Enter        ←表示処理
最頻値:  0.8222067115636862          ←結果
```

この例から最頻値は 0.82220671156368621 となる．最頻値をヒストグラムの中に示す処理を次に示す.

例. ヒストグラム中に最頻値を示す（先の例の続き）

```
>>> f = plt.figure( figsize=(6,2) )  Enter    ←作図の開始
>>> g = plt.hist(dat,bins=80)  Enter    ←ヒストグラムを作成
>>> r = plt.xlim(0,4)
>>> g2 = plt.vlines([md], 0, g[0].max(), lw=3, color='red' ) Enter ←最頻値の位置に垂直線を描画
>>> plt.show()  Enter    ←作図の実行
```

この結果，図 68 のようにヒストグラム中に最頻値が図示される.

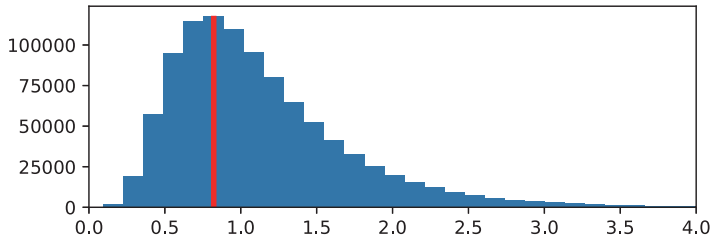


図 68: ヒストグラムの最も度数の大きい区間を赤線で図示

【digitize，bincount を用いて最頻値を求める方法】

先に示した方法は matplotlib の hist 関数の戻り値から最頻値を求めるものであるが，次に，NumPy の digitize, bincount を用いて最頻値を求める方法を示す.

基本的な方法は，先の digitize，bincount に関する説明のところで示したものの応用である.

例. digitize，bincount による度数集計（先の例の続き）

```
>>> b = np.linspace( dat.min(), dat.max(), 81 )  Enter    ←区間の境界のデータを作成
>>> dat2 = np.digitize( dat, bins=b )  Enter    ←各データを区間のインデックスでラベル付け
>>> s = np.bincount( dat2 )  Enter    ←上のデータの集計
```

この例で得られた区間データ b と集計結果 s を用いてヒストグラムを作成する作業を次に示す.

例. bar 関数による棒グラフでヒストグラムを描く（先の例の続き）

```
>>> offset = (b[1] - b[0]) / 2 Enter ← bar 関数をヒストグラムに応用する際の横位置のオフセット
>>> f = plt.figure( figsize=(6,2) )  Enter    ←作図の開始
>>> g = plt.bar( b+offset, s[1:], width=0.12 ) Enter ← bar 関数によるヒストグラムの作成
>>> r = plt.xlim(0,4)
>>> plt.show()  Enter    ←作図の実行
```

今回は，bar 関数によるヒストグラム作成における横軸のずれを補正する方法を取った．すなわち，区間の幅の半分（上記 offset）ほどグラフを右にずらす方法を取った．この結果，図 69 のようなヒストグラムが表示される.

[53] より厳密には，度数最大区間の前後の度数も考慮した補間式によって最頻値を推定する．ここでは便宜的に区間の中点を最頻値とした.

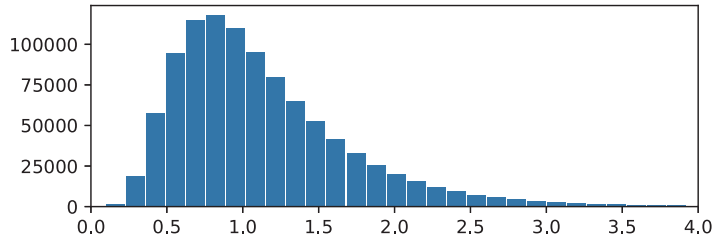


図 69: bar 関数で描画したヒストグラム

データの最頻の区間は集計結果（上記 s ）の最大値の要素のインデックスから判断する．

例. 集計結果の最大値のインデックス（先の例の続き）

```
>>> s.argmax()   Enter      ←最大値のインデックスを求める
np.int64(6)      ←インデックスは 6
```

この結果から判断し，`b[5]` と `b[6]` の中間が最頻値であると判断する[54] ．

注意）digitize 関数における集計区間のインデックスの考え方は，hist 関数の場合とは異なるので注意すること．

次に最頻値を求める例を示す．

例. 最頻値の算出（先の例の続き）

```
>>> md2 = (b[5] + b[6]) / 2   Enter   ←中間値（最頻値）の算出
>>> print( '最頻値:', md2 )   Enter          ←表示処理
最頻値:  0.8222067115636862          ←結果
```

この例から最頻値は 0.8222067115636862 となる．最頻値をヒストグラムの中に示す処理を次に示す．

例. ヒストグラム中に最頻値を示す（先の例の続き）

```
>>> f = plt.figure( figsize=(6,2) )   Enter      ←作図の開始
>>> g = plt.bar( b+offset, s[1:], width=0.12 ) Enter ← bar 関数によるヒストグラムの作成
>>> r = plt.xlim(0,4)
>>> g2 = plt.vlines([md2], 0, s.max(), lw=3, color='red' ) Enter ←最頻値の位置に垂直線を描画
>>> plt.show()   Enter      ←作図の実行
```

この結果，図 70 のようにヒストグラム中に最頻値が図示される．

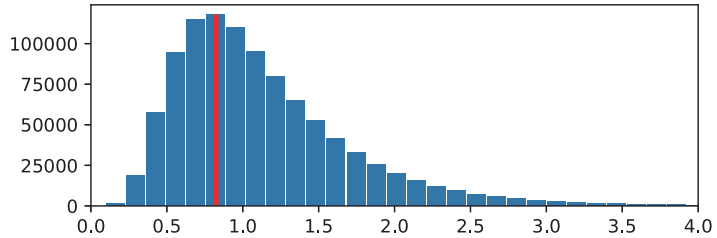


図 70: ヒストグラムの最も度数の大きい区間を赤線で図示

[54] より厳密には，度数最大区間の前後の度数も考慮した補間式によって最頻値を推定する．ここでは便宜的に区間の中点を最頻値とした．

3.1.17.7　相関係数

関数 corrcoef を用いると，2 つのデータ列の**相関係数**を求めることができる．

書き方：　corrcoef(データ列 1, データ列 2)

この関数は「データ列 1」と「データ列 2」の相関係数を**相関行列**の形で返す．以下に，サンプルデータを作成して相関係数を求める例を示す．

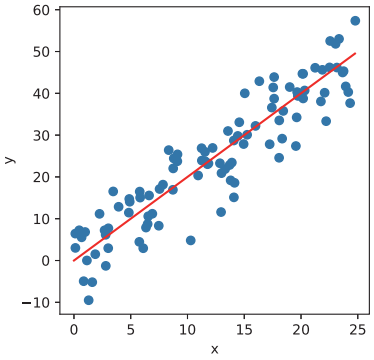
例. サンプルデータの作成

```
>>> import numpy as np   Enter
>>> rng = np.random.default_rng(0)   Enter
>>> x = np.linspace(0,25,100); y = 2*x   Enter     ← y = 2x を満たすデータ列 x, y を作成する．
>>> xr = x + rng.normal(0,1,100)   Enter     ← x を乱数で撹乱したデータ列
>>> yr = y + rng.normal(0,6,100)   Enter     ← y を乱数で撹乱したデータ列
```

x，y を理想的なデータとし，それらをノイズで撹乱したデータ列 xr，yr ができた．これの散布図（p.127 「3.1.18.2 散布図」で解説する）を表示してデータの概観を確認する（次の例）

例. 上で作成したデータ列 xr，yr の散布図（先の例の続き）

```
>>> import matplotlib.pyplot as plt   Enter
>>> f = plt.figure(figsize=(4,4))   Enter
>>> g1 = plt.plot(x,y,color='red')   Enter
>>> g2 = plt.scatter(xr,yr)   Enter
>>> t1 = plt.xlabel('x'); t2 = plt.ylabel('y')   Enter
>>> plt.show()   Enter
```



これを実行すると右のような散布図が表示される．横軸が xr，縦軸が yr である．撹乱する前のデータ x，y を直線で表示している．

次にデータ列 xr，yr の相関係数を算出する例を示す．

例. 相関係数を求める（先の例の続き）

```
>>> np.corrcoef(xr,yr)   Enter          ←相関係数（相関行列）の算出
array([[1.         , 0.92193488],        ←相関行列
       [0.92193488, 1.        ]])
```

得られた相関行列の非対角成分が 1.0 に近いので，ここで作成した xr，yr はかなり強い相関があるサンプルである．次に，互いに相関のないデータ列についても相関係数を求める例を示す．

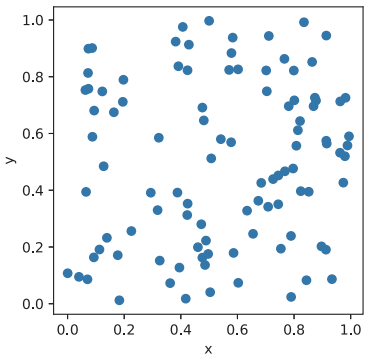
例. サンプルデータの作成

```
>>> xr2 = rng.random(100)   Enter     ← 100 個の一様乱数
>>> yr2 = rng.random(100)   Enter     ← 100 個の一様乱数
```

これによって得られる xr2，yr2 は一様乱数であり，互いに相関は無い．そのことを散布図で確認する．(次の例)

例. 上で作成したデータ列 xr2，yr2 の散布図（先の例の続き）

```
>>> f = plt.figure( figsize=(4,4) )   Enter
>>> g = plt.scatter(xr2,yr2)   Enter
>>> t1 = plt.xlabel('x'); t2 = plt.ylabel('y')   Enter
>>> plt.show()   Enter
```



これを実行すると右のような散布図が表示される．横軸が xr2，縦軸が yr2 である．

次にデータ列 xr2，yr2 の相関係数を算出する例を示す．

例. 相関係数を求める（先の例の続き）

```
>>> np.corrcoef(xr2,yr2)   Enter          ←相関係数（相関行列）の算出
array([[1.         , 0.10652495],        ←相関行列
       [0.10652495, 1.        ]])
```

得られた相関行列の非対角成分が小さい（0.10652495）ので，xr2，yr2 の間に相関はほぼ見られないことがわかる．

3.1.17.8　データのシャッフル

RNG に対する permutation，shuffle メソッドを使用して配列をシャッフルすることができる．前者は元のデータを変更せずにシャッフルした配列を返し，後者は配列そのものにシャッフル処理を行う．

例. RNG とサンプルデータの作成

```
>>> import numpy as np  Enter
>>> rng = np.random.default_rng(0)  Enter    ← RNG の作成
>>> a = np.arange(0,10,1)  Enter    ←サンプルデータの作成
>>> a  Enter    ←確認
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])    ←サンプルデータ
```

例. permutation によるシャッフル（先の例の続き）

```
>>> rng.permutation(a)  Enter    ←実行すると
array([4, 6, 2, 7, 3, 5, 9, 0, 8, 1])    ←シャッフルされた配列が新たに作られる
>>> a  Enter    ←元の配列は
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])    ←変わらない
```

例. shuffle によるシャッフル（先の例の続き）

```
>>> rng.shuffle(a)  Enter    ←実行して
>>> a  Enter    ←確認すると
array([2, 9, 3, 6, 0, 4, 8, 7, 5, 1])    ←元の配列がシャッフルされている
```

参考） permutation メソッドには引数に整数値のみ与えることもできる．その場合は 0 からその整数値未満の整数列をシャッフルした配列が得られる．

例. 整数列のシャッフル（先の例の続き）

```
>>> rng.permutation(10)  Enter    ← 0〜9 の数列を
array([5, 4, 9, 0, 8, 2, 1, 6, 7, 3])  ←シャッフルしたものが得られる
```

■ 旧 API での方法

np.random.permutation，np.random.shuffle を使用して配列をシャッフルすることができる．前者は元のデータを変更せずにシャッフルした配列を返し，後者は配列そのものにシャッフル処理を行う．

例. 配列データのシャッフル

```
>>> a = np.arange(0,10,1)  Enter    ← 0〜9 の配列を作成
>>> a  Enter    ←内容確認
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])    ←結果表示
>>> np.random.permutation(a)  Enter    ←シャッフル (1)
array([8, 2, 3, 6, 7, 1, 9, 5, 4, 0])    ←シャッフル結果
>>> a  Enter    ←元の配列の内容確認
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])    ←変化なし
>>> np.random.shuffle(a)  Enter    ←シャッフル (2)
>>> a  Enter    ←元の配列の内容確認
array([3, 6, 5, 2, 8, 9, 0, 1, 7, 4])    ←シャッフルされている
>>> np.random.permutation(10)  Enter    ← 0〜9 の数列を
array([1, 0, 6, 5, 2, 4, 7, 3, 9, 8], dtype=int32)  ←シャッフルしたものも得られる
```

permutation，shuffle は　np.random.seed(種)　によって処理の状態を設定することが可能である．また，先に説明した RandomState オブジェクトに対してメソッドとして実行することもできる．

3.1.18　データの可視化（2）

3.1.18.1　ヒストグラム

ヒストグラムのプロットには matplotlib の hist 関数を使用する．

書き方：　hist(データ配列, bins=階級の数)

「データ配列」の要素を「階級の数」に分類して度数分布図を作成する．この関数は階級のデータ列とそれに対する度数のデータ列などをタプルにして返す．‘bins=’ は省略することができ，デフォルト設定 ‘bins="auto"’ となる[55]．

以下に，正規乱数をヒストグラムにする例を挙げて解説する．

例. サンプルデータ（正規乱数）の作成

```
>>> import numpy as np Enter
>>> rng = np.random.default_rng(0)  Enter      ← RNG の作成
>>> a = rng.normal( 50, 10, 100000 )  Enter      ← μ = 50, σ = 10 の正規乱数を 10^5 個生成
```

例. ヒストグラムの描画（先の例の続き）

```
>>> import matplotlib.pyplot as plt Enter
>>> f = plt.figure( figsize=(5,2) )  Enter      ←描画サイズの設定
>>> d = plt.hist(a,bins=20)  Enter      ←ヒストグラムの作成（階級数は 20）
>>> plt.show()  Enter      ←描画実行
```

この例の実行の結果，図 71 のようなヒストグラムが表示される．

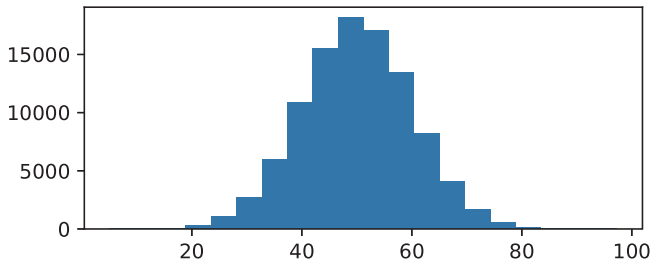


図 71: ヒストグラムの表示

上の例では hist 関数の戻り値を変数 d に得ている．この内容を確認する．

例. hist 関数の戻り値の確認（先の例の続き）

```
>>> print( d[0] )  Enter      ←戻り値の最初の要素の確認
[1.0000e+00 1.7000e+01 8.2000e+01 3.1400e+02 1.0930e+03 2.7340e+03   ←階級毎の度数の
6.0090e+03 1.0851e+04 1.5494e+04 1.8156e+04 1.7038e+04 1.3461e+04     配列
8.1830e+03 4.1010e+03 1.6910e+03 5.7400e+02 1.5800e+02 3.3000e+01
9.0000e+00 1.0000e+00]
>>> print( d[1] )  Enter      ←戻り値の 2 番目の要素の確認
[ 5.0588296 9.67186696 14.28490433 18.89794169 23.51097906 28.12401642   ←階級の境界値
32.73705378 37.35009115 41.96312851 46.57616588 51.18920324 55.80224061      の配列
60.41527797 65.02831534 69.6413527 74.25439006 78.86742743 83.48046479
88.09350216 92.70653952 97.31957689]
```

このように，hist 関数の戻り値は度数分布の集計結果を与える．戻り値の 2 番目の要素（上記 `d[1]`）は階級の境界値の配列であり，隣接する 2 つの値 b_{n-1}, b_n が意味する階級の区間は $[b_{n-1}, b_n)$ すなわち「b_{n-1} 以上 b_n 未満」である．ただし，最終の区間のみ $[b_{n-1}, b_n]$「b_{n-1} 以上 b_n 以下」（データの最大値を含む）である．

■ 指定した区間で度数を集計する方法

先に digitize 関数のところ（p.120「参考：指定した区間で度数を集計する方法」）で，階級の区切りを指定する方法について解説したが，hist 関数でもそれと類似の方法で階級の境界値を指定することができる．（次の例参照）

[55] Sturges の公式，FD 規則（Freedman–Diaconis rule）などを適宜使って階級の数が策定される．

例. 集計区間を明に指定してヒストグラムを作成する（先の例の続き）

```
>>> b = np.linspace( 0, 100, 21 )  Enter    ←集計区間の境界値の配列を作成
>>> f = plt.figure( figsize=(5,2) )  Enter    ←描画サイズの設定
>>> d2 = plt.hist(a,bins=b)  Enter    ←集計区間を指定してヒストグラムを作成
>>> plt.show()  Enter    ←描画実行
```

この例の実行の結果，図 72 のようなヒストグラムが表示される．

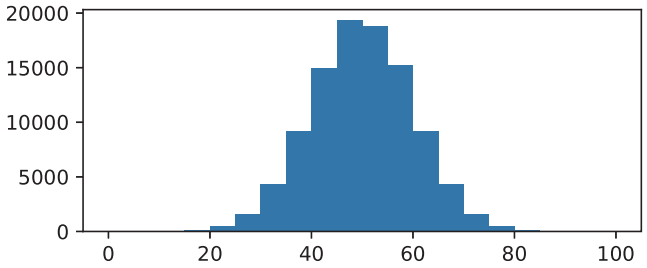


図 72: ヒストグラムの表示（階級指定）

上の例では hist 関数の戻り値を変数 d2 に得ている．この内容を確認する．

例. hist 関数の戻り値の確認（先の例の続き）

```
>>> print( d2[0] )  Enter    ←戻り値の最初の要素の確認
[0.0000e+00 2.0000e+00 2.2000e+01 1.2800e+02 5.0300e+02 1.6300e+03   ←階級毎の度数の
  4.3670e+03 9.2000e+03 1.4971e+04 1.9348e+04 1.8815e+04 1.5257e+04     配列
  9.1550e+03 4.3470e+03 1.6230e+03 4.9200e+02 1.0700e+02 3.0000e+01
  2.0000e+00 1.0000e+00]
>>> print( d2[1] )  Enter    ←戻り値の 2 番目の要素の確認
[  0.    5.   10.   15.   20.   25.   30.   35.   40.   45.   50.   55.   ←階級の境界値
  60.   65.   70.   75.   80.   85.   90.   95.  100.]                      の配列
```

注意）NumPy の digitize 関数と matplotlib の hist 関数では，区間の考え方が同じではないので注意すること．

3.1.18.2 散布図

散布図のプロットには scatter 関数を使用する．

書き方： **scatter(横軸データ, 縦軸データ)**

オプションとして，不透明度（アルファ値）を与える引数「alpha=」があり，0（透明）〜1.0（不透明）の数値を与える[56]．

サンプルプログラムを nplot04.py に示す．

プログラム： nplot04.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # データ列の生成
5  rng = np.random.default_rng(0)            # RNG作成
6  datx = rng.lognormal(0,0.35,4000)         # 対数正規乱数
7  daty = rng.lognormal(0,0.35,4000)         # 対数正規乱数
8
9  # 散布図の表示
10 plt.figure(figsize=(5,5))
11 plt.scatter(datx,daty,alpha=0.15)
12 plt.xlabel('x')
13 plt.ylabel('y')
14 plt.grid(ls='--',lw=0.8,alpha=1.0)
15 plt.title('Lognormal: mean=0, sigma=0.35, size=4000')
16 plt.show()
```

これは 4,000 件の 2 次元のデータ列の散布図を作成する例で，それぞれの軸のデータが対数正規分布（$\mu = 0$, $\sigma = 0.35$）に従う乱数になっている．

[56]特にたくさんのマーカーをプロットする場合に，マーカーを少し透明にすると，重なりによる濃淡が表現できる．

このプログラムを実行すると図 73 のようなプロットが表示される．

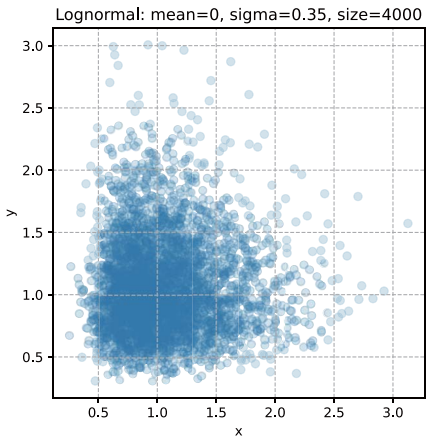


図 73: プロットの表示

参考）scatter 関数にキーワード引数 ‘s=**サイズ**’ を与えると，散布図のマーカーのサイズを指定できる．

3.1.18.3　棒グラフ

縦の棒グラフのプロットには bar 関数を，横の棒グラフのプロットには barh 関数を使用する．

書き方：　bar(横軸データ, 縦軸データ)

ここで説明する棒グラフは，先に説明したヒストグラムとは異なり，任意に作成した横軸データに対する縦軸データをプロットするものである．オプションの引数として「width= **幅**」を与えて，各棒の幅を指定することができる．幅の数値は，座標上の幅 1.0 に対する比率で与える．

barh 関数は，表示上の横軸と縦軸を入れ替えて水平の棒グラフを描画する．

書き方：　barh(横軸データ, 縦軸データ)

各棒の高さを設定するオプション引数「height=**高さ**」を与えることができる．

サンプルプログラムを nplot04-2.py に示す．

プログラム：nplot04-2.py

```
1  import numpy as np                    # NumPyの読込み
2  import matplotlib.pyplot as plt       # matplotlibの読込み
3
4  # データ列の生成
5  datx = np.arange(-1.0,1.2,0.2)
6  daty = -datx**2 + 1
7
8  # 棒グラフの表示
9  (fig,ax) = plt.subplots( 1,2, figsize=(9,4) )
10 plt.subplots_adjust(wspace=0.4)
11
12 ax[0].bar(datx,daty,width=0.17)                  # 縦の棒グラフ
13 ax[0].set_xlabel('x');  ax[0].set_ylabel('y')
14 ax[0].set_xlim(-1,1)
15 ax[0].set_title('-x^2+1')
16
17 ax[1].barh(datx,daty,height=0.17)                # 横の棒グラフ
18 ax[1].set_xlabel('y');  ax[1].set_ylabel('x')
19 ax[1].set_ylim(-1,1)
20 ax[1].set_title('-x^2+1')
21
22 plt.show()
```

この例では，$y = -x^2 + 1$ の関数を離散化してプロットデータにしている．棒の太さは bar 関数のキーワード引数 ‘width=’ に，あるいは barh 関数のキーワード引数 ‘height=’ に指定する．

このプログラムを実行すると図 74 のようなプロットが表示される．

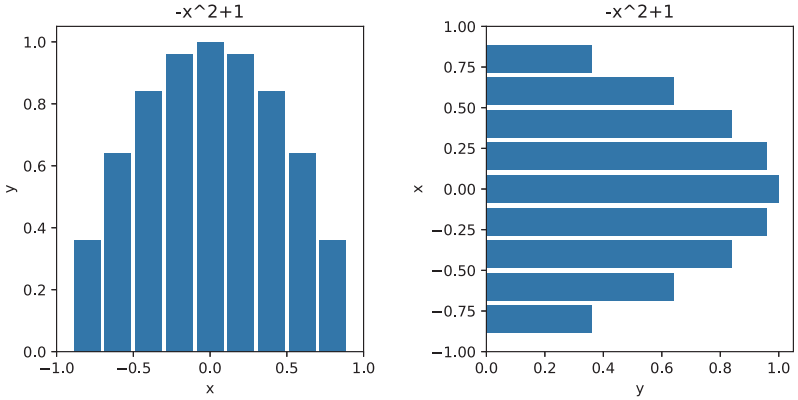


図 74: 棒グラフの表示

barh 関数による描画の場合は，グラフの横軸と縦軸が bar 関数の場合と逆になっている．このため，x 軸のラベル，y 軸のラベルを取り違えないように注意すること．

bar，barh 関数は定義域にラベル（文字列のリストなど）を取ることもできる．これに関する例をサンプルプログラム nplot04-5.py に示す．

プログラム：nplot04-5.py

```
1  import numpy as np                    # NumPyの読込み
2  import matplotlib.pyplot as plt       # matplotlibの読込み
3
4  # データ列の生成
5  daty = [ 13515271, 8839469, 7483128, 2610353 ]
6  lbl  = [ 'Tokyo', 'Osaka', 'Aichi', 'Kyoto' ]
7
8  # 棒グラフの表示
9  (fig,ax) = plt.subplots( 1,2, figsize=(8,4) )
10 plt.subplots_adjust(wspace=0.4)
11 fig.suptitle('Populations 2015')
12
13 ax[0].bar( lbl, daty )               # 縦の棒グラフ
14 ax[0].set_xlabel('prefecture')
15 ax[0].set_ylabel('population')
16
17 ax[1].barh( lbl, daty )              # 横の棒グラフ
18 ax[1].set_xlabel('population')
19 ax[1].set_ylabel('prefecture')
20
21 plt.show()
```

この場合の棒の幅（width），高さ（height）は，グラフ上のラベル間の距離を 1.0 として，その比率で与える．

このプログラムを実行すると図 75 のようなプロットが表示される．

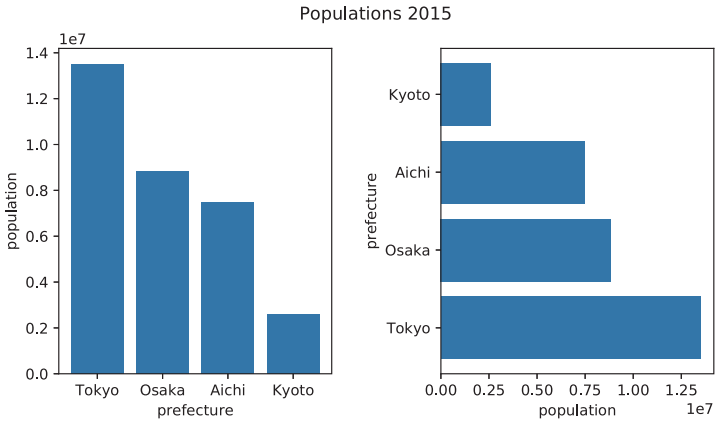


図 75: 定義域にラベルを取る棒グラフ

3.1.18.4　円グラフ

円グラフを作成するには pie 関数を使用する．可視化するデータは 1 次元の数値データ列であり，全要素の合計を 1.0 とする各要素の割合を円弧で図示する．

書き方：　pie(データ列, 各種キーワード引数…)

「各種キーワード引数」として使用頻度の高いものについて解説する．

■ 各円弧に与えるラベル：　labels=[ラベルのリスト]

円グラフの各円弧に表示するラベル（文字列）をリストにして与える．

■ 開始角度：　startangle=角度の値　（単位は度）

データ列に対応する円弧を描き始める位置（角度）を度数法（1 周が 360 度）で与える．角度の基準（0 ° ）は，円グラフの中心から右側水平の半径である．

■ 描画方向：　counterclock=[True/False]

描画方向を真理値で与える，True を与えると反時計回り，False を与えると時計回りの方向に円弧を描画する．暗黙値は True（反時計回り）である．

■ 各データの割合の表示：　autopct=書式

「書式」に従って各データ（各円弧）の割合の値を表示する．書式は文字列で与える．小数点形式で表示する場合は　**'%表示の長さ. 小数点以下の桁数 f'**　と記述する．整数形式で表示する場合は　**'%表示の長さ d'**　と記述する．数値の表示の末尾にパーセント記号を付ける場合は「%%」を書式の末尾に記述する．

■ 円グラフの中心から離れた円弧を描く：　explode=[中心からの距離のリスト]

円グラフの各円弧を描く際の「中心からの距離」をリストにして与える．距離の単位は円グラフの半径であり，例えば 0.1 という値は半径の 10 分の 1 の距離だけ中心から離れた位置を意味する．

■ 円弧の色：　colors=[色名のリスト]

各円弧の色名を文字列で与えることができる．また '#rrggbb' の形式の色コードで与えても良い．リストの要素に 0〜1.0 の数値を与えるとグレースケールとなり，その際の数値は明るさ（0 は黒，1.0 は白）を意味する．

■ ラベルのスタイルの指定：　textprops=ラベルのスタイル

各円弧のラベルや割合の数値のスタイルを辞書オブジェクト「ラベルのスタイル」として与える．例えばスタイルを白色の太字にするには

```
textprops={'color':'white', 'weight':'bold'}
```

とする．

円グラフを様々な形で描く例を次のサンプルプログラム nplot04-4.py に示す．

プログラム：nplot04-4.py

```
1  import numpy as np                    # NumPyの読込み
2  import matplotlib.pyplot as plt      # matplotlibの読込み
3
4  # データの作成
5  x = np.array( [1,3,5,7,9] )              # 値の配列
6
7  #--- 作図 ---
8  (fig,ax) = plt.subplots( 2, 3, figsize=(12,8) )
9  # データのみ（各種引数なし）の作図
10 ax[0,0].pie( x )
11 ax[0,0].set_title('(a) no args')
12
13 # 各種引数を指定して作図
14 lbl = ['1st','2nd','3rd','4th','5th']   # ラベルのリスト
15 ax[0,1].pie( x, labels=lbl, startangle=90, counterclock=False,
16             autopct='%.1f%%' )
17 ax[0,1].set_title('(b) with args')
18
19 # explodeの例
20 expd = [ 0, 0, 0.1, 0, 0 ]
```

```
ax[0,2].pie( x, labels=lbl, startangle=90, counterclock=False,
             autopct='%.1f%%', explode=expd )
ax[0,2].set_title('(c) with explode')

# 色指定の例
clr = [ '#00ff00', '#0000ff', '#00ffff', '#ff00ff', '#ffff00' ]
ax[1,0].pie( x, labels=lbl, startangle=90, counterclock=False,
             autopct='%.1f%%', explode=expd, colors=clr )
ax[1,0].set_title('(d) color change')

# 濃淡の例
clr = [ '0.4', '0.5', '0.6', '0.7', '0.8' ]      # 明るさ（暗0～1.0明）
ax[1,1].pie( x, labels=lbl, startangle=90, counterclock=False,
             autopct='%.1f%%', explode=expd, colors=clr )
ax[1,1].set_title('(e) gray scale')

# ラベルのスタイル指定の例
ax[1,2].pie( x, labels=lbl, startangle=90, counterclock=False,
             autopct='%.1f%%', explode=expd,
             textprops={'color': 'white', 'weight': 'bold'} )
ax[1,2].set_title('(f) label style')

plt.show()
```

このプログラムでは，円グラフを頂点から開始して，時計回り方向の回転にしている．このプログラムを実行すると図 76 のような円グラフが作成される．

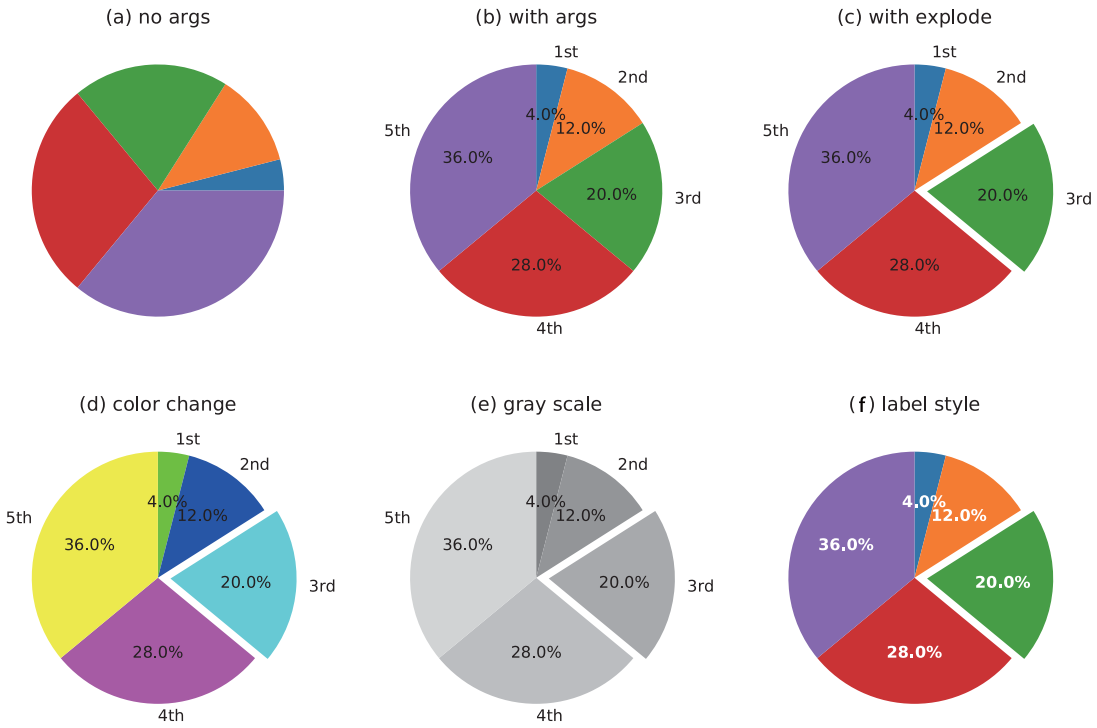


図 76: 円グラフ

図 76 の (a) は pie 関数の引数にデータ列のみを与えた描画例である．また (b) は，円弧のラベルと割合の値を表示した例，(c) は中心からはみ出た円弧を描いた例，(d) は円弧の色を指定した例，(e) は円弧をグレースケールにした例，(f) は円弧のラベルと割合の数値を白色太字にした例である．

3.1.18.5　箱ひげ図

箱ひげ図の作成には boxplot 関数を使用する．

書き方：　boxplot([データ配列のリスト], tick_labels=[ラベルのリスト])

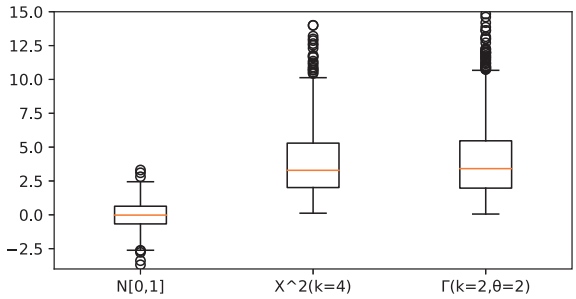
サンプルプログラムを nplot04-3.py に示す．

プログラム：nplot04-3.py

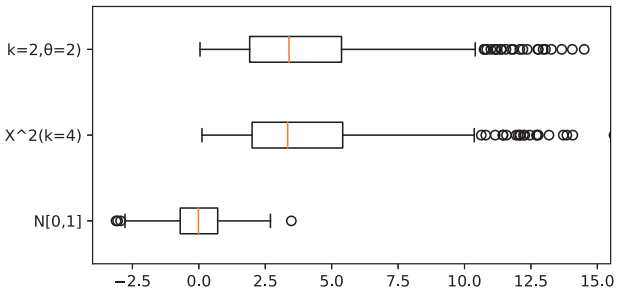
```
import numpy as np
```

```
2  import matplotlib.pyplot as plt
3
4  # データの作成
5  rng = np.random.default_rng(0)  # RNGの作成
6  d1 = rng.normal(0,1,1000)       # N[0,1]
7  d2 = rng.chisquare(4,1000)      # 自由度4のχ^2分布
8  d3 = rng.gamma(2,2,1000)        # 形状母数2, 尺度母数2のガンマ分布
9
10 # プロット
11 plt.figure( figsize=(6,3) )
12 #plt.boxplot( [d1,d2,d3], tick_labels=['N[0,1]','X^2(k=4)','Γ(k=2,θ=2)'],
13 #                                        showfliers=False )
14 plt.boxplot( [d1,d2,d3], tick_labels=['N[0,1]','X^2(k=4)','Γ(k=2,θ=2)'] )
15 plt.ylim(-4,15)
16 plt.show()
```

このプログラムでは配列 d1 に正規分布（$\mu = 0, \sigma = 1$），配列 d2 に χ^2 分布（$k = 4$），配列 d3 にガンマ分布（$k = 2, \theta = 2$）に沿った乱数列がそれぞれ格納される．このプログラムを実行すると図 77(a) のようなプロットが表示される．



(a) デフォルト



(b) vert=False でプロット

図 77: 箱ひげ図の表示

boxplot 関数に引数「`showfliers=False`」を与えると外れ値の「○」が非表示になる．プログラム nplot04-3.py の該当箇所のコメントを切り替えて（12〜14 行目の範囲を編集して）試されたい．

boxplot 関数に引数「`vert=False`」を与えると横方向の箱ひげ図が描画される．プログラム nplot04-3.py の 14〜15 行目を

```
plt.boxplot( [d1,d2,d3], tick_labels=['N[0,1]','X^2(k=4)','Γ(k=2,θ=2)'], vert=False )
plt.xlim(-4,15.5)
```

と書き換えて（plt.xlim を少し補正している）実行すると図 77(b) のようなグラフがプロットされる．

【解説】箱ひげ図

箱ひげ図はデータの分布の外観を表すだけでなく，データのばらつきを要約して視覚化するものであり，25 パーセント点（$Q_{1/4}$），50 パーセント点（$Q_{2/4}$：中央値），75 パーセント点（$Q_{3/4}$）の**四分位点**を「箱」で表示する．また，箱の長さ $Q_{3/4} - Q_{1/4}$ を IQR（interquartile range）として，有効なデータの範囲を $Q_{1/4} - 1.5 \times IQR \sim Q_{3/4} + 1.5 \times IQR$ であると考え，その範囲内にないデータは「外れ値」とみなす．外れ値は「○」で表す．（図 78）

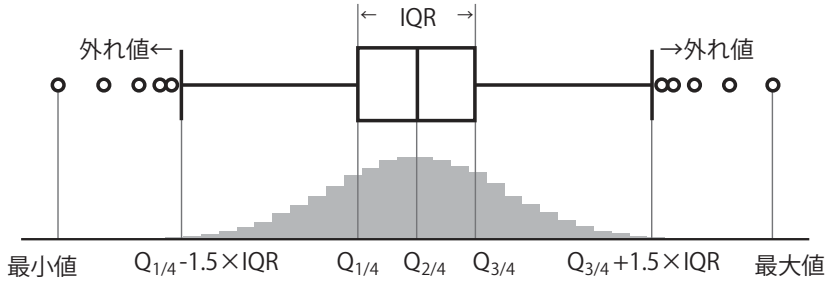


図 78: 箱ひげ図

3.1.19　データの可視化：3 次元プロット

ここでは，3 次元のプロットを実現するために必要な基礎知識を解説し，サンプルプログラムを示しながら具体的な方法について解説する．

3.1.19.1　メッシュ（格子）の考え方

高次元の関数 $\mathbb{R}^n \xrightarrow{f} \mathbb{R}$ の例として次のような関数について考える．

$$z = \cos(\sqrt{x^2 + y^2})$$

この関数の概形を図 79 に示す．

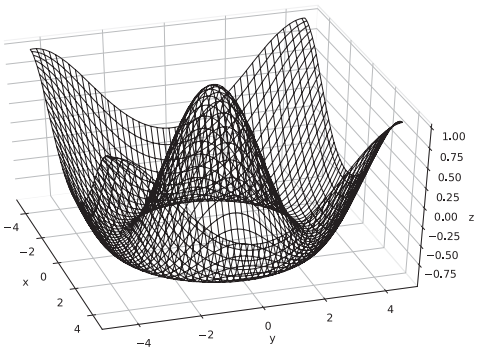


図 79: $z = \cos(\sqrt{x^2 + y^2})$ の概形

これは先の関数を 3 次元の空間にプロットしたものであるが，詳しく見ると，x，y の各座標が作る格子（グラフ底面のメッシュ）の交点上に z の値がプロットされていることがわかる．すなわち，3 次元のプロットは，x-y 座標メッシュの交点の座標（**定義域の格子**）と，それに対する z 軸の値による．

定義域の格子を生成するには，それを構成する各軸の 1 次元配列を meshgrid 関数の引数に与える．

例. $x \in [-4.5, 4.5)$，$y \in [-4.5, 4.5)$ の定義域の格子を生成する

```
>>> x = np.arange(-4.5, 4.5, 0.1)   Enter    ← x 軸のデータ列を生成
>>> y = np.arange(-4.5, 4.5, 0.1)   Enter    ← y 軸のデータ列を生成
>>> (X,Y) = np.meshgrid(x,y)   Enter    ← x-y 定義域の格子を生成
>>> Z = np.cos(np.sqrt(X**2+Y**2))   Enter    ←関数の値域の生成
```

これにより，X,Y,Z に関数の定義域と値域のデータ配列ができる．これら X, Y, Z は ndarray である．（次の例参照）

例. meshgrid によって生成された定義域の配列と値域の配列

```
>>> X.shape   Enter    ←配列 X の形状の調査
(90, 90)        ← 2 次元（90 × 90）の配列であることがわかる
>>> Y.shape   Enter    ←配列 Y の形状の調査
(90, 90)        ← 2 次元（90 × 90）の配列であることがわかる
>>> Z.shape   Enter    ←配列 Z の形状の調査
(90, 90)        ← 2 次元（90 × 90）の配列であることがわかる
```

3.1.19.2　meshgrid 関数の働き

meshgrid 関数が格子状の座標の並びを作成する様子について解説する．

2 次元の平面の 2 つの軸を 1 次元の配列で表し，それらから 2 次元の格子を作成する例を示す．

例. meshgrid で作成される配列

```
>>> x = np.array([1,2,3,4])   Enter     ← x 軸の 1 次元配列
>>> y = np.array([5,6,7,8])   Enter     ← y 軸の 1 次元配列
>>> (X,Y) = np.meshgrid(x,y)  Enter     ←格子を生成
>>> print(X)   Enter     ← 1 つ目の戻り値（配列）の確認
[[1 2 3 4]
  [1 2 3 4]          ←配列の内容（図 80 の右から 2 番目）
  [1 2 3 4]
  [1 2 3 4]]
>>> print(Y)   Enter     ← 2 つ目の戻り値（配列）の確認
[[5 5 5 5]
  [6 6 6 6]          ←配列の内容（図 80 の右端）
  [7 7 7 7]
  [8 8 8 8]]
```

この例で作成された配列 X，Y を図 80 に図解する．

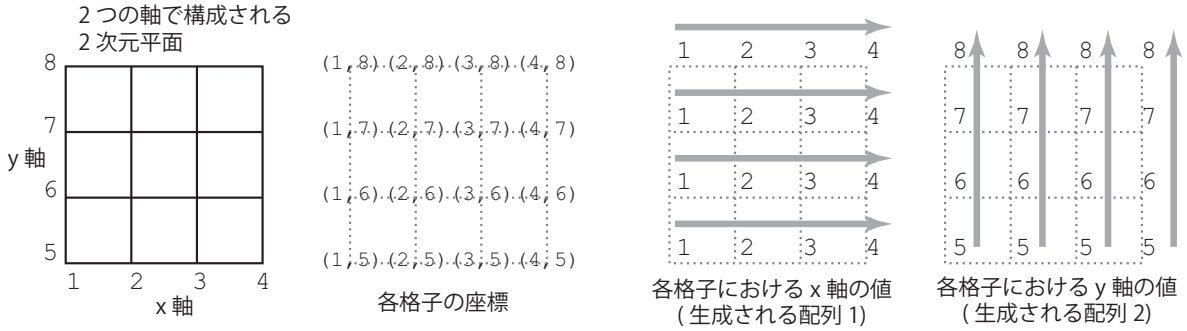


図 80: meshgrid が作成する配列

meshgrid が返す 2 つの配列は先の例における配列 X，Y に対応する．配列のインデックスの順序の関係上，配列 Y の上下の表示順序は図 80 の右端のものと比べると逆になっている．

3.1.19.3　3 次元プロットの準備

3 次元プロットのための描画の準備の具体的な方法は matplotlib の版で異なる．プロットに使用する環境によっては古い版の matplotlib が必要になる場合もあり，新旧両方の版に対応した形で解説する．

■ 新しい matplotlib（3.5 版以降）の方法

3.5 版以降の matplotlib で 3 次元のプロットを実現するには，次のようにして fig，ax オブジェクトを作成する．

例. `fig, ax = plt.subplots(subplot_kw={'projection':'3d'})`

以後，この ax オブジェクトに対して 3 次元描画のメソッドを実行する．

■ 古い matplotlib（3.4 版まで）の方法

3.4 版以前の matplotlib で 3 次元のプロットを実現するには Axes3D クラスを使用する．このクラスを使用するには，次のようにして必要なモジュールを読み込む．

```
from mpl_toolkits.mplot3d import Axes3D
```

Axes3D オブジェクトを生成するには，コンストラクタの引数に figure オブジェクトを与える．

例. ax = Axes3D(plt.figure())

以後，この ax オブジェクトに対して 3 次元描画のメソッドを実行する．

3.1.19.4　ワイヤフレーム

meshgrid で得られた定義域の格子データ（x-y 平面）と，それに対する高さのデータ（z 軸データ）を 3 次元のワイヤーフレームとして描画するには，Axes オブジェクトに対して `plot_wireframe` メソッドを実行する．

書き方：　Axes オブジェクト.plot_wireframe(X,Y,Z [, 描画オプション])

「X,Y」に定義域である xy 格子データを,「Z」に高さのデータを与える.「描画オプション」には p.87 の表 26 に挙げ

るようなものが指定できる.

先に挙げた関数 $z = \cos(\sqrt{x^2+y^2})$ の**ワイヤフレーム**をプロットするプログラムを nplot05_new.py（新版用）, nplot05_old.py（旧版用）に示す.

プログラム：nplot05_new.py（新版用）

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # meshgridの作成
5  x = np.arange(-4.5, 4.5, 0.1)
6  y = np.arange(-4.5, 4.5, 0.1)
7  (X,Y) = np.meshgrid(x,y)
8
9  Z = np.cos(np.sqrt(X**2+Y**2))  # 関数の算出
10
11 # 関数のプロット
12 fig, ax = plt.subplots(subplot_kw={'projection':'3d'})
13 ax.plot_wireframe(X,Y,Z, lw=0.5, color='black')
14 ax.set_xlabel('x')
15 ax.set_ylabel('y')
16 ax.set_zlabel('z')
17 #ax.set_box_aspect( (1,1,1) )
18 plt.show()
```

13 行目にある plot_wireframe メソッドでワイヤフレームを描画している.

プログラム：nplot05_old.py（旧版用）

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from mpl_toolkits.mplot3d import Axes3D     # 旧版用
4
5  # meshgridの作成
6  x = np.arange(-4.5, 4.5, 0.1)
7  y = np.arange(-4.5, 4.5, 0.1)
8  (X,Y) = np.meshgrid(x,y)
9
10 Z = np.cos(np.sqrt(X**2+Y**2))  # 関数の算出
11
12 # 関数のプロット
13 ax = Axes3D(plt.figure())   # 旧版用
14 ax.plot_wireframe(X,Y,Z, lw=0.5, color='black')
15 ax.set_xlabel('x')
16 ax.set_ylabel('y')
17 ax.set_zlabel('z')
18 plt.show()
```

これらプログラムを実行すると図 81 のようなプロットが表示される.

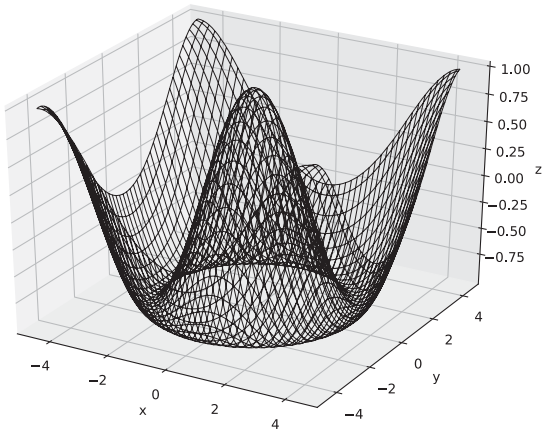


図 81: プロットの表示

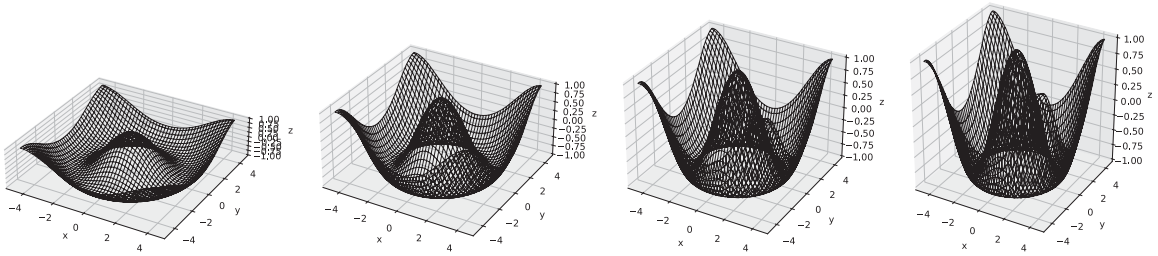
3.1.19.5　3 次元プロットのアスペクト比

3 次元プロットのアスペクト比は，Axes オブジェクトに対する set_box_aspect メソッドで設定する．

書き方：　Axes オブジェクト.set_box_aspect((X,Y,Z))

引数に与えるタプル **(X,Y,Z)** で，各軸の縮尺を指定する．デフォルトでは引数に None が与えられた状態であり，各軸の縮尺はシステムが自動的に調整する．

先のプログラム nplot05_new.py の 17 行目のコメントを外し，高さの軸（z 軸）のアスペクト比を様々に変えて実行した結果を図 82 に示す．



(1,1,0.25)　(1,1,0.5)　(1,1,0.75)　(1,1,1)

図 82: 高さの軸（z 軸）の縮尺の変更：set_box_aspect の引数を変えてプロット

3.1.19.6　面プロット（surface plot）

先のワイヤフレーム描画で使用した plot_wireframe メソッドの代わりに plot_surface メソッドを使用すると**面プロット**（surface plot）ができる．この際，matplotlib の**カラーマップ**モジュールを使用することで面にカラーマップを施すことができる．このためには必要なモジュールを次のようにして読み込んでおく．

```
from matplotlib import cm
```

サンプルプログラム nplot05-2_new.py（新版用），nplot05-2_old.py（旧版用）の実行を例にして面プロットの実行結果を見る．

プログラム：nplot05-2_new.py（新版用）

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from matplotlib import cm
4
5  # meshgridの作成
6  x = np.arange(-4.5, 4.5, 0.1)
7  y = np.arange(-4.5, 4.5, 0.1)
8  (X,Y) = np.meshgrid(x,y)
9
10 Z = np.cos(np.sqrt(X**2+Y**2))  # 関数の算出
11
12 # 関数のプロット
13 fig, ax = plt.subplots(subplot_kw={'projection':'3d'})
14
15 #ax.plot_surface(X, Y, Z, cmap=cm.gray, shade=True)
16 #ax.plot_surface(X, Y, Z, cmap=cm.hot, shade=True)
17 #ax.plot_surface(X, Y, Z, cmap=cm.cool, shade=True)
18 #ax.plot_surface(X, Y, Z, cmap=cm.bwr, shade=True)
19 ax.plot_surface(X, Y, Z, cmap=cm.seismic, shade=True)
20
21 ax.set_xlabel('x')
22 ax.set_ylabel('y')
23 ax.set_zlabel('z')
24 plt.show()
```

プログラムの 15〜19 行目が面プロットを実行する部分であり，コメントを切り替えてこの内の 1 つを実行することでカラーマップの様子を見ることができる．カラーマップは plot_surface のキーワード引数 cmap= に与える．

プログラム：nplot05-2_old.py（旧版用）

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from mpl_toolkits.mplot3d import Axes3D     # 旧版用
4  from matplotlib import cm
```

```
5
6  # meshgridの作成
7  x = np.arange(-4.5, 4.5, 0.1)
8  y = np.arange(-4.5, 4.5, 0.1)
9  (X,Y) = np.meshgrid(x,y)
10
11 Z = np.cos(np.sqrt(X**2+Y**2))  # 関数の算出
12
13 # 関数のプロット
14 ax = Axes3D(plt.figure())   # 旧版用
15
16 #ax.plot_surface(X, Y, Z, cmap=cm.gray, shade=True)
17 #ax.plot_surface(X, Y, Z, cmap=cm.hot, shade=True)
18 #ax.plot_surface(X, Y, Z, cmap=cm.cool, shade=True)
19 #ax.plot_surface(X, Y, Z, cmap=cm.bwr, shade=True)
20 ax.plot_surface(X, Y, Z, cmap=cm.seismic, shade=True)
21
22 ax.set_xlabel('x')
23 ax.set_ylabel('y')
24 ax.set_zlabel('z')
25 plt.show()
```

このプログラムを実行して表示されるグラフのバリエーションを図 83 に示す.

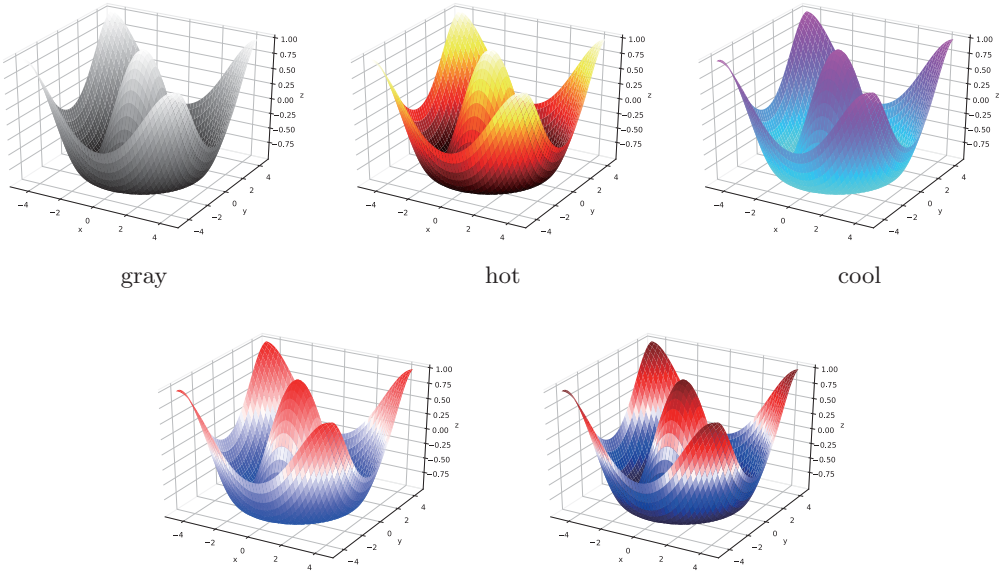


図 83: いくつかのカラーマップの例

単純で便利なカラーマップを表 34 に示す.

表 34: 単純で便利なカラーマップ

cmap	説明	cmap	説明
'Reds'	赤の濃淡	'Reds_r'	赤の濃淡（逆順）
'Greens'	緑の濃淡	'Greens_r'	緑の濃淡（逆順）
'Blues'	青の濃淡	'Blues_r'	青の濃淡（逆順）
'Greys'	黒の濃淡	'Greys_r'	黒の濃淡（逆順）

* 'Grays' ではなく'Greys' であることに注意

参考）面プロットには，カラーマップの色と値の対応を図示する**カラーバー**を付けることができる．
カラーバーについては，後の「カラーバーの表示」（p.138）で解説する．

カラーマップを使用した例を，後の「3.1.27.4 サンプルプログラム：画像の三色分解」（p.171）でも示す.

3.1.19.7　等高線のプロット（3D，2D）

3 次元の Axes オブジェクトに対して contour3D メソッドを実行することで 3 次元の**等高線**プロットができる.

書き方：　Axes オブジェクト.contour3D(X,Y,Z , levels=レベル数, [, 描画オプション])

「X,Y,Z」に関しては plot_wireframe に準ずる．描画する等高線の本数は「レベル数」[57] に与える．(デフォルトでは 7 本)

同様に，2 次元の Axes オブジェクトに対して contour メソッドを実行することで 2 次元の等高線プロットができる．

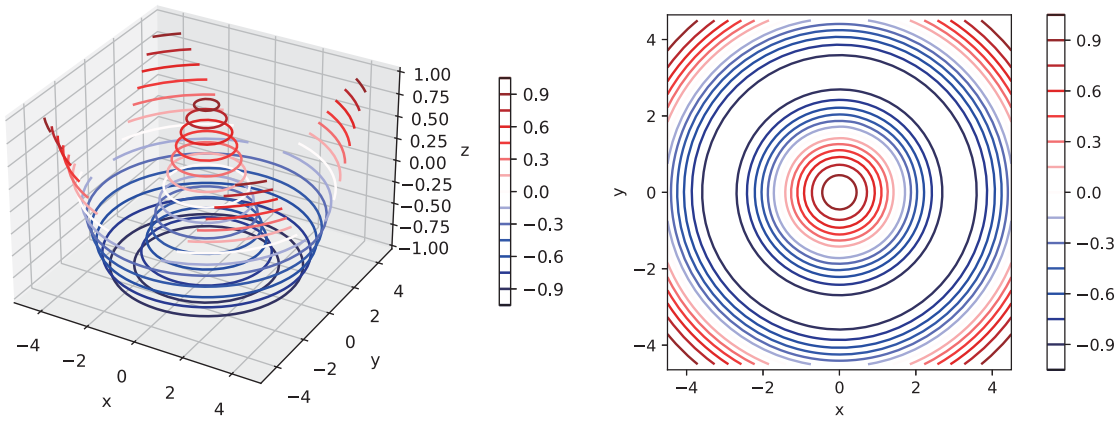
先に挙げた関数 $z = \cos(\sqrt{x^2 + y^2})$ の等高線を描くサンプルを nplot05-4.py に示す．

プログラム：nplot05-4.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from matplotlib import cm
4
5  #=== データ作成 ===
6  x = np.linspace(-4.5, 4.5, 91)  # x軸
7  y = np.linspace(-4.5, 4.5, 91)  # y軸
8  X, Y = np.meshgrid(x, y)        # xy格子
9  Z = np.cos(np.hypot(X, Y))      # z軸：関数の値
10
11 #=== 作図(3D) ===
12 fig, ax = plt.subplots(subplot_kw={'projection':'3d'})
13 g = ax.contour3D(X, Y, Z, levels=15, cmap=cm.seismic)   # 3D等高線プロット
14 fig.colorbar(g, ax=ax, pad=0.13, shrink=0.5)    # カラーバーを追加
15 ax.set_xlabel('x')       # x軸ラベル
16 ax.set_ylabel('y')       # y軸ラベル
17 ax.set_zlabel('z')       # z軸ラベル
18 plt.show()
19
20 #=== 作図(2D) ===
21 fig, ax = plt.subplots(figsize=(5,4))
22 g = ax.contour(X, Y, Z, levels=15, cmap=cm.seismic) # 2D等高線プロット
23 fig.colorbar(g, ax=ax, pad=0.08, shrink=1.0)    # カラーバーを追加
24 ax.set_xlabel('x')       # x軸ラベル
25 ax.set_ylabel('y')       # y軸ラベル
26 ax.set_aspect('equal',adjustable='datalim')
27 plt.show()
```

9 行目の np.hypot 関数は，引数に与えられた数の2 乗和の平方根を算出するものである．

このプログラムを実行すると図 84 のようなグラフが表示される．



(a) 3D 等高線プロット　　(b) 2D 等高線プロット

図 84: 等高線プロット

■ カラーバーの表示

Figure オブジェクトに対して メソッドを実行することで，グラフに**カラーバー**を付けることができる．

書き方：　Figure オブジェクト.colorbar(mappable オブジェクト, ax=Axes オブジェクト,
pad=間隔, shrink=カラーマップのサイズ)

「mappable オブジェクト」はプロット結果として得られるオブジェクトであり，例えば，先のプログラム nplot05-4.py の，contour3D や contour メソッドの戻り値がそれに該当する．「Axes オブジェクト」には，カラーマップを添える

[57] 各等高線の高さを配列として与えることも可能．

対象のものを指定する.「間隔」にはグラフとカラーマップの間隔を，当該 Axes オブジェクトのサイズに対する比率で指定する.「カラーマップのサイズ」も当該 Axes オブジェクトのサイズに対する比率で指定する.

3.1.19.8　3 次元の棒グラフ

3 次元の棒グラフを作成するには bar3d を使用する.

$z = \cos(\sqrt{x^2 + y^2}) + 1$ の棒グラフをプロットするサンプルプログラム nplot05-3.py を示す.

プログラム：nplot05-3.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # meshgridの作成
5  x = np.arange(-4.5, 4.5, 0.5)
6  y = np.arange(-4.5, 4.5, 0.5)
7  (X0,Y0) = np.meshgrid(x,y)                # プロット点の作成
8  X = np.ravel(X0);   Y = np.ravel(Y0)      # 1次元に展開
9
10 Z = np.cos(np.hypot(X,Y)) + 1             # 関数の算出
11
12 # 関数のプロット
13 fig, ax = plt.subplots( figsize=(10,5), subplot_kw={'projection':'3d'} )
14 Btm = np.zeros_like(Z)                    # 棒グラフの底
15 ax.bar3d(X, Y, Btm, 0.15, 0.15, Z, color='gray', shade=True)
16 ax.set_xlabel('x')
17 ax.set_ylabel('y')
18 ax.set_zlabel('z')
19 ax.set_box_aspect( (1,1,0.4) )
20 plt.show()
```

5〜7 行目では，関数の定義域の格子（X0,Y0）を作成している．bar3d メソッドは meshgrid で作成したグリッドは扱えず，x 軸，y 軸共に 1 次元の配列で与える．上記プログラムでは，ravel メソッドの処理（8 行目）で 1 次元のデータ列 X，Y に展開し，それに対する関数の値のデータ列 Z を生成（10 行目）している.

bar3d の引数は次のように与える.

書き方：　bar3d(x の配列, y の配列, 棒の底の値の配列, 棒の幅, 棒の奥行, z の配列,
color=色, shade=[True/False])

このメソッドには「棒の底」を与える必要があり，上の例では 14 行目で底を 0 とする配列 Btm を作成している.

このプログラムを実行して描画したグラフを図 85 に示す.

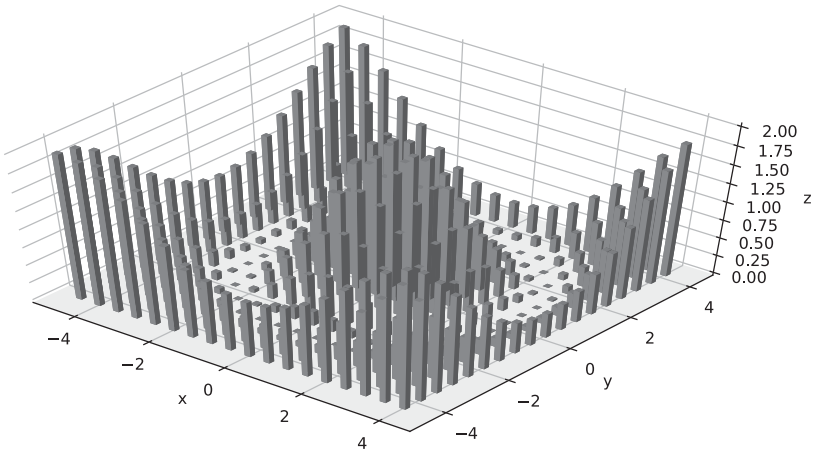


図 85: $z = \cos(\sqrt{x^2 + y^2}) + 1$ の棒グラフ

3.1.19.9　3 次元の散布図

3 次元の座標に対しても plot を使用することができるが，散布図作成には scatter を使用する[58] ことが推奨される.

書き方：　Axes オブジェクト.scatter(X, Y, Z)

[58] scatter3D というメソッドもあるが，これは scatter の別名である.

x 軸，y 軸，z 軸のデータ列「X」,「Y」,「Z」をマーカーでプロットする．

散布図を描画するサンプルプログラムを scatter3d01.py に示す．

プログラム：scatter3d01.py

```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(0)       # RNGの作成

# データの作成
x1 = rng.normal(0,0.2,100)
y1 = rng.normal(0,0.2,100)
z1 = rng.normal(0,0.2,100)
#
x2 = rng.normal(1,0.2,100)
y2 = rng.normal(1,0.2,100)
z2 = rng.normal(0,0.2,100)

# 3次元散布図
fig, ax = plt.subplots(subplot_kw={'projection':'3d'})
ax.scatter(x1,y1,z1)
ax.scatter(x2,y2,z2)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_box_aspect( (1,1,0.6) )
plt.show()
```

このプログラムは乱数で構成される 2 つのグループの点をプロットするもので，実行すると図 86 のようなグラフが表示される．

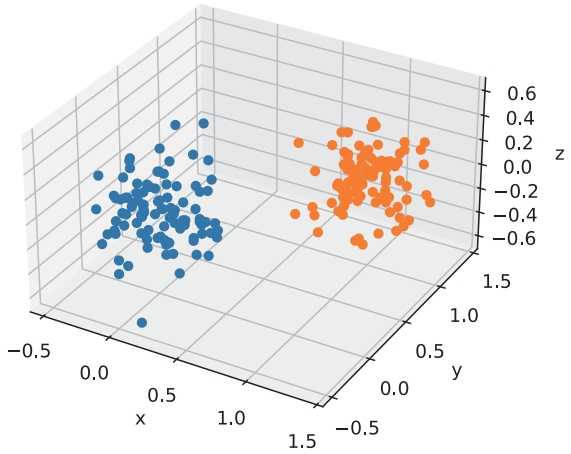


図 86: 3 次元の散布図の例

3.1.20　データの可視化：その他

3.1.20.1　ヒートマップ

2 次元の配列（ndarray，リスト）や meshgrid を用いて作成した 3 次元データをヒートマップとして表示するには pcolor 関数を使用する．この関数にキーワード引数 cmap= を与えることで，ヒートマップの表現に適用するカラーマップ（先の面プロットと同様）を指定する．

参考）pcolor と同じ使い方ができる pcolormesh もあり，大きなヒートマップを作成する際に推奨される．

次に示すサンプルプログラム heatmap01.py は 2 次元配列から，heatmap02.py は 3 次元データからヒートマップを描画するものである．

プログラム：heatmap01.py

```
import numpy as np
import matplotlib.pyplot as plt

w = 0.5      # 幅
```

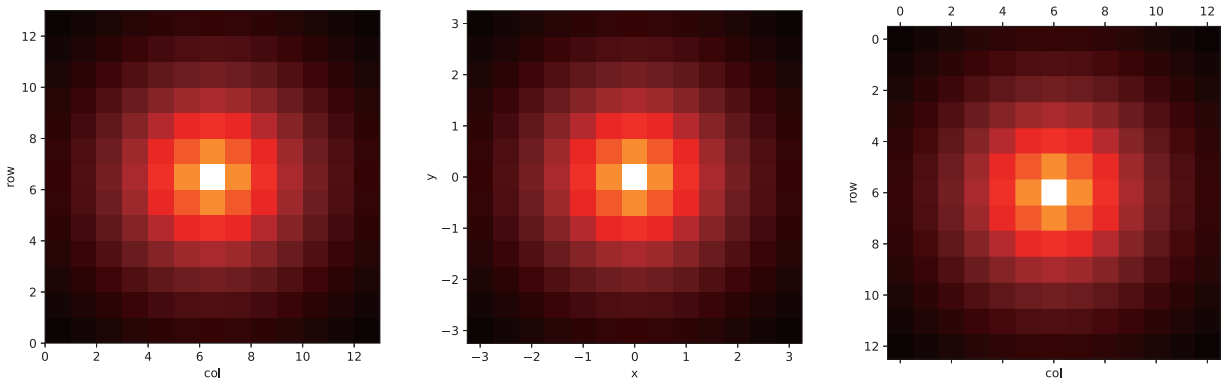
```
x = np.arange(-3,3+w,w) # 横軸の値
y = np.arange(-3,3+w,w) # 縦軸の値

# 2次元データの作成
z = np.zeros( (len(y),len(x)) )
for i in range(len(y)):
    for j in range(len(x)):
        z[i,j] = 1 / (np.sqrt(x[j]**2+y[i]**2)+1)

# ヒートマップの描画
plt.figure( figsize=(5,5) )
plt.pcolor( z, cmap=plt.cm.hot )
plt.xlabel('col');  plt.ylabel('row')
plt.show()
```

このプログラムを実行して作成したヒートマップが図 87 の (a) である．配列の開始位置（インデックスの [0,0]）のピクセルは左下となる．

プログラム：heatmap02.py

```
import numpy as np
import matplotlib.pyplot as plt

w = 0.5      # 幅
x = np.arange(-3,3+w,w) # 横軸の値
y = np.arange(-3,3+w,w) # 縦軸の値
# 3次元データの作成
(X,Y) = np.meshgrid(x,y)
Z = 1 / (np.sqrt(X**2+Y**2)+1)

# ヒートマップの描画
plt.figure( figsize=(5,5) )
plt.pcolor( X,Y,Z, cmap=plt.cm.hot )
plt.xlabel('x');    plt.ylabel('y')
plt.show()
```

このプログラムを実行して作成したヒートマップが図 87 の (b) である．



(a) 2 次元のリストのヒートマップ-1　(b) 3 次元データのヒートマップ　(c) 2 次元のリストのヒートマップ-2

図 87: ヒートマップの表示

meshgrid を用いて 3 次元データを作成する場合は，座標の**区間**が描画されるので，2 次元配列のヒートマップと比較して表示の体裁に若干の差異が生じる．

ヒートマップを作図する matshow も存在する．先のプログラム heatmap01.py と同様の処理を行うプログラム heatmap03.py を示す．

プログラム：heatmap03.py

```
import numpy as np
import matplotlib.pyplot as plt

w = 0.5      # 幅
x = np.arange(-3,3+w,w) # 横軸の値
```

```
y = np.arange(-3,3+w,w) # 縦軸の値

# 2次元データの作成
z = np.zeros( (len(y),len(x)) )
for i in range(len(y)):
    for j in range(len(x)):
        z[i,j] = 1 / (np.sqrt(x[j]**2+y[i]**2)+1)

# ヒートマップの描画
fig,ax = plt.subplots( figsize=(5,5) )
ax.matshow( z, cmap=plt.cm.hot )
ax.set_xlabel('col');   ax.set_ylabel('row')
plt.show()
```

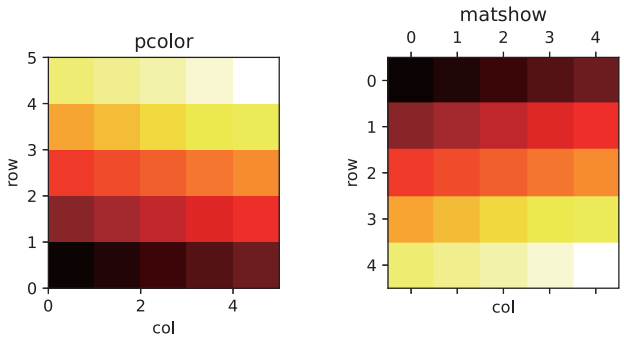
このプログラムを実行して作成したヒートマップが図 87 の (c) である．matshow は Axes オブジェクトに対するメソッド[59] である．

matshow による作図では，2 次元配列の開始位置（インデックスの [0,0]）が左上であり，pcolor による作図と上下が逆（行の順番が逆）になっている．このことは次のプログラム heatmap04.py で確認できる．

プログラム：heatmap04.py

```
import numpy as np
import matplotlib.pyplot as plt
# データの作成
a = np.arange(0,25,1).reshape( (5,5) )
# ヒートマップの描画
(fig,ax) = plt.subplots(1,2,figsize=(6,3))
ax[0].pcolor( a, cmap=plt.cm.hot )
ax[0].set_title('pcolor')
ax[0].set_xlabel('col')
ax[0].set_ylabel('row')
ax[0].set_aspect('equal')
ax[1].matshow( a, cmap=plt.cm.hot )
ax[1].set_title('matshow')
ax[1].set_xlabel('col')
ax[1].set_ylabel('row')
ax[1].set_aspect('equal')
plt.tight_layout()
plt.show()
```

このプログラムを実行すると図 88 のようなヒートマップが表示される．



(a) pcolor によるヒートマップ　(b) matshow によるヒートマップ

図 88: 上下の異なるヒートマップ

【カラーバーの表示】

pcolor，matshow で作成したヒートマップにはカラーバーを添えることができる[60] ．基本的な手順は，

1) 作図結果（pcolor，matshow）の戻り値を取得する

2) 上の値を colorbar メソッドに与えてカラーバーを作成する

である．colorbar は Figure オブジェクトに対するメソッドである．カラーバーを表示するサンプルプログラム

[59] `plt.matshow` という関数もあるが，これは推奨されない．

[60] p.138 で解説したものと同じである．

heatmap05.py を次に示す.

プログラム：heatmap05.py

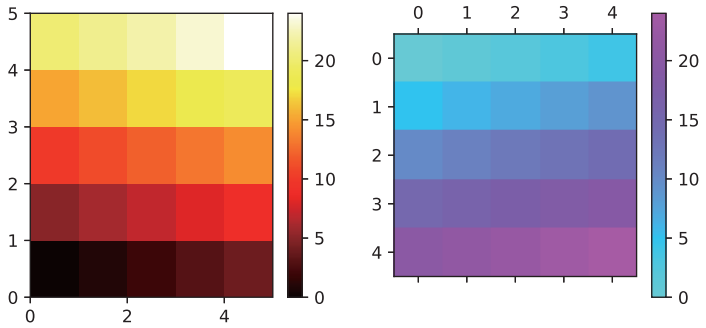
```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  # データの作成
4  a = np.arange(0,25,1).reshape( (5,5) )
5  # ヒートマップの描画
6  (fig,ax) = plt.subplots(1,2,figsize=(7,3))
7  a0 = ax[0].pcolor( a, cmap=plt.cm.hot )      # 作図結果の戻り値を取得
8  print( 'type of pcolor: ', type(a0) )
9  a1 = ax[1].matshow( a, cmap=plt.cm.cool )    # 作図結果の戻り値を取得
10 print( 'type of matshow:', type(a1) )
11 fig.colorbar( a0, ax=ax[0] )     # pcolor にカラーバーを添える
12 fig.colorbar( a1, ax=ax[1] )     # matshow にカラーバーを添える
13 plt.show()
```

プログラムの 7，9 行目でヒートマップを作成し，その戻り値を a0，a1 に取得している．それらの値を colorbar メソッドの第 1 引数に与えて（11，12 行目）カラーバーを作成している．

書き方：　colorbar(描画結果の戻り値, ax=カラーバーを添える描画面)

「カラーバーを添える描画面」は対象の Axes オブジェクトである．

heatmap05.py を実行すると図 89 のようにカラーバー付きのヒートマップが表示される．



(a) pcolor によるヒートマップ　(b) matshow によるヒートマップ

図 89: カラーバーの表示

3.1.20.2　表の作成

Axes オブジェクトに対する table メソッドを使用すると，2 次元の配列を表の形で描くことができる．

書き方：　Axes オブジェクト.table(cellText=配列, bbox=[左位置, 底辺位置, 幅, 高さ])

キーワード引数 'cellText=' に表として描画する配列を，'bbox=' にグラフ上の表を描く位置と大きさを指定する．

整数の乱数を 3 × 3 の配列として生成し，表として描くプログラムの例を pltable01.py に示す．

プログラム：pltable01.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  #--- テーブル（2次元配列）の作成 ---
5  rng = np.random.default_rng(0)       # RNGの作成
6  rtbl = rng.integers(0,100,(3,3))
7  print(rtbl)      # 表示処理（文字：標準出力）
8
9  #--- 表の作成(1)：最も素朴な表示 ---
10 fig,ax = plt.subplots( 1,2, figsize=(6,2) )
11 fig.subplots_adjust(wspace=0.6)
12 ax[0].table( cellText=rtbl, bbox=[0,0,1,1] )
13 ax[0].axis('off')    # 軸の目盛りを非表示
14 ax[0].set_title('Simple table')
15
16 #--- 表の作成(2)：見出し，色，水平位置の指定 ---
17 # 色の定義
18 Ccell = [['#dddddd','#ffffff','#ffffff'],    # セルの色
19          ['#ffffff','#dddddd','#ffffff'],
```

```
20            ['#ffffff','#ffffff','#dddddd']]
21  Ccol =  ['#ffbbbb','#bbffbb','#bbbbff']      # 列見出しの色
22  Crow =  ['#bbffff','#ffbbff','#ffffbb']      # 行見出しの色
23  # プロット
24  ax[1].table(
25          cellText=rtbl,
26          bbox=[0,0,1,1],
27          cellLoc='center',
28          colLabels=['A','B','C'],
29          rowLabels=['X','Y','Z'],
30          cellColours=Ccell,
31          colColours= Ccol,
32          rowColours= Crow
33      )
34  ax[1].axis('off')    # 軸の目盛りを非表示
35  ax[1].set_title('row/col label, colours, location')
36  plt.show()
```

プログラムの 6 行目で乱数配列を生成し，内容確認のため 7 行目でそれをターミナルウィンドウに表示する．10〜14 行目でそれを表として描画する．表の描画は 12 行目で行い，グラフ上の表示位置の範囲は (0,0)〜(1,1) としている．この際，13 行目の記述により縦横の軸の目盛りの表示を抑止している．

18 行目以降では，カラムや行の見出しを付け，セルの色を設定するなどして表を描く処理を記述している．

■ セル内の表示位置の設定

table メソッドのキーワード引数 'cellLoc=' に，セル内のデータの表示位置を与える（上記プログラム 27 行目）ことができる．設定する値は 'left'（左寄せ），'center' （中央揃え），'right'（右寄せ）から選ぶ．

■ カラム，行の見出しの設定

table メソッドのキーワード引数 'colLabels=', 'rowLabels=' にそれぞれカラムの見出しと行の見出しを配列の形で与えることができる．（上記プログラム 28〜29 行目）

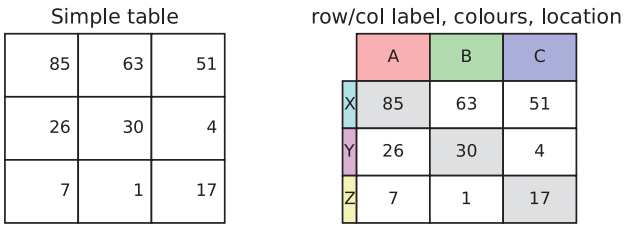
■ セルの背景色の設定

table メソッドのキーワード引数 'cellColours=', 'colColours=', 'rowColours=' にデータのセル，カラム見出し，行見出しそれぞれの背景色を与えることができる．背景色は対応するセルの色（上記プログラムでは 16 進数の色指定をしている）を配列の形で用意（上記プログラム 18〜22 行目）する．

上記プログラムを実行すると，標準出力（ターミナルウィンドウ）に

```
[[85 63 51]
 [26 30  4]
 [ 7  1 17]]
```

と表示され，図 90 の (a)，(b) の表が表示される．



(a) 素朴な形の表　　(b) 行, カラムの見出しと着色

図 90: 表の描画

3.1.21　高速フーリエ変換（FFT）

フーリエ変換は，時間 t の関数 $h(t)$ を別の変数 ω の関数 $H(\omega)$ に変換する次のような操作である．

$$H(\omega) = \int_{-\infty}^{\infty} h(t)\, \exp(-i\omega t)\, dt$$

また $\omega = 2\pi f$ と解釈すると，時間の関数から周波数の関数への変換であると見ることができ，この変換を応用すると，時間軸で解釈される振動（波形）を周波数成分に展開することができる．

関数 $H(\omega)$ は次のような**フーリエ逆変換**で元の関数 $h(t)$ に戻る．

$$h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(\omega)\, \exp(i\omega t)\, d\omega$$

これらフーリエ変換と逆変換は信号解析をはじめとする工学的応用に不可欠な処理である．ここでは NumPy が提供するフーリエ変換と逆変換の機能の使用方法について導入的に解説する．

3.1.21.1　時間領域から周波数領域への変換：フーリエ変換

NumPy の fft パッケージに含まれる fft 関数を使用することで時間の関数を周波数の関数に変換することができる．

書き方：　np.fft.fft(**データ列**)

これにより，与えたデータ列（時間領域）をフーリエ変換して周波数領域に変換したデータ列を返す．フーリエ変換が対象とするデータは**複素数**[61] であり，変換によって得られる周波数領域のデータ列も複素数である．通常の信号解析では，扱う波形データは実数で構成されることが一般的であるが，フーリエ変換の結果得られるデータは複素数である．

フーリエ変換により得られたデータ列の横軸（周波数）を求めるには fftfreq 関数を用いる．

書き方：　np.fft.fftfreq(**データ個数**, d=**サンプリングの時間間隔**)

データ個数は fft 関数に与えたデータ列の長さである．**サンプリングの時間間隔**には，サンプリング周波数の逆数を与える．fftfreq 関数は周波数のデータ列を返す．この配列の要素と，fft 関数で得られた周波数成分のデータ列の要素が対応する．

■ fftfreq が返す配列

fftreq 関数は，fft 関数の処理結果に対応する周波数軸のデータを返すが，その配列の要素の順序には特徴がある．これに関して例を示して解説する．

例. fftfreq が返す周波数軸：データ数 15，サンプリング周波数 15Hz の例

```
>>> import numpy as np   Enter      ← NumPy の読込み
>>> n = 15   Enter      ←データ数 n の設定
>>> r = 15   Enter      ←サンプリング周波数 r の設定
>>> f = np.fft.fftfreq(n,d=1.0/r)   Enter      ←周波数軸 f の生成
>>> print(f)   Enter      ←内容確認
 [ 0.   1.   2.   3.   4.   5.   6.   7.  -7.  -6.  -5.  -4.  -3.  -2.  -1.]
```

このように，fftfreq が返す周波数軸は，正の領域（0〜7Hz）の後に負の領域（-7〜 -1Hz）の順序となっている．この形式は API 実装上の事情に由来するもの[62] である．

参考）上の例では，fftfreq の戻り値の配列の中の最大値が 7Hz であり，サンプリング周波数である 15Hz の半分の周波数（**ナイキスト周波数**）を丸めたものになっていることがわかる．ナイキスト周波数を超える成分は折り返して負の周波数領域に現れる（**エイリアシング**）．

■ fft 関数が返す値について

fft が返す配列は各周波数の成分から成るが，振幅の値として解釈するには，データ数 n で除算して正規化する必要がある．例えば，振幅が 1，周波数が f の正弦波形のデータが n 個の要素の配列として与えられた場合，fft 関数で処理すると，得られたデータの中の当該周波数 f の値が n となる．

[61] NumPy での複素数の扱いについては後の「3.1.22 複素数の計算」(p.150) を参照のこと．

[62] NumPy 以外の FFT 関連ツールやライブラリの多くがこの形式になっている．

3.1.21.2　周波数領域から時間領域への変換：フーリエ逆変換

周波数領域のデータ列を時間領域のデータ列に変換（**フーリエ逆変換**）するには ifft 関数を使用する．

書き方：　np.fft.ifft(**周波数領域のデータ列**)

この関数は，先の fft 関数の逆関数であり，fft が返す値をそのまま ifft に渡すと，元の波形データが得られる．（ただし，微小な誤差に注意すること）

フーリエ変換，フーリエ逆変換の処理を行うサンプルプログラムを npfft01.py に示す．

プログラム：npfft01.py

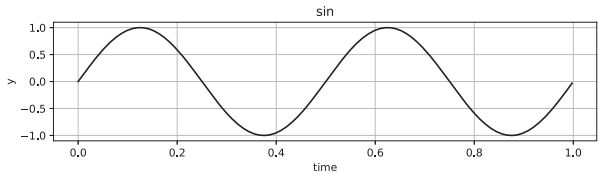
```
import numpy as np
import matplotlib.pyplot as plt

############################################################
#   波形データの生成                                       #
############################################################
fs = 400            # サンプリング周波数
#----- 時間軸データ -----
d_t = np.array( [i/fs for i in range(fs)] )

#----- 正弦波 -----
d_sin = np.sin( 4.0 * np.pi * d_t )

# 波形のプロット
plt.figure(figsize=(8,2.5))
plt.plot(d_t, d_sin, lw=1.5, color='black')
plt.xlabel('time')  # 横軸ラベル
plt.ylabel('y')     # 縦軸ラベル
plt.title('sin')
plt.grid()
plt.tight_layout()
plt.show()

#----- 鋸歯状波（ノコギリ波）-----
d_saw = np.array( [i/100.0-1.0 for i in range(200)]*2 )

# 波形のプロット
plt.figure(figsize=(8,2.5))
plt.plot(d_t, d_saw, lw=1.5, color='black')
plt.xlabel('time')  # 横軸ラベル
plt.ylabel('y')     # 縦軸ラベル
plt.title('Saw')
plt.grid()
plt.tight_layout()
plt.show()

#----- 三角波 -----
tmp = [i/50.0-1.0 for i in range(100)]
d_tri = np.array( (tmp+tmp[::-1])*2 )

# 波形のプロット
plt.figure(figsize=(8,2.5))
plt.plot(d_t, d_tri, lw=1.5, color='black')
plt.xlabel('time')  # 横軸ラベル
plt.ylabel('y')     # 縦軸ラベル
plt.title('Triangle')
plt.grid()
plt.tight_layout()
plt.show()

#----- 方形波 -----
d_rct = np.array( ([-1.0]*100+[1.0]*100)*2 )

# 波形のプロット
plt.figure(figsize=(8,2.5))
plt.plot(d_t, d_rct, lw=1.5, color='black')
plt.xlabel('time')  # 横軸ラベル
plt.ylabel('y')     # 縦軸ラベル
plt.title('Rect')
plt.grid()
```

```
61 plt.tight_layout()
62 plt.show()
63
64 ############################################################
65 #   フーリエ変換                                           #
66 ############################################################
67 # 周波数軸データの作成
68 n = len(d_t)            # データ長
69 frq = np.fft.fftfreq(n,d=1.0/fs)
70
71 #----- 正弦波の解析 -----
72 f_sin = np.fft.fft(d_sin)
73 f_sin_n = np.abs( f_sin ) / n   # 振幅に変換
74
75 # 振幅スペクトルのプロット
76 plt.figure(figsize=(8,2.5))
77 plt.bar(frq, f_sin_n, color='black')
78 plt.xlim(-8,8)
79 plt.xlabel('Frequency (Hz)')    # 横軸ラベル
80 plt.ylabel('Amplitude')         # 縦軸ラベル
81 plt.title('Amplitude spectrum of sin')
82 plt.grid()
83 plt.tight_layout()
84 plt.show()
85
86 #----- 鋸歯状波（ノコギリ波）の解析 -----
87 f_saw = np.fft.fft(d_saw)
88 f_saw_n = np.abs( f_saw ) / n   # 振幅に変換
89
90 # 振幅スペクトルのプロット
91 plt.figure(figsize=(8,2.5))
92 plt.bar(frq, f_saw_n, color='black')
93 plt.xlim(-60,60)
94 plt.xlabel('Frequency (Hz)')    # 横軸ラベル
95 plt.ylabel('Amplitude')         # 縦軸ラベル
96 plt.title('Amplitude spectrum of Saw')
97 plt.grid()
98 plt.tight_layout()
99 plt.show()
100
101 #----- 三角波の解析 -----
102 f_tri = np.fft.fft(d_tri)
103 f_tri_n = np.abs( f_tri ) / n   # 振幅に変換
104
105 # 振幅スペクトルのプロット
106 plt.figure(figsize=(8,2.5))
107 plt.bar(frq, f_tri_n, color='black')
108 plt.xlim(-30,30)
109 plt.xlabel('Frequency (Hz)')    # 横軸ラベル
110 plt.ylabel('Amplitude')         # 縦軸ラベル
111 plt.title('Amplitude spectrum of Triangle')
112 plt.grid()
113 plt.tight_layout()
114 plt.show()
115
116 #----- 方形波の解析 -----
117 f_rct = np.fft.fft(d_rct)
118 f_rct_n = np.abs( f_rct ) / n   # 振幅に変換
119
120 # 振幅スペクトルのプロット
121 plt.figure(figsize=(8,2.5))
122 plt.bar(frq, f_rct_n, color='black')
123 plt.xlim(-60,60)
124 plt.xlabel('Frequency (Hz)')    # 横軸ラベル
125 plt.ylabel('Amplitude')         # 縦軸ラベル
126 plt.title('Amplitude spectrum of Rect')
127 plt.grid()
128 plt.tight_layout()
129 plt.show()
130
131 ############################################################
132 #   フーリエ逆変換                                         #
```

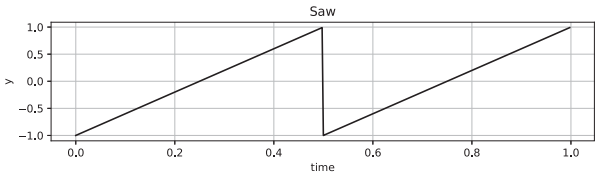
```
133 ############################################################
134 #----- 方形波の逆変換 -----
135 i_rct = np.fft.ifft(f_rct)
136
137 # 波形のプロット
138 plt.figure(figsize=(8,2.5))
139 plt.plot(d_t, i_rct.real, lw=1.5, color='black')
140 plt.xlabel('time')  # 横軸ラベル
141 plt.ylabel('y')     # 縦軸ラベル
142 plt.title('Rect (Fourier inverse transform)')
143 plt.grid()
144 plt.tight_layout()
145 plt.show()
```

解説：

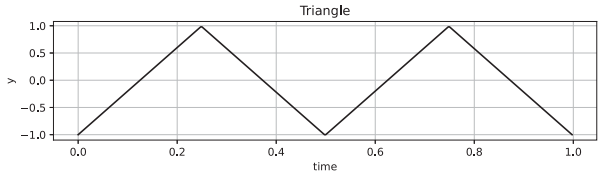
7〜62 行目で，各種の波形データ（正弦波，鋸歯状波，三角波，方形波）を生成してそれらをプロットしている．この部分により表示されるグラフを図 91 に示す．



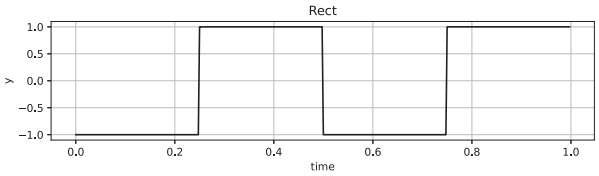
(a) 正弦波



(b) 鋸歯状波（ノコギリ波）



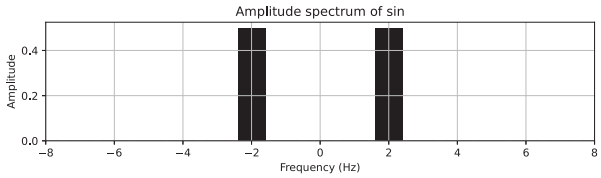
(c) 三角波



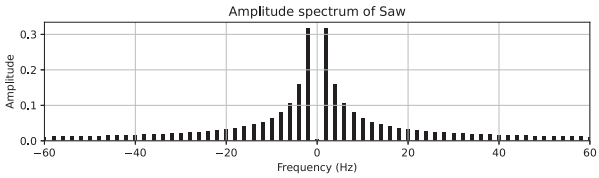
(d) 方形波

図 91: 波形データ（時間の関数）

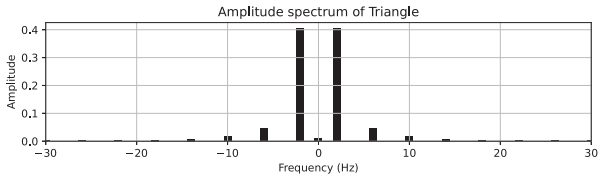
これらの波形データは振幅 ±1.0 で周波数は 2Hz である．すなわち，最大の時刻は 1 で，その間に 2 回のサイクルを繰り返すものである．これらの波形データをフーリエ変換してプロットしているのが 68〜129 行目の部分である．プロットに必要となる横軸（周波数スケール）のデータは 68〜69 行目で生成している．この部分の実行により得られる周波数領域のプロット（振幅スペクトル）を図 92 に示す．（棒グラフの描画には matplotlib の bar 関数を使用している）



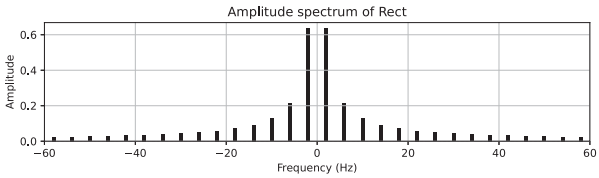
(i) 正弦波



(ii) 鋸歯状波（ノコギリ波）



(iii) 三角波



(iv) 方形波

図 92: 振幅スペクトル（周波数の関数）

この結果，各波形とも周波数は 2Hz であるので，主たる成分である 2Hz の成分が最も強く表示されている．当然であるが正弦波はそれ自身が 1 つの周波数成分であるので 2Hz のみの成分から成ることがわかる．フーリエ変換の結果として得られる周波数の成分は，正負の両方の周波数領域にわたる形となる．また，このプログラムでは，各周波数の振幅を求める際に，データ数 n で除算して正規化している．

NumPy の fft は正規化を行わないため，得られるスペクトルの振幅はサンプル数 n に比例する．振幅スペクトルを正しく比較するには，このプログラムのような正規化が必要となる．ifft は内部でこの正規化を採用しているので，fft → ifft の往復で元の波形が復元される．

この処理で得られた方形波の周波数領域のデータをフーリエ逆変換により再度時間領域のデータ列に戻している．（135 行目）　それをプロットしているのが 138〜145 行目の部分であり，プロット結果を図 93 に示す．

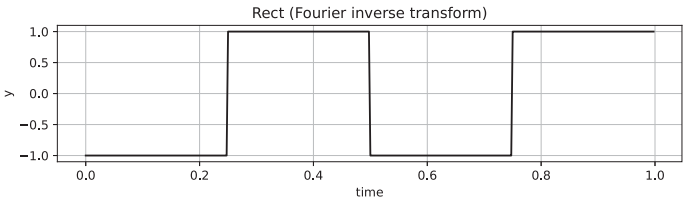


図 93: フーリエ逆変換で再構成した方形波

3.1.21.3　離散フーリエ変換を用いる際の注意

有限長の離散データ列に対してフーリエ変換（FFT）を適用すると，そのデータが周期的に繰り返されるものと見なしたうえでの周波数成分が得られる．このため，時間軸の両端がつながらない場合には不連続が生じ，実際の波形には含まれない成分がスペクトル上に現れることがある（これを**スペクトルリーケージ**と呼ぶ）．この問題を避けるには，解析区間の取り方を工夫したり，時間領域でデータに窓関数を掛けて端点を滑らかにするなどの処理が一般に行われる．

信号解析のための具体的な工夫に関しては本書の範囲を超えるため割愛する．

3.1.22　複素数の計算

NumPy の各種関数やメソッドの多くは複素数の配列を扱うことができる．配列の実数成分，虚数成分には，当該配列の real，imag それぞれのプロパティを介してアクセスできる．

例. 配列要素の実部，虚部へのアクセス

```
>>> a = np.zeros( 3, dtype='complex' )  Enter    ←複素数型のゼロの配列
>>> a  Enter    ←内容確認
array([0.+0.j, 0.+0.j, 0.+0.j])    ←結果表示
>>> a.real = [1, 2, 3]  Enter    ←全要素の実部を一度に設定
>>> a  Enter    ←内容確認
array([1.+0.j, 2.+0.j, 3.+0.j])    ←結果表示
>>> a.imag = [4, 5, 6]  Enter    ←全要素の虚部を一度に設定
>>> a  Enter    ←内容確認
array([1.+4.j, 2.+5.j, 3.+6.j])    ←結果表示
>>> a.real  Enter    ←全要素の実部を参照
array([1., 2., 3.])    ←結果表示
```

3.1.22.1　複素数の平方根

要素が複素数である配列に対する sqrt 関数の値は複素数である．

例. 複素数の平方根

```
>>> a = np.array([-1+0j,4])  Enter    ←先頭の要素が −1 + 0j の複素数
>>> np.sqrt(a)  Enter    ←各要素の平方根を求める
array([0.+1.j, 2.+0.j])    ←結果が複素数の配列として得られる
```

要素が全て実数である配列に対して sqrt を実行すると，実数の範囲で計算する．(次の例参照)

例. 平方根の結果が複素数として得られないケース

```
>>> a = np.array([-1,4])  Enter    ←要素が全て実数の配列
>>> np.sqrt(a)  Enter    ←各要素の平方根を求める
<stdin>-54:1:  RuntimeWarning:  invalid value encountered in sqrt    ←警告メッセージ
array([nan, 2.])    ←実数とならない要素が nan（非数）になっている
```

sqrt 関数の引数に与える配列が複素数でない場合でも，オプション引数 'dtype='complex'' を与えると，結果が複素数で得られる．

例. 上の例において，結果を複素数で得る方法（先の例の続き）

```
>>> np.sqrt(a,dtype='complex')  Enter    ←各要素の平方根を求める
array([0.+1.j, 2.+0.j])    ←結果が複素数の配列として得られる
```

参考）配列が実数か複素数かを判定すには isreal，iscomplex 関数を用いる．

例. 配列が複素数かを調べる（先の例の続き）

```
>>> np.iscomplex( np.array([-1+2j,3]) ).any()  Enter    ←「1 つでも複素数があるか」を検査
np.True_    ←複素数を含む（複素数の配列である）
```

3.1.22.2　複素数のノルム

複素数のノルム（長さ，絶対値）は absolute 関数で求めることができる．

例. 複素数のノルム

```
>>> a = np.array([1+1j,3+4j,-2])  Enter    ←複素数の配列
>>> np.absolute(a)  Enter    ←各要素のノルムを求める
array([1.41421356, 5.     , 2.     ])    ←計算結果
```

参考）absolute 関数には別名 abs もあり，上の例で np.abs(a) としても同じ結果が得られる．

3.1.22.3　共役複素数

共役複素数を求めるには conj メソッド，conj 関数，もしくは conjugate 関数を使用する．

例. 共役複素数：その 1

```
>>> a = np.array([1+1j,2-2j,3+3j,4-4j])  Enter    ←複素数の配列
>>> np.conj(a)  Enter    ←各要素の共役複素数を求める（conj 関数）
array([1.-1.j, 2.+2.j, 3.-3.j, 4.+4.j])    ←計算結果
>>> a.conj()  Enter    ←各要素の共役複素数を求める（conj メソッド）
array([1.-1.j, 2.+2.j, 3.-3.j, 4.+4.j])    ←計算結果（上と同じ）
```

例. 共役複素数：その 2（先の例の続き）

```
>>> np.conjugate(a)  Enter    ←各要素の共役複素数を求める（conjugate 関数）
array([1.-1.j, 2.+2.j, 3.-3.j, 4.+4.j])    ←計算結果
```

3.1.22.4 複素数の偏角（位相）

複素数の偏角（位相）を求めるには angle 関数を使用する.

例. 複素数の偏角（位相）

```
>>> a = np.array([1,1+1j,0+1j])  Enter    ←複素数の配列
>>> np.angle(a)  Enter    ←各要素の偏角を求める
array([0.     , 0.78539816, 1.57079633])    ←計算結果
```

得られた要素の内，後ろ 2 つが $\pi/4, \pi/2$ となっていることを次の例で確認できる.

例. $\pi/4, \pi/2$ を求めて上の例と比較する（先の例の続き）

```
>>> np.pi/4, np.pi/2  Enter    ← π/4, π/2 を求めてみる
(0.7853981633974483, 1.5707963267948966)    ←計算結果
```

参考）angle 関数にオプション引数 'deg=True' を与えると，結果を度数法の単位で得ることができる.

例. angle 関数の結果を度数法にする（先の例の続き）

```
>>> np.angle(a,deg=True)  Enter    ←各要素の偏角を求める
array([ 0., 45., 90.])    ←度数法による値
```

3.1.22.5 微小な虚部の切り落としによる実数化

real_if_close 関数を使用すると，微小な虚部を無視することによって，複素数を実数（浮動小数点数）に変換できることがある．ただし，虚部が微小でない場合は実数化できない.

例. 実数化できる場合 (1)

```
>>> a = np.array([1+0j,2+0j,3+0j])  Enter    ←虚部が 0 の複素数の配列は
>>> np.real_if_close(a)  Enter    ←実数化することが
array([1., 2., 3.])    ←できる
```

例. 実数化できる場合 (2)

```
>>> a = np.array([1+0j,2+0j,3+1e-100j])  Enter    ←虚部の微小な複素数の配列は
>>> np.real_if_close(a)  Enter    ←実数化することが
array([1., 2., 3.])    ←できる
```

これに対し，虚部が十分に小さくない要素を持つ配列は実数化できない.

例. 実数化できない場合

```
>>> a = np.array([1+0j,2+0j,3+0.1j])  Enter    ←虚部が微小でない複素数を含む配列は
>>> np.real_if_close(a)  Enter    ←実数化することが
array([1.+0.j , 2.+0.j , 3.+0.1j])    ←できない
```

参考）real_if_close 関数にオプション引数 'tol=' を与えて実数化の判定基準を調整することができるが，これに関する詳細は NumPy の公式インターネットサイトの情報を参照すること.

3.1.23 行列，ベクトルの計算（線形代数のための計算）

NumPy は，数値（複素数）から成る**行列**（matrix）と**ベクトル**を扱うための各種の機能を提供している.

3.1.23.1 行列の和と積

行列の和を求めるには通常の加算演算子 '+' が使用できる.

例. 行列の和

```
>>> a1 = np.array([[1,2],[3,4]])  Enter     ←行列 a1 を生成
>>> a2 = np.array([[5,6],[7,8]])  Enter     ←行列 a2 を生成
>>> a1 + a2  Enter              ←加算の実行
array([[ 6, 8],                 ←処理結果
       [10, 12]])
```

通常の乗算演算子 '*' を用いて行列同士を計算すると，各要素毎の積を要素とする行列が得られる．行列同士の積を求めるには dot 関数を使用する.

例. 行列の積（先の例の続き）

```
>>> a1 * a2  Enter              ←'*' の実行
array([[ 5, 12],                ←処理結果
       [21, 32]])
>>> np.dot(a1,a2)  Enter        ←行列の積
array([[19, 22],                ←処理結果
       [43, 50]])
```

dot 関数と同様の計算を行う `@` 演算子もある[63] .

例. `@` 演算子による行列の積（先の例の続き）

```
>>> a1 @ a2  Enter              ←行列の積
array([[19, 22],                ←処理結果
       [43, 50]])
```

3.1.23.2 単位行列，ゼロ行列，他

全ての要素がゼロや 1 であるような行列や，単位行列を生成する例を示す.

例. 全てゼロの配列の生成： zeros 関数

```
>>> np.zeros( 3 )  Enter            ←ゼロが 3 つの配列
array([ 0., 0., 0.])                ←処理結果
>>> np.zeros( (2,3) )  Enter        ← 2 行 3 列のゼロ行列
array([[ 0., 0., 0.],               ←処理結果
       [ 0., 0., 0.]])
```

既存の行列（配列）のサイズに合わせた形のゼロ行列を作成するには zeros_like を使用する.

例. 既存の行列と同じ形のゼロ行列

```
>>> a = np.array([[1,2,3,4],[5,6,7,8]])  Enter   ← 2 行 4 列の行列 a
>>> np.zeros_like(a)  Enter     ←行列 a と同じサイズのゼロ行列を作成
array([[0, 0, 0, 0],            ← 2 行 4 列のゼロ行列
       [0, 0, 0, 0]])
```

例. 全て 1 の配列の生成： ones 関数（先の例の続き）

```
>>> np.ones( 3 )  Enter             ← 1 が 3 つの配列
array([ 1., 1., 1.])                ←処理結果
>>> np.ones( (2,3) )  Enter         ← 2 行 3 列の 1 ばかりの行列
array([[ 1., 1., 1.],               ←処理結果
       [ 1., 1., 1.]])
```

既存の行列（配列）のサイズに合わせた形で，全ての要素が 1 である行列を作成するには `ones_like` を使用する.

例. 既存の行列と同じ形の 1 ばかりの行列（先の例の続き）

```
>>> np.ones_like(a)  Enter      ←行列 a と同じサイズの 1 ばかりの行列を作成
array([[1, 1, 1, 1],            ← 2 行 4 列の 1 ばかりの行列
       [1, 1, 1, 1]])
```

[63] NumPy 1.10 の版から導入され，使用が推奨されている.

例. 単位行列の生成： identity 関数

```
>>> np.identity( 5 )  Enter        ← 5 × 5 の単位行列
array([[ 1., 0., 0., 0., 0.],       ←処理結果
       [ 0., 1., 0., 0., 0.],
       [ 0., 0., 1., 0., 0.],
       [ 0., 0., 0., 1., 0.],
       [ 0., 0., 0., 0., 1.]])
```

3.1.23.3 行列の要素を全て同じ値にする

既存の配列の全ての要素を同じ値で置き換えるには fill メソッドを用いる．次の例は，全ての要素が 1 である配列を作成した後，全要素を 3.1416 で置き換えるものである．

例. 指定した値で既存の配列の要素を全て置き換える

```
>>> a = np.ones( (3,4) )  Enter        ← 3 行 4 列の 1 ばかりの行列を生成
>>> a  Enter                          ←内容の確認
array([[ 1., 1., 1., 1.],             ←結果表示
       [ 1., 1., 1., 1.],
       [ 1., 1., 1., 1.]])
>>> a.fill( 3.1416 )  Enter          全ての要素を 3.1416 で満たす
>>> a  Enter                          ←内容の確認
array([[ 3.1416, 3.1416, 3.1416, 3.1416],          ←結果表示」
       [ 3.1416, 3.1416, 3.1416, 3.1416],
       [ 3.1416, 3.1416, 3.1416, 3.1416]])
```

参考）疎な行列の作成

疎な行列（スパース行列：sparse matrix ）[64] を生成するには，予め 0 ばかりの要素の行列を生成しておき，添字を指定して要素の値を設定するという方法が良い．

全要素が同じ値である行列を新規に作成するには full 関数を用いる．

例. 全要素が 7 である行列の新規作成

```
>>> a = np.full( (3,6), 7 )  Enter   ← 3 行 6 列の 7 ばかりの行列を生成
>>> a  Enter                          ←内容の確認
array([[7, 7, 7, 7, 7, 7],            ←結果表示
       [7, 7, 7, 7, 7, 7],
       [7, 7, 7, 7, 7, 7]])
```

3.1.23.4 ベクトルの内積

dot 関数に 1 次元配列（ベクトル）を与えると，それらの**内積**を計算する．

例. ベクトルの内積

```
>>> v1 = np.array( [1,2,3] )  Enter   ←ベクトル v1 = (1, 2, 3) の作成
>>> v2 = np.array( [4,5,6] )  Enter   ←ベクトル v2 = (4, 5, 6) の作成
>>> np.dot(v1,v2)  Enter   ←内積 v1 · v2 の計算
np.int64(32)          ←内積の値
```

@ 演算子も内積を求めるのに使用できる．

例. @による内積の算出（先の例の続き）

```
>>> v1 @ v2  Enter   ←内積の計算
np.int64(32)          ←内積の値
```

inner 関数でも内積を求めることができる．

例. inner 関数による内積の計算（先の例の続き）

```
>>> np.inner(v1,v2)  Enter   ←内積 v1 · v2 の計算
np.int64(32)          ←内積の値
```

[64] 大部分の要素が 0 であるような行列.

3.1.23.5　対角成分，対角行列

diag 関数を用いると，行列の対角成分を抽出することができる．

例. 行列の対角成分の抽出

```
>>> a = np.arange(1,10).reshape(3,3)  Enter      ←サンプル行列の作成
>>> a  Enter                          ←内容の確認
array([[1, 2, 3],                     ←結果表示
       [4, 5, 6],
       [7, 8, 9]])
>>> np.diag(a)  Enter      ←対角成分の抽出
array([1, 5, 9])        ←結果表示（対角成分の配列が得られている）
```

diag にキーワード引数 ‘k=**開始インデックス**’ を与えると，対角成分を抽出する開始位置（先頭行のインデックス）を指定することができる．

例. 開始位置を指定して対角成分を抽出（先の例の続き）

```
>>> np.diag(a,k=1)  Enter     ←先頭行のインデックス 1（2 番目の要素）から抽出
array([2, 6])                  ←結果表示
>>> np.diag(a,k=2)  Enter     ←先頭行のインデックス 2（3 番目の要素）から抽出
array([3])                     ←結果表示
>>> np.diag(a,k=3)  Enter     ←先頭行のインデックス 3（存在しない）から抽出
array([], dtype=int32)         ←結果表示（空）
```

開始インデックスに負の値を与えることもできる．

例. 負の開始位置を指定して対角成分を抽出（先の例の続き）

```
>>> np.diag(a,k=-1)  Enter    ← 2 番目の行から抽出
array([4, 8])
>>> np.diag(a,k=-2)  Enter    ← 3 番目の行から抽出
array([7])
>>> np.diag(a,k=-3)  Enter    ← 4 番目の行は存在しないので
array([], dtype=int64)     ←空
```

diag 関数の引数に 1 次元の配列やリストを与えると，それを対角成分とする**対角行列**を生成する．

例. 対角行列の生成

```
>>> np.diag([11,22,33])  Enter     ←対角成分を与えて対角行列を生成
array([[11,  0,  0],                ←結果表示
       [ 0, 22,  0],
       [ 0,  0, 33]])
```

3.1.23.6　行列の転置

行列を転置するには transpose 関数を使用する．

例. 行列の転置

```
>>> a = np.array([[1,0,-2],[5,2,-3],[2,-1,3]])  Enter     ← 3 行 3 列の行列を生成
>>> a  Enter                          ←内容の確認
array([[ 1,  0, -2],                  ←結果表示
       [ 5,  2, -3],
       [ 2, -1,  3]])
>>> np.transpose(a)  Enter     ←転置の実行 (1)
array([[ 1,  5,  2],                ←処理結果
       [ 0,  2, -1],
       [-2, -3,  3]])
>>> a.T  Enter                ←転置の実行 (2)
array([[ 1,  5,  2],                ←処理結果
       [ 0,  2, -1],
       [-2, -3,  3]])
```

この例にあるように，T プロパティからも転置行列を取り出すことができる．

3.1.23.7　行列式と逆行列

linalg パッケージの det 関数で，行列式を，inv 関数で逆行列を求めることができる．

例. 行列式を求める（先の例の続き）

```
>>> np.linalg.det(a)  Enter      ←行列式を求める
20.999999999999989               ←結果表示
```

例. 逆行列を求める（先の例の続き）

```
>>> np.linalg.inv(a)  Enter                          ←逆行列を求める
array([[ 0.14285714, 0.0952381 , 0.19047619],        ←結果表示
       [-1.        , 0.33333333, -0.33333333],
       [-0.42857143, 0.04761905, 0.0952381 ]])
```

3.1.23.8　固有値と固有ベクトル

行列を転置するには linalg パッケージの eig 関数を使用する．

例. 行列の固有値と固有ベクトルを求める（先の例の続き）

```
>>> (e,ev) = np.linalg.eig(a)  Enter     ←固有値と固有ベクトルを求める
>>> e  Enter     ←固有値の配列の内容を確認
array([ 0.82433266+2.03631557j, 0.82433266-2.03631557j, 4.35133469+0.j ])  ←結果表示
>>> ev  Enter     ←固有ベクトルの配列の内容を確認
array([[-0.01760985-0.35786298j, -0.01760985+0.35786298j, -0.21323532+0.j ],  ←結果表示
       [-0.85880917+0.j        , -0.85880917-0.j        , -0.90931800+0.j ],
       [-0.36590772-0.01350281j, -0.36590772+0.01350281j, 0.35731146+0.j ]])
```

固有値は複素数のスカラーであり，与えられた行列が固有値を持つ場合はそれらを配列にして返す．個々の固有値にはそれぞれ対応する固有ベクトルがあり，それを行列（配列）にしたものを返す．

eig 関数は固有値と固有ベクトルを EigResult オブジェクトとして返すが，これの先頭要素が**固有値の配列**（1 次元），次の要素が**固有ベクトルの配列**（2 次元）である．上の例ではそれらを e，ev の変数に分割代入の手法で受け取っている．

eig 関数が返した固有値は，固有ベクトルの配列の列要素に順番に対応する．

3.1.23.9　行列のランク

行列のランクを求めるには linalg パッケージの matrix_rank 関数を使用する．

例. 行列のランクを求める：　matrix_rank 関数

```
>>> a1 = np.array([[2,-1,5],[-3,0,7],[9,4,-6]])  Enter     ←行列 a1 を生成
>>> a1  Enter                    ←内容の確認
array([[ 2, -1, 5],          ←結果表示
       [-3, 0, 7],
       [ 9, 4, -6]])
>>> np.linalg.matrix_rank( a1 )  Enter       ←ランクの算出
np.int64(3)          ←結果表示（ランクは 3）
>>> a2 = np.array([[2,-1,5],[-3,0,7],[-4,2,-10]])  Enter     ←行列 a2 を生成
>>> a2  Enter                    ←内容の確認
array([[ 2, -1, 5],          ←結果表示
       [ -3, 0, 7],
       [ -4, 2, -10]])
>>> np.linalg.matrix_rank( a2 )  Enter       ←ランクの算出
np.int64(2)          ←結果表示（ランクは 2）
```

3.1.23.10　線形方程式の求解

linalg パッケージの solve 関数を使用すると**線形方程式**（連立一次方程式）を解くことができる．例として，次のような線形方程式を考える．

$$\begin{pmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 8 \\ -11 \\ -3 \end{pmatrix}$$

係数行列（左辺の 3 × 3 行列）と右辺の定数ベクトルを solve 関数の引数に与えて解を求める例を次に示す.

例. 線形方程式の求解

```
>>> A = np.array([[2,1,-1],[-3,-1,2],[-2,1,2]])  Enter    ←係数行列
>>> b = np.array([8, -11, -3])  Enter    ←方程式右辺の定数ベクトル
>>> sol = np.linalg.solve(A, b)  Enter    ←求解処理
>>> sol  Enter    ←解の確認
array([ 2., 3., -1.])    ←解
```

解が $x = 2$, $y = 3$, $z = -1$ と得られていることがわかる.
係数行列が正則でない場合は解は得られず，エラーとなる.（次の例）

例. 係数行列が正則でないケース

```
>>> A = np.array([[1,2,3],[2,4,6],[3,6,9]])  Enter    ←係数行列（正則でない）
>>> b = np.array([6, 12, 18])  Enter    ←方程式右辺の定数ベクトル
>>> np.linalg.solve(A, b)  Enter    ←求解試みると
Traceback (most recent call last):          ←エラーとなる
  File "<stdin>", line 1, in <module>
    np.linalg.solve(A, b)
    ^^^^^^^^^^^^^^^^^^^^^
      :
   （途中省略）
      :
numpy.linalg.LinAlgError:  Singular matrix
```

3.1.23.11　複素共役行列

先の「3.1.22.3 共役複素数」（p.150）で解説した機能を用いて複素共役行列を求めることができる.

例. 複素共役行列を求める：　conjugate 関数

```
>>> ca = np.array([[1-2j,0],[0,-3+1j]])  Enter    ←行列 ca を生成
>>> ca  Enter                  ←内容の確認
array([[ 1.-2.j, 0.+0.j],        ←結果表示
       [ 0.+0.j, -3.+1.j]])
>>> np.conjugate(ca)  Enter              ←複素共役行列を求める
array([[ 1.+2.j, 0.-0.j],        ←結果表示
       [ 0.-0.j, -3.-1.j]])
```

3.1.23.12　エルミート共役行列

エルミート共役行列を求めるには conjugate 関数（conj 関数と同じ）と転置を組み合わせる.

例. エルミート共役行列を求める

```
>>> ca = np.array([[1-2j,-5+7j],[4-6j,-3+1j]])  Enter    ←行列 ca を生成
>>> ca  Enter                  ←内容の確認
array([[ 1.-2.j, -5.+7.j],        ←結果表示
       [ 4.-6.j, -3.+1.j]])
>>> np.conjugate( ca.T )  Enter    ←エルミート共役行列を求める
array([[ 1.+2.j, 4.+6.j],        ←結果表示
       [-5.-7.j, -3.-1.j]])
```

3.1.23.13　ベクトルのノルム

ベクトルのノルムを求めるには linalg パッケージの norm 関数を使用する.

例. ベクトルのノルムを求める：　norm 関数

```
>>> v = np.array([1,1])  Enter    ←ベクトル v を生成
>>> np.linalg.norm(v)  Enter    ←ノルム（長さ）を求める
np.float64(1.4142135623730951)          ←結果表示
```

3.1.24　構造化配列

NumPy の配列（ndarray）は基本的には数値演算の一括処理を目的としており，1 つの配列オブジェクトの全ての要素は同一の型に統一されている．しかし，データ処理の実務においては，型の異なる複数の列データ（カラムデータ）を束ねて管理することが多い[65]．NumPy においても，列データを単位とするデータ管理の方法を提供している．ここでは，そのような目的に使用する**構造化配列**について解説する．

構造化配列は，オブジェクトとしては ndarray であるが，作成時のデータ型の指定方法がに特徴がある．具体的には，配列作成時のキーワード引数 'dtype=' の指定方法に特徴がある．通常は次のように，全ての要素の型を統一した形で配列を作成する．

例. int64 型の 2 次元配列

```
>>> a = np.array([[1,2],[3,4]],dtype=np.int64)  Enter      ← dtype 引数で全要素の型を指定
>>> a  Enter      ←確認
array([[1, 2],
       [3, 4]])
```

これに対して，構造化配列の作成時には dtype 引数に特別な dtype オブジェクトを与える．dtype コンストラクタの書き方は次の通り．

書き方：　dtype([(名前 1, 型 1), (名前 2, 型 2),…])

1 つのタプルで 1 つの列（カラム）の名前と型を記述する．「名前」,「型」は文字列で与える．「型」は「3.1.2.3 型の表記に関する事柄」（p.54）で解説したものである．

例. dtype オブジェクトの作成

```
>>> dt = np.dtype([('name','U10'),('age','i1'),('height','f4'),('weight','f4')])  Enter
```

この例で作成している dtype オブジェクト dt は Unicode 文字 10 文字の「'name' カラム」，1 バイト（8 ビット）整数の「'age' カラム」，4 バイト（32 ビット）浮動小数点数の「'height' カラム」と「'weight' カラム」という構造を表す．この dtype オブジェクトの構造を反映した配列オブジェクトを作成する例を次に示す．

例. 構造化配列の作成（先の例の続き）

```
>>> a = np.array([('Nakamura',59,164,65),('Tanaka',32,159,53),('Satoh',44,171,85)], Enter
                 dtype=dt)  Enter        ←構造化配列の作成
>>> a  Enter      ←内容確認
array([('Nakamura', 59, 164., 65.), ('Tanaka', 32, 159., 53.), ('Satoh', 44, 171., 85.)],
      dtype=[('name', '<U10'), ('age', 'i1'), ('height', '<f4'), ('weight', '<f4')])
```

これで，列毎に型の異なる，名前を持った 3 列の構造化配列 a ができた．この配列は名前を指定して各列にアクセスすることができる．(次の例)

例. カラム（列）単位で配列にアクセス（先の例の続き）

```
>>> a['name']  Enter      ← 'name' 列の配列の参照
array(['Nakamura', 'Tanaka', 'Satoh'], dtype='<U10')      ← Unicode 文字列の配列
>>> a['age']  Enter      ← 'age' 列の配列の参照
array([59, 32, 44], dtype=int8)      ← 8 ビット整数の配列
>>> a['height']  Enter      ← 'height' 列の配列の参照
array([164., 159., 171.], dtype=float32)      ← 32 ビット浮動小数点数の配列
>>> a['weight']  Enter      ← 'weight' 列の配列の参照
array([65., 53., 85.], dtype=float32)      ← 32 ビット浮動小数点数の配列
```

構造化配列は，複数の 1 次元配列に名前を与えて束ねて管理する場合に便利である．また，後に解説する CSV データは，文字情報，数値情報など多様なデータ系列を保持するものであり，そのような情報構造を NumPy の配列として保持する場合に構造化配列は役立つ．

[65] pandas ライブラリにおける DataFrame などが典型例である．

3.1.25　入出力：配列オブジェクトのファイル I/O

実際のデータ解析においては，扱うデータを適宜ファイルに出力して保存したり，それを読み込んで処理をするという作業となる．ここでは，NumPy で扱うデータをファイルに保存する，あるいはファイルから読み込む方法について説明する．

配列オブジェクトのファイル I/O において 3 種類の形式を選ぶことができる．1 つはテキスト形式であり，CSV や TSV などの形式に沿った入出力により，他の処理系とのデータ連携が容易になる．

3.1.25.1　テキストファイルへの保存

NumPy の乱数発生機能を用いて作成した乱数表を CSV 形式[66] のテキストファイルとして保存する例を示す．

例. 100 行 10 列の一様乱数を CSV データとして保存する

```
>>> import numpy as np   Enter      ← NumPy の読込み
>>> rng = np.random.default_rng(0)   Enter      ← RNG を作成
>>> rnd = rng.random( (100,10) )   Enter      ←乱数表を生成
>>> np.savetxt('random.csv',rnd,delimiter=',')   Enter      ← CSV データとして保存
```

この例では savetxt 関数を使用して配列オブジェクトをファイルに保存している．この関数の使用方法は次の通りである．

書き方：　np.savetxt(**出力先ファイルのパス, 配列オブジェクト**, delimiter=**区切り文字**)

この方法で保存したテキストファイル 'random.csv' を Microsoft 社の表計算ソフト Excel で開いた例を図 94 に示す．

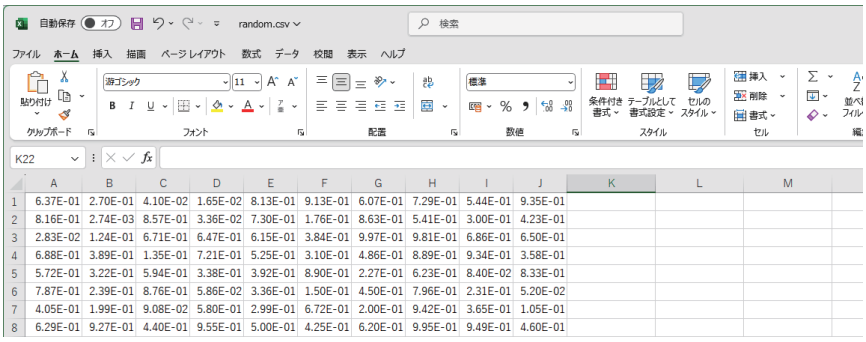


図 94: Microsoft 社の表計算ソフト Excel でファイルを開いたところ

■ 保存時の書式指定

保存する値の書式（桁数や型）を指定するには，savetxt 関数にキーワード引数 'fmt=' を与える．これに関して例を挙げて説明する．

サンプルとして次のような配列データ r を用意する．

例. サンプルデータの作成（先の例の続き）

```
>>> r = rng.normal( 50, 30, (3,3) )   Enter      ← 3 行 3 列の乱数配列
>>> print( r )   Enter      ←内容確認
[[ 52.50973937   76.89726501 116.1147237 ]
  [ 71.9533652     7.8701436  -29.70834842]      ←内容表示
  [ 47.16613494   52.14165306   15.14380082]]
```

この配列の各列に書式を指定して CSV ファイルとして保存する例を示す．

例. 書式付き保存（先の例の続き）

```
>>> np.savetxt('test01.csv',r,delimiter=',',   Enter
...             fmt=['%3d','%5.1f','%.2f'])   Enter
```

出力結果のファイル：test01.csv

```
1  52, 76.9,116.11
2  71,  7.9,-29.71
3  47, 52.1,15.14
```

この例のように，キーワード引数 'fmt=' には保存する配列の各列に対する書式設定のリストを与える．小数点数の書式は

'%**値の桁数. 小数点以下の桁数 f** '

[66] コンマ ',' で区切られたテキスト形式の表データである．詳しくは IETF が定める技術仕様 RFC 4180 を参照のこと．

とし，整数の書式は

’%値の桁数 d’

とする.「値の桁数」を省略すると桁数は自動的に調整される.

■ ヘッダーを付けて保存する

出力するファイルの先頭に見出し行を付けるには，savetxt 関数にキーワード引数 ’header=’ を与える．これに関して例を挙げて説明する.

例. ヘッダーを付けて保存（先の例の続き）

```
>>> np.savetxt('test02.csv',r,delimiter=',',  Enter
...     encoding='utf-8', Enter ←エンコーディングの指定
...     fmt=['%3d','%5.1f','%.2f'],  Enter
...     header='1 列目,2 列目,3 列目')  Enter
```

出力結果のファイル：test02.csv

```
1 # 1列目,2列目,3列目
2  52, 76.9,116.11
3  71,  7.9,-29.71
4  47, 52.1,15.14
```

この例では，日本語の見出し行を出力するためにエンコーディング指定「encoding='utf-8'」を与えている．ヘッダー行はデータとは無関係であるため，出力されたファイルのヘッダー行の先頭にはコメント行であることを意味する記号「#」が自動的に付けられる．ヘッダー行の先頭に「#」を付けずに出力するにはキーワード引数「comments=''」を与える．これに関する例を示す.

例. ヘッダー先頭に「#」を付けずに保存（先の例の続き）

```
>>> np.savetxt('test03.csv',r,delimiter=',',  Enter
...     encoding='utf-8',  Enter
...     fmt=['%3d','%5.1f','%.2f'],  Enter
...     header='1 列目,2 列目,3 列目',  Enter
...     comments='' )  Enter  ←「#」を付けない指定
```

出力結果のファイル：test03.csv

```
1 1列目,2列目,3列目
2  52, 76.9,116.11
3  71,  7.9,-29.71
4  47, 52.1,15.14
```

■ 数値以外のデータの保存

統計学における**カテゴリデータ**[67] は非数値の文字列（ラベル）として表現されることが一般的であり，そのような要素から成る配列をファイルに出力するには書式を

’%s’

とする．文字列の配列をファイルに出力する例を以下に示す.

例. サンプルデータ（文字列の配列）の作成（先の例の続き）

```
>>> catd = np.array([[' みかん','orange'],[' りんご','apple'],
                     [' バナナ','banana']])  Enter        ←文字列の配列
>>> print( catd )  Enter      ←内容確認
[[' みかん' 'orange']
  [' りんご' 'apple']      ←配列の内容
  [' バナナ' 'banana']]
```

例. サンプルデータのファイルへの出力（先の例の続き）

```
>>> np.savetxt('catd.csv', catd, delimiter=',', encoding='utf-8', fmt='%s',
                header='Japanese,English', comments='')  Enter    ←ファイルへの出力
```

これを実行した結果，右のようなファイル ‘catd.csv’ が作成される.

出力結果のファイル：catd.csv

```
1 Japanese,English
2 みかん,orange
3 りんご,apple
4 バナナ,banana
```

3.1.25.2　テキストファイルからの読み込み

関数 loadtxt を使用するとテキストファイルからデータを読み込むことができる.

書き方：　np.loadtxt(入力用ファイルのパス, delimiter=区切り文字)

ファイルの読込みが正常に終了すると，その内容を配列として返す．この関数は，基本的にファイルの内容を np.float64

[67] **カテゴリカルデータ**（categorical data），**質的データ**とも呼ばれる.

の形式で読み取る．読取り時のデータ型を指定する場合はオプションの引数「dtype=型」を与える．(注：キーワード引数「fmt=」は指定できない)

p.158 の例で変数 rnd に作成した乱数データ（100 行 10 列）を random.csv という CSV ファイルに保存したが，それを読み込む例を示す．

例. CSV 形式のデータを読み込む（先の例の続き）

```
>>> rnd2 = np.loadtxt('random.csv',delimiter=',')  Enter     ← CSV データの読み込み
```

これでファイル random.csv 内容が配列として rnd2 に受け取られた．p.158 の例で作成された変数 rnd の内容と一致することを確認する．(次の例)

例. 配列の内容を比較する（先の例の続き）

```
>>> chk = (rnd == rnd2)  Enter     ←元の配列との比較を行う
>>> chk  Enter     ←比較結果の確認
array([[ True, True, True, True, True, True, True, True, True, True],   ←比較結果
       [ True, True, True, True, True, True, True, True, True, True],
                          ⋮
                    (途中省略)
                          ⋮
       [ True, True, True, True, True, True, True, True, True, True]], dtype=bool)
>>> chk.all()  Enter     ←全ての要素が True かどうかを調べる
np.True_
```

この実行結果から，保存したデータを再度読み込んだものが，元の配列と同じ内容であることが検証できた．(行列同士の比較に関しては「3.1.26 行列の検査」を参照のこと)

注意） 上の例のように配列の同値性を「==」で検証する場合，両者が完全に同じでなければ結果は False となる．
実務上は allclose 関数[68] を使用して「ほぼ同じ」という形の判定も有効な場合がある．

■ 見出し行のある CSV 形式データを読み込む際の注意

CSV 形式データは 1 行目が見出し行となっていることがあり，そのようなデータを先の方法で読み込むとエラーが発生する．見出し行には NumPy の配列として扱えない型（文字列など）のデータが使用されることが多いというだけでなく，そもそも計算の対象とするデータではないので，見出し行は読み込みの際に無視する必要がある．次に示すように，loadtxt 関数のキーワード引数 skiprows= に整数値を指定すると，入力ファイルの先頭から指定した行数の読み込みを無視することができる．

書き方 (2)： np.loadtxt(**入力元ファイル名**, delimiter=**区切り文字**, skiprows=**スキップする行数**)

入力データの先頭 1 行が見出し行となっている場合は skiprows=1 を指定することで，正しく入力処理が行われる．

■ 数値以外のデータの読込み

数値以外のデータ（文字列）から構成されるカテゴリデータなどを読み込む際は loadtxt にキーワード引数「dtype=**データタイプ**」を与える．読み込むデータが Unicode 文字である場合は「dtype='U'」とする[69] ．先の例で作成した CSV データ 'catd.csv' を読み込む例を示す．

例. 文字列から成る CSV ファイルの読込み

```
>>> catd2 = np.loadtxt('catd.csv',delimiter=',', encoding='utf-8', skiprows=1,
                                   dtype='U' )  Enter     ←読込み処理
>>> print( catd2 )          ←内容確認
[['みかん' 'orange']
 ['りんご' 'apple']        ←配列の内容
 ['バナナ' 'banana']]
```

■ 特定の列（カラム）の読込み

CSV ファイルの内容の内，指定した特定の列（カラム）のみを抽出して読み込むことができる．作成したサンプルデータを用いてこのことを例示する．

[68]「3.1.2.9 配列の『近さ』の判定」(p.59) で解説している．

[69] dtype='object' と指定するとより安全である．

例. サンプルデータの作成

```
>>> rdat = np.zeros( (4,10) )  Enter    ←ここからサンプルデータの作成
>>> for i in range(10):  rdat[:,i] = np.arange(i*10,i*10+4)  Enter
...  Enter    ← for 文の終了
>>> print( rdat )  Enter    ←サンプルデータの内容確認
[[ 0.  10.  20.  30.  40.  50.  60.  70.  80.  90.]
  [ 1.  11.  21.  31.  41.  51.  61.  71.  81.  91.]
  [ 2.  12.  22.  32.  42.  52.  62.  72.  82.  92.]      ← 4 行 10 列の配列
  [ 3.  13.  23.  33.  43.  53.  63.  73.  83.  93.]]
```

例. サンプルデータの保存（先の例の続き）

```
>>> np.savetxt('rdat01.csv',rdat, delimiter=',', fmt='%3d' )  Enter    ←ファイルへの保存
```

この処理により次のような CSV ファイル 'rdat01.csv' が出来上がる.

出力結果のファイル：rdat01.csv

```
1    0, 10, 20, 30, 40, 50, 60, 70, 80, 90
2    1, 11, 21, 31, 41, 51, 61, 71, 81, 91
3    2, 12, 22, 32, 42, 52, 62, 72, 82, 92
4    3, 13, 23, 33, 43, 53, 63, 73, 83, 93
```

次に，このファイルから特定の列（カラム）を抽出して読み込む例を示す.

例. 'rdat01.csv' からインデックス位置が 1（左から 2 番目）の列を読み込む（先の例の続き）

```
>>> r1 = np.loadtxt('rdat01.csv', delimiter=',', usecols=1 ) Enter ←列指定読込み
>>> print( r1 )  Enter    ←読み取り結果の確認
[10.  11.  12.  13.]    ←結果表示
```

この例のように，loadtxt 関数にキーワード引数

usecols=列のインデックス

を与えると,「列のインデックス」に指定した列のみを読み込んで一次元配列の形で返す.

usecols 引数には複数のインデックスをリストや配列の形で与えることも可能であり，その場合は，指定した複数の列をまとめて読み込むことができる.

例. 複数の列をまとめて読み込む（先の例の続き）

```
>>> r135 = np.loadtxt('rdat01.csv', delimiter=',', usecols=[1,3,5] ) Enter ←複数列の読込み
>>> print( r135 )  Enter    ←読み取り結果の確認
[[10.  30.  50.]
  [11.  31.  51.]     ← CSV ファイルの中のインデックス位置 1,3,5 の列が得られている
  [12.  32.  52.]
  [13.  33.  53.]]
```

この例のように，loadtxt 関数にキーワード引数

usecols=$[n_1, n_2, n_3, \cdots]$

を与えると，$n_1, n_2, n_3, \cdots$ のインデックス位置の列を抽出して二次元配列として返す.

3.1.25.3 テキストファイル読込みの高度な方法

先の loadtxt は欠損値や文字列項目などに対応できず，そのような項目を含んだテキストファイルを読み込むとエラーとなる．例えば次のような CSV ファイル tattered01.csv を読み込む事例について考える.

サンプルデータ：tattered01.csv

```
1  #ラベル, 時間(sec), 温度(℃), 圧力(Pa)
2         , 0,         20.1,     101325
3         , 1,         20.3,     101300
4  event1 , 2,         N/A,      101280
5         , 3,         20.5,     N/A
6  event2 , 4,         20.7,     N/A
7         , 5,         20.6,     101200
```

これを loadtxt 関数で読込みを試みるとエラーとなる.

例. loadtxt で起こるエラー（先の例の続き）

```
>>> dat = np.loadtxt('tattered01.csv',delimiter=',',encoding='utf-8',skiprows=1)  Enter
Traceback (most recent call last):              ←上の方法で読込みを試みるとエラーとなる
  File "<stdin>", line 1, in <module>
    dat = np.loadtxt('tattered01.csv',delimiter=',',encoding='utf-8',skiprows=1)
          :
        （途中省略）
          :
ValueError:  could not convert string '        ' to float64 at row 0, column 1.
```

loadtxt 関数に対して，genfromtxt 関数は数値以外のデータの読込みにも対応している．

例. genfromtxt による CSV ファイルの読込み（先の例の続き）

```
>>> dat = np.genfromtxt('tattered01.csv',delimiter=',',encoding='utf-8')  Enter
>>> dat  Enter      ←読取り結果の確認
array([[        nan, 0.00000e+00, 2.01000e+01, 1.01325e+05],
       [        nan, 1.00000e+00, 2.03000e+01, 1.01300e+05],
       [        nan, 2.00000e+00,         nan, 1.01280e+05],
       [        nan, 3.00000e+00, 2.05000e+01,         nan],
       [        nan, 4.00000e+00, 2.07000e+01,         nan],
       [        nan, 5.00000e+00, 2.06000e+01, 1.01200e+05]])
```

このように，数値でない項目は非数（nan）として読込み処理を終えている．また，コメント行も自動的に認識してスキップしていることがわかる．genfromtxt 関数は概ね loadtxt 関数と同様の引数を取る．ただし，先頭行を明示的にスキップする場合は skiprows 引数ではなく「skip_header=スキップする行数」を与える．

■ 構造化配列として CSV ファイルの内容を読み込む方法

様々なデータ型の値から構成される CSV ファイルの内容は**構造化配列**として入力すると良い．構造化配列を扱うには，「3.1.24 構造化配列」（p.157）で解説したように，dtype オブジェクトで各列（カラム）の名前と型の構造を定義する．そして，CSV データを読み込む際に dtype オブジェクトに沿って各カラムの内容の型を決定する．例えば，先の tattered01.csv を読み込むための dtype オブジェクトを次のような形式で作成する．

例. dtype オブジェクトの作成（先の例の続き）

```
>>> dt = np.dtype([('label','U10'),('time','i4'),('temperature','f4'),('pressure','i4')])  Enter
```

次に，この dtype オブジェクトに沿った形で CSV ファイルの内容を読み込む．

例. CSV ファイルの内容を構造化配列として読み込む（先の例の続き）

```
>>> dat = np.genfromtxt('tattered01.csv',delimiter=',',encoding='utf-8',  Enter
...                          dtype=dt,autostrip=True)  Enter
```

これで，CSV ファイルの内容が構造化配列 dat として得られた．構造化配列はカラム名を指定して，各列の配列にアクセスすることができる．（次の例）

例. カラム（列）単位で構造化配列を参照する（先の例の続き）

```
>>> dat['label']  Enter     ←カラム名 'label' を参照
array(['', '', 'event1', '', 'event2', ''], dtype='<U10')
>>> dat['time']  Enter     ←カラム名 'time' を参照
array([0, 1, 2, 3, 4, 5], dtype=int32)
>>> dat['temperature']  Enter     ←カラム名 'temperature' を参照
array([20.1, 20.3,  nan, 20.5, 20.7, 20.6], dtype=float32)
>>> dat['pressure']  Enter     ←カラム名 'pressure' を参照
array([101325, 101300, 101280,     -1,     -1, 101200], dtype=int32)
```

浮動小数点数のカラムに含まれる異常値は nan，整数のカラムに含まれる異常値は -1 に変換される．

genfromtxt の役立つ引数を表 35 に挙げる．

3.1.25.4　バイナリファイルへの I/O

関数 save, load を使用することでバイナリファイルに対する I/O ができる．save 関数で配列をファイルに保存すると，NumPy の配列データとしての情報が属性を含めて全て保存される．

書き方：　np.save(保存先ファイルのパス, 配列データ)

表 35: genfromtxt の引数（一部）

引　　数	解　　説
missing_values=値	データが欠損している場合に「値」で置き換える.
filling_values=値	異常なデータを「値」で置き換える.
names=True	CSV のヘッダー行から各カラム（列）の名前を付ける.
dtype=dtype オブジェクト	dtype オブジェクトで各カラムの名前と型を決定する.
skip_header=行数	先頭から「行数」だけ読み飛ばす.
skip_footer=行数	末尾の「行数」分は読まない.
autostrip=True	文字列の前後になる空白を削除する.
comments=文字	「文字」で始まる行はコメントとする.
delimiter=文字	「文字」を区切り文字として項目を分離する.

この方法で保存されるファイルは NumPy 独自の形式であり，ファイル名の拡張子としては「*.npy」とする（以後，この形式のファイルを「npy ファイル」と呼ぶ）ことが推奨されている．save 関数の戻り値はない．(None)

load 関数は読み取ったファイルの内容を，元の形式の配列として返す．

書き方：　np.load(入力元ファイルのパス)

p.158 の例で変数 rnd に作成した乱数データ（100 行 10 列）を save 関数によって random.npy というバイナリファイルに保存する例を示す．

例. バイナリファイルに対する I/O（先の例の続き）

```
>>> np.save('random.npy',rnd)   Enter    ←配列 rnd をファイル 'random.npy' に保存
>>> rnd2 = np.load('random.npy')   Enter    ←ファイル 'random.npy' から読み込み
>>> np.allclose(rnd2,rnd)   Enter    ←元の配列との比較を試みる
True        ←両者は等しい
```

この例でも保存と読み込みが正常に行われたことがわかる．

3.1.25.5　書庫形式での保存と読込み

NumPy には ZIP 書庫のフォーマットでデータをファイルとして入出力する機能がある．savez 関数は，複数の配列オブジェクトにアクセス用の名前を付与して 1 つのファイルに束ねて保存する．

書き方：　np.savez(出力先ファイルのパス, 名前 1=配列 1, 名前 2=配列 2, …)

複数の配列オブジェクト「配列 1」,「配列 2」…にそれぞれ「名前 1」,「名前 2」，…と名称を与えて束ね，ファイルに保存する．ファイル名の拡張子としては「*.npz」とする（以後，この形式のファイルを「npz ファイル」と呼ぶ）ことが推奨されている．この関数は値を返さない．(None)

load 関数は npz ファイルを開くことができる．

書き方：　np.load(npz ファイルのパス)

先に解説した load 関数であるが，npz ファイルのパスを引数に与えると，そのファイルの内部にアクセスするための NpzFile オブジェクトを返す．この段階ではまだファイルの内容は読み込まれない．NpzFile オブジェクトから必要な配列を取り出す際に，実際に読込み処理が行われる．必要な配列を取り出すには NpzFile オブジェクトに対して，配列データの名前をキーとして指定する．

書き方：　NpzFile オブジェクト [配列データの名前]

「配列データの名前」は，savez 関数でデータを保存する際に付与した配列の名前である．npy ファイルと異なり，npz ファイルは，その使用が終わった時点で close メソッドで閉じる必要がある．

複数の配列データを作成して，それらを npz ファイルに保存する例を次に示す．

例. サンプルデータを作成して npz ファイルに保存する

```
>>> import numpy as np  Enter      ← NumPy の読込み
>>> rng = np.random.default_rng(0)  Enter      ← RNG 作成
>>> d11 = rng.random( size=(100000) )  Enter      ←データの作成 (1)：一様乱数
>>> d12 = rng.normal( size=(100000) )  Enter      ←データの作成 (2)：正規乱数
>>> d13 = rng.lognormal( 0,0.5,size=(100000) )  Enter      ←データの作成 (3)：対数正規乱数
>>> np.savez( 'datset1.npz', uniform=d11, normal=d12, lognormal=d13 )  Enter
```

この例では，3 種類の乱数データ d11，d12，d13 を作成して，それらを束ねて datset1.npz という npz ファイルとして保存している．この際に d11 に 'uniform'，d12 に 'normal，d13 に 'lognormal' という名前を付与している．

この npz ファイルを開き，それぞれのデータを読み込む例を次に示す．

例. npz ファイルを開いて内部の配列データを取り出す（先の例の続き）

```
>>> dset = np.load( 'datset1.npz' )  Enter      ← npz ファイルを開く
>>> d21 = dset['uniform']  Enter      ←データの読込み (1)
>>> d22 = dset['normal']  Enter      ←データの読込み (2)
>>> d23 = dset['lognormal']  Enter      ←データの読込み (3)
>>> dset.close()  Enter      ← npz ファイルを閉じる
```

この例では，npz ファイルを開き，得られた NpzFile オブジェクトを dset に受け取っている．これに対して配列の名前をキーに指定して，ファイル内の 3 つの内容を読み取り，それぞれ d21，d22，d23 に受け取っている．この段階で，ファイルの内容が実際に読み込まれる．読込み処理が終わった後で NpzFile オブジェクトを閉じている．

以上の処理で得られた d21，d22，d23 のヒストグラムを表示する例を次に示す．

例. 先の例で読み取ったデータを可視化（先の例の続き）

```
>>> import matplotlib.pyplot as plt  Enter
>>> fig,ax = plt.subplots(1,3,figsize=(15,6))  Enter
>>> fig.subplots_adjust(wspace=0.26)  Enter
>>> g1 = ax[0].hist(d21,bins=15)  Enter
>>> t11 = ax[0].set_xlabel('value')  Enter
>>> t12 = ax[0].set_title('uniform')  Enter
>>> g2 = ax[1].hist(d22,bins=15)  Enter
>>> t21 = ax[1].set_xlabel('value')  Enter
>>> t22 = ax[1].set_title('normal')  Enter
>>> g3 = ax[2].hist(d23,bins=35)  Enter
>>> r3 = ax[2].set_xlim(0,4)  Enter
>>> t31 = ax[2].set_xlabel('value')  Enter
>>> t32 = ax[2].set_title('lognormal')  Enter
>>> plt.show()  Enter
```

これによって図 95 のようなグラフが表示される．

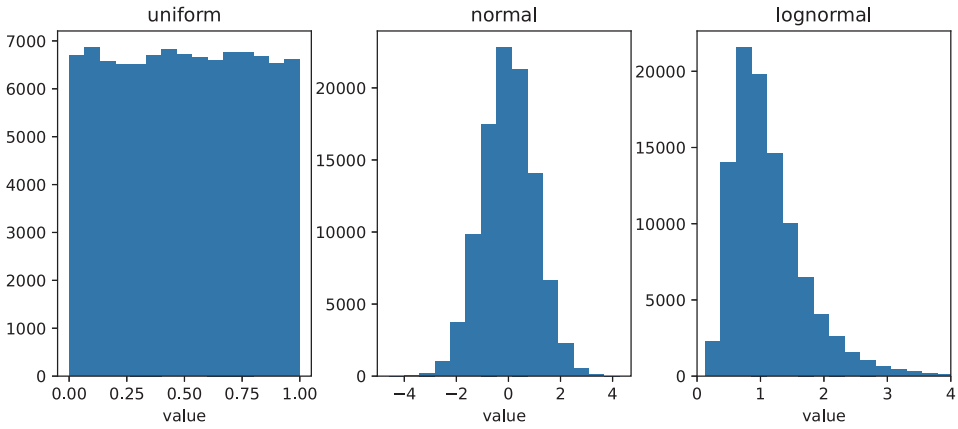


図 95: 3 種類のヒストグラムが表示される

■ データ保存時の圧縮処理について

先に解説した savez 関数では，保存時のデータ圧縮は行わず，ZIP 書庫の「書庫」としての機能のみを利用する．実際に保存時に ZIP 圧縮を施すには，savez_compressed 関数を使用（引数の指定方法は savez と同じ）する．

3.1.25.6 配列をバイトデータに変換して入出力に使用する方法

配列オブジェクトに対して tobytes メソッドを実行することで，それをバイト列に変換することができる．

例. 配列をバイト列に変換する

```
>>> a = np.array( [1,2,3,4,5,1,2,3,4,5], dtype='int16' )  Enter    ← 1 次元配列を作成
>>> b = a.tobytes()  Enter    ←それをバイト列のデータに変換する
>>> len(b)  Enter    ←バイト列の長さを調べる
20                   ← 20 バイトである
>>> type(b)  Enter    ←型を調べる
<class 'bytes'>      ←バイト列であることが確認できる
```

この例で得られたバイト列 b をファイルに保存する例を示す．

例. 作成したバイト列をファイルに保存する（先の例の続き）

```
>>> fo = open( 'array01.dat', 'wb' )  Enter    ←出力用ファイルをバイナリモードで開く（作成する）
>>> fo.write(b)  Enter    ←そのファイルに対して，先に作成したバイト列 b を出力する
20                  ← 20 バイト出力した
>>> fo.close()  Enter    ←ファイルを閉じる
```

この処理で，バイト列 b がファイル 'array01.dat' に出力される．

次に，改めてこのファイルから内容を読み取って，それを配列オブジェクトに変換する処理の例を示す．

例. バイナリデータを読み込んで配列オブジェクトにする（先の例の続き）

```
>>> fi = open( 'array01.dat', 'rb' )  Enter    ←ファイルを入力用にバイナリモードで開く
>>> buf = fi.read()  Enter    ←そのファイルからバイトデータを（全て）読み取る
>>> fi.close()  Enter    ←ファイルを閉じる
>>> a2 = np.frombuffer( buf, dtype='int16' )  Enter    ←読み取ったバイトデータを配列に変換する
>>> print( a2 )  Enter    ←出来上がった配列を表示する
[1 2 3 4 5 1 2 3 4 5]    ←結果表示：最初に作成した配列と同じものが得られている
```

この例で示したように frombuffer を使用することで，バイト列を配列オブジェクトに変換することができる．

書き方： frombuffer(バイト列, dtype=データ型)

結果として，最初に作成した配列オブジェクトをファイルとして保存し，それを読み取って別の配列オブジェクトとして復元することができていることがわかる．

重要）frombuffer は対象バイト列の**バッファビュー**を返す．従って元のバイト列が失われると frombuffer で得られた配列の中身も失われるので注意すること．

注意）

他のプラットフォームとの間でデータを交換する際にはデータのバイトオーダーに注意すること．バイトオーダーを明に指定して frombuffer を実行する場合は，キーワード引数 'dtype=' に指定するものを「3.1.2.3 型の表記に関する事柄」（p.54）で解説した内部表現形式にすると良い．また，tobytes には型やバイトオーダーを変換する機能は無いので，それを行うには astype メソッドなどを使用して変換し，それを tobytes でバイト列にする必要がある．

応用例）

frombuffer を使ってバイト列を配列に変換する方法を応用すると，WAV 形式サウンドファイルから波形データを読み取って，それを NumPy の配列オブジェクトに変換する[70] ことができる．

3.1.25.7 配列を直接的にファイル入出力に使用する

NumPy の配列の構造は，実メモリ上では基本的には C 言語の配列と互換性があるので，C 言語で作成したシステムとのデータ交換が容易である．NumPy の配列オブジェクトを直接的にファイルに出力するには tofile メソッドを使

[70] これに関しては拙書「Python3 入門」で解説しています．

用するのがより簡便である．

書き方：　配列オブジェクト.tofile(出力先ファイルのパス)

「配列オブジェクト」をバイナリ形式のファイルとして出力する．このメソッドは，テキストファイルに出力する機能も持つが，テキストファイルへの出力は先に解説した savetxt 関数を使用することが推奨される．またこのメソッドは，型やバイトオーダーを変換する機能を持たないので，その必要がある場合は astype メソッドなどを使用して変換すること．

以下に具体例を挙げ，NumPy の配列をバイナリファイルとして出力し，それを C 言語プログラムで読み取る例を示す．

例. サンプルデータの作成と保存

```
>>> import numpy as np  Enter
>>> a = np.arange(15,dtype=np.float64).reshape(3,5)  Enter    ←サンプルデータの作成
>>> a  Enter    ←内容確認
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.],
       [10., 11., 12., 13., 14.]])
>>> a.tofile('BinFileTest.bin')        ←バイナリファイルとして保存
```

この処理によって，配列 a の内容がファイル 'BinFileTest.bin' に出力される．このファイルの内容を読み取って標準出力に出力する C 言語のプログラム BinFileTest02.c を示す．

プログラム：BinFileTest02.c

```
1  #include    <stdio.h>
2
3  int main() {
4      FILE    *f;
5      double  x[3][5];
6      int     i, j;
7      f = fopen("BinFileTest.bin","rb");
8      fread(x,sizeof(double),3*5,f);
9      fclose(f);
10     for ( i = 0; i < 3; i++ ) {
11         for ( j = 0; j < 5; j++ ) {
12             printf("%5.1f,",x[i][j]);
13         }
14         printf("\n");
15     }
16 }
```

左のプログラムをコンパイルして実行すると，標準出力に

```
  0.0,  1.0,  2.0,  3.0,  4.0,
  5.0,  6.0,  7.0,  8.0,  9.0,
 10.0, 11.0, 12.0, 13.0, 14.0,
```

と出力される．

これは，ファイル 'BinFileTest.bin' を作成した Python 言語処理システムと同じ計算機環境でコンパイルして実行したものである．Python 言語と C 言語の実行環境が異なる場合はそれぞれの計算機アーキテクチャのバイトオーダーに注意すること．

次に，C 言語で作成したバイナリファイルを Python 側で受け取って NumPy の配列にする例を示す．

プログラム：BinFileTest01.c

```
1  #include    <stdio.h>
2
3  int main() {
4      FILE    *f;
5      double  x[3][5], n=0.0;
6      int     i, j;
7      for ( i = 0; i < 3; i++ ) {
8          for ( j = 0; j < 5; j++ ) {
9              x[i][j] = n++;
10             printf("%5.1f,",x[i][j]);
11         }
12         printf("\n");
13     }
14     f = fopen("BinFileTest.bin","wb");
15     fwrite(x,sizeof(double),3*5,f);
16     fclose(f);
17 }
```

左のプログラムをコンパイルして実行すると，標準出力に

```
  0.0,  1.0,  2.0,  3.0,  4.0,
  5.0,  6.0,  7.0,  8.0,  9.0,
 10.0, 11.0, 12.0, 13.0, 14.0,
```

と出力される．

このプログラムは，double 型数値の配列 x の内容をファイル 'BinFileTest.bin' に出力する．

この後，ファイル 'BinFileTest.bin' を Python 言語処理システムで読み取って NumPy の配列に変換する例を示す．

例. バイナリファイルの内容を NumPy の配列にする

```
>>> import numpy as np  Enter
>>> a = np.fromfile('BinFileTest.bin', dtype=np.float64)  Enter    ←読込み
>>> a = a.reshape(3, 5)  Enter    ← 3 行 5 列に変形
>>> print(a)  Enter    ←内容確認
[[ 0.   1.   2.   3.   4.]
 [ 5.   6.   7.   8.   9.]
 [10.  11.  12.  13.  14.]]
```

この例においても，同じ計算機環境で，C 言語プログラムと Python 言語処理システムを実行した．それぞれのプログラムを異なる計算機環境で実行する場合は，取り扱うバイナリファイルのバイトオーダーには注意すること．

3.1.26 行列の検査

3.1.26.1 全て真，少なくとも 1 つは真： all，any

all メソッドは配列の要素が全て真（True）の場合に真を返す．

例. all メソッドによる「全て真である」判定

```
>>> b = np.full( (2,3), True )  Enter    ←全て True の 2 次元配列
>>> b  Enter          ←内容確認
array([[ True, True, True],          ←配列の内容
       [ True, True, True]])
>>> b.all()  Enter    ←全て真（True）かの検査
np.True_    ←検査結果
```

当然のことではあるが，配列の要素に 1 つでも False が含まれると all の結果は偽（False）となる．

例. 1 つの偽を含む配列に対する all（先の例の続き）

```
>>> b[0,1] = False  Enter    ←配列の要素の 1 つを False にする
>>> b  Enter          ←内容確認
array([[ True, False, True],          ←配列の内容（False が含まれている）
       [ True, True, True]])
>>> b.all()  Enter    ←全て真（True）かの検査
np.False_    ←検査結果
```

注意）処理環境や NumPy のバージョンによって，真理値が `np.True_`，`np.False_` と表示されることがある．

any メソッドは配列の要素が 1 つでも真（True）を含む場合に真を返す．

例. any メソッドによる「真が含まれる」判定

```
>>> b = np.full( (2,3), False )  Enter    ←全て False の 2 次元配列
>>> b  Enter          ←内容確認
array([[False, False, False],          ←配列の内容
       [False, False, False]])
>>> b.any()  Enter    ←真（True）が含まれるかの検査
np.False_    ←検査結果
>>> b[0,1] = True  Enter    ←配列の要素の 1 つを True にする
>>> b  Enter          ←内容確認
array([[False, True, False],          ←配列の内容（True が含まれている）
       [False, False, False]])
>>> b.any()  Enter    ←真（True）が含まれるかの検査
np.True_    ←検査結果
```

any も行列の比較に応用することができる．次の例は 2 つの行列を比較するもので，対応する位置の要素が等しいものが 1 箇所でもあるかどうかを判定するものである．

例. any メソッドを使用した「少なくとも 1 つの要素が True である」判定

```
>>> a1 = np.array([[1,2],[3,4]])  Enter        ←配列 a1 を生成
>>> a2 = np.array([[1,2],[5,4]])  Enter        ←配列 a2 を生成
>>> a3 = np.array([[-1,-2],[-5,-4]])  Enter    ←配列 a3 を生成
>>> (a1==a2).all()  Enter      ← a1 と a2 の要素が全て等しいか検査
np.False_                        ←異なるものがあることがわかる
>>> (a1==a2).any()  Enter      ← a1 と a2 の要素で互いに等しいものが 1 つでもあるかを検査
np.True_                         ←等しい要素があることがわかる
>>> (a2==a3).any()  Enter      ← a2 と a3 の要素で互いに等しいものが 1 つでもあるかを検査
np.False_                        ←等しい要素がないことがわかる
```

all，any は数値の配列に対しても使用することができる．その場合，ゼロを False，非ゼロを True とみなす．

例. 数値配列に対する all の判定

```
>>> a = np.array([3,2,1,2,3])  Enter    ←全て非ゼロの配列
>>> a.all()  Enter    ←全て非ゼロかの判定
np.True_
>>> a = np.array([3,2,1,0,1,2,3])  Enter    ←ゼロを含む配列
>>> a.all()  Enter    ←全て非ゼロかの判定
np.False_
```

3.1.27　画像データの扱い

NumPy では画素の RGB 値の配列（図 96 参照）を画像として表示することができる．

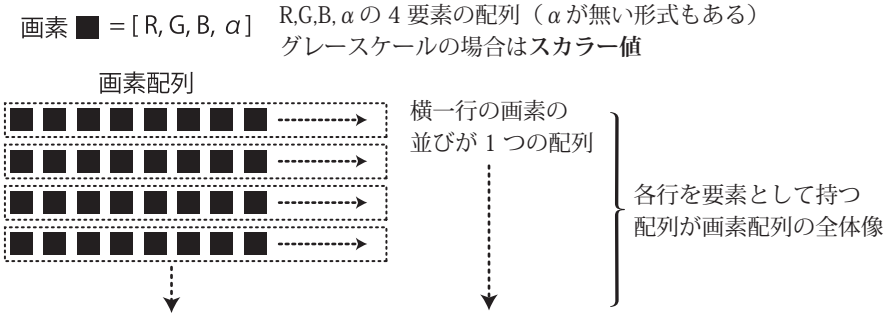


図 96: NumPy における画素の配列

画素の配列は全体としては 3 次元の構造である．このようなデータを画像として表示する方法について，例を示しながら説明する．

図 97 のような画像を構成する画素の配列を作成することを考える．

図 97: サンプル：3 色の画像

この画像を構成する画素の配列は [赤, 赤,…, 緑, 緑,…, 青, 青,…] の行を多数束ねた配列であると見ることができる．すなわち，

```
[ [赤, 赤,…, 緑, 緑,…, 青, 青,…]
  [赤, 赤,…, 緑, 緑,…, 青, 青,…]
  [赤, 赤,…, 緑, 緑,…, 青, 青,…]]
                ⋮
```

のような配列を作成することになる．(次の例)

例. 図 97 を画素の配列として作成する

```
>>> import numpy as np   Enter      ← NumPy の読込み
>>> r = [[255,0,0,255]]*100   Enter      ←赤の画素の配列
>>> g = [[0,255,0,255]]*100   Enter      ←緑の画素の配列
>>> b = [[0,0,255,255]]*100   Enter      ←青の画素の配列
>>> imA = np.array([r+g+b]*200,dtype=np.uint8)  Enter ←上記 3 つを連結してそれを 200 行分作成
```

これで imA に画素の配列ができた．標準的な各色 8 ビット（24 ビットカラー）の構成にするため，符号なし 8 ビット整数の配列として imA を作成している．次にこれを matplotlib によって可視化する．(次の例)

例. 作成した配列を画像として表示する．(先の例の続き)

```
>>> import matplotlib.pyplot as plt   Enter      ← matplotlib の読込み
>>> f = plt.figure()   Enter      ←描画処理の開始
>>> g = plt.imshow( imA )   Enter      ←画素の配列を表示するメソッド
>>> plt.show()   Enter      ←描画の実行
```

この結果，imshow メソッドによって図 98 のように表示される．

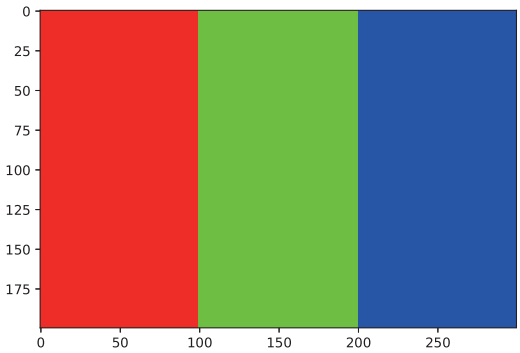


図 98: 画素の配列を表示した例

3.1.27.1　画素の配列から画像データへの変換（PIL ライブラリとの連携）

Pillow ライブラリ（p.22）を用いると NumPy で作成した画素の配列を画像データに変換することができる．

例. NumPy の画素配列を Pillow の Image オブジェクトに変換する（先の例の続き）

```
>>> from PIL import Image    Enter      ← Pillow ライブラリから Image クラスを読み込む
>>> im = Image.fromarray( imA ) Enter ←画素配列を Pillow の Image オブジェクトに変換
>>> im.show()    Enter      ←表示処理
>>> im.save('rgb01.png')    Enter      ←画像をファイルに保存
```

処理の手順としては，NumPy の画素配列を **8 ビット符号無し整数**の配列に変換（uint8 メソッド）し，それを Pillow の fromarray メソッドによって Image オブジェクトに変換する．この例では，出来上がった Image オブジェクトを Image クラスの show メソッドでディスプレイに表示している．（画像ビューワが開いて図 97 のような画像が表示される）

3.1.27.2　画像データから画素の配列への変換（PIL ライブラリとの連携）

NumPy の asarray メソッドを使用すると，Pillow の Image オブジェクトを NumPy の画素配列に変換することができる．（次の例参照）

例. Image オブジェクトを画素配列に変換（先の例の続き）

```
>>> imA2 = np.asarray(im)    Enter      ← Image オブジェクトを画素配列に変換
>>> f = plt.figure()    Enter      ←描画処理の開始
>>> g = plt.imshow( imA2 )    Enter      ←画素の配列を表示
>>> plt.show()    Enter      ←描画の実行
```

この結果，図 98 のように表示される．

課題）図 99 のような画像を，画素配列として作成して matplotlib の imshow メソッドで表示せよ．

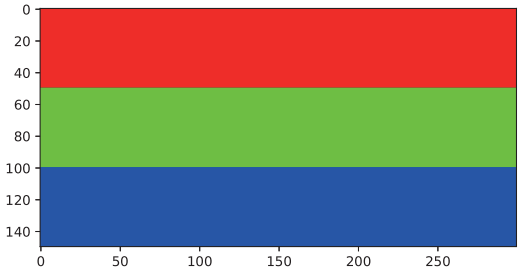


図 99: 作成する画像

Image オブジェクトをファイルに保存する方法など，Pillow に関することは「1.2 Pillow」（p.22）を参照のこと．

3.1.27.3 OpenCV における画素の配列

OpenCV ライブラリ[71] による画像の扱いでは，画素の RGB 成分の並びが異なるので注意が必要である．OpenCV では，例えば次のようにして画像ファイルを読み込むことができる．

例. OpenCV ライブラリの imread メソッドによる画像の読込み（先の例の続き）

```
>>> import cv2  Enter      ← OpenCV ライブラリの読込み
>>> imcv = cv2.imread( 'rgb01.png', cv2.IMREAD_UNCHANGED ) Enter ←画像ファイルの読込み
>>> type( imcv )  Enter      ←データ型を調べる
<class 'numpy.ndarray'>     ← NumPy の配列である
>>> imcv.shape  Enter      ←配列の形状を調べる
(200, 300, 4)     ←配列の形状
```

これは，先の例で作成した画像ファイル 'rgb01.png' を読み込む例であるが，このように imread メソッドで画像ファイルを読み込むと，直接 NumPy の配列（ndarray）の形で画素が得られる．ただし，画素の色の成分の並びが **B,G,R,** α（青，緑，赤，α）の順であるので，次のようにして表示すると，元の表示（図 98）と異なる結果となる．

例. 得られた画素の配列を表示する（先の例の続き）

```
>>> g = plt.imshow( imcv )  Enter      ←画像を表示
>>> plt.show()  Enter      ←作図の実行
```

この結果，図 100 のように表示されてしまう．

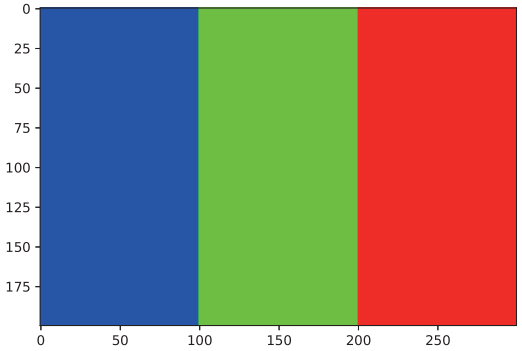


図 100: OpenCV の imread メソッドで読み込んだ配列を表示

上の例で得られた画素配列 imcv の色成分の順序を R,G,B, α にするには，例えば次のような変換処理を行う．

例. 色成分の並びの変更（先の例の続き）

```
>>> imcv2 = imcv[:,:,[2,1,0,3]]  Enter      ←変換処理
```

こうすることで，imcv の画素の縦と横の並び順はそのままで，画素の色成分の部分の値の並びを変更することができる．（変換結果を imcv2 に得ている）

この処理で得られた imcv2 を表示する例を次に示す．

例. 色成分の順序を修正した画像を表示（先の例の続き）

```
>>> g = plt.imshow( imcv2 )  Enter      ←修正した画像を表示
>>> plt.show()  Enter      ←作図の実行
```

この結果，図 101 のように表示される．

3.1.27.4 サンプルプログラム：画像の三色分解

PIL ライブラリ経由で読み取った画像を 3 つの原色（RGB）に分解して，それぞれ表示するプログラム imshow01.py を示す．

プログラム： imshow01.py

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from PIL import Image
4
```

[71] p.1「1.1 OpenCV」で解説している．

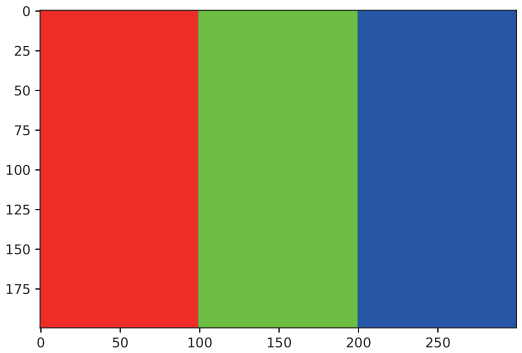


図 101: 色成分の順序を修正した画像

```
5  IM = Image.open('ImgCat.jpg')    # 画像読み込み
6  (R, G, B) = IM.split()           # 3色分解
7
8  # NumPy 配列に変換
9  im = np.asarray(IM)
10 r  = np.asarray(R)
11 g  = np.asarray(G)
12 b  = np.asarray(B)
13 gr = np.asarray(IM.convert("L"))    # グレースケール配列
14
15 # 表示（オリジナルとグレースケール）
16 fig, axs = plt.subplots(1, 2, figsize=(8, 4))
17 axs[0].imshow(im)
18 axs[0].set_title('Original')
19 axs[1].imshow(gr, cmap='gray')
20 axs[1].set_title('Grayscale')
21 plt.show()
22
23 # 表示（R, G, B 各チャンネル）
24 fig, axs = plt.subplots(1, 3, figsize=(12, 4))
25 axs[0].imshow(r, cmap='Reds');   axs[0].set_title('Red')
26 axs[1].imshow(g, cmap='Greens'); axs[1].set_title('Green')
27 axs[2].imshow(b, cmap='Blues');  axs[2].set_title('Blue')
28 plt.show()
```

解説：

PIL の Image オブジェクトに対して split メソッドを実行（6 行目）すると 3 色（RGB）に分解された 3 つの Image オブジェクトが返される．それらを配列に変換（9〜12 行目）し，imshow で表示している．

このプログラムを実行すると図 102 のように表示される．

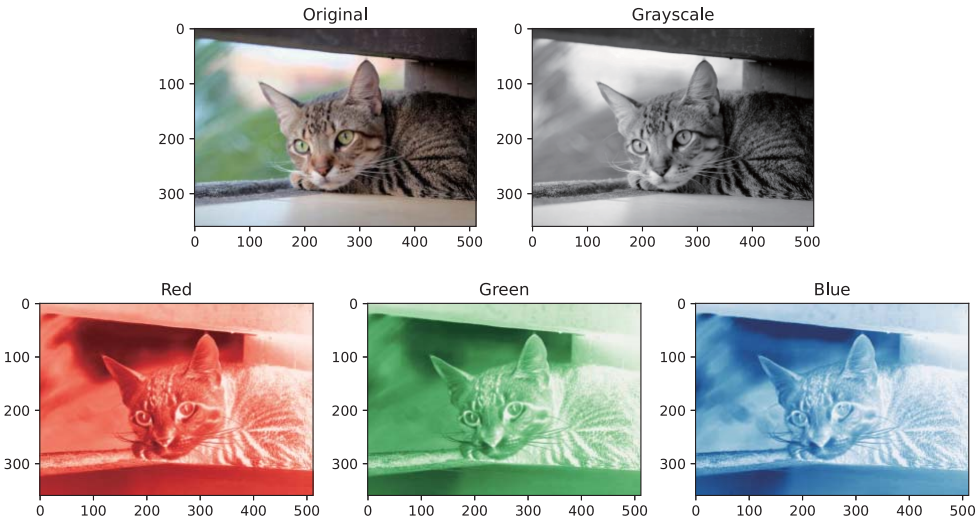


図 102: グレースケール表示と 3 色分解した画像の表示

3.1.28　図形の描画：matplotlib.patches

matplotlib.patches は各種の幾何学図形を表示する機能を提供する．ここでは，matplotlib.patches の描画機能の内の一部[72] について解説する．

ここで解説する描画機能は matplotlib によるものであり，使用に際して下記のように matplotlib.pyplot と matplotlib.patches を読み込む必要がある．

```
import matplotlib.pyplot as plt
import matplotlib.patches as pat
```

これにより，幾何学図形の API を接頭辞「pat.」をつけて呼び出すことができる．

図形描画の手順：

matplotlib.patches の各種図形クラスのオブジェクトを作成し，それを Axes （p.95）に対して add_patch メソッドで描画する．

3.1.28.1　四角形，円，楕円，正多角形

■ 四角形：　Rectangle(xy=四角形の左下の座標, width=横幅, height=高さ,
fc=塗りの色, ec=線の色, lw=線の太さ, angle=回転の角度)

キーワード引数「angle=」には四角形の傾き（反時計回りの回転の角度）を 360 進法の**度**で与える．この場合の回転の中心は，四角形の左下の座標である．

■ 円：　Circle(xy=中心の座標, radius=半径, fc=塗りの色, ec=線の色, lw=線の太さ)

■ 楕円：　Ellipse(xy=中心の座標, width=横幅, height=高さ,
fc=塗りの色, ec=線の色, lw=線の太さ, angle=回転の角度)

width 以降の引数については四角形の場合に準ずる．

■ 正多角形：　RegularPolygon(xy=中心の座標, radius=半径, numVertices=頂点の数,
fc=塗りの色, ec=線の色, lw=線の太さ, orientation=回転の角度)

キーワード引数「orientation=」には多角形の傾き（反時計回りの回転の角度）を与えるが，この場合は**弧度法（ラジアン）**の値で与えることに注意しなければならない．それ以外の引数については円の場合に準ずる．

これらの図形を表示するサンプルプログラム mPolygon2D01.py を示す．

プログラム：mPolygon2D01.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import matplotlib.patches as pat
4
5  # 四角形
6  p1 = pat.Rectangle( xy=(0,0), width=1.5, height=1, fc='blue', ec='red', lw=4 )
7  p2 = pat.Rectangle( xy=(0,1), width=1.5, height=1, fc='blue', ec='red', lw=4,
8                      angle=30 )
9
10 # 円
11 p3 = pat.Circle( xy=(2.2,0.5), radius=0.5, fc='yellow', ec='blue', lw=8 )
12
13 # 楕円
14 p4 = pat.Ellipse( xy=(3.7,0.5), width=1.5, height=1,
15                     fc='cyan', ec='magenta', lw=6 )
16 p5 = pat.Ellipse( xy=(3.7,1.8), width=1.5, height=1,
17                     fc='cyan', ec='magenta', lw=6, angle=30 )
18
19 # 正多角形
20 p6 = pat.RegularPolygon( xy=(5.2,0.5), radius=0.5, numVertices=5,
21                     fc='magenta', ec='green', lw=8 )
22 p7 = pat.RegularPolygon( xy=(5.2,1.8), radius=0.5, numVertices=5,
23                     fc='magenta', ec='green', lw=8, orientation=np.pi/2 )
```

[72]詳しくは matplotlib の公式インターネットサイト（https://matplotlib.org/）を参照のこと．

```
24
25 # 描画
26 plt.figure( figsize=(8,3.53) )
27 ax = plt.gca()
28 ax.add_patch(p1);   ax.add_patch(p2);   ax.add_patch(p3);   ax.add_patch(p4)
29 ax.add_patch(p5);   ax.add_patch(p6);   ax.add_patch(p7)
30 plt.xlim(-0.8,6.0); plt.ylim(-0.2,2.8)
31 plt.show()
```

このプログラムを実行すると図 103 のように表示される．

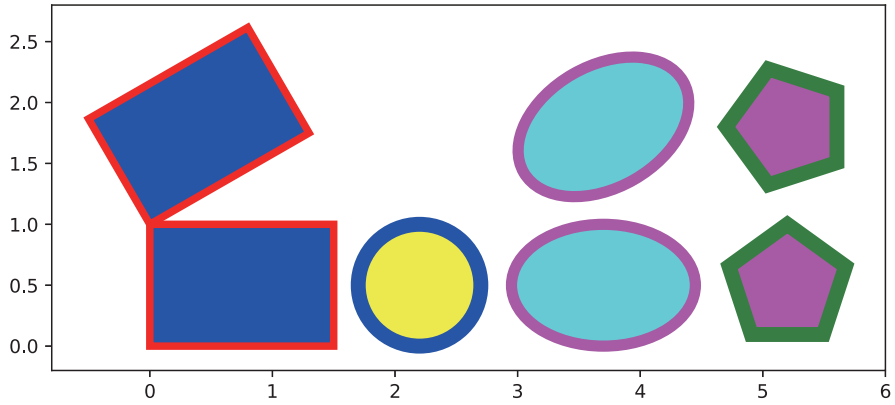


図 103: mPolygon2D01.py の実行結果

3.1.28.2　円弧（楕円弧）

円弧は Arc オブジェクトとして描画する．

Arc(xy=中心の座標, width=横幅, height=高さ, ec=線の色, lw=線の太さ,
theta1=開始の角度, theta2=終了の角度, angle=回転の角度)

「開始の角度」で始まり「終了の角度」で終わる円弧（楕円弧）を作成する．それ以外の引数については楕円の場合に準ずる．角度は反時計回り，360 進法の**度**で与える．

Arc を使用したサンプルプログラム mPolygon2D02.py を示す．

プログラム：mPolygon2D02.py

```
1  import matplotlib.pyplot as plt
2  import matplotlib.patches as pat
3
4  # 円弧
5  p1 = pat.Arc( xy=(0,0.5), width=0.8, height=0.8,
6                ec='red', lw=12, theta1=30, theta2=330 )
7  p2 = pat.Arc( xy=(0.3,0.5), width=0.8, height=0.8,
8                ec='red', lw=12, theta1=330, theta2=30 )
9  p3 = pat.Arc( xy=(2,0.5), width=1.8, height=0.8,
10               ec='blue', lw=12, theta1=290, theta2=250 )
11 p4 = pat.Arc( xy=(4,0.5), width=1.8, height=0.8,
12               ec='blue', lw=12, theta1=290, theta2=250, angle=30 )
13
14 # 描画
15 plt.figure( figsize=(14,3.2) )
16 ax = plt.gca()
17 ax.add_patch(p1);   ax.add_patch(p2)
18 ax.add_patch(p3);   ax.add_patch(p4)
19 plt.xlim(-0.8,5.2); plt.ylim(-0.2,1.3)
20 plt.show()
```

このプログラムを実行すると図 104 のように 4 つの円弧が表示される．

3.1.28.3　ポリゴン

与えられた複数の座標を結ぶ**ポリゴン**は Polygon オブジェクトとして作成する．

Polygon(xy=座標のリスト, fc=塗りの色, ec=線の色, lw=線の太さ)

キーワード引数「fc=」以降の引数については四角形の場合に準ずる．

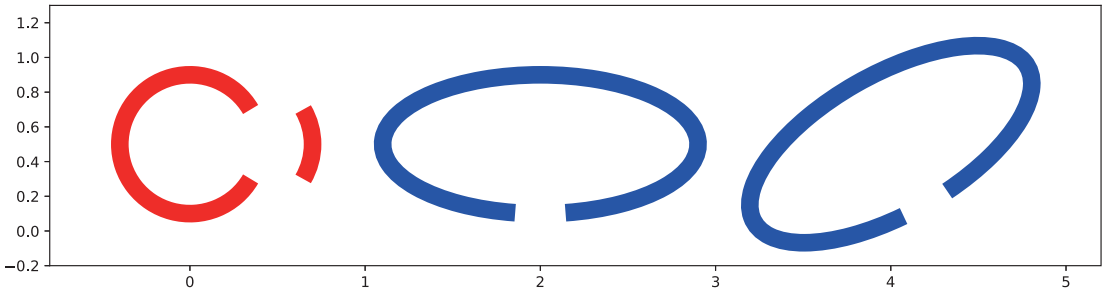


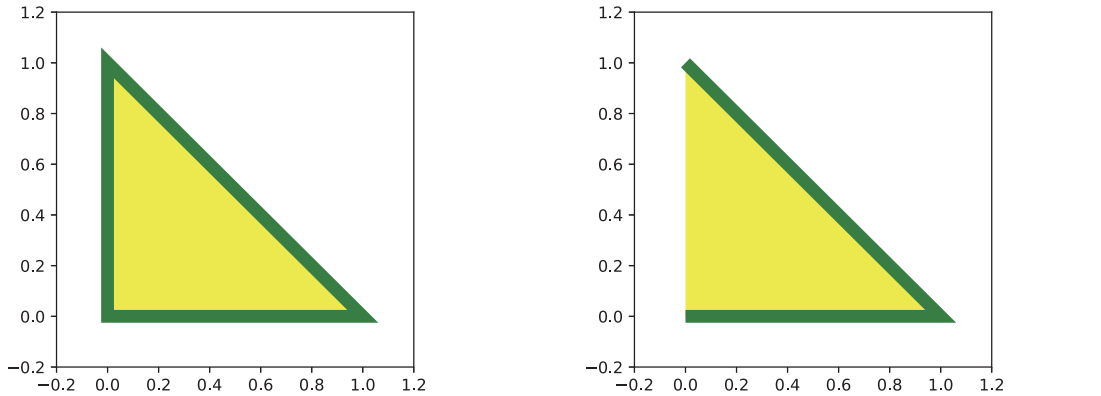
図 104: mPolygon2D02.py の実行結果

Polygon を使用したサンプルプログラム mPolygon2D03.py を示す.

プログラム：mPolygon2D03.py

```
1  import matplotlib.pyplot as plt
2  import matplotlib.patches as pat
3
4  # ポリゴン
5  p1 = pat.Polygon( xy=[(0,0),(1,0),(0,1)], fc='yellow', ec='green', lw=8 )
6
7  # 描画
8  plt.figure( figsize=(4,4) )
9  ax = plt.gca()
10 ax.add_patch(p1)
11 plt.xlim(-0.2,1.2); plt.ylim(-0.2,1.2)
12 plt.show()
```

このプログラムを実行すると図 105 の (a) ようなポリゴンが表示される.



(a) mPolygon2D03.py の実行で表示されるポリゴン　(b) Polygon の引数に 'closed=False' を与えた場合

図 105: mPolygon2D03.py の実行結果

「座標のリスト」の始点と終点は同じにする必要はなく，自動的に閉じられる．引数 'closed=False' を与えると始点と終点を開いた状態の描画ができる．この引数を与える形で mPolygon2D03.py を改変して実行すると，図 105 の (b) ようなポリゴンが表示される.

3.1.29　3 次元のポリゴン表示

先の「3.1.19 データの可視化：3 次元プロット」(p.133) で解説した 3 次元プロット空間にポリゴンを表示する最も簡単な方法[73] について説明する．3 次元のポリゴンオブジェクトは mpl_toolkits.mplot3d.art3d が提供する Poly3DCollection オブジェクトであり，これを 3 次元空間に表示するためには次のようにして必要なライブラリを読み込んでおく.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
```

書き方：　Poly3DCollection(ポリゴンリスト, facecolors=[色 1, 色 2, …])

「ポリゴンリスト」にはポリゴンの頂点の座標のリストを

[[(x11,y11,z11),(x12,y12,z12),…], [(x21,y21,z21),(x22,y22,z22),…], …]

[73] 詳しくは matplotlib の公式インターネットサイト（https://matplotlib.org/）を参照のこと.

の形式で与える．(x,y,z) の座標タプルのリストで 1 つのポリゴンを構成し，それらを更にリストで束ねて複数のポリゴンを作成する．「色 1」,「色 2」，… は個々のポリゴンの色指定である．作成した Poly3DCollection オブジェクトは 3 次元の Axes オブジェクトに対して `add_collection3d` メソッドを使用することで表示する．

Poly3DCollection を使用したサンプルプログラム mPolygon3D00.py を示す．

プログラム：mPolygon3D00.py

```
1  import matplotlib.pyplot as plt
2  from mpl_toolkits.mplot3d.art3d import Poly3DCollection
3
4  # 4つのポリゴン
5  verts = [
6      [(0,0,0), (0,2,0), (1,1,0)],     # ポリゴン1
7      [(0,0,0), (1,1,1), (1,1,0)],     # ポリゴン2
8      [(0,2,0), (1,1,1), (1,1,0)],     # ポリゴン3
9      [(0,0,0), (1,1,1), (0,2,0)]      # ポリゴン4
10 ]
11
12 # Poly3DCollectionをまとめて作成
13 poly = Poly3DCollection(verts, facecolors=['red','green','blue','yellow'])
14
15 # 描画
16 fig,ax = plt.subplots(subplot_kw={'projection':'3d'})
17 ax.add_collection3d(poly)
18 ax.set_xlim(-0.2, 1.2); ax.set_ylim(-0.2, 2.5); ax.set_zlim(-0.2, 1.0)
19 ax.set_xlabel('x'); ax.set_ylabel('y'); ax.set_zlabel('z')
20 plt.show()
```

このプログラムを実行すると図 106 のようなポリゴンが表示される．

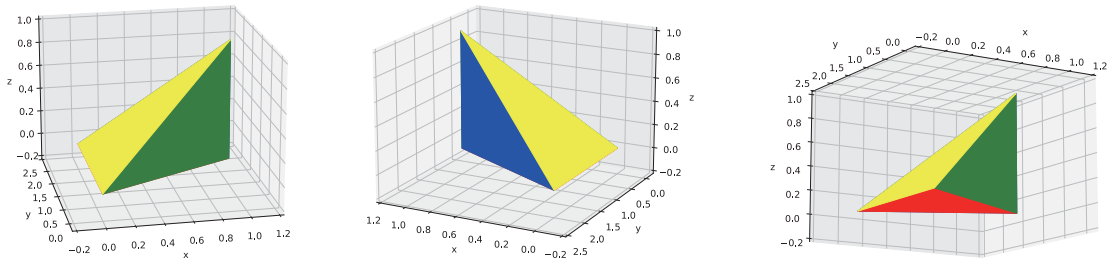


図 106: mPolygon3D00.py の実行結果（同一のグラフを様々な角度で表示した例）

この方法では add_collection3d メソッドで追加した順序でポリゴンが重なり合う．従って**隠線処理**が正しく施されない　（図 107）ことに注意すること．

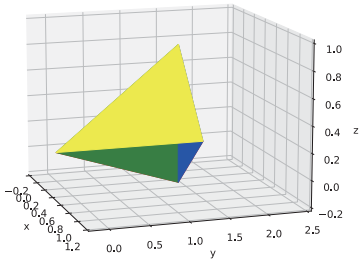


図 107: 不自然なポリゴンの配置の例

3.1.30　日付，時刻の扱い

3.1.30.1　datetime64 クラス

NumPy は日付と時刻を表現するための datetime64 クラスを提供する．このクラスのインスタンスは 1 つの**タイムスタンプ**を表す．ISO8601 形式の日付，時刻をコンストラクタに与えて datetime64 オブジェクトを生成する例を次に示す．

例. 2022 年 1 月 1 日 00:00:00 を表す datetime64 オブジェクトの作成

```
>>> import numpy as np  Enter      ← NumPy の読込み
>>> d1 = np.datetime64('2022-01-01T00:00:00')  Enter      ← datetime64 オブジェクトの生成
>>> d1  Enter                                    ←確認
np.datetime64('2022-01-01T00:00:00')             ←作成されたオブジェクト
```

注意）NumPy 1.11.0 以降の datetime64 ではタイムゾーン情報を扱わないとしている．従って，タイムゾーンは指定せずに（UTC の解釈で）datetime64 を扱うこと．

コンストラクタに与える日付，時刻の文字列は省略した形で記述することが可能である．その場合は，その記述が示す日付時刻の開始時点のタイムスタンプと同じものとなる．(次の例)

例. 省略した記述を与えて作成した datetime64 オブジェクト（先の例の続き）

```
>>> np.datetime64('2022-01-01') == d1  Enter      ←時刻を省略したもの
True                          ←先の d1 と同じ
>>> np.datetime64('2022-01') == d1  Enter      ←「日」以降を省略したもの
True                          ←先の d1 と同じ
>>> np.datetime64('2022') == d1  Enter      ←「月」以降を省略したもの
True                          ←先の d1 と同じ
```

この例からわかるように，解像度の荒い側の値の単位が比較の基準となる．(処理環境や NumPy のバージョンによっては True が `np.True_` と表示されることがある．)

コンストラクタに整数値を与えて datetime64 オブジェクトを作成することができる．

書き方：　datetime64(整数値, 単位)

この形を取る場合，'1970-01-01T00:00:00' を基準[74] とした datetime64 オブジェクトが作成される．またその際「整数値」が意味するものは「単位」（文字列）で指定した経過時間となる．

例. '1970-01-01T00:00:00' と基準とした datetime64 オブジェクトの作成

```
>>> np.datetime64(1,'Y')  Enter      ← 1 年後
numpy.datetime64('1971')              ←'1971-01-01T00:00:00' となる
>>> np.datetime64(-1,'s')  Enter      ← 1 秒前
numpy.datetime64('1969-12-31T23:59:59')
```

この例では「1 年後」と「1 秒前」の日付を作成しているが「単位」の部分に指定できるものを表 36 に示す．

表 36: 時間の単位（一部）

単位	意味	単位	意味	単位	意味	単位	意味
'Y'	年	'M'	月	'W'	週	'D'	日
'h'	時	'm'	分	's'	秒	'ms'	ミリ秒
'us'	ﾏｲｸﾛ秒	'ns'	ナノ秒				

参考）ナノ秒よりも小さな単位も扱える．

3.1.30.2　現在時刻の取得

datetime64 コンストラクタの引数に 'now' を与えると，UTC の現在時刻の情報が得られる．

例. 現在時刻の取得

```
>>> np.datetime64('now')  Enter      ←現在時刻の取得
np.datetime64('2025-08-24T07:13:19')      ← UTC 時刻
```

[74] UNIX エポック

3.1.30.3　timedelta64 クラス

datetime64 オブエジェクトの差を通常の減算演算子「-」で求めることができる.（次の例）

例. datetime64 オブジェクトの差（その 1）

```
>>> d1 = np.datetime64('2022-01-01')  Enter      ← 1 つ目
>>> d2 = np.datetime64('2023-01-01')  Enter      ← 2 つ目
>>> td = d2 - d1  Enter              ←それらの差
>>> td  Enter                        ←確認
numpy.timedelta64(365,'D')           ←結果（365 日）
```

この例では，ちょうど 1 年の差がある datetime64 オブジェクトの減算を実行したものである．減算の結果は timedelta64 クラスのオブジェクトとして得られる．このクラスのオブジェクトは

timedelta64(値, 単位)

の形式で時間の長さを表す．この例では「365 日の差」を表すものが得られている．減算に使用する datetime64 オブジェクトによって減算の結果の単位が異なるものとなる.（次の例）

例. datetime64 オブジェクトの差（その 2）

```
>>> d1 = np.datetime64('2022-01-01T00:00:00')  Enter      ← 1 つ目
>>> d2 = np.datetime64('2023-01-01T00:00:00')  Enter      ← 2 つ目
>>> td = d2 - d1  Enter              ←それらの差
>>> td  Enter                        ←確認
numpy.timedelta64(31536000,'s')      ←結果（365 日の秒数）
```

この例は先の例と同様の処理を実行したものであるが，減算に使用した datetime64 オブジェクトが秒単位であることから，減算結果の timedelta64 オブジェクトも秒単位になっている．

■ グレゴリオ暦

datetime64，timedelta64 は**グレゴリオ暦**を反映している．以下に**閏年**が関わる処理の例を示す．

例. 2020 年（閏年）と 2021 年の差

```
>>> d1 = np.datetime64('2020')  Enter      ←閏年
>>> d2 = np.datetime64('2021')  Enter      ←平年
>>> d2 - d1  Enter              ←差を取ると…
numpy.timedelta64(1,'Y')        ← 1 年の差
```

これは「年」を単位として差を算出したものであり，結果は「1 年」であることがわかる．更に「日」を単位として差を算出すると次のようになる．

例. 「日」を単位とした時間の長さ

```
>>> np.datetime64('2021-01-01') - np.datetime64('2020-01-01')  Enter  ←閏年の 1 年
numpy.timedelta64(366,'D')              ← 366 日
>>> np.datetime64('2022-01-01') - np.datetime64('2021-01-01')  Enter  ←平年の 1 年
numpy.timedelta64(365,'D')              ← 365 日
```

時間の長さについて考える際は，単位と暦を意識する必要がある．

3.1.30.4　日付，時刻の応用例

datetime64，timedelta64 オブジェクトを要素として持つ配列を作成する例を示す．

例. 15 分おきの datetime64 の配列

```
>>> t1 = np.datetime64('2022-01-01T00:00')  Enter  ←自（分単位）
>>> t2 = np.datetime64('2022-01-01T01:00')  Enter  ←至（分単位）
>>> a12 = np.arange(t1,t2,np.timedelta64(15,'m'))  Enter  ← 15 分おきの配列作成
>>> a12  Enter                  ←確認
array(['2022-01-01T00:00', '2022-01-01T00:15', '2022-01-01T00:30',
       '2022-01-01T00:45'], dtype='datetime64[m]')          ←得られた配列（分単位）
```

例. 10 マイクロ秒おきの datetime64 の配列（先の例の続き）

```
>>> t3 = np.datetime64('2022-01-01T00:00:00')  Enter  ←自（秒単位）
>>> t4 = np.datetime64('2022-01-01T00:00:00.000040')  Enter  ←至（マイクロ秒単位）
>>> a34 = np.arange(t3,t4,np.timedelta64(10,'us'))  Enter  ← 10 マイクロ秒おきの配列作成
>>> a34  Enter  ←確認
array(['2022-01-01T00:00:00.000000', '2022-01-01T00:00:00.000010',
       '2022-01-01T00:00:00.000020', '2022-01-01T00:00:00.000030'],
      dtype='datetime64[us]')                    ←得られた配列（マイクロ秒単位）
```

例. 上の例で得られた配列 a12，a34 の 1 階差分の配列（先の例の続き）

```
>>> np.diff(a12)  Enter  ← a12 の 1 階差分
array([15, 15, 15], dtype='timedelta64[m]')  ←分単位の timedelta64 オブジェクトの配列
>>> np.diff(a34)  Enter  ← a34 の 1 階差分
array([10, 10, 10], dtype='timedelta64[us]')  ←マイクロ秒単位の timedelta64 オブジェクトの配列
```

3.1.31　その他の機能

3.1.31.1　基準以上／以下のデータのクリッピング

ある基準と比較して，それより大きい／小さいデータをクリッピングする関数に maximum ，minimum がある．ここでいう「クリッピング」とは，基準を超える／下回る要素を基準値で置き換える処理のことである．

書き方：　maximum(データ配列, 基準値の配列)
書き方：　minimum(データ配列, 基準値の配列)

maximum 関数は「基準値の配列」以上のものを「データ配列」からクリッピングして返す．このとき，データ列の要素の内で基準値未満の要素に関しては，対応する基準値で置き換えたものを戻り値の要素とする．同様に，minimum 関数は「基準値の配列」以下のものを「データ配列」からクリッピングする．このとき，データ配列の要素の内で基準値より大きい要素に関しては，対応する基準値の要素で置き換えたものを戻り値の要素とする.「データ配列」と「基準値の配列」の長さは同じものであるとする.（NumPy のブロードキャスト規則により，基準値がスカラーであればすべての要素に同じ基準が適用される.）

考え方：　maximum は「大きい方を取る」，minimum は「小さい方を取る」関数である．

これら関数を使用するサンプルプログラム maximum_minimum01.py を示す．

プログラム：maximum_minimum01.py

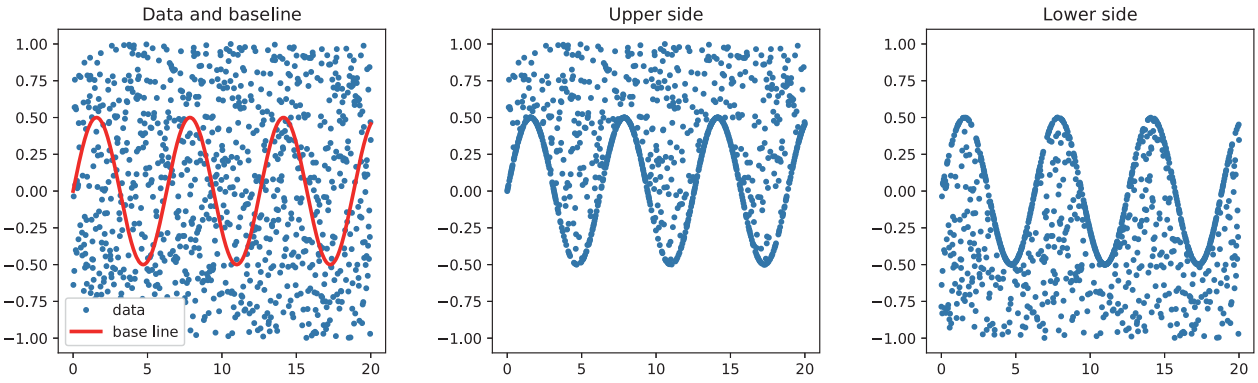
```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # データの作成
5  x = np.linspace(0,20,1000)
6  y = np.sin(x) / 2.0              # 基準列
7  rng = np.random.default_rng(1)   # RNG作成
8  r = rng.random(1000)*2 - 1       # データ列
9
10 # データとベースラインのプロット
11 plt.figure(figsize=(4,4))
12 plt.plot(x,r,label='data',ls='None',marker='.')
13 plt.plot(x,y,label='base line',lw=2.5,c='red')
14 plt.ylim(-1.1,1.1)
15 plt.legend()
16 plt.title('Data and baseline')
17 plt.show()
18
19 # maximumによる抽出
20 r1 = np.maximum(r,y)          # 基準列以上のデータを取り出す
21 plt.figure(figsize=(4,4))
22 plt.plot(x,r1,ls='None',marker='.')
23 plt.ylim(-1.1,1.1)
24 plt.title('Upper side')
25 plt.show()
26
27 # minimumによる抽出
28 r2 = np.minimum(r,y)          # 基準列以下のデータを取り出す
29 plt.figure(figsize=(4,4))
30 plt.plot(x,r2,ls='None',marker='.')
31 plt.ylim(-1.1,1.1)
32 plt.title('Lower side')
33 plt.show()
```

このプログラムは，データ配列 r を基準値の配列 y と比較してクリッピングする例である．5〜17 行目でデータ配列と基準値の配列を生成して，それらをプロット（図 108 の a）している．20〜25 行目で基準値以上のデータ（図 108 の b）を，28〜33 行目で基準値以下のデータ（図 108 の c）をクリッピングしてプロットしている．

maximum，minimum 関数の第 2 引数（基準値）にスカラー値を与えることもできる．サンプルプログラム maximum_minimum02.py でそれを示す．

プログラム：maximum_minimum02.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
```

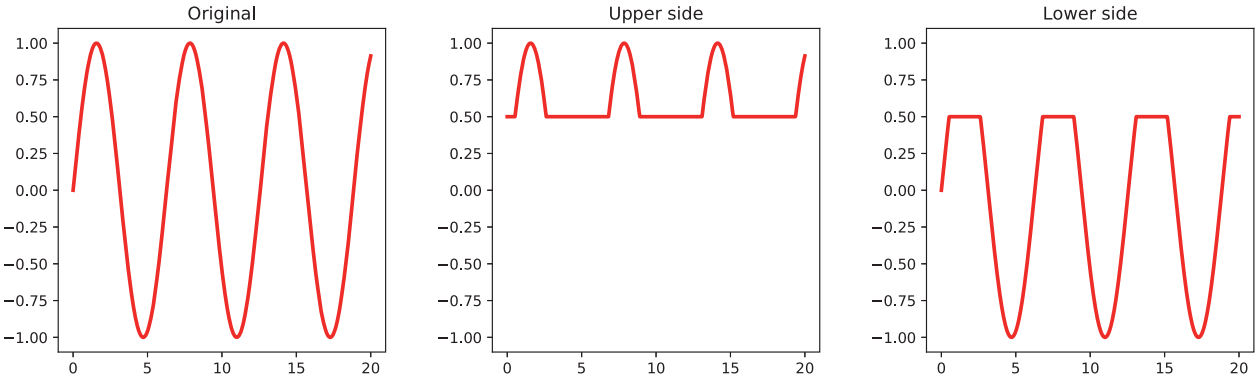


(a) データ配列と基準値の配列　(b) 基準値以上のデータ　(c) 基準値以下のデータ

図 108: 上側，下側のデータのクリッピング

```
4  # データの作成
5  x = np.linspace(0,20,1000)
6  y = np.sin(x)
7
8  # 元のデータのプロット
9  plt.figure(figsize=(4,4))
10 plt.plot(x,y,lw=2.5,c='red')
11 plt.ylim(-1.1,1.1)
12 plt.title('Original')
13 plt.show()
14
15 # スカラーによる抽出（上側）
16 y1 = np.maximum(y,0.5)
17 plt.figure(figsize=(4,4))
18 plt.plot(x,y1,label='base line',lw=2.5,c='red')
19 plt.ylim(-1.1,1.1)
20 plt.title('Upper side')
21 plt.show()
22
23 # スカラーによる抽出（下側）
24 y1 = np.minimum(y,0.5)
25 plt.figure(figsize=(4,4))
26 plt.plot(x,y1,label='base line',lw=2.5,c='red')
27 plt.ylim(-1.1,1.1)
28 plt.title('Lower side')
29 plt.show()
```

このプログラムを実行すると，図 109 のようなグラフが順番に表示される．



(a) 元のデータ列　(b) 基準値以上のデータ　(c) 基準値以下のデータ

図 109: スカラー値を基準にして上側，下側データをクリッピング

このように，スカラーの基準値を与えると単純なクリッピングができる．

3.1.31.2　指定した範囲のデータのクリッピング

clip 関数を用いることで配列の要素の値を指定した範囲内にクリッピングすることができる.

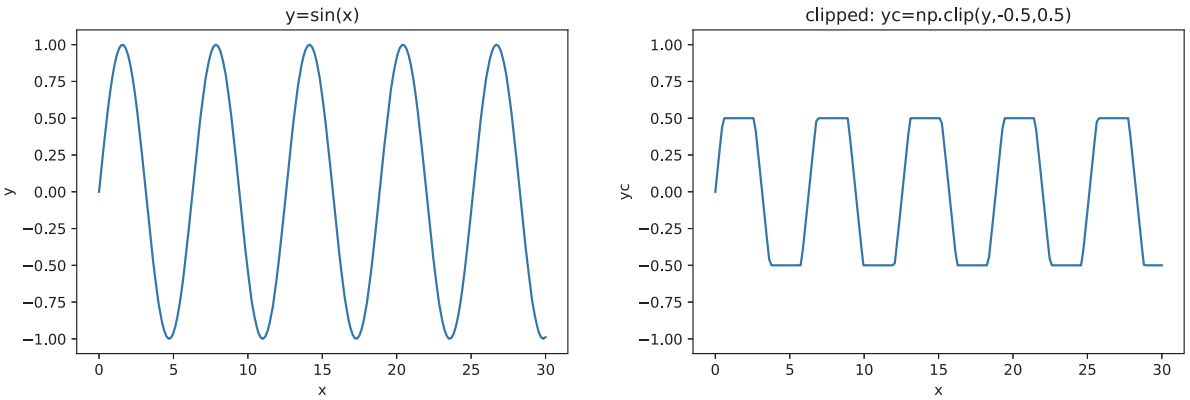
書き方：　clip(配列, 最小値, 最大値)

「配列」の要素の内,「最小値」よりも小さいものを「最小値」に,「最大値」よりも大きいものを「最大値」に置き換える．次に示すサンプルプログラム npclip01.py は，$y = sin(x)$ を $-0.5 \leq y \leq 0.5$ の範囲にクリッピングしてプロットするものである.

プログラム：npclip01.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # データの生成
5  x = np.linspace(0,30,200)
6  y = np.sin(x)
7
8  # プロット：clipなし
9  plt.figure(figsize=(6,4))
10 plt.plot( x, y )
11 plt.xlabel('x');    plt.ylabel('y')
12 plt.ylim(-1.1,1.1)
13 plt.title('y=sin(x)')
14 plt.show()
15 print('min:',y.min(),' max:',y.max())
16
17 # プロット：clipあり
18 yc = np.clip( y, -0.5, 0.5 )
19 plt.figure(figsize=(6,4))
20 plt.plot( x, yc)
21 plt.xlabel('x');    plt.ylabel('yc')
22 plt.ylim(-1.1,1.1)
23 plt.title('clipped: yc=np.clip(y,-0.5,0.5)')
24 plt.show()
25 print('min:',yc.min(),' max:',yc.max())
```

このプログラムを実行すると，図 110 のようなグラフが順番に表示される.



(a) クリッピングなし　　(b) clip で -0.5〜0.5 の範囲にクリッピング

図 110: clip 関数による値のクリッピング

y の値を $-0.5 \leq y \leq 0.5$ の範囲にクリッピングしている様子がわかる.

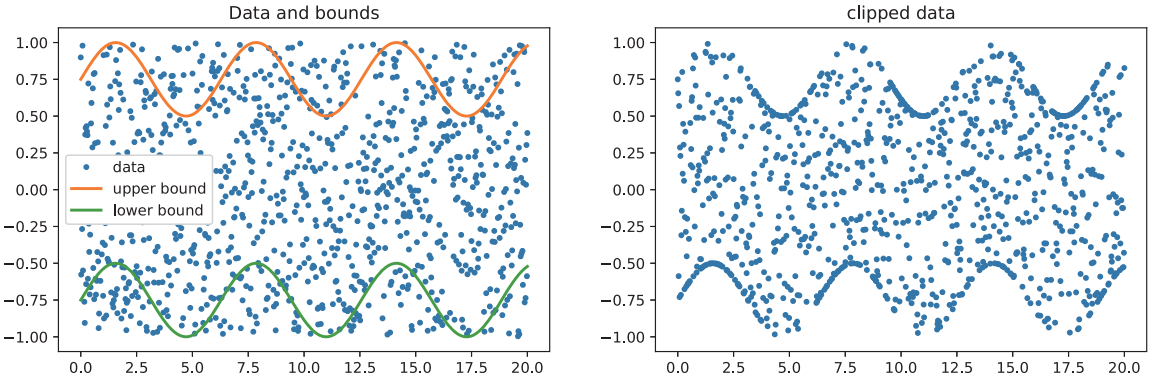
clip 関数の第 2, 第 3 引数（最小値，最大値）に範囲を表すデータ列を与えることもできる．そのことを次のサンプルプログラム npclip02.py で示す.

プログラム：npclip02.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # データの作成
5  x = np.linspace(0,20,1000)
6  y1 = np.sin(x) / 4.0 + 0.75      # 上限
```

```
y2 = np.sin(x) / 4.0 - 0.75      # 下限
rng = np.random.default_rng(1)   # RNGの作成
r = rng.random(1000)*2 - 1       # データ列

# データ，下限，上限のプロット
plt.figure(figsize=(6,4))
plt.plot(x,r,label='data',ls='None',marker='.')
plt.plot(x,y1,label='upper bound',lw=2)
plt.plot(x,y2,label='lower bound',lw=2)
plt.ylim(-1.1,1.1)
plt.legend()
plt.title('Data and bounds')
plt.show()

# プロット：clipによる制限
rc = np.clip( r, y2, y1 )
plt.figure(figsize=(6,4))
plt.plot( x, rc, ls='None', marker='.' )
plt.ylim(-1.1,1.1)
plt.title('clipped data')
plt.show()
```

このプログラムを実行すると，図 111 のようなグラフが順番に表示される．



(a) データ列，下限，上限　　(b) clip 関数でクリッピングした結果

図 111: 下限，上限の列を clip 関数に与える例

指定した下限，上限の範囲にデータがクリッピングされている様子がわかる．

3.2 科学技術計算用ライブラリ： SciPy

SciPy は NumPy を基礎にして構築されたライブラリであり，科学，工学のための高度な数値計算のための機能（表 37）を提供する．SciPy 本体と関連情報は公式インターネットサイト https://www.scipy.org/ から得られる．

表 37: SciPy に含まれるパッケージ

パッケージ	説　　明	パッケージ	説　　明
constants	物理定数と変換係数	cluster	階層的クラスタリング，ベクトル量子化
fft	離散フーリエ変換	integrate	数値積分
interpolate	データの補間	io	データ入出力
linalg	線形代数	misc	ユーティリティ系
ndimage	多次元の画像処理	optimize	線形計画法を含む最適化計算
signal	信号処理ツール	sparse	スパース行列の取り扱い
spatial	KD 木，最近傍，距離関数	special	その他の特別な機能
stats	統計関連の機能		

本書では SciPy の機能の一部を抜粋して説明する．本書で触れない事柄に関しては SciPy の公式インターネットサイトなどの情報を参照のこと．

3.2.1 信号処理ツール: scipy.signal

3.2.1.1 基本的な波形の生成

■ **矩形波（方形波）**： square 関数

書き方：　square(時間軸の配列, duty=パルス幅)

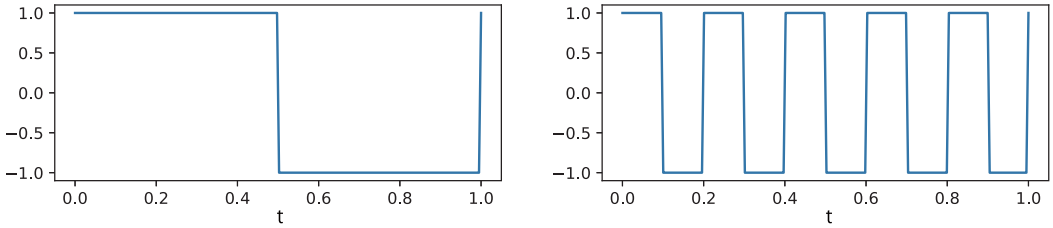
「時間軸の配列」に対する矩形波の波形（周期は 2π）を生成し，それを配列として返す．戻り値の範囲は -1.0〜1.0 である．「パルス幅」には 0〜1.0 の値を指定する．パルス幅の暗黙値は 0.5 である．

次の例は，時間軸 $t = 0$〜1 に対する周波数 $f = 1$ Hz の矩形波 $square(2\pi ft)$ を生成して波形をプロットするものである．

例. 周期 1 つ分の矩形波

```
>>> import matplotlib.pyplot as plt  Enter    ←プロット用ライブラリの読込み
>>> import numpy as np  Enter    ← NumPy の読込み
>>> from scipy import signal  Enter    ← scipy.signal の読込み
>>> t = np.linspace(0,1,200)  Enter    ←時間軸の生成
>>> f = 1  Enter    ←周波数（1Hz）の設定
>>> y = signal.square(2*np.pi*f*t)  Enter    ←矩形波の生成
>>> f = plt.figure(figsize=(5,2))  Enter    ←作図の開始
>>> r1 = plt.plot(t,y)  Enter    ←波形のプロット
>>> r2 = plt.xlabel('t')  Enter    ←グラフの横軸ラベル
>>> plt.show()  Enter    ←作図の実行
```

この実行結果として図 112 の (a) が表示される．周波数を f=5 として実行すると同図の (b) が表示される．



(a) $f = 1$ の場合の波形　　(b) $f = 5$ の場合の波形

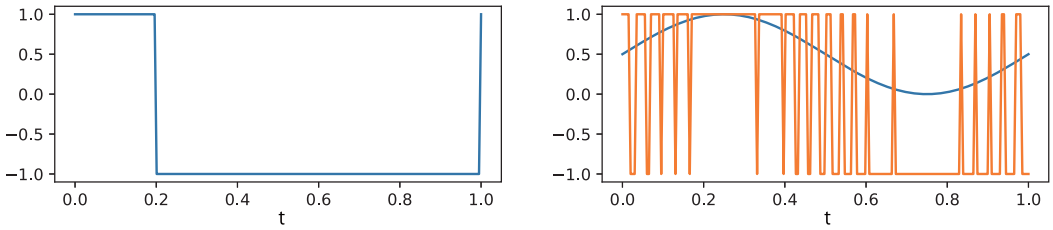
図 112: 矩形波の生成

次の例は，キーワード引数 'duty=' の値（パルス幅の比率）を 0.2 として実行するものである．

例. duty=0.2 として実行した例（先の例の続き）

```
>>> f = 1   Enter      ←周波数（1Hz）の設定
>>> y = signal.square(2*np.pi*f*t,duty=0.2)   Enter      ←矩形波の生成
>>> f = plt.figure(figsize=(5,2))   Enter      ←作図の開始
>>> r1 = plt.plot(t,y)   Enter      ←波形のプロット
>>> r2 = plt.xlabel('t')   Enter      ←グラフの横軸ラベル
>>> plt.show()   Enter      ←作図の実行
```

この実行結果として図 113 の (a) が表示される．



(a) duty=0.2 の場合の波形　　(b) パルス幅を連続的に変化させた場合の波形

図 113: 矩形波の生成

パルス幅は連続的に変化させること（**パルス幅変調**，PWM：pulse width modulation）ができる．そのためには，時間軸配列の各要素に対応するパルス幅の値を配列にしてキーワード引数 'duty=' に与える．
次の例は，正弦波 $(\sin(2\pi t)+1)/2$ でパルス幅を変調する例である．

例. パルス幅を連続的に変化させる例（先の例の続き）

```
>>> m = (np.sin(2*np.pi*t)+1)/2   Enter      ←変調用の正弦波 (sin(2πt)+1)/2 の配列
>>> f = 30   Enter      ←周波数（30Hz）の設定
>>> y = signal.square(2*np.pi*f*t,duty=m)   Enter      ←矩形波の生成
>>> f = plt.figure(figsize=(5,2))   Enter      ←作図の開始
>>> r1 = plt.plot(t,m)   Enter      ←変調用正弦波のプロット
>>> r2 = plt.plot(t,y)   Enter      ← PWM の結果の波形のプロット
>>> r3 = plt.xlabel('t')   Enter      ←グラフの横軸ラベル
>>> plt.show()   Enter      ←作図の実行
```

この実行結果として図 113 の (b) が表示される．このグラフには，変調用の正弦波 $(\sin(2\pi t)+1)/2$ と変調された矩形波が重ねて表示される．

■ 鋸歯状波（ノコギリ波）：sawtooth 関数

書き方：　sawtooth(時間軸の配列, width=立ち上がり幅)

「時間軸の配列」に対する鋸歯状波の波形（周期は 2π）を生成し，それを配列として返す．戻り値の範囲は -1.0〜1.0 である．「立ち上がり幅」には 0〜1.0 の値を指定する．width の暗黙値は 0.5（三角波）である．

次の例は，時間軸 $t = 0$〜1 に対する周波数 $f = 1$ Hz の鋸歯状波 $sawtooth(2\pi ft)$ を生成して波形をプロットするものである．

例. 周期 1 つ分の鋸歯状波（先の例の続き）

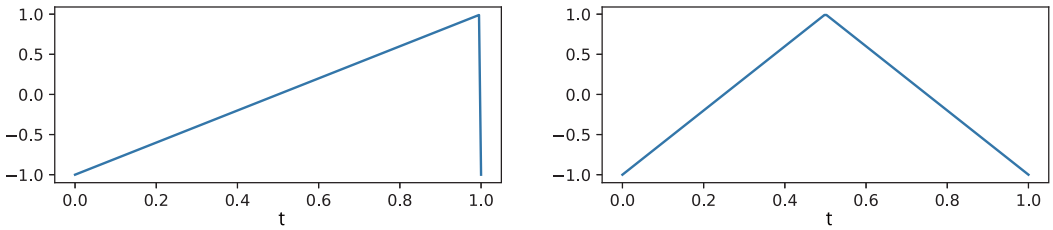
```
>>> f = 1   Enter      ←周波数（1Hz）の設定
>>> y = signal.sawtooth(2*np.pi*f*t)   Enter      ←鋸歯状波の生成
>>> f = plt.figure(figsize=(5,2))   Enter      ←作図の開始
>>> r1 = plt.plot(t,y)   Enter      ←波形のプロット
>>> r2 = plt.xlabel('t')   Enter      ←グラフの横軸ラベル
>>> plt.show()   Enter      ←作図の実行
```

この実行結果として図 114 の (a) が表示される．

先の例において，鋸歯状波を生成する文を

```
y = signal.sawtooth(2*np.pi*f*t,width=0.5)
```

として実行すると，図 114 の (b) が表示される．



(a) 鋸歯状波の波形　　(b) 立ち上がり幅を 0.5 にした場合の波形

図 114: 鋸歯状波の生成

立ち上がり幅は連続的に変化させることができる．そのためには，時間軸配列の各要素に対応する立ち上がり幅の値を配列にしてキーワード引数 ‘width=’ に与える．
次の例は，正弦波 $(\sin(2\pi t)+1)/2$ で立ち上がり幅を変調する例である．

例. 立ち上がり幅を連続的に変化させる例（先の例の続き）

```
>>> f = 5   Enter      ←周波数（5Hz）の設定
>>> y = signal.sawtooth(2*np.pi*f*t,width=m)   Enter      ←鋸歯状波の生成
>>> f = plt.figure(figsize=(5,2))   Enter      ←作図の開始
>>> r1 = plt.plot(t,m)   Enter      ←変調用正弦波のプロット
>>> r2 = plt.plot(t,y)   Enter      ←変調結果の波形のプロット
>>> r3 = plt.xlabel('t')   Enter      ←グラフの横軸ラベル
>>> plt.show()   Enter      ←作図の実行
```

この実行結果として図 115 が表示される．

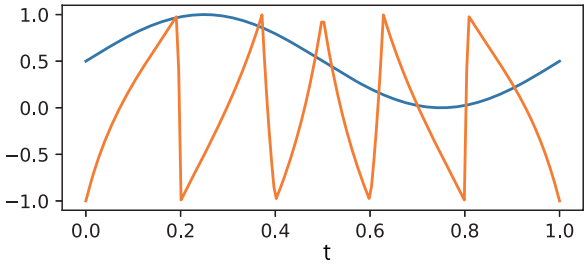


図 115: 鋸歯状波の立ち上がり幅の変調

このグラフには，変調用の正弦波 $(\sin(2\pi t)+1)/2$ と，立ち上がり幅が変調された鋸歯状波が重ねて表示される．

3.2.2 WAV ファイル入出力ツール: scipy.io.wavfile

scipy.io.wavfile は **WAV 形式**[75] 音声データファイルの入出力のための機能を提供する．Python 処理系には WAV 形式音声ファイルを扱うために wave モジュール[76] が標準的に提供されているが，scipy.io.wavfile の方が高機能である．

3.2.2.1 WAV 形式ファイル出力： write 関数

書き方： write(ファイル名, サンプリング周波数, データ配列)

波形データの「データ配列」を「ファイル名」のファイルに出力する．「サンプリング周波数」[77] は整数値で与える．出力する WAV ファイルの**量子化ビット数**[78] は「データ配列」の要素の型から自動的に設定される．

1 秒の長さの 440Hz の正弦波のサウンドをサンプリング周波数 44.1KHz で出力する例を次に示す．

例. 正弦波の波形を WAV 形式ファイルとして出力する

```
>>> from scipy.io import wavfile  Enter     ← scipy.io.wavfile の読み込み
>>> import numpy as np  Enter     ← NumPy の読み込み
>>> r = 44100; f = 440  Enter     ←サンプリング周波数 r とサウンドの周波数 f の設定
>>> t = np.linspace( 0, 1, r, endpoint=False )  Enter     ←時間軸の生成（0〜1 秒）
>>> yL = np.sin(2*np.pi*f*t)  Enter     ←正弦波の波形の生成
>>> wavfile.write( 'scipyWavINT16.wav', r, (32767*yL).astype('int16') )  Enter     ←出力 (1)
>>> wavfile.write( 'scipyWavUINT8.wav', r, (127*(yL+1)).astype('uint8') )  Enter     ←出力 (2)
>>> wavfile.write( 'scipyWavFLT32.wav', r, yL.astype('float32') )  Enter     ←出力 (3)
```

「出力 (1)」は 16 ビット整数型の出力であり，市販の音楽 CD で標準的に採用されている形式である．「出力 (2)」は符号なし 8 ビット整数型の出力であり，音質を犠牲にして出力データの大きさを小さく抑える場合に適している．波形データを整数型で出力する場合，波形の値は出力用の整数型の値の範囲でなければならない．これを調べるには NumPy の iinfo [79] を使用すると良い．

「出力 (3)」は 32 ビット浮動小数点数による出力であり，高品質のサウンドデータ（いわゆる**ハイレゾリューションオーディオ**あるいは**ハイレゾ**）を作成する場合に適している．

出力する音声の品質はデータ型だけでなく，サンプリング周波数にも依存する．サンプリング周波数は標準的には 44.1KHz，ハイレゾの場合は 48KHz 以上（96KHz など），音質を求めない場合は 22.05KHz 以下の値を採用する．

この例で作成した WAV 形式ファイルは，多くのサウンド再生用アプリケーションソフトで再生することができる．

【ステレオサウンドの出力】

左右 2 チャンネルのサウンドを出力するにはそれら波形データを 2 次元の配列として構成する．具体的には，

```
[ [左 0, 右 0],
  [左 1, 右 1],
      :
  [左 n, 右 n] ]
```

という形式でデータ配列を作成する．

例. ステレオサウンドの出力（先の例の続き）

```
>>> yR = np.cos(2*np.pi*f*t)  Enter     ←余弦関数の波形の生成
>>> yST = np.hstack( (yL.reshape(-1,1), yR.reshape(-1,1)) )  Enter     ← 2 次元配列の波形
>>> wavfile.write( 'scipyWavFLT32stereo.wav', r, yST.astype('float32') )  Enter     ←出力
```

この例では余弦関数の波形データを変数 yR に作成している．これを右チャンネル，先の例で作成した yL を左チャンネルとして合成して 2 次元配列 yST を作成し，それを WAV 形式ファイルとして出力している．

---

[75] Microsoft 社と IBM 社によって開発されたデータフォーマット

[76] これに関しては拙書「Python3 入門」で解説しています．

[77] 1 秒間の音声を何個のデータとしてサンプリングするかの個数．

[78] 音声波形の 1 つの値を表現するビット長

[79] 「3.1.2.6 扱える値の範囲」（p.57）を参照のこと．

3.2.2.2 WAV 形式ファイル入力： read 関数

書き方： read(ファイル名)

「ファイル名」の WAV 形式ファイルの内容を読み取って **(サンプリング周波数, データ配列)** のタプルを返す．先に作成した WAV 形式ファイルを読み込み，波形をプロットする例を次に示す．

例. WAV 形式ファイルの内容をプロットする（先の例の続き）

```
>>> (r2,yST2) = wavfile.read( 'scipyWavFLT32stereo.wav' )  Enter  ← WAV ファイルの読込み
>>> r2  Enter      ←サンプリング周波数の確認
44100      ←単位は Hz
>>> yST2.shape  Enter      ←データ配列の形状の確認
(44100, 2)      ← 2 列の 2 次元配列（左右 2 チャンネル）
>>> yL2 = yST2[:,0]  Enter      ←左チャンネルの取り出し
>>> yR2 = yST2[:,1]  Enter      ←右チャンネルの取り出し
>>> import matplotlib.pyplot as plt  Enter      ← matplotlib の読込み
>>> (fig,ax) = plt.subplots( 2,1, figsize=(8,3) )  Enter      ←描画手順の開始
>>> a00 = ax[0].plot(yL2[:300])  Enter      ←左チャンネルの波形のプロット（先頭 300 個）
>>> a01 = ax[0].set_ylabel('Left')
>>> a02 = ax[0].grid()
>>> a10 = ax[1].plot(yR2[:300])  Enter      ←右チャンネルの波形のプロット（先頭 300 個）
>>> a11 = ax[1].set_ylabel('Right')
>>> a12 = ax[1].grid()
>>> plt.show()  Enter      ←描画の実行
```

この実行の結果，図 116 のような波形のプロットが表示される．

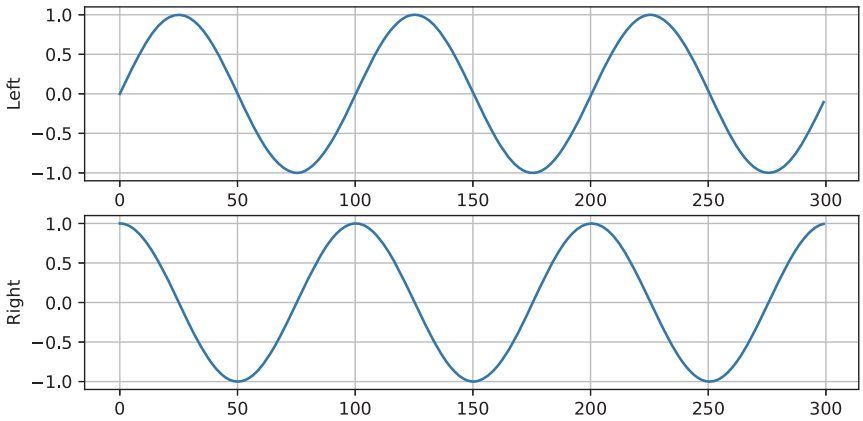


図 116: ステレオサウンドの波形のプロット

3.2.2.3 32-bit floating-point の WAV 形式サウンドデータ

32-bit floating-point の WAV 形式サウンドデータは -1.0〜1.0 の範囲で波形の値が表現される．従って，この範囲を超える値はサウンドデータとしては**クリッピング**される．この様子を次に示す．

例. -2.0〜2.0 の正弦波形の WAV ファイルの作成（先の例の続き）

```
>>> y1 = 2*np.sin(2*np.pi*f*t)  Enter      ←振幅の大きな正弦波形
>>> wavfile.write( 'scipyWavFLT32over.wav', r, y1.astype('float32') )  Enter ←ファイル出力
```

この処理で作成した WAV 形式ファイル 'scipyWavFLT32over.wav' をオープンソースの音声編集ソフト Audacity [80] で開いた様子を図 117 に示す．

[80] 公式インターネットサイト：https://www.audacityteam.org/

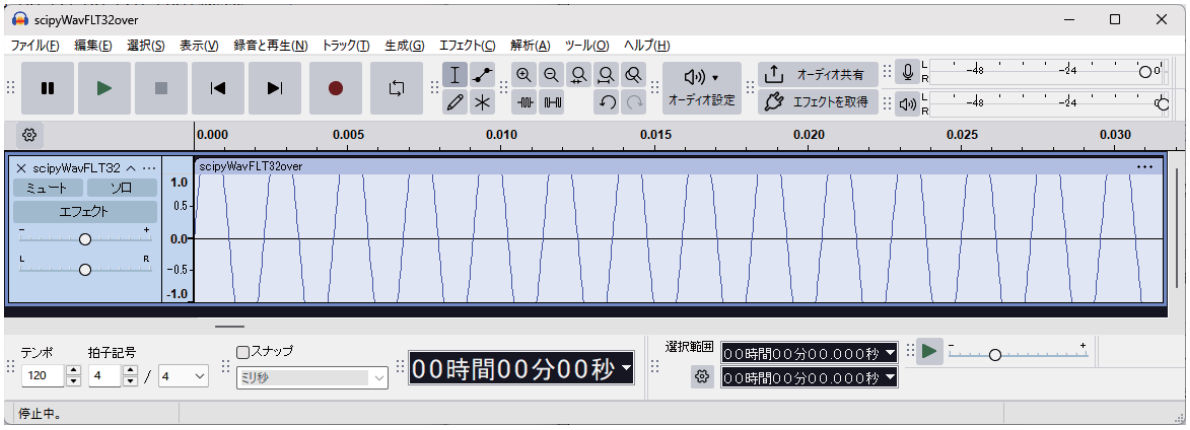


図 117: 振幅の大きな（-2.0〜2.0）正弦波形を Audacity で開いたところ

-1.0〜1.0 の範囲からはみ出した波形の上下の部分がクリッピング（切り取り）されているのがわかる．また，これをサウンドとして再生するとクリッピングが原因となる音の歪みが起こる．ただし，32-bit floating-point の WAV 形式サウンドデータは，クリッピングされた部分の波形の情報を保持しており，振幅を縮小して波形の値の範囲を -1.0〜1.0 にすることで元の波形を再現することができる[81]．このことを図 118 に示す．

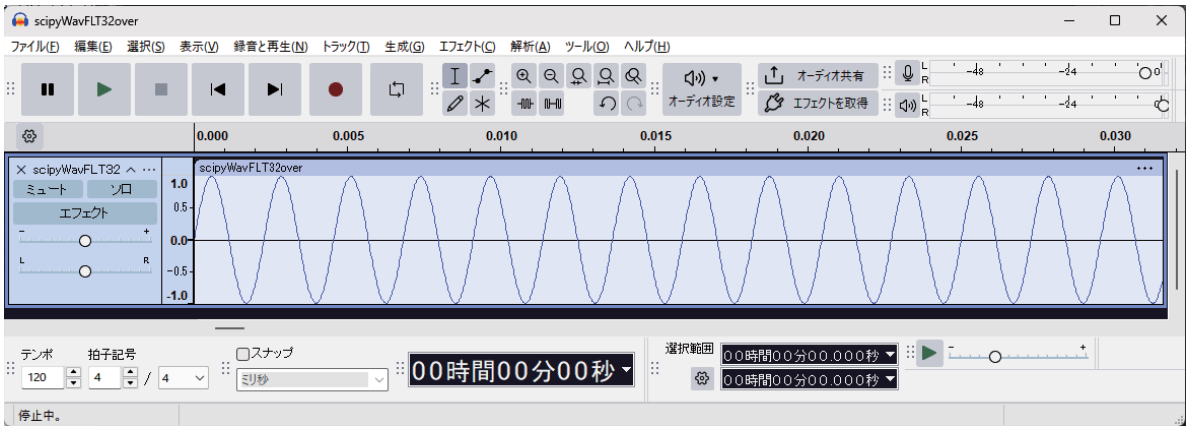


図 118: 振幅を半分に縮小（-6db 増幅）したところ：元の波形が再現されている

整数値として作成された WAV 形式データでは，クリッピングされた部分の波形情報は失われ，振幅を事後で縮小してもクリッピングされた部分の波形は復元されない．

float32 の型で表現される数値の仮数部は 23 ビットの長さ[82] があり，32-bit floating-point の WAV 形式サウンドデータはハイレゾサウンドとしての音質を実現することができる．また，先に示したように 0db を超えるレベルのサウンドの波形を保持できるので，整数型の WAV 形式に比べて有利な点が多い．ただし，32-bit floating-point の WAV 形式サウンドデータは同じ再生時間の 16 ビット整数型 WAV 形式データに比べると 2 倍のデータサイズとなることに留意する必要がある．

注意）scipy.io.wavfile は 24-bit WAV の読み込みに対応していないので注意すること．

[81] クリッピングされた部分を保持しないアプリケーションもあるので注意すること．

[82] IEEE 754 で規定されている．

3.3 数式処理ライブラリ： SymPy

SymPy は Python に数式処理機能を提供するパッケージ[83] である．**数式処理システム**（CAS: Computer Algebra System）とは，数式を記号的に処理するシステムであり，代数的な計算を記号のまま実行する．例えば，$a + a$ という式を評価すると $2a$ という形（あるいは $2 * a$ という形）で結果が得られる．

SymPy は，数式やそれを構成する記号を独特のオブジェクトとして扱うため，一般的に知られる数式処理システム[84] と比較すると，記号の扱いに違いがある．本書では他の数式処理システムとの違いを意識しながら，SymPy の使用方法について導入的なレベルで説明する．SymPy に関するより詳しい情報についてはインターネットサイト https://www.sympy.org/ を参照のこと．

3.3.1 モジュールの読込みに関する注意

SymPy は多くの関数やメソッドを提供している．そのため，クラス名や関数名などが他のパッケージと重複する可能性が大きくなるので，Python に読み込む際には注意が必要である．例えば，

```
from sympy import *
```

などとしてパッケージを読み込むと，オブジェクトの生成や関数呼び出しにおいてパッケージ名の指定を省略することができて操作が簡便である反面，他のモジュールを読み込んで併用する場合にクラスや関数の名前が衝突するという問題が起こる．Python に SymPyのみを読み込んで，数式処理ツールとして利用する場合は上記の形でモジュールを読み込んでも特に問題はないが，数式処理機能を応用プログラムに組み込んで利用する場合は，名前の衝突に関しては注意を払う必要がある．すなわち，SymPy を読み込む際に，

```
from sympy import 関数名,…
```

などとして，使用する関数名やクラス名を明に指定する方がよい．あるいは，

```
import sympy as sp
```

としてパッケージを読み込んで，各種 API にアクセスする際に，

```
sp.API 名
```

としてパッケージ名（この場合は sp）を明に指定するのが良い．

SymPy の別名：import の際に別名は自由に設定して良いが，sp という別名が一般的である．

SymPy では，多くの機能が関数としてだけでなく，各クラスのメソッドとしても実装されているので，極力メソッドの形式で数式処理機能を使用するのが安全である．

3.3.2 基礎事項

SymPy による数式の計算には通常の算術記号（+, -, *, /）が使える．また冪乗は '**' である．ただし，他の数式処理システムと違う点として，Python の変数と，数式を構成する記号は全く別のものであるということがある．次の実行例について考える．

誤った操作の例

```
>>> import sympy as sp   Enter   ← SymPy の読込み
>>> x + x   Enter                ←数式の簡単化を試みる
Traceback (most recent call last):        エラーメッセージが表示される
  File "<stdin>", line 1, in <module>
    x + x
    ^
NameError:  name 'x' is not defined
```

エラーメッセージが表示されているが，原因は，未定義の変数同士を加算しようとしたことにある．SymPy で記号的計算を実行するには，計算対象の記号を予め生成しておく必要がある．次の例について考える．

[83] 文献参照：“Open source computer algebra systems: SymPy”, David Joyner, Ondřej Čertík, et.al., *ACM Communications in Computer Algebra archive* Volume 45 Issue 3/4, Sep./Dec. 2011, pp.225-234,ACM New York, USA

[84] ウルフラム・リサーチ社の *Mathematica*，ウォータールー大学（カナダ）で開発された Maple，フリーソフトウェア（GNU GPL）の Maxima などが有名である．

正しい操作の例

```
>>> x = sp.Symbol('x')  Enter      ←'x' という記号オブジェクトを生成している
>>> x + x  Enter                   ←数式の簡単化を試みる
2*x                  ←正しい結果が得られている
```

この例では，SymPy の記号オブジェクト'x' を生成して，それを Python の変数 x に与えている．すなわち

「変数 x に SymPy の記号オブジェクト 'x' が代入されている」

と考えると良い．他の数式処理システムでは，変数と代数記号の区別がないが，SymPy の利用においては，このことを常に念頭に置く必要がある．

3.3.2.1 記号オブジェクトの生成

Symbol 関数を用いることで数式処理のための記号オブジェクトを生成することができる．

書き方： Symbol(**文字列**)

文字列として記述されたものを表す 1 つの数式記号オブジェクトを生成して返す．また関数 symbols を使用（先頭が小文字であることに注意）すると，複数の記号を空白で（あるいはコンマで）区切った形の文字列を引数に与えて，複数の数式記号を同時に生成することができる．この場合の戻り値は生成した記号オブジェクトのタプルである．またこの場合，引数に与えた文字列中に記述した順でシンボルオブジェクトが得られる．

書き方： symbols(**文字列**)

例. 記号オブジェクトの生成例

```
>>> import sympy as sp  Enter      ←パッケージの読み込み
>>> w = sp.Symbol('w')  Enter      ←数式記号の生成（1 個）
>>> (x,y,z) = sp.symbols('x y z')  Enter      ←数式記号の生成（3 個）
>>> w + x + x  Enter               ←記号的計算
w + 2*x                          ←計算結果
```

上の例に示したように，生成した記号オブジェクトは，同じ名前の変数に割り当てると，その記号を扱う上で判り易い．このような割り当て処理を簡単な形で実行するために var 関数がある．

書き方： var(**文字列**)

引数に与える「文字列」は symbols 関数の場合に準ずる．生成された記号オブジェクトは，同名の大域変数（グローバル変数）に割り当てられる．

例. 記号オブジェクトを生成して，同じ名前の変数に割り当てる（先の例の続き）

```
>>> sp.var('a,b')  Enter           ←記号オブジェクト 'a', 'b' の生成
(a, b)          ←記号オブジェクトが生成された
>>> a+b+a+b+a+2  Enter          ←それらを使った計算
3*a + 2*b + 2                ←計算結果
```

var 関数も先の symbols 関数と同様に，生成した記号オブジェクトのタプルを返す．

3.3.2.2 数式の簡単化（評価）

数式記号を生成すると，それらを算術記号（+, -, *, /）でつなげることで計算（簡単化，評価）されるが，もっと単純に，文字列として記述した数式を関数 simplify の引数に与えることでも計算結果が得られる．

書き方： simplify(**文字列**)

例. 数式の簡単化

```
>>> import sympy as sp  Enter      ←パッケージの読み込み
>>> f = sp.simplify('a + b + a + c')  Enter  ←計算
>>> f  Enter  ←確認
2*a + b + c      ←計算結果
```

この例のような処理では，数式記号を明に生成しなくても良い．すなわち，simplify 関数を使用すると，より簡単に（文字列型の式から）数式オブジェクトを生成することができる．

simplify 関数に与える文字列は，正しい数式の表現になっていなければならない．

例. 正しくない表現の文字列を与えた場合（先の例の続き）

```
>>> sp.simplify('a+*b')  Enter  ←正しくない表現を与えると
ValueError:  Error from parse_expr with transformed code:  "Symbol ('a' )+*Symbol ('b' )"

The above exception was the direct cause of the following exception:      ←エラーとなる
                    ⋮
      （SymPy 内部のエラーメッセージが多数表示される）
                    ⋮
sympy.core.sympify.SympifyError:  Sympify of expression 'could not parse 'a+*b'' failed,
because of exception being raised:
SyntaxError:  invalid syntax (<string>, line 1)
```

このように，正しくない表現を simplify 関数に与えるとエラー（SympifyError）となる．

■ 簡単化できないや式未定義の関数の評価

評価済みの既に簡単化された式を simplify に与えると，その式がそのまま返される．

例. 既に簡単化された式の評価（先の例の続き）

```
>>> sp.simplify('a+b')  Enter     ←既に簡単化された式の評価
a + b                  ←そのまま返される
```

定義されていない関数を simplify に与えると，それがそのまま返される．（エラーにはならない）

例. 未定義の関数の評価（先の例の続き）

```
>>> sp.simplify('f(x)')  Enter     ←未定義の関数 f の評価
f(x)                   ←そのまま返される
```

※ Python の様々なオブジェクトを SymPy の式に変換する関数 sympify が存在する．詳しくは SymPy の公式インターネットサイトを参照のこと．

3.3.2.3 評価なしの数式作成

先に解説した simplify は**数式の評価**が主な目的であり，与えられた文字列表現の式や SymPy の数式オブジェクトの評価結果を返す．次に解説する parse_expr 関数は，文字列として与えられた数式を SymPy の数式のオブジェクトに変換するのが主たる目的である．parse_expr 関数は SymPy の数式オブジェクトが与えられた場合は SymPy の数式オブジェクトを返すので，結果的に両関数は同じ働きをするように見える．ただし，parse_expr 関数は，オプションとして引数 'evaluate=False' を与える[85] と，数式の評価を抑制し，文字列表現の数式を評価せずにそのまま SymPy の数式オブジェクトに変換したものを返す．

例. 評価機能を抑制して数式を作成する（先の例の続き）

```
>>> sp.parse_expr( 'a+a', evaluate=False )  Enter     ←評価を抑制して数式作成
a + a           ←簡単化されていない
```

この機能は後で解説する各種の遅延実行（導関数や積分の未評価状態の保持など）の際に必要となるので重要である．

本書では数式の作成において simplify，parse_expr を適宜使い分ける形で SymPy の機能について解説する．

3.3.2.4 数式からのオブジェクトの取り出し

数式オブジェクトに対して atoms メソッドを使用すると，その数式に含まれるオブジェクトを集合の形で取得することができる．

例. 数式から Symbol オブジェクトを取り出す

```
>>> import sympy as sp  Enter       ←パッケージの読み込み
>>> s = sp.simplify('a+b+a+2*b+c+pi+f(x)+g(y)')  Enter  ←計算
>>> s  Enter              ←内容の確認
2*a + 3*b + c + f(x) + g(y) + pi       ←計算結果が保持されている
>>> s.atoms(sp.Symbol)  Enter  ←記号の取り出し
{y, x, c, a, b}          ← Symbol オブジェクト（代数記号）の集合（セット）が得られている
```

[85] デフォルトでは 'evaluate=True'.

例. 数式から数値，関数を取り出す（先の例の続き）

```
>>> s.atoms(sp.Number)  Enter  ←数値の取り出し
{2, 3}              ←数値の集合（セット）が得られている
>>> s.atoms(sp.Function)  Enter  ←関数の取り出し
{g(y), f(x)}              ←関数の集合（セット）が得られている
```

このように atoms メソッドの引数に Symbol，Number，Function を指定することで，各種オブジェクトの集合（セット）が得られるので，これを list コンストラクタでリストに変換すると，数式を構成するオブジェクトのリストを得ることができる．

式の中に含まれる，束縛されていない（値が代入されていない）変数記号は `free_symbols` プロパティから取得できる．

例. 束縛されていない Symbol の取得（先の例の続き）

```
>>> s.free_symbols  Enter     ←束縛されていない Symbol を
{a, y, b, x, c}               ← set 形式で取得
```

3.3.2.5 式 f(x,y,...) の構造 （頭部と引数列の取り出し）

f(x,y,...) の形をした式の**頭部**と**引数列**はそれぞれ，func, args プロパティから得られる．

例. 式の頭部と引数列の取り出し

```
>>> import sympy as sp  Enter       ←パッケージの読み込み
>>> s = sp.simplify('f(x,y,z)')  Enter     ←式の生成
>>> sf = s.func  Enter       ←頭部の取り出し
>>> sf  Enter            ←結果を調べる
f                        ←頭部が得られているのがわかる
>>> sa = s.args  Enter     ←引数列の取り出し
>>> sa  Enter            ←結果を調べる
(x, y, z)                ←引数の列（タプル）が得られている
```

頭部が sf に，引数のタプルが sa に得られている．

式の頭部として得られたオブジェクトは，そのまま式の構成に利用できる．（次の例）

例. 式の再構成（先の例の続き）

```
>>> sf( sa[0], sa[1], sa[2] )  Enter     ←式の再構成
f(x, y, z)                ←元の式と同じものが構成されている
>>> sf( 1, 2, 3 )  Enter         ←引数を変えて式を構成する例
f(1, 2, 3)                ←構成結果
```

以上のことは算術で構成された式についても同様である．

例. 多項式の頭部と引数列の取り出し（先の例の続き）

```
>>> s = sp.simplify('x+y+z')  Enter     ←式の生成
>>> sf = s.func  Enter       ←頭部の取り出し
>>> sf  Enter            ←結果を調べる
<class 'sympy.core.add.Add'>     ←頭部（加算演算子）が得られているのがわかる
>>> sa = s.args  Enter     ←引数列の取り出し
>>> sa  Enter            ←結果を調べる
(x, y, z)                ←引数の列（タプル）が得られている
```

例. 式の再構成（先の例の続き）

```
>>> sf( sa[0], sa[1], sa[2] )  Enter     ←式の再構成
x + y + z                ←元の式と同じものが生成されている
>>> sf(1,2,3)  Enter         ←引数を変えて式を構成する例
6                ←再構成の結果
```

3.3.2.6　定数

数式の中で使用できる定数の一部を表 38 に示す.

表 38: SymPy の定数（一部）

定　数	解　説	定　数	解　説
E	ネイピア数	pi	円周率
I	虚数単位	nan	非数
oo	正の無限大	zoo	複素無限大
GoldenRatio	黄金比　$\varphi = \frac{1+\sqrt{5}}{2}$		

注意） oo，zoo，nan は厳密には数ではない.

例. 定数の使用

```
>>> sp.simplify('sin(pi)')   Enter   ← sin(π) の計算
0          ←計算結果
>>> sp.simplify('I * I')   Enter   ← i∗i の計算
-1         ←計算結果
```

定数は sympy パッケージのオブジェクトとしても使用できる．すなわち，

```
sp.E, sp.pi, sp.I, sp.oo
```

などとして参照することができる.

3.3.2.7　数式の表示に関すること

SymPy の数式オブジェクトは，使用するインターフェース（コマンドプロンプトウィンドウ，IPython ノートブック，Web のノートブックなど）に応じて異なる出力形式になることがある．また，出力のためのメソッドによっても異なることがある．例えば積分の式[86]

```
s = sp.parse_expr('Integral(f(x),x)')
```

を通常の print 関数で　`print(s)`　として出力すると，出力環境にかかわらず

```
Integral(f(x), x)
```

と出力される．また，SymPy の pprint 関数で　`sp.pprint(s)`　として出力すると，出力環境にかかわらず

```
⌠
⎮ f(x) dx
⌡
```

のように文字の 2 次元配置[87] で出力される.

数式の清書機能を備えたインターフェース[88] で SymPy の数式オブジェクトを直接出力すると

$$\int f(x)dx$$

のように TEX ベースの清書出力がなされることがある.

3.3.3　基本的な数式処理機能

先に説明した simplify に加えて，次に挙げるような基本的な数式処理機能が使用できる.

3.3.3.1　式の展開

expand メソッドを使用すると数式を展開することができる.

例. 式の展開

```
>>> s = sp.simplify('(a+b)**10')   Enter   ←数式の生成
>>> s2 = s.expand()   Enter        ←展開の処理
>>> s2   Enter                     ←結果の確認
a**10 + 10*a**9*b + 45*a**8*b**2 + 120*a**7*b**3 + 210*a**6*b**4 + 252*a**5*b**5 +
210*a**4*b**6 + 120*a**3*b**7 + 45*a**2*b**8 + 10*a*b**9 + b**10     ←処理結果
```

[86] 後の「3.3.4.5 integrate の遅延実行」（p.198）で解説する.

[87] キャラクターアート

[88] Jupyter Notebook（Anaconda や Google Colaboratory のノートブック）が有名.

同様の処理を　sp.expand(s)　として関数の形式で実行することもできる.

3.3.3.2　因数分解

factor メソッドを使用すると数式の因数分解ができる.

例.（先の例の続き）

```
>>> s2.factor()  Enter        ←因数分解の処理
(a + b)**10     ←処理結果
```

同様の処理を　sp.factor(s2)　として関数の形式で実行することもできる.

3.3.3.3　指定した記号による整理

collect メソッドを使用すると，指定した記号で整理することができる.

例. collect による式の整理

```
>>> s = sp.simplify('(a+b+x)**2').expand()  Enter  ←数式の生成：(a + b + x)^2 の展開
>>> s  Enter          ←結果の確認
a**2 + 2*a*b + 2*a*x + b**2 + 2*b*x + x**2      ←処理結果
>>> s.collect('x')  Enter          ←記号 x で整理
a**2 + 2*a*b + b**2 + x**2 + x*(2*a + 2*b)      ←処理結果
```

同様の処理を　sp.collect(s,'x')　として関数の形式で実行することもできる.

3.3.3.4　約分：分数の簡単化 (1)

cancel メソッドを使用すると，分数を約分することができる．複雑な分数

$$\frac{ax^2+2axy+ay^2+bx^2+2bxy+by^2}{acx+acy+adx+ady+bcx+bcy+bdx+bdy}$$

が約分される様子を例示する.

例. 約分

```
>>> s1 = sp.simplify('a*x**2 + 2*a*x*y + a*y**2 +
           b*x**2 + 2*b*x*y + b*y**2')  Enter        ←数式の生成 (分子)
>>> s2 = sp.simplify('a*c*x + a*c*y + a*d*x + a*d*y +
           b*c*x + b*c*y + b*d*x + b*d*y')  Enter        ←数式の生成 (分母)
>>> s = s1 / s2  Enter                          ←複雑な分数の作成
>>> s  Enter          ←結果の確認
(a*x**2 + 2*a*x*y + a*y**2 + b*x**2 + 2*b*x*y + b*y**2)/
  (a*c*x + a*c*y + a*d*x + a*d*y + b*c*x + b*c*y + b*d*x + b*d*y)   ←処理結果 (複雑な分数)
>>> s.cancel()  Enter  ←約分の実行
(x + y)/(c + d)     ←処理結果
```

約分した結果が

$$\frac{x+y}{c+d}$$

として得られている.

同様の処理を　sp.cancel(s)　として関数の形式で実行することもできる.

3.3.3.5　部分分数

apart メソッドを使用すると，分数を部分分数にすることができる.（ただし，複数の記号オブジェクトからなる分数は処理できない）　分数

$$\frac{5x^3+6x^2+x+4}{x^4+4x^3+4x^2+4x+3}$$

が部分分数に変形される様子を例示する.

例. 部分分数

```
>>> s1 = sp.simplify('5*x**3 + 6*x**2 + x + 4')  Enter  ←数式の生成 (分子)
>>> s2 = sp.simplify('x**4 + 4*x**3 + 4*x**2 + 4*x + 3')  Enter  ←数式の生成 (分母)
>>> s = s1 / s2  Enter  ←1 つの長い分数の作成
>>> s  Enter  ←結果の確認
(5*x**3 + 6*x**2 + x + 4)/(x**4 + 4*x**3 + 4*x**2 + 4*x + 3)  ←処理結果 (1 つの長い分数)
>>> s3 = s.apart()  Enter  ←部分分数への変形
>>> s3  Enter  ←結果の確認
-1/(x**2 + 1) + 4/(x + 3) + 1/(x + 1)  ←処理結果
```

部分分数 $$-\frac{1}{x^2+1}+\frac{4}{x+3}+\frac{1}{x+1}$$

に変換されていることがわかる.

同様の処理を `sp.apart(s)` として関数の形式で実行することもできる.

3.3.3.6 分数の簡単化 (2)

ratsimp メソッドを使用すると，分数をまとめることができる.（通分の処理を含む） 先の例（部分分数分解）で得られた結果の s3 に対して ratsimp を適用した例を示す.

例. 1 つの分数にまとめる（先の例の続き）

```
>>> s3.ratsimp()  Enter  ←簡単化の処理
(5*x**3 + 6*x**2 + x + 4)/(x**4 + 4*x**3 + 4*x**2 + 4*x + 3)  ←処理結果
```

元の式に戻り，1 つの分数としてまとめられていることがわかる.

同様の処理を `sp.ratsimp(s3)` として関数の形式で実行することもできる.

3.3.3.7 代入（記号の置換）

subs メソッドを使用すると，記号（Symbol）を別の記号に置き換えることができる.

例. 代入（記号の置き換え）処理

```
>>> (a,b,x) = sp.symbols('a b x')  Enter  ←記号の生成
>>> s = 2*a + 3*b  Enter  ←式の生成
>>> s.subs(a,x)  Enter  ←記号 a を記号 x に置き換える
3*b + 2*x  ←処理結果
>>> s.subs(a+b,x)  Enter  ←式を別のものに置き換えることは…
2*a + 3*b  ←できない.
```

この例の様に，式を別のものに置き換えることはできない．複数の記号の置換処理には，置換規則を辞書オブジェクトにして与える.

例. 複数の記号の置き換え（先の例の続き）

```
>>> s.subs( {a:x,b:1} )  Enter  ←記号 a を記号 x に，b を 1 に置き換える
2*x + 3  ←処理結果
```

3.3.3.8 各種の数学関数

SymPy では，Python の math モジュールが提供する各種の数学関数と同じ名前のものが概ね使用できるが，対数関数は log の表記を基本とする．ln の表記でも入力できるが，それは log として扱われる.

例. 対数関数（先の例の続き）

```
>>> sp.simplify('ln(1)')  Enter  ← ln の表記も使用できる
0  ←計算結果
>>> sp.simplify('ln(x)')  Enter  ← ln を代数的に記述すると
log(x)  ← log の表記に統一される
```

3.3.3.9 式の型

SymPy で扱う式には様々な「型」があり，'is_' の接頭辞で始まるプロパティでそれを検査できる.

例. オブジェクトの型の検査

```
>>> import sympy as sp   Enter      ←モジュールの読込み
>>> s = sp.Symbol('x')   Enter   ← s にシンボルをセット
>>> s.is_symbol   Enter   ← s がシンボルかどうかを検査
True          ←真である
>>> s.is_number   Enter   ← s が数値かどうかを検査
False         ←偽である
```

この例が示すように，検査対象のオブジェクトの 'is_' で始まるプロパティからその型がわかる．is_で始まるプロパティの一部を表 39 に示す．

表 39: オブジェクトの型を検査する 'is_' プロパティ（一部）

プロパティ	解説	プロパティ	解説
is_number is_Number	数値かどうかを検査する． （←こちらが推奨される）	is_symbol is_Symbol	シンボルかどうかを検査する． （←こちらが推奨される）
is_Add	加算の式かどうかを検査する．	is_Mul	乗算の式かどうかを検査する．
is_Pow	べき乗の式かどうかを検査する．		

is_で始まるプロパティにはこの他にも様々な検査をするものがあるが，詳しくは SymPy のドキュメントの「assumptions」に関する解説を参照のこと．

3.3.4 解析学的処理

ここでは，微分と積分を基本とする解析学的な処理機能について説明する．

3.3.4.1 極限

与えられた式の極限を求めるには limit メソッドを使用する．

書き方： **式**.limit(**対象の変数, 向かう極限**)

次に，

$$\lim_{x \to 1} \frac{1}{x-1}$$

を求める例を示す．

例. 極限（先の例の続き）

```
>>> x = sp.symbols('x')   Enter      ←記号 x の生成
>>> s = 1 / (x - 1)   Enter              ←式 1/(x-1) の生成
>>> s.limit(x,1)   Enter          ← x → 1 の極限を求める
oo                        ←処理結果：∞
```

ただしこの例の式では，x の数直線上における「右から左」の極限であり,「左から右」の極限では計算結果が異なる．極限に向かう方向を指定して厳密に計算するには，limit メソッドに 3 番目の引数として '+' もしくは '-' を与える．（次の例を参照）

例. 方向を指定した極限（先の例の続き）

```
>>> s.limit(x,1,'+')   Enter      ←「右から左」の極限
oo                       ←処理結果：∞
>>> s.limit(x,1,'-')   Enter      ←「左から右」の極限
-oo                      ←処理結果：−∞
```

この例において同様の処理を　`sp.limit(s,x,1,'-')`　として関数の形式で実行することもできる．

3.3.4.2 導関数

与えられた式を，**ある変数を定義域として値域を与える関数**として見た場合，diff メソッドを使用して，その変数についての導関数を求める（偏微分する）ことができる．

例. 導関数（先の例の続き）

```
>>> x = sp.symbols('x')  Enter          ←記号 x の生成
>>> s = sp.simplify('sin(x)')  Enter  ←式の生成
>>> s.diff(x)  Enter          ← x についての導関数を求める
cos(x)                        ←処理結果
```

これは，

$$\frac{d}{dx}\sin(x)$$

を求めた例である．

この例において同様の処理を　`sp.diff(s,x)`　として関数の形式で実行することもできる．

3.3.4.3　微分操作の遅延実行

Derivative クラスを使用すると，parse_expr 関数を使って，導関数を求める処理（微分操作）を**遅延実行**することができる．

例. 微分操作の遅延実行

```
>>> s = sp.parse_expr('Derivative(sin(x),x)',evaluate=False) Enter ←式の生成（評価せず）
>>> s  Enter                  ←結果の確認
Derivative(sin(x), x)         ←元のままの式が得られている
>>> s.doit()  Enter           ←式の「実行」
cos(x)                        ←処理結果：導関数が得られている
```

この例のように doit メソッドを使用すると微分操作が実行される．Derivative を用いた導関数の表現は，微分演算の抽象的な表現を数式として記述可能[89] にする．これにより，微分方程式を記述することが可能となるだけでなく，後の「3.3.10.1 LaTeX」のところで説明する書式変換などにおいても有効な表現手段を与える．

注意）simplify 関数を使用する，あるいは parse_expr 関数で引数 evaluate=False）を与えない場合は，遅延実行の式も評価されてしまうことがあるので注意すること．（次の例）

例. 遅延実行されないケース（先の例の続き）

```
>>> sp.simplify('Derivative(sin(x),x)')  Enter    ←これは
cos(x)                  ←即座に評価されてしまう
```

3.3.4.4　原始関数

先の導関数の算出と逆の処理をするには integrate メソッドを使用する．

例. 原始関数（先の例の続き）

```
>>> x = sp.symbols('x')  Enter          ←記号 x の生成
>>> s = sp.simplify('cos(x)')  Enter    ←式の生成
>>> s.integrate(x)  Enter          ← diff(x) と逆の処理
sin(x)                          ←処理結果
```

得られた値に定数項が付いていないので，厳密な意味ではこの処理では原始関数を求めたことにはならない．厳密な意味での原始関数を求めるには「3.3.5 各種方程式の求解」で説明する**微分方程式の求解**の方法を参照のこと．

この例において同様の処理を　`sp.integrate(s,x)`　として関数の形式で実行することもできる．

3.3.4.5　integrate の遅延実行

Integral クラスを使用すると，parse_expr 関数を使って，integrate による処理を**遅延実行**することができる．

例. integrate の遅延実行

```
>>> s = sp.parse_expr('Integral(cos(x),x)',evaluate=False) Enter ←式の生成（評価なし）
>>> s  Enter                  ←結果の確認
Integral(cos(x), x)           ←元のままの式が得られている
>>> s.doit()  Enter           ←式の「実行」
sin(x)                        ←処理結果
```

[89]関数に対する操作であるので，Derivative は広義の**汎関数**である．

注意）simplify 関数を使用する，あるいは parse_expr 関数で引数 evaluate=False）を与えない場合は，遅延実行の式も評価されてしまうことがあるので注意すること．（次の例）

例. 遅延実行される／されないケース（先の例の続き）

```
>>> sp.simplify('Integral(1,x)')  Enter    ←この遅延実行は
Integral(1, x)    ←評価されない
>>> sp.simplify('Integral(0,x)')  Enter    ←この遅延実行は
0                 ←即座に評価される
```

3.3.4.6　定積分

integrate を使用して定積分を求めることができる．

例. $\int_1^{\infty} \frac{1}{x^2}\, dx$ を求める

```
>>> x = sp.symbols('x')  Enter          ←記号 x の生成
>>> inf = sp.simplify('oo')  Enter      ←無限大記号の生成
>>> s = sp.simplify('1/x**2')  Enter    ←関数の生成
>>> s.integrate( (x,1,inf) )  Enter     ←定積分の実行
1                    ←積分結果
```

遅延実行の形で同様の処理を行うこともできる．（次の例参照）

例. 先の例の遅延実行（先の例の続き）

```
>>> s = sp.parse_expr('Integral(1/x**2,(x,1,oo))',evaluate=False) Enter  ←式の生成
>>> s  Enter    ←式の確認
Integral(1/x**2, (x, 1, oo))      ←実行が遅延されている
>>> s.doit()  Enter                ←処理の実行
1                    ←積分結果
```

3.3.4.7　級数展開

与えられた式の級数展開（テイラー展開／マクローリン展開）を得るには series メソッドを使用する．

例. $\exp(x)$ の展開

```
>>> x = sp.symbols('x')  Enter          ←記号 x の生成
>>> s = sp.simplify('exp(x)')  Enter    ←式の生成
>>> s.series(x)  Enter                  ←級数展開
1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + O(x**6)   ← 6 番目の項まで計算される
>>> s.series(x,0,8)  Enter              ←級数展開：0 を起点に 8 番目まで
1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + x**6/720 + x**7/5040 + O(x**8)   ←処理結果
>>> s.series(x,1,6)  Enter              ←級数展開：1 を起点に 6 番目まで
E + E*(x - 1) + E*(x - 1)**2/2 + E*(x - 1)**3/6 + E*(x - 1)**4/24 +
                    E*(x - 1)**5/120 + O((x - 1)**6, (x, 1))      ←処理結果
```

この例において同様の処理を　sp.series(s,x)　のようにして関数の形式で実行することもできる．

3.3.5　各種方程式の求解

ここでは，各種の方程式の求解のためのメソッドを紹介する．

3.3.5.1　代数方程式の求解

代数方程式の解を求めるには solve 関数を使用する．

1) $f(x) = 0$ の x についての求解

例. $x + 1 = 0$ を x について解く

```
>>> x = sp.symbols('x')  Enter          ←記号 x の生成
>>> s = sp.simplify('x+1')  Enter       ←式の生成（=0 の部分は書かない）
>>> sp.solve(s,x)  Enter                ← s = 0 を満たす x の求解
[-1]                 ←解は x = -1 の 1 個
```

2) $f_1(x) = f_2(x)$ の x についての求解

等式を表現するには Eq を使用する．

例. $2x + 1 = 3x - 5$ を x について解く

```
>>> x = sp.symbols('x')  Enter          ←記号 x の生成
>>> s = sp.simplify('Eq(2*x+1,3*x-5)')  Enter     ←方程式の生成
>>> sp.solve(s,x)  Enter              ← s を満たす x の求解
[6]                        ←解は x = 6 の 1 個
```

■ 等式オブジェクト：Eq

Eq は 2 つの式から成る等式を表すオブジェクトである．

書き方：　Eq(左辺, 右辺)

「左辺=右辺」の等式を表す．両辺が明らかに等しい場合は True，明らかに等しくない場合は False となる．

例. 等式を表す Eq

```
>>> sp.simplify( 'Eq(f(x),g(x))' )  Enter     ←等式 f(x) = g(x)
Eq(f(x), g(x))
>>> sp.simplify( 'Eq(1,1)' )  Enter     ←両辺が明らかに等しい場合
True
>>> sp.simplify( 'Eq(1,0)' )  Enter     ←両辺が明らかに等しくない場合
False
>>> sp.parse_expr( 'Eq(1,0)', evaluate=False )  Enter  ←明らかな場合は parse_expr でも
False          ←即座に評価される
```

Eq は，方程式をはじめ，等式を扱う際に用いられる．

solve 関数では 4 次の代数方程式まで一般解が得られる．

例. 2 次の代数方程式 $ax^2 + bx + c = 0$ の求解

```
>>> x = sp.symbols('x')  Enter          ←記号 x の生成
>>> s = sp.simplify('a*x**2 + b*x + c')  Enter     ←方程式の生成
>>> sp.solve(s,x)  Enter              ← s = 0 を満たす x の求解
[(-b + sqrt(-4*a*c + b**2))/(2*a), -(b + sqrt(-4*a*c + b**2))/(2*a)]  ←解は 2 個
```

■ 連立方程式の求解

solve 関数に与える方程式と変数をリスト（タプルも可）にして与えることで連立方程式を解くことができる．次の例は

$$\begin{cases} ax + by = e \\ cx + dy = f \end{cases}$$

の解を求めるものである．

例. 連立方程式の求解

```
>>> eq = sp.parse_expr('[a*x + b*y - e, c*x + d*y - f]')  Enter  ←方程式の作成
>>> eq  Enter     ←確認
[a*x + b*y - e, c*x + d*y - f]     ←方程式のリスト
>>> v = sp.parse_expr('[x,y]')  Enter     ← Symbol のリストの作成
>>> v  Enter     ←確認
[x, y]     ← Symbol のリスト
>>> sp.solve(eq,v)  Enter          ←求解
{x:  (-b*f + d*e)/(a*d - b*c), y:  (a*f - c*e)/(a*d - b*c)}     ←解が得られている
```

このように，解は辞書オブジェクトの形で得られる．

3.3.5.2　微分方程式の求解

微分方程式の解を求めるには dsolve 関数を使用する．この関数には，解くべき方程式と，求めるべき解の関数を引数に指定する．

例. $\dfrac{d}{dx}f(x) - \dfrac{1}{\sin(x)} = 0$ の $f(x)$ についての求解

```
>>> eq = sp.parse_expr('Derivative(f(x),x)-1/sin(x)')  Enter    ←微分方程式
>>> f = sp.simplify('f(x)')  Enter          ←求めるべき関数 f(x)
>>> sol = sp.dsolve(eq,f)  Enter          ←求解
>>> sol     Enter                          ←解の確認
Eq(f(x), C1 + log(cos(x) - 1)/2 - log(cos(x) + 1)/2)     ←解
```

解として，　$f(x) = C_1 + \dfrac{1}{2}\log(\cos(x)-1) - \dfrac{1}{2}\log(\cos(x)+1)$　が得られている．(C_1 は定数項)

dsolve 関数の引数に与える方程式は `Eq(...)` の形でも良い．

不定積分によって原始関数を求める場合は dsolve 関数を使用する．

SymPy の dsolve には解けない微分方程式も多数存在する．次に示す例は，非線形の微分方程式

$$\frac{d}{dx}f(x) = f(x)^2 + x$$

の求解を試みるものである．

例. 求解できないケース（非線形微分方程式の一例：Riccati 方程式）

```
>>> eq = sp.parse_expr('Derivative(f(x),x)-f(x)**2-x')  Enter    ←微分方程式
>>> f = sp.simplify('f(x)')  Enter          ←求めるべき関数 f(x)
>>> sp.dsolve(eq,f)  Enter          ←求解を試みると
Traceback (most recent call last):          ←エラーとなる
  File "<stdin>", line 1, in <module>
    sp.dsolve(eq,f)
    ^^^^^^^^^^^^^^^
                :
          (途中省略)
                :
TypeError:  bad operand type for unary -:  'list'
```

■ 偏微分方程式

偏微分方程式の解を求めるには pdsolve 関数を使用する．次の例は，偏微分方程式

$$\frac{\partial}{\partial x}u(x,y) + \frac{\partial}{\partial y}u(x,y) = 0$$

において $u(x,y)$ を求めるものである．

例. 偏微分方程式の求解

```
>>> eq = sp.parse_expr('Derivative(u(x,y),x)+Derivative(u(x,y),y)')  Enter    ←偏微分方程式
>>> sol = sp.pdsolve(eq)  Enter          ←求解
>>> sol     Enter                          ←解の確認
Eq(u(x, y), F(x - y))     ←解
```

解として　$u(x,y) = F(x-y)$　が得られている．(F は任意の関数)

dsolve の場合と同様に，pdsolve の第 2 引数に求めるべき関数を指定することができる．

例. 上と同じ処理（先の例の続き）

```
>>> f = sp.simplify('u(x,y)')  Enter          ←求めるべき関数 f(x)
>>> sp.pdsolve(eq,f)  Enter          ←求解
Eq(u(x, y), F(x - y))     ←解
```

以下に，偏微分方程式の事例をいくつか挙げる．

事例）減衰項を加えた一次元の移流方程式（advection equation）

$$\frac{\partial}{\partial t}u(x,t) + c\frac{\partial}{\partial x}u(x,t) + a{\cdot}u(x,t) = 0$$

例. 求解処理

```
>>> eq = sp.parse_expr('Derivative(u(x,t),t)+c*Derivative(u(x,t),x)+a*u(x,t)')  Enter
>>> sp.pdsolve(eq)  Enter          ←求解
Eq(u(x, t), F(-c*t + x)*exp(-a*(c*x + t)/(c**2 + 1)))     ←解
```

解が　$u(x,t) = F(x-ct)\cdot \exp\left(-\dfrac{a(cx+t)}{c^2+1}\right)$　として得られていることがわかる．

SymPy の pdsolve には解けない微分方程式も多数存在する．

事例）波動方程式（求解できないケース）

$$\frac{\partial^2}{\partial t^2}u(x,t) + c^2\frac{\partial^2}{\partial x^2}u(x,t) = 0$$

例. 求解の試み

```
>>> eq = sp.parse_expr('Derivative(u(x,t),t,2)-c**2*Derivative(u(x,t),x,2)')  Enter
>>> sp.pdsolve(eq)  Enter          ←求解を試みると
Traceback (most recent call last):          ←エラーとなる
  File "<stdin>", line 1, in <module>
    sp.pdsolve(eq)
    ^^^^^^^^^^^^^^
                :
          (途中省略)
                :
    raise NotImplementedError(dummy + "solve" + ":  Cannot solve " + str(eq))
NotImplementedError:  psolve:  Cannot solve -c**2*Derivative(u(x, t), (x, 2)) +
Derivative(u(x, t), (t, 2))
```

3.3.5.3　階差方程式の求解（差分方程式，漸化式）

rsolve 関数を使用すると，階差方程式（差分方程式）を解くことができる．これは漸化式の一般化も含む．

例. $f(n+1) - rf(n) = 0$ の $f(n)$ についての求解

```
>>> s = sp.parse_expr('f(n+1)-r*f(n)')  Enter  ←階差方程式の作成
>>> f = sp.simplify('f(n)')  Enter       ←求めるべき関数 f(n)
>>> sp.rsolve(s,f)  Enter                ←求解
C0*r**n                                  ←解
```

解が $C_0 \cdot r^n$ として得られている．

また，初期値を辞書オブジェクトの形で与えることもできる．

例. （先の続き）初期値 $f(0) = a$ を与える

```
>>> ini = sp.simplify('{f(0):a}')  Enter  ←初期値の生成
>>> sp.rsolve(s,f,ini)  Enter             ←求解
a*r**n                                    ←解
```

解が $a \cdot r^n$ として得られている．

SymPy の rsolve には解けない階差方程式も多数存在する．次に示す例は，非線形の階差方程式　$a_{n+1} = a_n^2 + 1$ の求解を試みるものである．

例. 求解できないケース：非線形階差方程式の一例

```
>>> s = sp.parse_expr('a(n+1)-a(n)**2-1')  Enter  ←階差方程式の作成
>>> f = sp.simplify('a(n)')  Enter       ←求めるべき関数 f(n)
>>> sp.rsolve(s,f)  Enter                ←求解を試みると
Traceback (most recent call last):        ←エラーとなる
  File "<stdin>", line 1, in <module>
    sp.rsolve(s,f)
    ^^^^^^^^^^^^^^
  …(途中省略)…
    raise ValueError(
        "'%s(%s + k)' expected, got '%s'" % (y.func, n, h))
ValueError:  'a(n + k)' expected, got 'a(n)**2'
```

SymPy の rsolve が対象としていないタイプの方程式であるという旨のエラーが起こっている．

3.3.6 線形代数

Matrix クラスを使用すると行列が表現できる．例えば，

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

を SymPy のオブジェクトとして生成するには次のようにする．

例. 行列の作成

```
>>> sp.var('a b c d')  Enter  ←記号の生成
(a, b, c, d)          ←生成された記号
>>> m1 = sp.Matrix([[a,b],[c,d]])  Enter  ←行列の生成
>>> sp.pprint(m1)  Enter  ←整形表示
```

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$ ←表示結果

この例の様に pprint 関数を使用すると，行列を整形表示する．この関数は行列以外の数式にも使用することができる．

Matrix オブジェクトの和，差，積には通常の算術記号が使用できる．

例. 行列の和，差，積（先の例の続き）

```
>>> sp.var('e f g h')  Enter  ←記号の生成
(e, f, g, h)          ←生成された記号
>>> m2 = sp.Matrix([[e,f],[g,h]])  Enter  ←行列の生成
>>> sp.pprint(m2)  Enter  ←整形表示
```

$$\begin{bmatrix} e & f \\ g & h \end{bmatrix}$$ ←表示結果

```
>>> sp.pprint( m1 + m2 )  Enter  ←行列同士の和
```

$$\begin{bmatrix} a+e & b+f \\ c+g & d+h \end{bmatrix}$$ ←和

```
>>> sp.pprint( m1 - m2 )  Enter  ←行列の差
```

$$\begin{bmatrix} a-e & b-f \\ c-g & d-h \end{bmatrix}$$ ←差

```
>>> sp.pprint( m1 * m2 )  Enter  ←行列の積
```

$$\begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$ ←積

3.3.6.1 行列の連結

Matrix.hstack，Matrix.vstack メソッドを用いると水平，垂直の方向に行列を連結することができる．

例. 水平方向の連結（先の例の続き）

```
>>> mh = sp.Matrix.hstack( m1, m2 )  Enter  ←行列の水平連結
>>> sp.pprint(mh)  Enter  ←整形表示
```

$$\begin{bmatrix} a & b & e & f \\ c & d & g & h \end{bmatrix}$$ ←連結結果（2 行 4 列）

例. 垂直方向の連結（先の例の続き）

```
>>> mv = sp.Matrix.vstack( m1, m2 )  Enter  ←行列の垂直連結
>>> sp.pprint(mv)  Enter  ←整形表示
```

$$\begin{bmatrix} a & b \\ c & d \\ e & f \\ g & h \end{bmatrix}$$ ←連結結果（4 行 2 列）

3.3.6.2　行列の形状

行列の形状（行，列のサイズ）は shape プロパティ[90] から得られる．

例. shape プロパティ（先の例の続き）

```
>>> mv.shape  Enter  ←行数と列数を調べる
(4, 2)  ← 4 行 2 列
```

この例のように，行と列のサイズのタプルが得られる．

3.3.6.3　行列の要素へのアクセス

Matrix オブジェクトの行や列は row(n)，col(m) メソッドで n 行，m 列（n,m はインデックス位置）を参照することができる．

例. 行の参照（先の例の続き）

```
>>> sp.pprint( mv.row(1) )  Enter  ← mv のインデックス 1 番目の行を整形表示
```

$$\begin{bmatrix} c & d \end{bmatrix}$$ ←参照した行

例. 列の参照（先の例の続き）

```
>>> sp.pprint( mv.col(0) )  Enter  ← mv のインデックス 0 番目の列を整形表示
```

$$\begin{bmatrix} a \\ c \\ e \\ g \end{bmatrix}$$ ←参照した列

row，col メソッドは Matrix オブジェクトを返す．

Matrix オブジェクトにはスライス ‘[n]’ を付けて n 番目の要素にアクセス（参照，値の設定）することができる．

例. Matrix オブジェクトの要素の参照（先の例の続き）

```
>>> mv[0],mv[1],mv[2],mv[3],mv[4],mv[5],mv[6],mv[7]  Enter  ← mv の全要素を並べたタプル
(a, b, c, d, e, f, g, h)    ←行列の左上から右下にかけて順に並んでいる
```

例. Matrix オブジェクトに直接的に要素を与える（先の例の続き）

```
>>> mv[2] = 2  Enter  ← mv のインデックス位置 2 の要素として「2」を与える
>>> mv[3] = 3  Enter  ← mv のインデックス位置 3 の要素として「3」を与える
>>> sp.pprint(mv)  Enter  ← mv を整形表示
```

$$\begin{bmatrix} a & b \\ 2 & 3 \\ e & f \\ g & h \end{bmatrix}$$ ←指定した位置に要素が与えられている

3.3.6.4　行列式

行列オブジェクトに対して det メソッドを使用すると行列式を得ることができる．

例. 行列式（先の例の続き）

```
>>> m1.det()  Enter  ←行列式を求める
a*d - b*c          ←処理結果
```

3.3.6.5　逆行列

正則な行列オブジェクトに対して inv メソッドを使用すると逆行列を得ることができる．

[90] NumPy の ndarray が持つ shape プロパティと似ている．

例. 逆行列（先の例の続き）

```
>>> im = m1.inv()   Enter   ←行列式を求める
>>> sp.pprint(im)   Enter   ←整形表示
```

$$\begin{bmatrix} \dfrac{d}{ad-bc} & -\dfrac{b}{ad-bc} \\ -\dfrac{c}{ad-bc} & \dfrac{a}{ad-bc} \end{bmatrix}$$

←表示結果

3.3.6.6 行列の転置

Matrix オブジェクトに対して transpose メソッドを使用すると，それを転置したものを返す．また，元の Matrix オブジェクトには変更はない．

例. 行列の転置

```
>>> sp.var('a b c d e f g h i')   Enter   ←記号の生成
(a, b, c, d, e, f, g, h, i)   ←生成された記号
>>> m = sp.Matrix([[a,b,c],[d,e,f],[g,h,i]])   Enter   ←行列の作成
>>> sp.pprint( m )   Enter   ←整形表示
```

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

←表示結果

```
>>> sp.pprint( m.transpose() )   Enter   ←転置処理（整形表示）
```

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

←転置された行列

3.3.6.7 ベクトル，内積

n 次元のベクトルは n 行 1 列の Matrix オブジェクトとして扱う．

例. 3 次元のベクトル

```
>>> sp.var('a b c d e f')   Enter   ←記号の生成
(a, b, c, d, e, f)   ←生成された記号
>>> v1 = sp.Matrix([a,b,c])   Enter   ←ベクトル v1 = (a, b, c) の作成
>>> v2 = sp.Matrix([d,e,f])   Enter   ←ベクトル v2 = (d, e, f) の作成
>>> sp.pprint(v1)   Enter   ←ベクトル v1 の整形表示
```

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

← 3 次元のベクトル（3 行 1 列の Matrix）

ベクトルの内積は dot メソッドで求める．

例. ベクトルの内積（先の例の続き）

```
>>> v1.dot(v2)   Enter   ←ベクトルの内積 v1 · v2 を求める
a*d + b*e + c*f   ←内積
```

3.3.6.8 固有値，固有ベクトル

行列オブジェクトに対して eigenvals メソッドを使用すると，固有値を求めることができる．固有ベクトルも共に求める場合は eigenvects メソッドを使用する．

例. サンプル行列の作成

```
>>> m = sp.Matrix([[3,1],[2,4]])   Enter   ←行列の作成
>>> sp.pprint(m)   Enter   ←整形表示
```

$$\begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix}$$

←表示結果

例. 固有値，固有ベクトルの算出（先の例の続き）

```
>>> m.eigenvals()  Enter  ←固有値を求める
{5:  1, 2:  1}     ←固有値と代数的重複度[91] の辞書オブジェクトが得られる
>>> ev = m.eigenvects()  Enter  ←固有値と固有ベクトルを求める
>>> sp.pprint(ev)  Enter  ←整形表示
```

$$\left[\left(2,\quad 1,\quad \left[\begin{bmatrix} -1 \\ \\ 1 \end{bmatrix}\right]\right),\quad \left(5,\quad 1,\quad \left[\begin{bmatrix} \frac{1}{2} \\ \\ 1 \end{bmatrix}\right]\right)\right] \qquad \leftarrow\text{表示結果}$$

3.3.7 総和

総和を表す式として Sum がある．これは総和を意味する式であり，doit メソッドにより評価される．

書き方： Sum(**式,(変数, 初期値, 終了値)**)

例えば Sum(f(k), (k, k0, n)) という式は，

$$\sum_{k=k_0}^{n} f(k)$$

を意味する．

例. 初項 a_1，公差 d の等差数列 $a_1, a_2, \cdots, a_n$ の n 番目までの総和

```
>>> s = sp.simplify('Sum( a1+(k-1)*d, (k,1,n) )')  Enter  ←一般項 a1 + (k − 1)d の形で与える
>>> sp.simplify( s.doit() )  Enter  ←評価の実行
n*(2*a1 + d*n - d)/2  ←評価結果
```

この例でもわかるように，Sum は遅延実行される式であり，doit により実際に評価される．

3.3.8 パターンマッチ

SymPy には数式の記号代数的な構造に沿った**パターンマッチ**のための機能が提供されている．例えば，

$$3x^2 + 5x + 1$$

という式があった場合に，x の 2 次の部分の係数とそれ以外の部分を取り出すことを考える．
この数式の記号的代数的な構造を

$$P_1\, {P_2}^2 + P_3$$

という**パターン**と見做して P_1, P_2, P_3 に該当する（マッチする）部分を元の式から抽出すると，

$$P_1 = 3,\ P_2 = x,\ P_3 = 5x + 1$$

のように対応する．SymPy ではこのような形でのパターンマッチが可能である．ここで示した例にある P_1, P_2, P_3 は SymPy においては Wild **オブジェクト**として扱われる．次に SymPy による例を手順を追って示す．

【例】 $3x^2 + 5x + 1$ を $P_1\, {P_2}^2 + P_3$ にマッチさせる

手順 0. モジュールの読込みと対象となる式の用意

```
>>> import sympy as sp  Enter  ← SymPy モジュールの読込み
>>> f = sp.sympify('3*x**2+5*x+1')  Enter  ←式 3x^2 + 5x + 1 を f にセット
```

手順 1. Wild オブジェクトの用意

```
>>> P1 = sp.Wild('P1')  Enter
>>> P2 = sp.Wild('P2',properties=[lambda V:V==sp.Symbol('x')])  Enter  ←条件付き
>>> P3 = sp.Wild('P3')  Enter
```

ここで，Wild オブジェクト P1 の内容を確認してみると

```
>>> P1  Enter  ← Wild オブジェクト P1 の内容確認
P1_  ← Wild オブジェクト P1 の内容
```

このように，アンダースコア '_' が付いた形で表示されるオブジェクトである．

[91] algebraic multiplicity

手順 2. マッチングの実行

```
>>> r = f.match( P1*P2**2+P3 )  Enter  ← match メソッドでマッチングする
>>> r  Enter  ←マッチングの結果である r の内容を確認
{P2_:  x, P1_:  3, P3_:  5*x + 1}  ← P1,P2,P3 への対応が辞書オブジェクト r として得られている.
>>> r[P3]  Enter  ←マッチング結果から P3 に対応する部分を取り出す.
5*x + 1  ←対応する部分が得られている
```

このように，対象となる数式に対して match メソッドを使用することでパターンマッチが実行される．match メソッドの引数には Wild オブジェクトから構成される**パターン**を与える．

上の例では，Wild オブジェクト P2 に条件を付けて,「`Symbol('x')` に限るもの」としている．Wind オブジェクト生成時に条件をつけるには

書き方：　Wild(名前, 条件)

と記述する.「条件」の部分には

exclude=除外するもののリスト
properties=関数リスト

といったものを与える．すなわち,「除外するもののリスト」に該当しない要素を与えたり，より柔軟に，真理値（True/False）を返す形の条件判定用の関数のリストを与えることもできる．

3.3.9　数値計算

3.3.9.1　素因数分解

整数の素因数分解には factorint 関数を使用する．

例. 整数の素因数分解

```
>>> sp.factorint( 1234567890 )  Enter  ←整数の素因数分解
{2:  1, 3:  2, 5:  1, 3607:  1, 3803:  1}  ←素因数とその指数が辞書オブジェクトとして得られる
```

これは 1,234,567,890 を $2 \times 3^2 \times 5 \times 3607 \times 3803$ に分解した例である．

3.3.9.2　素数

sympy ライブラリは，素数を生成する関数やそれを判定する関数を提供する．関数 primerange は指定した範囲にある素数のジェネレータを作成する．

書き方：　primerange(N1, N2)

N1 以上 N2 未満の範囲にある素数を取得するためのジェネレータを返す.

例. 指定した範囲にある素数の取得

```
>>> p = sp.primerange(2,20)  Enter  ← 2 以上 20 未満の素数のジェネレータを取得
>>> list( p )  Enter  ←それをリストに変換
[2, 3, 5, 7, 11, 13, 17, 19]  ←得られた素数のリスト
```

primerange の第 2 引数には「それ未満」の値を与える関係上，第 2 引数に素数を与えるとその値は得られない.

例. 2 以上 19 未満の素数

```
>>> list( sp.primerange(2,19) )  Enter  ←素数のリストを作成
[2, 3, 5, 7, 11, 13, 17]  ←得られた素数のリスト
```

素数は 2，3，5，7，… と続くが，最初の 2 を「1 番目の素数」として「N 番目の」素数を求める関数 prime がある．

書き方：　prime(N)

この関数は「N 番目の」素数を返す.

例. 100,000 番目の素数

```
>>> sp.prime(100000)  Enter  ← 100,000 番目の素数
1299709  ←得られた素数
```

与えられた数が素数かどうかを判定するには関数 isprime を使用する．

例. 素数かどうかを判定

```
>>> sp.isprime( 23 )  Enter        ← 23 は…
True                    ←素数である
>>> sp.isprime( 24 )  Enter        ← 24 は…
False                   ←素数ではない
```

指定した数以下の範囲に存在する素数の個数を調べるには関数 primepi を使用する．

例. 素数の個数を調べる

```
>>> sp.primepi(19)  Enter     ← 19 以下の範囲にある素数の個数を求める
8           ←これだけ素数がある
```

N 以下の範囲にある全ての素数の積を**素数階乗**と言い $N\#$ と書く．SymPy には「N 番目までの素数の全ての積」を求める関数 primorial が存在し，先の primepi と共に用いることで素数階乗を求めることができる．

例. $5\#$ を求める

```
>>> n = sp.primepi(5)  Enter     ← 5 以下の素数の個数を求める
>>> n  Enter     ←確認
3                    ← 3 個
>>> sp.primorial(n)  Enter     ← 3 番目までの素数の積を求める
30                    ←得られた値
```

SymPy は，ここで紹介したもの以外にも素数や合成数に関する関数を提供する．

3.3.9.3 近似値

数式の近似値（数値）を求めるには evalf メソッドを使用する．

例. 円周率を 70 桁の精度で求める

```
>>> sp.pi.evalf(70)  Enter  ←数値近似を求める
3.14159265358979323846264338327950288419716939937510582097494459230781 6
```

例. Matrix オブジェクトの数値近似

```
>>> s = sp.sympify('sqrt(2)'); t = sp.sympify('sqrt(3)')  Enter  ← s = √2, t = √3
>>> u = sp.sympify('sqrt(5)'); v = sp.sympify('sqrt(7)')  Enter  ← u = √5, v = √7
>>> m = sp.Matrix([[s,t],[u,v]])  Enter  ←行列を作成
>>> sp.pprint( m )  Enter  ←整形表示
```

$$\begin{bmatrix} \sqrt{2} & \sqrt{3} \\ \sqrt{5} & \sqrt{7} \end{bmatrix}$$

```
>>> sp.pprint( m.evalf(20) )  Enter  ←数値近似
```

$$\begin{bmatrix} 1.4142135623730950488 & 1.7320508075688772935 \\ 2.2360679774997896964 & 2.6457513110645905905 \end{bmatrix}$$

←数値近似した Matrix

当然ではあるが，数値化できない式には無意味である．（次の例参照）

例. 数値近似が得られないもの

```
>>> s = sp.simplify('a+b')  Enter  ←記号のみからなる数式
>>> s.evalf(70)  Enter  ←数値近似を求めようとしても…
a + b       ←できない．
```

3.3.9.4 数式を数値化する際の工夫

SymPy の数式オブジェクトは記号代数の処理が主な目的であるが，シミュレーションや可視化のために，具体的な数値（近似値）への変換が求められることがある．基本的には，先に解説した evalf メソッドでそれが可能であるが，NumPy による数値演算や matplotlib による可視化を行う際には，更に高速な処理が求められる．ここでは，SymPy の数式を効率よく数値に変換する方法を解説する．

■ lambdify による関数の生成

lambdify 関数を用いると，SymPy の数式を各種の数値演算ライブラリに適した形の関数に変換できる場合がある．

書き方：　lambdify(引数, SymPy の式, modules=対象ライブラリ)

「SymPy の式」には対象の数式を与え，それを,「引数」に対して数値を算出する関数に変換したものを返す.「引数」は Symbol オブジェクトの形で与える.「SymPy の式」を複数の引数を取る関数に変換する場合は,「引数」に必要なだけ Symbol オブジェクトのリスト（などのイテラブル）を与える．また，数値化の際に必要となる数学関数として「対象ライブラリ」（表 40）のものを使用する．

表 40: 指定できるライブラリ名（一部）

'math'	'cmath'	'mpmath'	'numpy'	'scipy'	'sympy'
'python'（デフォルト：組み込み関数+math）					

正弦関数 sin(x) を例に挙げて，SymPy の式を数値演算のための関数に変換する例を示す．

例. SymPy の数式 'sin(x)' を NumPy 用の関数に変換する

```
>>> import sympy as sp   Enter      ← SymPy の読込み
>>> f = sp.parse_expr('sin(x)')   Enter      ← SymPy の数式 'sin(x)' の作成
>>> x = sp.Symbol('x')   Enter      ←引数用の変数記号 'x' の作成
>>> f_np = sp.lambdify( x, f, modules='numpy' )   Enter      ← NumPy 用の関数に変換
```

この例では，f に作成された SymPy の数式 'sin(x)' を NumPy 用の関数 f_np に変換している．NumPy の関数は，配列（ndarray オブジェクト）に対する一括演算が可能である.（次の例）

例. NumPy 用に変換された関数を用いて一括演算する（先の例の続き）

```
>>> import numpy as np   Enter      ← NumPy の読込み
>>> px = np.linspace( -4*np.pi, 4*np.pi, 200 )   Enter      ←定義域データの配列 px を作成
>>> py = f_np(px)   Enter      ← px の全ての要素に対して関数 f_np を一括計算
```

この例では -4π〜4π の範囲を 200 個の値として配列 px に作成している．そしてその全ての要素に対して一括して関数 f_np の値を算出し，結果を配列 py として受け取っている．

以上の処理で，定義域データの配列 px（横軸）と，値域データの配列 py（縦軸）のデータができたので，matplotlib で可視化することができる.（次の例）

例. 得られた配列を可視化する（先の例の続き）

```
>>> import matplotlib.pyplot as plt   Enter      ← matplotlib の読込み
>>> fg = plt.figure( figsize=(10,2) )   Enter      ←描画サイズの設定
>>> pl = plt.plot(px,py)   Enter      ←プロット処理
>>> plt.show()   Enter      ←作図の実行
```

この結果，図 119 のようなグラフが表示される．

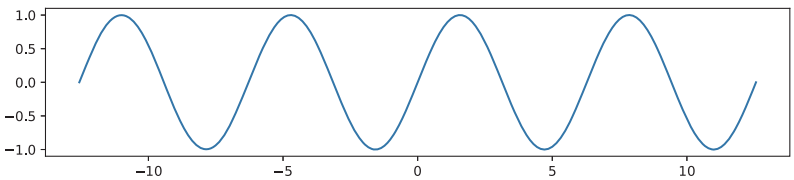


図 119: プロット結果　（横軸:px，縦軸:py）

上に示した事例は単純な数式 'sin(x)' を数値化するものであるが，同様の手法によって，より複雑な SymPy の数式を NumPy 用の関数に変換して可視化する例を sympy3Dplt01.py に示す．

プログラム：sympy3Dplt01.py

```
1 import sympy as sp
2 import numpy as np
3 import matplotlib.pyplot as plt
4
```

```
# 2変数を取る関数（3D描画用）
def w3df(x,y):
    r = sp.sqrt(x**2 + y**2)
    z = sp.exp(-r/5) * sp.cos(2*r)
    return z

sp.var('x y')    # 上記関数のための引数のSymbol
f = w3df(x,y)    # SymPy数式として取得
print('SymPyの式',f)
print('をNumPy用関数に変換して一括計算し，matplotlibで可視化します．')

# SymPyの数式をNumPy用の関数に変換
w3df_np = sp.lambdify( (x, y), f, modules='numpy' )

# NumPyで一括計算
px0 = np.linspace(-8,8,120)      # x軸データの配列
py0 = np.linspace(-8,8,120)      # y軸データの配列
px, py = np.meshgrid(px0, py0)   # 3Dプロット用の平面に変換
pz = w3df_np(px,py)              # z軸データを一括計算

# matplotlibで可視化
fig, ax = plt.subplots(subplot_kw={'projection':'3d'})
ax.plot_surface(
    px, py, pz,          # 座標データ
    cmap='hot',          # 面のカラーマップ
    alpha=0.7,           # 不透明度（0=完全透明，1=不透明）
    edgecolor='black',   # エッジ線の色
    linewidth=0.3        # エッジ線の太さ
)
ax.set_xlabel('px')
ax.set_ylabel('py')
ax.set_zlabel('pz')
plt.show()
```

このプログラムでは

$$\exp\left(-\frac{1}{5}\sqrt{x^2+y^2}\right)\cdot\cos\left(2\sqrt{x^2+y^2}\right)$$

を関数 w3df として定義しており，この式を SymPy の数式オブジェクトとして返す．lambdify 関数でこの式を NumPy 用の関数 w3df_np に変換した後，一括計算と可視化を行っている．

このプログラムを実行すると，標準出力に

```
SymPy の式 exp(-sqrt(x**2 + y**2)/5)*cos(2*sqrt(x**2 + y**2))
を NumPy 用関数に変換して一括計算し，matplotlib で可視化します．
```

と出力した後，図 120 のようなグラフが表示される．

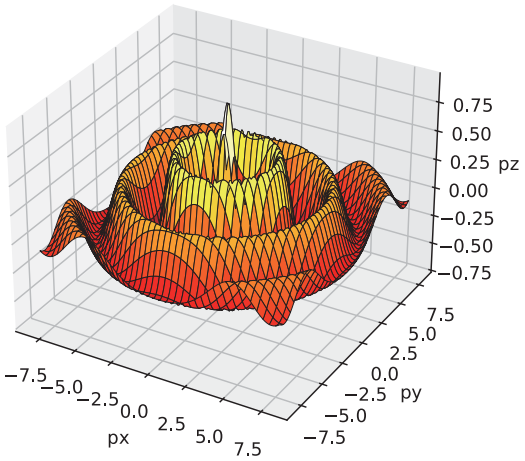


図 120: sympy3Dplt01.py の実行によって表示されるグラフ

3 次元のプロットに関しては「3.1.19 データの可視化：3 次元プロット」（p.133）で解説している．

■ 代入と evalf メソッドを組み合わせる方法

SymPy の数式を lambdify で数値計算用の関数に変換することが困難な場合は，数値の代入と，evalf メソッドによる近似値の算出を組み合わせる方法が有効である．次に示す例は，敢えて lambdfy を使わずに SymPy の数式 'sqrt(x)' を数値に変換して可視化するものである．

例. SymPy の式として $f = \sqrt{x}$ を作る

```
>>> import sympy as sp   Enter      ← SymPy の読込み
>>> f = sp.parse_expr('sqrt(x)')   Enter      ← SymPy の数式 'sqrt(x)' の作成
>>> x = sp.Symbol('x')   Enter      ←引数用の変数記号 'x' の作成
```

この例は SymPy の数式オブジェクトとして $f = \sqrt{x}$ と，変数 x を作るものである．この式を元にして，定義域と値域の数値データの並びを作る例を次に示す．

例. 数値データの作成（先の例の続き）

```
>>> px = [ v for v in range(101) ]   Enter      ←定義域の数値データ（0〜100）
>>> py = [ f.subs(x,v).evalf(6) for v in px ]   Enter      ←値域の数値データを算出
```

この例の `f.subs(x,v).evalf(6)` の部分が実際に個々の数値を算出している．このようにして得られた数値データを可視化する例を次に示す．

例. 得られた数値データのプロット（先の例の続き）

```
>>> import matplotlib.pyplot as plt   Enter      ← matplotlib の読込み
>>> fg = plt.figure( figsize=(10,3) )   Enter      ←描画サイズの設定
>>> pl = plt.plot(px,py)   Enter      ←プロット処理
>>> plt.show()   Enter      ←作図の実行
```

この処理の結果，図 121 のようなグラフが表示される．

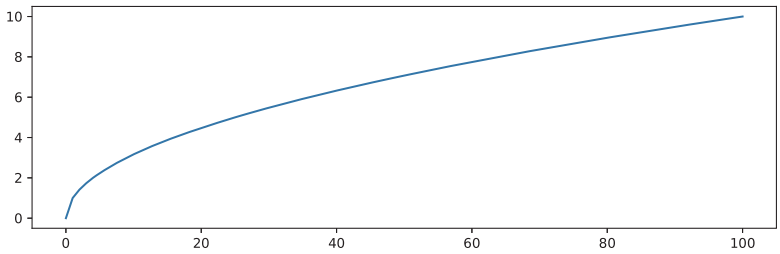


図 121: 表示されるグラフ

上に示した方法は，SymPy の数式から数値データを生成するためのより柔軟なものである．ただし，多数の数値の算出において，処理の速度は NumPy ほど高速ではないことに留意すること．

3.3.10 書式の変換出力

SymPy の式を別の言語（MathML,LaTeX など）の表現に変換する方法が用意されている．

3.3.10.1 LaTeX

SymPy のオブジェクトとして表現した部分積分の公式は，

```
Eq(Integral(u,v),v*u-Integral(v,u))
```

であるが，これを LaTeX の式に変換する例を次に示す．

例. LaTeX 表現の作成

```
>>> s = sp.parse_expr('Eq(Integral(u,v),v*u-Integral(v,u))')   Enter      ←部分積分の公式
>>> s   Enter      ←内容確認
Eq(Integral(u, v), u*v - Integral(v, u))      ←内容表示
>>> print(sp.latex(s))   Enter      ← LaTeX の形式に変換して表示
\int u\, dv = u v - \int v\, du      ←内容表示
```

これを LaTeX で処理すると，

$$\int u\,dv = uv - \int v\,du$$

と表示される．このように latex 関数を使用することで LaTeX の表現を生成できる．

3.3.10.2 MathML

sympy のオブジェクトを MathML の式に変換するには mathml 関数を使用する．

例. MathML 表現の作成（先の例の続き）

```
>>> print(sp.mathml(s))  Enter  ← MathML の形式に変換して表示
<apply><eq/><apply><int/><bvar><ci>v</ci></bvar><ci>u</ci></apply>
<apply><minus/><apply><times/><ci>u</ci><ci>v</ci></apply>
<apply><int/><bvar><ci>u</ci></bvar><ci>v</ci></apply></apply></apply>      ←内容表示
```

3.3.11 グラフのプロット

SymPy は関数のグラフを描く機能（グラフのプロット）を提供する．SymPy はその内部で matplotlib の描画機能を応用してグラフをプロットするが，matplotlib に関するきめ細かい制御を要求する場合は，先の「3.3.9.4 数式を数値化する際の工夫」（p.208）で解説した方法を取る方が良い．ここで紹介する機能は，あくまで簡易的なものと考える方が良い．

1 変数の関数のプロット（2 次元のグラフ）を作成するには plot 関数を使用する．

書き方： plot(**式, (変数, 最小値, 最大値)**)

例. 正弦関数のプロット

```
>>> import sympy as sp  Enter  ← SymPy の読込み
>>> f = sp.parse_expr('sin(x)')  Enter  ← f に正弦関数の式を与える
>>> x = sp.Symbol('x')  Enter  ←プロット用の変数
>>> g = sp.plot(f,(x,-4,4))  Enter  ←プロット実行
```

この処理の結果，図 122 のようなグラフが表示される．

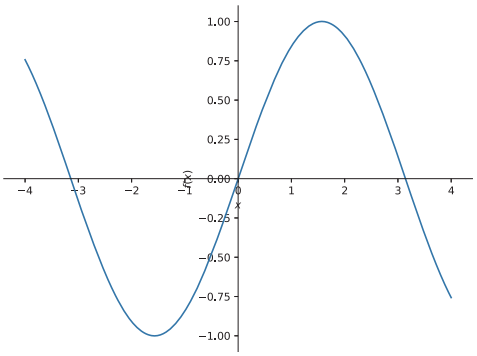


図 122: プロットの表示

2 変数の関数のプロット（3 次元のグラフ）を作成するには plot3d 関数を使用する．この関数は，モジュール sympy.plotting にあるため，使用に際してはこのモジュールをインポートしておく必要がある．

書き方： plot3d(**式, (変数 1, 最小値, 最大値), (変数 2, 最小値, 最大値)**)

例. $\cos\left(\sqrt{x^2+y^2}\right)$ のプロット

```
>>> import sympy as sp  Enter  ← SymPy の読込み
>>> from sympy.plotting import plot3d  Enter  ← 3D プロット機能のインポート
>>> f = sp.sympify('cos((x**2+y**2)**(1/2))')  Enter  ← f に関数の式を与える
>>> x = sp.Symbol('x')  Enter  ←プロット用の変数 x
>>> y = sp.Symbol('y')  Enter  ←プロット用の変数 y
>>> g = plot3d(f,(x,-4.5,4.5),(y,-4.5,4.5))  Enter  ←プロット実行
```

この処理の結果図 123 のようなグラフが表示される．

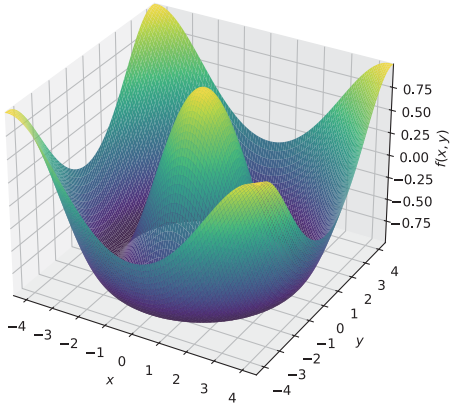


図 123: 3 次元のプロット

3.3.11.1　グラフを画像ファイルに保存する方法

plot 関数や plot3d 関数で作成したグラフを，画像ファイルとして保存するには，それら関数の戻り値に対して save メソッドを使用する．(次の例参照)

例. グラフの保存（先の例の続き）

```
>>> g.save('sympyFig01.eps')  Enter    ← eps 形式で保存
>>> g.save('sympyFig01.png')  Enter    ← png 形式で保存
```

save メソッドの引数に，保存先のファイル名を与える．画像ファイルのフォーマットは，ファイル名の拡張子によって自動的に判断される．(扱えないフォーマットもあるので注意)

3.4 多倍長精度の数値演算用ライブラリ：gmpy2

gmpy2 は Python 上で多倍長精度の数値演算を実行するためのライブラリ[92] である．このライブラリは C/C++ 用に構築された GMP，MPFR ライブラリ[93] を Python から利用利用できるようにしたものである．

GMP，MPFR ライブラリは多倍長精度の数値演算を実行するための高性能なライブラリとして普及しており，これを応用した gmpy2 を利用すると，同じ目的のライブラリである mpmath を大きく上回る性能が得られる．

gmpy2 は標準ライブラリではなく，その利用に先立って Python 処理系に別途導入する必要がある．

pip コマンド[94] によるインストールの例： `pip install gmpy2`
conda コマンド[95] によるインストールの例： `conda install -c conda-forge gmpy2`

gmpy2 を読み込むには

```
import gmpy2
```

とする．「from gmpy2 import *」という形式で読み込むのは，API の名前の衝突の問題もあり，推奨されない．以下の解説では，上のようにして gmpy2 を読み込んだことを前提とする．

3.4.1 基本的なデータ型

gmpy2 は，整数，有理数，浮動小数点数，複素数のためのデータ型としてそれぞれ，mpz，mpq，**mpfr**，mpc を提供しており，それらは Python の数値のクラス階層に組み込まれ，数値に関する各種の組み込みの演算子（+，-，*，/，**，//，%）を用いて演算することができる．

■ 他倍長整数型：mpz

書き方：　mpz(初期値)

与えた「初期値」を持つ mpz インスタンスを返す．「初期値」には int，float，str など，様々な型の表現が使える．

例. mpz コンストラクタ

```
>>> gmpy2.mpz(2)   Enter      ← int の初期値
mpz(2)                        ←得られたオブジェクト
>>> gmpy2.mpz(3.14)  Enter    ← float の初期値
mpz(3)                        ←小数点以下切り捨て
>>> gmpy2.mpz('1234567890987654321')  Enter   ← str の初期値
mpz(1234567890987654321)              ←得られたオブジェクト
```

このように様々な型で初期値を与えることができる．
また mpz オブジェクトは print 関数で整形出力することができる．

例. print 関数による整形出力

```
>>> print( gmpy2.mpz('1234567890987654321') )   Enter
1234567890987654321            ←整形された結果
```

fractions.Fraction のオブジェクトを初期値に与えることもできる．

例. Fraction オブジェクトの初期値

```
>>> from fractions import Fraction   Enter    ← Fraction クラスの読込み
>>> gmpy2.mpz( Fraction(7,2) )   Enter    ←初期値に Fraction オブジェクトを与える
mpz(3)                                    ←「分子/分母」の整数部分
```

ただし，文字列表現の分数は初期値に与えることができない．(次の例)

[92] https://pypi.org/project/gmpy2/，https://github.com/aleaxit/gmpy
[93] https://gmplib.org/，https://www.mpfr.org/
[94] PSF 版 Python における標準的なライブラリ管理コマンド．
[95] Anaconda ディストリビューションの Python におけるライブラリ管理コマンド．

例. 文字列表現の分数を初期値に与える試み

```
>>> gmpy2.mpz( '7/2' )  Enter     ←これは
Traceback (most recent call last):      ←エラーとなる
  File "<stdin>", line 1, in <module>
    gmpy2.mpz( '7/2' )
    ^^^^^^^^^^^^^^^^^^
ValueError:  invalid digits
```

mpz オブジェクト同士，あるいは Python 元来の数値型との算術演算の結果は mpz オブジェクトとなる．

例. mpz オブジェクトの算術演算

```
>>> x = gmpy2.mpz(3)  Enter     ← mpz オブジェクト
>>> x + x  Enter     ← mpz 同士の演算
mpz(6)               ← mpz オブジェクト
>>> x + 2  Enter     ← mpz と int の演算
mpz(5)               ← mpz オブジェクト
>>> 2 + x  Enter     ← int と mpz の演算
mpz(5)               ← mpz オブジェクト
```

■ 有理数型：mpq

書き方：　mpq(初期値)

与えた「初期値」を持つ mpq インスタンスを返す.「初期値」には int，float，str, fractions.Fraction など，様々な型の表現が使える．また，明示的に分子，分母を与えることもできる．

書き方：　mpq(分子, 分母)

この場合,「分子」,「分母」には int もしくは mpz の型の値を与える．mpq オブジェクトの値もこの形式である．

例. mpz コンストラクタ

```
>>> gmpy2.mpq(2)  Enter     ← 2 を分数にすると
mpq(2,1)                  ← 2/1
>>> gmpy2.mpq(0.5)  Enter     ← 0.5 を分数にすると
mpq(1,2)                  ← 1/2
>>> gmpy2.mpq('2222/3333')  Enter     ←これは
mpq(2,3)                        ←約分される
>>> gmpy2.mpq(3,4)  Enter     ←分子，分母を明示的に与える
mpq(3,4)
```

※分子，分母を明示的に与える場合，整数以外のものを与えるとエラーとなるので注意すること．

mpq の初期値に float を与えると，分子，分母とも意図せず大きな値になることがある．(次の例)

例. 分子，分母が大きくなってしまう例

```
>>> gmpy2.mpq(3.14)  Enter     ←この初期値の場合は
mpq(7070651414971679,2251799813685248)     ←分子，分母が大きくなってしまう
```

分母の大きさを制限するには fractions.Fraction の機能を応用すると良い．

例. 分母の大きさを制限する

```
>>> from fractions import Fraction  Enter     ← Fraction クラスの読込み
>>> q = Fraction(3.14).limit_denominator(100)  Enter   ←分母を100 以下に制限して Fraction に変換
>>> q  Enter     ←内容確認
Fraction(157, 50)          ←制限されている
>>> gmpy2.mpq(q)  Enter     ←その値を mpq に変換
mpq(157,50)
```

mpq は常に約分された形で保持され，算術の結果も常に約分される．また，print 関数で出力する際は整形される．

例. 常に約分される mpq

```
>>> x = gmpy2.mpq(2,4) Enter      ← 2/4
>>> y = gmpy2.mpq(3,9) Enter      ← 3/9
>>> print( 'x =',x,', y =',y ) Enter      ←値を整形された形で確認
x = 1/2 , y = 1/3      ← 1/2 と 1/3（約分されている）
>>> print( x + y ) Enter      ←計算結果は
5/6                ←約分される
```

■ 他倍長浮動小数点数型：mpfr

書き方：　mpfr(初期値)

与えた「初期値」を持つ mpfr インスタンスを返す.「初期値」には int，float，str，mpz，mpq など様々なものを与えることができる.

例. mpfr オブジェクトによる $\sqrt{2}$ の近似値の算出

```
>>> x = gmpy2.mpfr(2)  Enter      ← 2
>>> x ** 0.5  Enter      ← √2
mpfr('1.4142135623730951')        ←近似値
```

mpf の値は print 関数で出力すると整形される.

例. print による整形出力（先の例の続き）

```
>>> print( x ** 0.5 )  Enter      ← √2 を整形出力すると
1.4142135623730951                ←一般的な浮動小数点数の形式
```

【mpfr の値の精度】

mpfr が保持できる値の精度は gmpy2 の**演算環境**に設定されている．演算環境にアクセスするには `get_context` 関数を使用する.

例. mpfr の値の精度を調べる（先の例の続き）

```
>>> gmpy2.get_context().precision  Enter      ←値の精度を参照する
53                                  ←仮数部の長さは 53 ビット
```

このように，`get_context` 関数が返す演算環境の precision 属性に現在の精度（ビット長）が保持されている.（デフォルトは IEEE-754 倍精度=53 ビット）
この属性の値を変更することで，mpfr の値の精度を変更することができる.

例. mpfr の値の精度の設定（先の例の続き）

```
>>> gmpy2.get_context().precision = 300  Enter  ←仮数部を 300 ビットに設定
>>> x ** 0.5  Enter      ← √2
mpfr('1.41421356237309504880168872420969807856967187 5      ←仮数部の精度が 300 ビットの場合の表現
3769480731766797379907324784621070388503875348',300)      ←オブジェクトは精度の値も保持
```

値の精度をビット長ではなく 10 進数表現の桁数で指定する場合は，次のようにして変換すると良い.

$$\text{精度のビット長} = \lceil n \times \log_2 10 \rceil \qquad (\text{n は 10 進桁数})$$

10 進数で 100 桁分の精度で先の計算を行う例を次に示す.

例. 10 進数表現で精度の桁数を指定する（先の例の続き）

```
>>> precDigit = 100  Enter      ← 10 進の桁数で「100 桁」
>>> precBits = int( gmpy2.ceil( precDigit * gmpy2.log2(10) ) ) Enter      ←それをビット長に変換
>>> precBits  Enter      ←確認
333                      ←ビット長
>>> gmpy2.get_context().precision = precBits  Enter      ←これを演算環境に設定
>>> x ** 0.5  Enter      ← √2
mpfr('1.41421356237309504880168872420969807856967187537694
807317667973799073247846210703885038753432764157271',333)
```

【mpfr の値の丸め】

演算の結果として得られる mpfr の値は適切に丸められるが，丸めの手法（表 41）を**演算環境**の round 属性に設定することができる．

表 41: mpfr の丸めのモード

定　数　名	値	振る舞い	小数点以下を丸める例
gmpy2.RoundToNearest	0	偶数丸め　（デフォルト）	1.5 → 2，2.5 → 2
gmpy2.RoundToZero	1	0 に向かって丸める	1.9 → 1，-1.9 → -1
gmpy2.RoundUp	2	正の方向に丸める	1.1 → 2，-1.9 → -1
gmpy2.RoundDown	3	負の方向に丸める	1.9 → 1，-1.1 → -2

例. 丸めのモードの確認

```
>>> gmpy2.get_context().round   Enter      ←確認
0              ← gmpy2.RoundToNearest
```

mpfr の値を明示的に丸める例を以下に示す．次のように x，y にそれぞれ 1.4，-1.4 が設定されているとする．

例. 正負の小数点数

```
>>> x = gmpy2.mpfr(1.4)   Enter
>>> y = gmpy2.mpfr(-1.4)   Enter
```

これら x，y の値を丸める処理を次に示す．

例. 切り上げ：ceil 関数（先の例の続き）

```
>>> gmpy2.ceil(x)   Enter
mpfr('2.0')
>>> gmpy2.ceil(y)   Enter
mpfr('-1.0')
```

例. 切り下げ：floor 関数（先の例の続き）

```
>>> gmpy2.floor(x)   Enter
mpfr('1.0')
>>> gmpy2.floor(y)   Enter
mpfr('-2.0')
```

例. 小数点以下切落とし：trunc 関数（先の例の続き）

```
>>> gmpy2.trunc(x)   Enter
mpfr('1.0')
>>> gmpy2.trunc(y)   Enter
mpfr('-1.0')
```

※ gmpy2 のビルドによっては偶数丸めのための gmpy2.round 関数や，最も近い整数値を求める gmpy2.nearest 関数が使える場合もある．

■ 複素数型：mpc

書き方：　mpc(初期値)

与えた「初期値」を持つ mpc インスタンスを返す.「初期値」には int，float，complex，str，mpz，mpq，mpfr，mpc など様々なものを与えることができる．このクラスのオブジェクトは，実部，虚部とも mpfr 型のオブジェクトを持つ．コンストラクタに実部，虚部の 2 つのオブジェクトを引数として与えることもできる．

書き方：　mpc(実部, 虚部)

この場合，引数には複素数を与えることはできない．

例. 複素数

```
>>> gmpy2.mpc(2+3j)   Enter
mpc('2.0+3.0j')        ← 2 + 3i
>>> gmpy2.mpc(2,3)   Enter
mpc('2.0+3.0j')        ←同上
```

mpc の値は print 関数で出力すると整形される．

例. mpc の値の整形出力

```
>>> print( gmpy2.mpc(2,3) )   Enter
2.0+3.0j              ← complex 型と同様の形式
```

3.4.2　演算結果の型

gmpy2 の数値（mpz，mpq，mpfr，mpc）は相互に算術演算できるだけでなく，Python の基本的な数値（int，fractions.Fraction，float，complex）とも相互に算術演算ができる．異なる型の数値同士の算術演算を実行すると，最も表現力の高い型で結果が得られる．

数値の表現力は，整数→有理数→浮動小数点数→複素数の順で高まる．この構造は**演算タワー**[96] と呼ばれており，異なる型の数値の算術演算の結果は，演算タワー上の高い位置の型となる．また，gmpy2 の数値型は，Python の基本的な数値型よりも高い位置にある．別の言い方で解説すると，「演算結果として，元の値を構成する情報が失われないような演算結果の型となる」とも表現できる．

例. mpz と mpq の加算

```
>>> vz = gmpy2.mpz(2)   Enter      ←整数
>>> vq = gmpy2.mpq(1,3)   Enter      ←有理数
>>> vz + vq   Enter      ←整数+有理数
mpq(7,3)              ←結果は有理数（7/3）
```

例. mpq と mpfr の加算（先の例の続き）

```
>>> vf = gmpy2.mpfr(3.14)   Enter      ←小数点数
>>> vq + vf   Enter      ←有理数+小数点数
mpfr('3.4733333333333336')      ←結果は小数点数
```

例. mpfr と float の加算（先の例の続き）

```
>>> vf + 0.0015926535   Enter
mpfr('3.1415926535000001')      ←結果は mpfr（誤差が生じている）
```

complex と mpz の加算においては complex の方が虚部を保持しているという意味で複雑であり，結果は mpc となる．（次の例）

例. complex と mpz の加算（先の例の続き）

```
>>> (1+5j) + vz   Enter
mpc('3.0+5.0j')        ←結果は mpc
```

3.4.3　数学関数

gmpy2 には多くの数学関数が提供されており，

gmpy2. **関数名 (引数並び)**

の形式で呼び出すことができる．標準ライブラリ math と同じ関数名のもの多いが，完全に同じではないので，詳しくは gmpy2 の公式インターネットサイトを参照のこと．

例. 円周率（math ライブラリとは異なる呼び出し方）

```
>>> gmpy2.const_pi()   Enter
mpfr('3.1415926535897931')
>>> gmpy2.const_pi( 200 )   Enter      ←引数に精度（ビット長）を与える
mpfr('3.1415926535897932384626433832795028841971693993751058209749445',200)
```

例. 余弦関数

```
>>> gmpy2.cos( gmpy2.const_pi() )   Enter      ← cos(π) を求める
mpfr('-1.0')
```

gmpy2 の数学関数は基本的に複素数に対応しており，引数に複素数を与えることで，複素数として値を返す．

例. $\sqrt{2}$ と $\sqrt{-2}$（その 1）

```
>>> gmpy2.sqrt( gmpy2.mpfr(2) )   Enter      ← √2 を求める
mpfr('1.4142135623730951')
>>> gmpy2.sqrt( gmpy2.mpfr(-2) )   Enter      ← √-2 を求める
mpfr('nan')        ←引数に実数を与えると算出できない
```

[96] PEP 3141 でも言及されている．

この例では，$\sqrt{-2}$ は算出されていない．しかし，引数を複素数として与えると算出できる．(次の例)

例. $\sqrt{2}$ と $\sqrt{-2}$ (その 2)

```
>>> gmpy2.sqrt( gmpy2.mpc(2) )   Enter     ← √2 を求める
mpc('1.4142135623730951+0.0j')            ←結果が複素数で得られる
>>> gmpy2.sqrt( gmpy2.mpc(-2) )   Enter    ← √-2 を求める
mpc('0.0+1.4142135623730951j')            ←結果が複素数で得られる
```

gmpy2 の数学関数の引数には，int，float，Fraction，complex といった型の値を与えても良い．

例. 数学関数の引数に int，complex の値を与える

```
>>> print( gmpy2.sqrt( 2 ) )   Enter     ←引数に int 型の 2 を与える
1.4142135623730951        ←計算できている（整形出力）
>>> print( gmpy2.sqrt( -2+0j ) )   Enter     ←引数に complex 型の値を与える
0.0+1.4142135623730951j        ←計算できている（整形出力）
```

【数学関数の複素数領域での演算】

数学関数の引数に複素数（mpc，complex）の値を与えると複素数領域での演算を実行するが，演算環境の `allow_complex` 属性に True を設定する（デフォルトは False）と，数学関数は複素数領域で演算する．

例. 複素数領域の演算の設定

```
>>> gmpy2.get_context().allow_complex = True   Enter     ←複素数領域の演算の設定
>>> print( gmpy2.sqrt( -2 ) )   Enter     ←引数に int 型の -2（実数領域の値）を与える
0.0+1.4142135623730951j          ←演算結果が得られている
```

3.4.4 数論関連の演算

3.4.4.1 mod 演算

Python の整数除算 `x//y` は $\lfloor x/y \rfloor$ の演算に基づいており，剰余もこれに基づいている[97]．このことを Python の組み込み関数 divmod で確認する．

例. Python の整数除算と剰余

```
>>> divmod(10,3)   Enter     ← 10/3 の商と剰余 (1)
(3, 1)
>>> divmod(-10,3)   Enter     ← -10/3 の商と剰余 (2)
(-4, 2)
>>> divmod(10,-3)   Enter     ← 10/(-3) の商と剰余 (3)
(-4, -2)
>>> divmod(-10,-3)   Enter     ← (-10)/(-3) の商と剰余 (4)
(3, -1)
```

C 言語や JavaScript では，剰余を求める際の整数除算の考えが Python と異なり，上の例の (2)，(3) における剰余の値が異なる．

例. C 言語，JavaScript での剰余

```
-10 % 3     →  -1    (上の例の (2) と異なる結果)
10 % -3     →   1    (上の例の (3) と異なる結果)
```

これは，C 言語や JavaScript が，剰余を求める際の除算において，商の小数点以下を単純に切り取る（truncate）ことに起因する．

gmpy2 は，整数除算の各種の考え方に対応する剰余を算出する関数を提供している．

■ 商と剰余 (1)： $\lfloor x/y \rfloor$ による整数除算に基づく商と剰余

書き方： `f_divmod( x, y )`

x ÷ y の商と剰余のタプル（要素は mpz）を返す．これは Python の組み込み関数 divmod と同じ考え方でである．ただし，divmod と異なり，x，y に整数（int，mpz）以外の値を与えるとエラー（TypeError）となる．

[97] Python における剰余は，数学の **mod 演算**（**モジュロ演算**）の考えに基づいている．

例. f_divmod 関数

```
>>> gmpy2.f_divmod(10,3)   Enter     ← 10/3 の商と剰余
(mpz(3), mpz(1))
>>> gmpy2.f_divmod(-10,3)   Enter     ← -10/3 の商と剰余
(mpz(-4), mpz(2))
>>> gmpy2.f_divmod(10,-3)   Enter     ← 10/(-3) の商と剰余
(mpz(-4), mpz(-2))
>>> gmpy2.f_divmod(-10,-3)   Enter     ← (-10)/(-3) の商と剰余
(mpz(3), mpz(-1))
```

次のように，引数に整数でないものを与えるとエラーとなる

例. f_divmod 関数に整数でない値を与える試み

```
>>> gmpy2.f_divmod( 10.0, 3.0 )   Enter     ←浮動小数点数を与えると
Traceback (most recent call last):      ←エラーとなる
  File "<stdin>", line 1, in <module>
    gmpy2.f_divmod( 10.0, 3.0 )
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~
TypeError:  cannot convert object to mpz
```

このようなエラーは後の関数においても注意すること．

■ 商と剰余 (2)： 小数点以下切り捨ての整数除算に基づく商と剰余

書き方： t_divmod(**x, y**)

x ÷ y の商と剰余のタプル（要素は mpz）を返す．これは C 言語や JavaScript と同じ考え方の整数除算による商と剰余である．ただし，x，y に整数（int，mpz）以外の値を与えるとエラー（TypeError）となる．

例. t_divmod 関数

```
>>> gmpy2.t_divmod(10,3)   Enter     ← 10/3 の商と剰余
(mpz(3), mpz(1))
>>> gmpy2.t_divmod(-10,3)   Enter     ← -10/3 の商と剰余
(mpz(-3), mpz(-1))
>>> gmpy2.t_divmod(10,-3)   Enter     ← 10/(-3) の商と剰余
(mpz(-3), mpz(1))
>>> gmpy2.t_divmod(-10,-3)   Enter     ← (-10)/(-3) の商と剰余
(mpz(3), mpz(-1))
```

■ 商と剰余 (3)： $\lceil x/y \rceil$ による整数除算に基づく商と剰余

書き方： c_divmod(**x, y**)

x ÷ y の商と剰余のタプル（要素は mpz）を返す．これは Python の組み込み関数 divmod と同じ考え方でである．x，y に整数（int，mpz）以外の値を与えるとエラー（TypeError）となる．

例. c_divmod 関数

```
>>> gmpy2.c_divmod(10,3)   Enter     ← 10/3 の商と剰余
(mpz(4), mpz(-2))
>>> gmpy2.c_divmod(-10,3)   Enter     ← -10/3 の商と剰余
(mpz(-3), mpz(-1))
>>> gmpy2.c_divmod(10,-3)   Enter     ← 10/(-3) の商と剰余
(mpz(-3), mpz(1))
>>> gmpy2.c_divmod(-10,-3)   Enter     ← (-10)/(-3) の商と剰余
(mpz(4), mpz(2))
```

f_divmod，t_divmod，c_divmod における剰余のみを返す f_mod ，t_mod，c_mod 関数もある．

例. -10/3 の剰余のみを求める

```
>>> gmpy2.f_mod(-10,3)   Enter
mpz(2)
>>> gmpy2.t_mod(-10,3)   Enter
mpz(-1)
>>> gmpy2.c_mod(-10,3)   Enter
mpz(-1)
```

注意）後の解説において，特に断りがない場合は `f_mod` の剰余を前提とする．

■ 2 のべき乗による剰余： $x \,\%\, 2^n$

書き方： `f_mod_2exp( x, n )`

法 2^n における x の値を mpz 型で返す．x，n は整数（int，mpz）である．

例. $25 \,\%\, 2^4$

```
>>> gmpy2.f_mod_2exp( 25, 4 )   Enter     ← 25 % 2^4
mpz(9)                  ← 9
```

■ べき剰余： $b^e \bmod m$

書き方： **powmod(b, e, m)**

法 m における b^e を mpz 型で返す．b，e，m は整数（int，mpz）である．

例. $2^{12} \bmod 10$

```
>>> gmpy2.powmod( 2, 12, 10 )   Enter
mpz(6)
```

■ モジュラ逆元：

書き方： **invert(x, m)**

法 m における x の逆元を mpz 型で返す．x，m は整数（int，mpz）である．

例. 法 7 における 3 の逆元

```
>>> inv = gmpy2.invert( 3, 7 )   Enter
>>> inv   Enter      ←逆元の確認
mpz(5)               ←法 7 における 3 の逆元は 5
>>> 3 * inv % 7   Enter      ←逆元のと元の値 3 との積を法 7 で求めると
mpz(1)               ←積の単位元 1
```

注意）モジュラ逆元が存在しない場合（x，m が互いに素でないケース）はエラー（ZeroDivisionError）となるので注意すること．

例. モジュラ逆元が存在しないケース

```
>>> gmpy2.invert( 6, 15 )   Enter     ←法 15 における 6 の逆元は
Traceback (most recent call last):        ←存在しないのでエラー
  File "<stdin>", line 1, in <module>
    gmpy2.invert( 6, 15 )
    ^^^^^^^^^^^^^^^^^^^^^
ZeroDivisionError:  invert() no inverse exists
```

3.4.4.2　素数の生成

`next_prime` 関数を用いると素数を生成することができる．

書き方： **next_prime(n)**

n より大きい最小の素数を mpz 型で返す．n は整数（int，mpz）である．

例. 2^{100} より大きい最小の素数

```
>>> x = gmpy2.mpz( 2**100 )  Enter      ← 2^100
>>> x  Enter      ←値の確認
mpz(1267650600228229401496703205376)
>>> p = gmpy2.next_prime( x )  Enter      ← 2^100 より大きい最小の素数を求める
>>> p  Enter      ←値の確認
mpz(1267650600228229401496703205653)      ←素数
```

3.4.4.3 約数の検査

書き方： is_divisible(n, d)

d が n の約数ならば（n が d で割り切れれば）True，そうでなければ False を返す．n，d は整数（int，mpz）である．

先の例で作成した x，p を用いた例を次に示す．

例. 約数の検査（先の例の続き）

```
>>> gmpy2.is_divisible( p, 2**50 )  Enter      ← 2^50 は p の約数か？
False                                    ←約数でない
>>> gmpy2.is_divisible( x, 2**50 )  Enter      ← 2^50 は 2^100 の約数か？
True                                     ←約数である
```

3.4.4.4 最大公約数，最小公倍数

書き方： gcd(x, y)
書き方： lcm(x, y)

gcd は x，y の最大公約数を，lcm は最小公倍数を返す．x，y は整数（int，mpz）, 戻り値は mpz である．

大きな素数 p1，p2，p3 の合成数 x = p1*p2，y = p2*p3，z = p1*p2*p2 を作り，x と y の gcd が p2 となることを判定し，x と y の lcm が z となる様子を次の例で示す．

例. 大きな素数 p1，p2，p3 の生成

```
>>> p1 = gmpy2.next_prime( 2**60 )  Enter      ←素数 p1 の生成
>>> p2 = gmpy2.next_prime( 2**61 )  Enter      ←素数 p2 の生成
>>> p3 = gmpy2.next_prime( 2**62 )  Enter      ←素数 p3 の生成
>>> print(p1,p2,p3,sep='\n')  Enter      ←出力して確認
1152921504606847009        ← p1
2305843009213693967        ← p2
4611686018427388039        ← p3
```

例. 合成数の作成（先の例の続き）

```
>>> x = p1 * p2  Enter
>>> y = p2 * p3  Enter
>>> z = p1 * p2 * p3  Enter
>>> print(x,y,z,sep='\n')  Enter      ←出力して確認
2658455991569831839194255993715294703                                  ← x
1063382396627932736369455300250226071 3                                 ← y
1225996432692711165642820571344251686379253878142225741 7   ← z
```

例. gcd，lcm の算出と検証（先の例の続き）

```
>>> g = gmpy2.gcd(x,y)  Enter      ←最大公約数（gcd）の算出
>>> g == p2  Enter      ←検証
True                 ←検証結果
>>> l = gmpy2.lcm(x,y)  Enter      ←最小公倍数（lcm）の算出
>>> l == z  Enter      ←検証
True                 ←検証結果
```

■ **拡張ユークリッドの互除法**

gmpy2 は，**拡張ユークリッドの互除法**（Extended Euclidean algorithm）で最大公約数を求める gcdext 関数を提供する．

書き方：　gcdext(x, y)

x，y の最大公約数 g とベズー係数（Bézout coefficients） s，t（次の式）のタプル (g,s,t) を返す．

$$g = s \cdot x + t \cdot y$$

引数 x，y は整数（int，mpz），戻り値のタプルの要素は mpz である．

例. gcd とベズー係数を求める（先の例の続き）

```
>>> g,s,t = gmpy2.gcdext(x,y)  Enter      ←最大公約数（gcd）とベズー係数の算出
>>> g == p2  Enter      ←検証
True                    ←検証結果
>>> s  Enter      ← 1 つ目のベズー係数の確認
mpz(1537228672809129345)
>>> t  Enter      ← 2 つ目のベズー係数の確認
mpz(-384307168202282336)
```

例. ベズー係数の検証（先の例の続き）

```
>>> s*x + t*y == g  Enter      ←検証
True                    ←検証結果
```

3.4.5　乱数生成

gmpy2 は，**線形合同法**（LCG：Linear Congruential Generator）で多倍長精度整数の乱数を生成する mpz_urandomb 関数を提供する．この関数は，**乱数状態オブジェクト**を元にして乱数を生成する．プログラミングの流れとして，まず乱数状態オブジェクトを作成し，その後，乱数生成用関数を必要なだけ呼び出すという手順となる．また，生成される乱数は LCG による確定的なもの[98] である．

■ **乱数状態オブジェクトの作成：　random_state**

書き方：　random_state(種)

整数（int，mpz）の「種」を与えると乱数状態オブジェクトを返す．

■ **指定したビット長の乱数の生成：　mpz_urandomb**

書き方：　mpz_urandomb(rs, ビット長)

乱数状態オブジェクト rs を元に，指定した整数（int，mpz）の「ビット長」の乱数を mpz として返す．

例. mpz_urandomb による乱数生成

```
>>> rs = gmpy2.random_state(1)  Enter      ←種「1」を与えて乱数状態オブジェクトを作成
>>> for _ in range(10):  Enter      ←乱数を 10 個生成するループ
...       print( gmpy2.mpz_urandomb(rs,16), end=',' )  Enter      ← 16 ビット長の整数乱数の生成
...   else:  print()  Enter      ←終了時に改行
...  Enter      ←ループの記述の終了
47419,61646,55250,29287,58479,34395,28898,9622,49803,48689,      ←生成された 10 個の乱数
```

この例の random_state(1) の種を変えると，生成される乱数の系列が変わる．

3.4.6　性能の評価

3.4.6.1　整数の算術演算の比較

以下に示すプログラム numSpdTest01.py は，再帰的アルゴリズムで整数の階乗を計算するものであり，計算に使用する数値の型ごとの実行時間を計測するものである．具体的には Python 元来の int 型と gmpy2 の mpz の実行時間比較する．また，本来は階乗の計算を浮動小数点数で実行するべきではないが，参考までに，mpmath の mpf と gmpy2 の mpfr での実行時間も計測する．

[98] **疑似乱数**という．

プログラム：numSpdTest01.py

```
import gmpy2, timeit, sys, math
from mpmath import mp

#--- 階乗計算の関数 ---
sys.setrecursionlimit(11000)     # 関数の再帰呼び出しの上限を再設定

def fct(n):
    if n <= 0: return 1
    return n * fct(n-1)

n = 10000
x = fct(n)                                          # 事前実行
dgt = int(math.floor(math.log10(abs(x)))+1)         # 桁数調査
print( f'{n}! は {dgt} 桁 ({int(dgt*3.33)} bit)       速度比' )
print('-'*44)

#--- intの場合の実行時間 ---
t1 = timeit.timeit('fct(n)',globals=globals(),number=10) / 10
print( f'   int による計算時間(sec): {t1:7.4f} {1.0:5.2f} 倍' )

#--- mpzの場合の実行時間 ---
t2 = timeit.timeit('fct(gmpy2.mpz(n))',globals=globals(),number=10) / 10
print( f'   mpz による計算時間(sec): {t2:7.4f} {t1/t2:5.2f} 倍')

#--- mpmathの場合の実行時間 ---
mp.dps = dgt                                        # 演算精度設定
t3 = timeit.timeit('fct(mp.mpf(n))',globals=globals(),number=10) / 10
print( f'mpmath による計算時間(sec): {t3:7.4f} {t1/t3:5.2f} 倍')

#--- mpfrの場合の実行時間 ---
gmpy2.get_context().precision = int(dgt*3.33)   # 演算精度設定
t4 = timeit.timeit('fct(gmpy2.mpfr(n))',globals=globals(),number=3) / 3
print( f'  mpfr による計算時間(sec): {t4:7.4f} {t1/t4:5.2f} 倍')
```

このプログラムを実行した例を次に示す.

実行例. Windows 環境のコマンドプロンプトウィンドウでの実行結果

```
10000!  は 35660 桁 (118747 bit)     速度比
--------------------------------------------
   int による計算時間(sec):  0.0162   1.00 倍
   mpz による計算時間(sec):  0.0082   1.97 倍
mpmath による計算時間(sec):  0.0382   0.42 倍
  mpfr による計算時間(sec):  0.0882   0.18 倍
```

この実行結果から，mpz による計算は int による計算よりも 2 倍近い速度であることがわかる．また当然のことであるが，多倍長精度の浮動小数点数の計算時間は大きく，特にこの例において，敢えて mpfr を採用すると最も速度が遅いことがわかる.

3.4.6.2 有理数の算術演算の比較

以下に示すプログラム numSpdTest02.py は，

$$\sum_{i=1}^{n} \frac{i}{i+1}$$

を高い精度の有理数で計算するものである．具体的には，Python の標準ライブラリとして提供されている fractions.Fraction と，gmpy2.mpq の実行時間を比較する.

プログラム：numSpdTest02.py

```
import gmpy2, timeit
from fractions import Fraction

#--- i/(i+1)の総和 ---
def sumQ(f,n):       # fの型でn回加算
    s = 0
    for i in range(1,n+1):
        s += f(i,i+1)
```

```
    return s

n = 10000
print('sum_{i=1}^{n}{i/(i+1)} の計算: n =',n,', 速度比')
print('-'*51)
#--- Fractionで計算 ---
t1 = timeit.timeit('sumQ(Fraction,n)',globals=globals(),number=10) / 10
print( f' Fractionによる計算時間: {t1:8.5f} (sec), {1.0:5.2f} 倍' )

#--- mpqで計算 ---
t2 = timeit.timeit('sumQ(gmpy2.mpq,n)',globals=globals(),number=10) / 10
print( f'gmpy2.mpqによる計算時間: {t2:8.5f} (sec), {t1/t2:5.2f} 倍' )

print('-'*51)
#--- 計算結果 ---
s1 = str(gmpy2.mpfr(sumQ(Fraction,n)))
s2 = str(gmpy2.mpfr(sumQ(gmpy2.mpq,n)))
print( f' Fractionによる計算結果の近似値: {s1}' )
print( f'gmpy2.mpqによる計算結果の近似値: {s2}' )
```

このプログラムを実行した例を次に示す.

実行例. Windows 環境のコマンドプロンプトウィンドウでの実行結果

```
sum_{i=1}^{n}{i/(i+1)} の計算:  n = 10000 , 速度比
-------------------------------------------------
 Fraction による計算時間:  0.04705 (sec),  1.00 倍
gmpy2.mpq による計算時間:  0.01490 (sec),  3.16 倍
-------------------------------------------------
 Fraction による計算結果の近似値:  9991.2122939739547
gmpy2.mpq による計算結果の近似値:  9991.2122939739547
```

この実行結果から，mpq による計算は Fraction による計算よりも 3 倍以上の速度であることがわかる.

3.4.6.3 浮動小数点数による特殊関数の算出の比較

以下に示すプログラム numSpdTest03.py は，e^e を高い精度の浮動小数点数で計算するものである．具体的には，mpmath.mpf と，gmpy2.mpfr の実行時間を比較する.

プログラム：numSpdTest03.py

```
import gmpy2, timeit
from mpmath import mp

#--- mpmathの場合の実行時間 ---
mp.dps = 20000                                          # 演算精度設定
t1 = timeit.timeit('mp.exp(mp.exp(1))',globals=globals(),number=10) / 10
print(f'mpmath による e**e の計算時間(sec):{t1:7.4f}, 速度比:{1.0:5.2f}')

#--- mpfrの場合の実行時間 ---
gmpy2.get_context().precision = int(20000*3.33) # 演算精度設定
t2 = timeit.timeit('gmpy2.exp(gmpy2.exp(1))',globals=globals(),number=10) / 10
print(f'  mpfr による e**e の計算時間(sec):{t2:7.4f}, 速度比:{t1/t2:5.2f}')

#--- 計算結果 ---
print('-'*78)
print('mpmathによる計算結果の近似値 :',str(mp.exp(mp.exp(1)))[:47])
print('  mpfrによる計算結果の近似値 :',str(gmpy2.exp(gmpy2.exp(1)))[:47])
```

このプログラムを実行した例を次に示す.

実行例. Windows 環境のコマンドプロンプトウィンドウでの実行結果

```
mpmath による e**e の計算時間 (sec):  0.1278, 速度比:  1.00
  mpfr による e**e の計算時間 (sec):  0.0112, 速度比:  11.36
------------------------------------------------------------------------------
mpmath による計算結果の近似値 :  15.154262241479264189760430272629911905528548 53
  mpfr による計算結果の近似値 :  15.154262241479264189760430272629911905528548 53
```

この実行結果から，mpfr による計算は mpmath の mpf による計算よりも 11 倍以上の速度であることがわかる.

mpfr による演算が常に高速であるとは限らない．次のプログラム numSpdTest04.py は，リーマンゼータ関数 $\zeta(s)$

を高い精度の浮動小数点数で計算するものであるが，mpfr の方が遥かに遅い事例である．

プログラム：numSpdTest04.py

```
import timeit
import gmpy2
from mpmath import mp

digits = 1000        # 精度（桁数）の設定
mp.dps = digits
gmpy2.get_context().precision = int(digits * 3.33)

# --- mpmathによるζ(3) ---
t1 = timeit.timeit('mp.zeta(3)', globals=globals(), number=10) / 10
print(f'mpmath による ζ(3) 計算時間 (sec): {t1:.5f}, 速度比:{1.0:8.4f}')

# --- mpfrによるζ(3) ---
t2 = timeit.timeit('gmpy2.zeta(3)', globals=globals(), number=10) / 10
print(f'  mpfr による ζ(3) 計算時間 (sec): {t2:.5f}, 速度比:{t1/t2:8.4f}')

# 計算結果の出力
print('---計算結果（近似値）'+'-'*39)
print('mpmath:', str(mp.zeta(3))[:52])
print('mpfr:  ', str(gmpy2.zeta(3))[:52])
```

このプログラムを実行した例を次に示す．

実行例． Windows 環境のコマンドプロンプトウィンドウでの実行結果

```
mpmath による ζ(3) 計算時間 (sec):  0.00042, 速度比:  1.0000
  mpfr による ζ(3) 計算時間 (sec):  0.18206, 速度比:  0.0023
---計算結果（近似値）---------------------------------------
mpmath:  1.20205690315959428539973816151144999076498629234049
mpfr:    1.20205690315959428539973816151144999076498629234049
```

このケースでは mpmath の方が gmpy2（mpfr）よりも約 450 倍早い．

3.4.6.4　巨大素数の生成

以下に示すプログラム numSpdTest06.py は，素数を生成してファイルに保存するものである．このプログラムでは，gmpy2 の next_prime 関数を用いて素数を生成している．この関数は Baillie-PSW テストや Miller-Rabin テストなどを用いて，極めて高い信頼性の素数[99] を生成する．

プログラム：numSpdTest06.py

```
import gmpy2, timeit

#--- 素数リストの生成 ---
primeList = []       # 素数を蓄積するリスト

def genPrimes(s,n):
    global primeList
    p = s
    for _ in range(n):
        p = gmpy2.next_prime(p)
        primeList.append(p)

#--- 実行時間の測定 ---
s = 10**78       # 開始位置
n = 10000        # 生成個数
print(n,'個の素数を生成します…')
print('開始ポイント:\n',s,sep='')
t = timeit.timeit('genPrimes(s,n)',globals=globals(),number=1) / 1
print('生成時間 :',t,'(sec)')

#--- 素数をファイルに出力 ---
fname = 'primenumbers.txt'
print('ファイル :',fname,'に出力します．')
f = open(fname,'w',encoding='utf-8')
```

[99] **疑似素数**と呼ばれるものであるが，暗号関連の実際の処理にも用いられている．

```
for p in primeList:
    print(int(p),file=f)
f.close()
```

このプログラムを実行した例を次に示す.

実行例. Windows 環境のコマンドプロンプトウィンドウでの実行結果

```
10000 個の素数を生成します…
開始ポイント:
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
生成時間 :  2.7033722000196576 (sec)
ファイル :  primenumbers.txt に出力します.
```

この結果，ファイル primenumbers.txt に次のような内容（10,000 個の素数）が保存される.

ファイル： primenumbers.txt

```
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000093
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000297
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000339
                                    ⋮
                                (途中省略)
                                    ⋮
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000001811437
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000001811499
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000001811673
```

この処理が 3 秒未満で完了していることがわかる.

3.4.6.5 乱数の品質と生成の速度

mpz_urandomb 関数による乱数生成と，secrets モジュール[100] の randbits 関数による乱数生成を比較するプログラムを numRandTest01.py に示す.

プログラム：numRandTest01.py

```
import gmpy2, secrets, time
import matplotlib.pyplot as plt

lenR = 128          # 乱数のビット長
numR = 10000        # 生成する乱数の個数

#--- gmpy2による乱数生成 ---
t1 = time.time()
rs = gmpy2.random_state(1)      # 乱数状態オブジェクト
R1 = []
for _ in range(numR):
    R1.append( gmpy2.mpz_urandomb(rs,lenR) )
t2 = time.time()

#--- secretsによる乱数生成 ---
t3 = time.time()
R2 = []
for _ in range(numR):
    R2.append( secrets.randbits(lenR) )
t4 = time.time()

#--- 生成時間の比較 ---
print( f'{lenR}ビットの乱数を{numR}個生成:' )
print( f' gmpy2.mpz_urandomb:{t2-t1:9.6f}(sec)' )
print( f'   secrets.randbits:{t4-t3:9.6f}(sec)' )

#--- 乱数の散布図 ---
rat = 2**lenR               # 正規化用係数
R1_n = [ float(r)/rat for r in R1 ]
R2_n = [ float(r)/rat for r in R2 ]
fig, ax = plt.subplots( 1,2, figsize=(14,5) )
ax[0].scatter(R1_n[::2],R1_n[1::2],alpha=0.4)
ax[0].set_title('gmpy2.mpz_urandomb')
ax[1].scatter(R2_n[::2],R2_n[1::2],alpha=0.4)
```

[100] Python の標準ライブラリの 1 つ.

```
35 ax[1].set_title('secrets.randbits')
36 plt.savefig('numRandTest01.eps')
37 plt.show()
```

このプログラムを実行すると，乱数生成に要した時間が出力される.（下記）

実行時の出力. Windows のコマンドウィンドウでの実行

```
128 ビットの乱数を 10000 個生成:
gmpy2.mpz_urandomb:  0.002003(sec)
  secrets.randbits:  0.003888(sec)
```

gmpy2 の乱数生成のほうが secrets のそれより約 2 倍早いことがわかる．また，上のプログラムは乱数を散布図にプロットして，偏りの様子を示す.

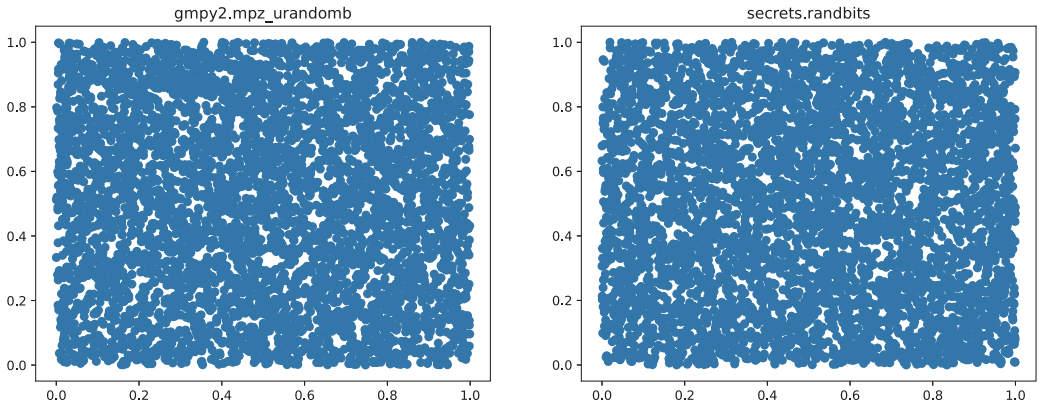


図 124: 生成した乱数を散布図にして眺めるたところ

生成した乱数の偶数インデックスの並びを横軸,
奇数インデックスの並びを縦軸にして散布図を作成した.

`mpz_urandomb` 関数が生成する乱数は十分な品質であるように見える.

4　セキュリティ関連

4.1　hashlib

hashlib は暗号学的な**メッセージダイジェスト**を生成するためのモジュールであり，Python に標準的に提供されている．hashlib が提供するダイジェスト作成アルゴリズムを表 42 に示す．

表 42: hashlib で利用できるダイジェスト生成アルゴリズム

名称（系統）	解　　説
MD5（非推奨）	2010 年頃まで Linux のパスワードのダイジェストとして採用されていた．
SHA-1（非推奨）	2016 年まで一般的に使用されていた．
SHA-2（現行標準）	アルゴリズム：SHA-224，SHA-256，SHA-384，SHA-512
SHA-3（次世代標準）	アルゴリズム：SHA3-224，SHA3-256，SHA3-384，SHA3-512
BLAKE2（高速安全）	アルゴリズム：BLAKE2b，BLAKE2s
SHAKE（可変長）	アルゴリズム：SHAKE128，SHAKE256

メッセージダイジェストを生成する機能は，**パスワード文字列の秘匿化**[101] や，文書の**デジタル署名**の生成に必要となる．このモジュールは使用に先立って，次のようにして必要なモジュールを読み込んでおく必要がある．

```
import hashlib
```

4.1.1　基本的な使用方法

ここでは，パスワード文字列を秘匿化する処理を例に挙げて hashlib の基本的な使用方法について説明する．

例. パスワード文字列 'MyPassword' の秘匿化（SHA-3 による）

```
>>> import hashlib  Enter      ←モジュールの読み込み
>>> m = hashlib.sha3_256(b'MyPassword') Enter  ←ダイジェスト生成用オブジェクトの生成（SHA-3）
>>> m.digest()  Enter      ←ダイジェスト生成          ↓得られたダイジェスト（バイト列）
b'F}(\xf6jXw8\x17\t;)\xac\xbf\xa3\x936\xfffFZ\xb7\xe1\xee%\xd7:L\xdb\x19\x86v'
>>> m.hexdigest() Enter ← 16 進数表現でダイジェスト生成      ↓得られたダイジェスト（文字列）
'467d28f66a58773817093b29acbfa39336ff66465ab7e1ee25d73a4cdb198676'
```

このように，ハッシュ化のアルゴリズム（この例では SHA-3）の名前のコンストラクタの引数に秘匿化したい（ハッシュ化したい）文字列をバイト列形式で与えた後，digest メソッドでダイジェスト（秘匿化されたデータ）を生成する．あるいは hexdigest メソッドを使用すると 16 進数表現の文字列としてダイジェストを得ることができる．

4.2　passlib

UNIX 系 OS（Linux など）の /etc/shadow に使用するために，パスワードのハッシュ文字列（**メッセージダイジェスト**）を生成するには passlib が利用できる．このモジュールに関する情報は，公式インターネットサイト

https://passlib.readthedocs.io/

から得られる．

【基本的な考え方】

平文のパスワードと**ソルト**[102]（salt）から，暗号化アルゴリズムによってメッセージダイジェストを生成する．

4.2.1　使用できるアルゴリズム

passlib で使用できる暗号化アルゴリズムを表 43 に示す．
次に，平文のパスワードからメッセージダイジェストを生成する例を示す．

[101] UNIX 系 OS（Linux など）の /etc/shadow に使用するためにパスワードのハッシュ文字列を生成するには，後の passlib を用いるのが良い．
[102] **ソルト**：メッセージダイジェストから平文を見破る**パスワードクラック**を難しくするために，メッセージダイジェストを生成する際に与える文字列．

表 43: passlib で利用できる主なハッシュ化アルゴリズム

分類	使用例（モジュールまたはクラス）
SHA-512 方式	passlib.hash.sha512_crypt
SHA-256 方式	passlib.hash.sha256_crypt
bcrypt 方式	passlib.hash.bcrypt，passlib.hash.bcrypt_sha256
Argon2 方式	passlib.hash.argon2
scrypt 方式	passlib.hash.scrypt
旧式 UNIX 互換（非推奨）	passlib.hash.md5_crypt，passlib.hash.des_crypt

例. SHA-512 アルゴリズムによるパスワードハッシュの生成

```
>>> from passlib.hash import sha512_crypt  Enter    ←モジュールの読込み
>>> sha512_crypt.hash('MyPassword')  Enter    ←ハッシュを生成（自動ソルト付与）
'$6$rounds=656000$qbtYg1GWSqs3iYO8$NBkLWTnqgrXRCRwTWbq5pT4hH/qcZLAwkof50xoy.mSb2bRqb2Hq
 JLTxIm2NwlnT5lMBcIW1u799us1v4ABtU0'    ←生成されたハッシュ文字列
```

この例では，sha512_crypt アルゴリズムを用いて，平文パスワード ‘MyPassword’ からメッセージダイジェストを生成している．

5　プログラムの高速化／アプリケーション構築

Python インタプリタによるプログラムの実行時間は，同様のアルゴリズムを実装した C 言語のプログラムと比べて数十倍〜百数十倍程度大きい．このことは計算量の多い処理を行う際に大きな問題となる．ここでは，プログラムの実行時間を小さくするための方法についていくつか紹介する．

5.1　Cython

Python のプログラムの実行時間を短縮するための方法の 1 つに **Cython 処理系** の利用がある．Cython 処理系は公式のインターネットサイト http://cython.org/ から入手できるが，必要となる C 言語処理系[103] も Cython の導入に先立って準備しておくこと．本書では Cython 処理系について導入的に解説する．Cython の詳細に関しては公式サイトをはじめとするドキュメント[104] を参照のこと．

Cython 処理系は Python の言語仕様を拡張した **Cython 言語**を扱う．Cython 言語で記述されたソースプログラムは一旦 C 言語のソースプログラムに翻訳され，更にそれが C 言語処理系によって実行形式のプログラムに翻訳される．最終的に Cython のプログラムは Python 処理系のためのモジュールとなり，Python のプログラムから呼び出すことができる．Cython 処理系を用いて作成されたモジュールプログラムの実行速度は，通常の Python プログラムの実行に比べて数倍からそれ以上となる．

5.1.1　使用例

サンプルプログラムを挙げ，Cython を用いることでプログラムの実行時間が短縮されることを例示する．次に示す fib.py はフィボナッチ数列を表示するプログラム[105] である．

プログラム：fib.py

```
1  import time
2
3  # フィボナッチ数列の生成
4  def fib1(n):
5      if n == 0 or n == 1:
6          return( 1 )
7      else:
8          f = fib1(n-1) + fib1(n-2)
9          return( f )
10
11 def fib(n):
12     t1 = time.time()
13     for i in range(n):
14         print( fib1(i), end=',' )
15     else:   print()
16     t = time.time() - t1
17     print(t,'秒')
```

このプログラムをモジュールとして Python 処理系に読み込んで実行した例を次に示す．

例. fib.fib(36) の実行

```
>>> import fib   Enter      ←モジュールとして読込み
>>> fib.fib(36)   Enter     ←フィボナチ数列表示の開始
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,
46368,75025,121393,196418,317811,514229,832040,1346269,2178309,3524578,5702887,9227465,
14930352,
3.096280097961426 秒          ←要した時間（Windows 11,Core i7-1195G7,2.9GHz）
```

次に，このプログラムを Cython によって高速化する手順を示す．

---

[103] 例：Windows 環境では Visual Studio，Apple Macintosh の場合は Xcode.

[104] Cython の日本語ドキュメントサイト：http://omake.accense.com/static/doc-ja/cython/

[105] ここに示すプログラムは，フィボナッチ数の再帰的な関数定義をそのまま実装したものである．フィボナッチ数の生成は**動的計画法**などのアルゴリズムを採用すると大幅に高速化できるが，ここでは大きな実行時間を要する例として，敢えてこの形のプログラムを示す．

【手順】

1． Cython のプログラムとして用意する

先のプログラム fib.py の拡張子を '.pyx' にしたものを用意する．プログラム自体は変更しない．

今回は Cython のプログラム fibC.pyx として用意する．

2． 翻訳用スクリプトを作成して実行する

Cython プログラムを翻訳するための次のような Python プログラム setup.py を用意する．

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
    cmdclass = {'build_ext':  build_ext},
    ext_modules = [Extension('fibC', ['fibC.pyx'])]      #  ←プログラム名を指定する
)
```

下から 2 行目にあるように，翻訳対象のプログラムの名前を指定する．

この翻訳用スクリプトを，build_ext –inplace という引数とオプションを付けて OS のコマンドとして実行する．

例. 翻訳処理

```
C:\Users\katsu> py setup.py build_ext --inplace   Enter     ←翻訳処理の開始
Compiling fibC.pyx because it changed.
[1/1] Cythonizing fibC.pyx
fibC.c
fibC.c(4986):  warning C4244:  '=':  'Py_ssize_t' から 'long' への変換です。
データ が失われる可能性があります。
  ライブラリ build\temp.win-amd64-cpython-313\Release\fibC.cp313-win_amd64.lib と
オブジェクト build\temp.win-amd64-cpython-313\Release\fibC.cp313-win_amd64.exp を作成中
コード生成しています。
コード生成が終了しました。
```

この処理の結果，モジュール fibC が生成される．

3． Python 処理系で実行する

Cython で作成したモジュール fibC を Python 処理系に読み込んで実行する例を次に示す．

例. コンパイルされた figC を読み込んで実行

```
>>> import fibC   Enter      ←モジュールとして読込み
>>> fibC.fib(36)   Enter      ←フィボナチ数列表示の開始
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,
46368,75025,121393,196418,317811,514229,832040,1346269,2178309,3524578,5702887,9227465,
14930352,
1.6023225784301758 秒          ←要した時間（Windows 11,Core i7-1195G7,2.9GHz）
```

先に示した fib.py を Python 処理系で実行する場合と比べて，実行速度が約 2 倍になっていることがわかる．

5.1.2　高速化のための調整

Cython は Python のプログラムを C 言語のプログラム変換して翻訳する．このため，Cython のプログラムを記述する際に変数や関数の型を明に宣言することにより，より効率的に C 言語のプログラムに変換されることがある．先の fib.py を元に，変数や関数の型を明に宣言する形にしたプログラム fibC2.pyx を次に示す．

プログラム：fibC2.pyx

```
1  import time
2
3  # フィボナッチ数列の生成
4  cdef int fib1( int n ):      # 型を指定した関数の定義
5      if n == 0 or n == 1:
```

```
6           return( 1 )
7       else:
8           f = fib1(n-1) + fib1(n-2)
9           return( f )
10
11  def fib( int n ):            # 引数の型の指定
12      cdef int    i            # 変数の型の指定
13      t1 = time.time()
14      for i in range(n):
15          print( fib1(i), end=',' )
16      else: print()
17      t = time.time() - t1
18      print(t,'秒')
```

このプログラムでは，変数や関数の仮引数の型を明に宣言し，外部から呼び出されない関数の型を指定している．型の指定には 'cdef' を用いる．このプログラムを翻訳して実行した結果の例を次に示す．

例. コンパイルされた figC2 を読み込んで実行

```
>>> import fibC2   Enter      ←モジュールとして読込み
>>> fibC2.fib(36)  Enter      ←フィボナチ数列表示の開始
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,
46368,75025,121393,196418,317811,514229,832040,1346269,2178309,3524578,5702887,9227465,
14930352,
0.10167288780212402 秒          ←要した時間（Windows 11,Core i7-1195G7,2.9GHz）
```

はじめに示した fib.py を Python 処理系で実行する場合と比べて，実行速度が約 30 倍になっていることがわかる．

5.2 Numba

Numba は LLVM [106] を用いて Python のプログラムを実行するためのモジュールであり，関連の情報はインターネットの公式サイト http://numba.pydata.org/ から入手できる．本書では Numba について導入的に解説する．Numba に関する詳しいことは公式サイトを参照のこと．

5.2.1 基本的な使用方法

Numba は **JIT コンパイラ**[107] を使用して Python のプログラムを実行する．具体的には，Python のソースプログラム中に JIT コンパイラに対する指示を**デコレータ**「@njit」,「@jit」として記述するという方法を取る．この方法によると，元々Python で記述したプログラムをあまり変更することなく実行時間の短縮が望める．ここでは先に挙げたフィボナッチ数を計算するプログラム fib.py を Numba によって高速化する過程を例示する．

fib.py を Numba で実行するために改訂したものが次に示す fibN1.py である．

プログラム： fibN1.py

```
1   from numba import jit, njit
2   import time
3
4   # フィボナッチ数列の生成
5   @njit
6   def fib1(n):
7       if n == 0 or n == 1:
8           return( 1 )
9       else:
10          f = fib1(n-1) + fib1(n-2)
11          return( f )
12
13  def fib(n):
14      _ = fib1(3)      # JIT用ウォームアップ
15      t1 = time.time()
```

[106] C 言語をはじめとする各種言語のためのコンパイラ基盤である.. 元々はイリノイ大学（米）で開発され，オープンソースとして公開されている．

[107] 「実行時コンパイラ」（Just-In-Time コンパイラ）．ソフトウェアの実行時にコードのコンパイルを行い実行速度の向上を図るコンパイラのこと．

```
    for i in range(n):
        print( fib1(i), end=',' )
    else: print()
    t = time.time() - t1
    print(t,'秒')
```

解説：

プログラムの 1 行目で Numba のパッケージを読み込んでいる．5 行目にある ‘@njit’ は，直下に記述した関数を JIT コンパイルの対象とすることを指示するデコレータである．

このプログラムを実行した結果の例を次に示す．

例. figN1 を読み込んで実行

```
>>> import fibN1   Enter      ←モジュールとして読込み
>>> fibN1.fib(36)   Enter     ←フィボナチ数列表示の開始
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,
46368,75025,121393,196418,317811,514229,832040,1346269,2178309,3524578,5702887,9227465,
14930352,
0.09687566757202148 秒          ←要した時間（Windows 11,Core i7-1195G7,2.9GHz）
```

はじめに示した fib.py を実行する場合と比べて，実行速度が 30 倍以上になっていることがわかる．

重要）

デコレータ「‘@njit’」による装飾の対象となる関数は，キーワード引数が使えないなど，様々な制約が求められる．「‘@jit’」による装飾では，実行時の最適化を弱めて，より柔軟な関数の記述が可能になる．これらデコレータにはオプションの引数を与えて，最適化に関する様々な調整が可能である．詳しくは公式インターネットサイトの情報を参照のこと．

5.2.2 型指定による高速化

JIT コンパイルする対象の関数の引数や戻り値のデータ型を指定することで，プログラムの実行時間が更に短縮できる場合がある．関数の型指定をする形で先の fibN1.py を改訂したプログラムを fibN2.py に示す．

プログラム：fibN2.py

```
from numba import njit, jit, i8
import time

# フィボナッチ数列の生成
@njit('i8(i8)')
def fib1(n):
    if n == 0 or n == 1:
        return( 1 )
    else:
        f = fib1(n-1) + fib1(n-2)
        return( f )

def fib(n):
    _ = fib1(3)          # JIT用ウォームアップ
    t1 = time.time()
    for i in range(n):
        print( fib1(i), end=',' )
    else: print()
    t = time.time() - t1
    print(t,'秒')
```

解説：

プログラムの 1 行目で Numba のパッケージを読み込み，5 行目にデコレータを記述している点は先の fibN1.py と共通するが，関数の引数と戻り値の型を ‘i8’ で指定している．これは「8 バイト整数型」を意味する表記である．（詳しくは公式サイトを参照のこと）

※ Numba による高速化の度合いは対象となる関数や使用する他のパッケージに大きく依存する点に留意すること．

5.3 ctypes

ctypes は標準のパッケージであり，C 言語と互換性のあるデータ型を提供し，動的リンク/共有ライブラリ内の関数呼び出しを可能にする．本書では ctypes について導入的に解説する．ctypes に関する詳しい情報は Python の公式サイトのドキュメントなどを参照すること．

5.3.1 C 言語による共有ライブラリ作成の例

C 言語で記述した関数を GNU の C コンパイラによって共有ライブラリにする手順を例示する．フィボナッチ数を求める C 言語のプログラム（関数）を fibCC.c に示す．

プログラム：fibCC.c

```
#include    <stdio.h>

long fib0( long n )
{
    if ( n == 0 ) {
        return( 1 );
    } else if ( n == 1 ) {
        return( 1 );
    } else {
        return( fib0(n-1) + fib0(n-2) );
    }
}

__declspec(dllexport) void fib( long n )
{
    long c;

    for ( c = 0; c < n; c++ ) {
        printf("%ld,",fib0(c));
    }
    printf("\n");
}
```

このプログラムを gcc コマンドによってコンパイルし，他の言語処理系から使用できる**共有ライブラリ**を作成するには，コンパイルオプション '-shared' を指定する．

例. MinGW 環境下（Windows）での共有ライブラリの作成

```
gcc -O2 -shared -o fibCC.dll fibCC.c
```

この処理が正常に終了すると共有ライブラリ fibCC.dll が作成される．

参考）Microsoft Visual Studio の cl.exe でコンパイルする場合は `/LD` オプションを与える．

例. Microsoft Visual Studio の cl.exe で dll を作成する

```
cl fibCC.c /LD /O2              （/O2 は最適化オプション.）
```

注意）dll を作成する場合は，アーキテクチャのビットモデル（32/64 ビット）を意識すること．

参考）C 言語側の関数のエクスポート

上のプログラム fibCC.c では fib 関数の記述の前に `__declspec(dllexport)` がある．これは外部に関数をエクスポートする際の記述であり，これを書かなくても問題なく dll を作成できるコンパイラ（例：GCC）もある．

5.3.2 共有ライブラリ内の関数を呼び出す例

ctypes モジュールの CDLL クラスを使用することで，外部の共有ライブラリを Python 処理系に読み込み，共有ライブラリ内の関数を呼び出すことができる．CDLL クラスのインスタンス（CDLL オブジェクト）を生成する際に，コンストラクタに外部の共有ライブラリを指定する．先の例で作成した共有ライブラリを読み込んで関数を呼び出す Python 側プログラムの例を fibCCpy.py に示す．

プログラム：fibCCpy.py

```
1  import ctypes
2  import time
3
4  # 共有ライブラリの読込み
5  ex = ctypes.CDLL('./fibCC.dll')
6
7  t1 = time.time()
8
9  # 共有ライブラリ中の関数の呼び出し
10 ex.fib(36)
11
12 t = time.time() - t1
13 print(t,'秒')
```

このプログラム例では，共有ライブラリ fibCC.dll を読み込んで CDLL オブジェクト ex を生成し，それに対するメソッドとして関数名を指定することで，関数 fib を呼び出している．このプログラムを実行した例を次に示す．

例．fibCCpy.py の実行（Windows 環境：MinGW の GCC で dll を作成）

```
C:\Users\katsu> py fibCCpy.py
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,
46368,75025,121393,196418,317811,514229,832040,1346269,2178309,3524578,5702887,9227465,
14930352,
0.028124094009399414 秒          ←要した時間（Windows 11,Core i7-1195G7,2.9GHz）
```

はじめに示した fib.py を実行する場合と比べて，実行速度が約 110 倍になっていることがわかる．

5.3.3　引数と戻り値の扱いについて

実用的な形で Python と C 言語の連携をするためには，相互にデータの受け渡しをする必要がある．ここでは，C 言語で作成した共有ライブラリと Python プログラムとの間で変数の値や配列を受け渡しするための基本的な方法を紹介する．

次に示す C 言語のプログラム ctypesTest01.c は，引数として受け取った値を処理して値を返す 3 つの関数を実装したものである．

C のプログラム：ctypesTest01.c

```
1  #include    <stdio.h>
2  #include    <string.h>
3
4  /*-------------------------------------------------------*
5   *  整数（int）の受け渡し                                *
6   *-------------------------------------------------------*/
7  int excgInt( int a )
8  {
9      int r;
10
11     r = 2 * a;
12
13     printf("C側)\t\t与えられた整数 %d を2倍します: %d\n",a,r);
14     return( r );
15 }
16
17 /*-------------------------------------------------------*
18  *  浮動小数点数（double）の受け渡し                     *
19  *-------------------------------------------------------*/
20 double  excgDouble( double a )
21 {
22     double  r;
23
24     r = 2.0 * a;
25
26     printf("C側)\t\t与えられた浮動小数点数 %lf を2.0倍します: %lf\n",a,r);
27     return( r );
28 }
```

```
29
30 /*-------------------------------------------------------*
31  *  文字列（char*）の受け渡し                            *
32  *-------------------------------------------------------*/
33 char    *excgString( char *a )
34 {
35     static char r[256];
36
37     strcpy(r,a);
38     strcat(r," <- を返します. ");
39
40     printf("C側)\t\t与えられた文字列 \'%s\' を加工します: '%s'\n",a,r);
41     return( r );
42 }
```

このプログラムで実装した関数は次の 3 つである.

- int excgInt(int a) : 引数 a の値を 2 倍した値を返す関数.
- double excgDouble(double a) : 引数 a の値を 2.0 倍した値を返す関数.
- char *excgString(char *a) : 引数 a で示す文字列を加工したもののポインタ返す関数.

これらの関数を呼び出す Python のプログラム ctypesTest01.py を次に示す.

Python 側プログラム：ctypesTest01.py

```
1  import ctypes
2
3  # 共有ライブラリの読込み
4  ex = ctypes.CDLL('./ctypesTest01.dll')
5
6  # 整数の受け渡し
7  r = ex.excgInt( 4 )          # 戻り値は暗黙で int 型
8  print('python側)\t戻り値:',r)
9
10 # 浮動小数点数の受け渡し
11 ex.excgDouble.restype = ctypes.c_double      # 戻り値を double と設定
12 a = ctypes.c_double(2.3)                     # 引数を double に変換
13 r = ex.excgDouble( a )                       # 関数呼び出し
14 print('python側)\t戻り値:',r)
15
16 # 文字列の受け渡し
17 ex.excgString.restype = ctypes.c_char_p      # 戻り値を char* と設定
18 a0 = '元の文字列'.encode('utf-8')             # 送るデータを作成
19 a = ctypes.c_char_p( a0 )                     # それを char* に変換
20 r0 = ex.excgString( a )                      # 関数呼び出し
21 r = r0.decode('utf-8')
22 print('python側)\t戻り値: \'',r,'\'',sep='')
```

解説：

整数（int 型）の値を受け渡しする方法は単純であり，7 行目の記述の通りである．整数以外の型の引数や戻り値を扱う場合は，関数呼び出しに先立って準備のための処理が必要となる.

【共有ライブラリの関数の戻り値の扱い】

C 言語で記述した関数の戻り値の型は restype 属性で指定する．今回のプログラム（ctypesTest01.py）の場合，excgDouble 関数は double 型の値を返すので，11 行目にあるように restype として ctypes.c_double を設定している．同様に，文字列のポインタを返す関数 excgString には，17 行目にあるように restype として ctypes.c_char_p を設定している.

● 関数からの戻り値の型の指定

CDLL オブジェクト. 共有ライブラリの関数名.restype = ctypes. **型指定**

C 言語の型を意味する ctypes の表現（一部）を表 44 に示すが，更に詳しい情報については Python の公式サイトを参照すること.

表 44: ctypes における C 言語のための型の指定：

ctypes での型	C 言語における型	ctypes での型	C 言語における型
c_int	int 型	c_long	long 型
c_uint	unsigned int 型	c_ulong	unsigned long 型
c_short	short int 型	c_ushort	unsigned short int 型
c_float	float 型	c_double	double 型
c_char, c_byte	char 型	c_ubyte	unsigned char 型
c_char_p	char のポインタ型	c_void_p	void のポインタ型

先のプログラム ctypesTest01.py を実行した例を次に示す.

実行例.

```
C 側)          与えられた整数 4 を 2 倍します： 8
python 側)     戻り値： 8
C 側)          与えられた浮動小数点数 2.300000 を 2.0 倍します： 4.600000
python 側)     戻り値： 4.6
C 側)          与えられた文字列 ’元の文字列’ を加工します： ’元の文字列 <- を返 します．’
python 側)     戻り値： ’元の文字列 <- を返します．’
```

【記憶の管理について】

先のプログラム例 ctypesTest01.c，ctypesTest01.py では，文字列の処理結果は関数 excgString 内の static の配列に格納されている．この形の実装では記憶の管理（配列の管理）が C 言語側に委ねられていることになるが，データ構造の管理を全て Python 側で行うには，Python 側のデータ構造のポインタのみを C の関数に渡すという形で実装する必要がある．また，C のプログラム側で malloc 関数などで記憶域を確保するというのも記憶域の解放といった管理を C 側に委ねなければならないので，Python との連携を安全な形で実現するには余り好ましくない.

次に，Python 側のリストを C 言語の配列のポインタとして受け渡す方法について説明する.

5.3.3.1 配列データの受け渡し

Python 側のリストなど，多数の要素を持ったデータ構造を C 言語の関数に渡すには，C 言語のポインタの形に変換する必要がある．また，C 言語の関数で処理した配列を Python 側で受け取るには，やはり配列のポインタとして Python プログラムが受け取り，それをリストなどのデータ構造に変換する必要がある.

ここでは，double 型の配列に格納された値から正弦関数の値を算出して，同じく double 型の配列として作成するプログラムを例に挙げ，C 言語の関数との間で配列を受け渡しする方法を紹介する．定義域の値の配列から正弦関数の値の配列を作成する C 言語のプログラム ctypesTest02.c を次に示す.

C のプログラム：ctypesTest02.c

```
 1 #include    <stdio.h>
 2 #include    <math.h>
 3
 4 /*-------------------------------------------------------*
 5  *  配列の受け渡し                                       *
 6  *-------------------------------------------------------*/
 7 int excgArray( double *a, int n, double *r )
 8 {
 9     int     x;
10
11     for ( x = 0; x < n; x++ ) {
12         r[x] = sin( a[x] );
13     }
14
15     printf("C側)\t\t正弦関数の値の列を算出しました．: %d個\n",x);
16
17     return( x );
18 }
```

このプログラムは，定義域のデータを保持する配列のポインタを double *a に受け取って正弦関数の値を算出する関数 excgArray を実装したものである．計算結果も配列に格納するが，そのための配列の記憶域も呼び出し元（Python プログラム側）が用意したものを使用する．（配列のポインタを仮引数 r に受け取る）すなわち，Python 側で用意した配列のポインタを関数 excgArray に渡すので，C 言語プログラムの側では，配列の確保といった記憶の管理はしない．

C 言語の関数 excgArray を呼び出す Python プログラム ctypesTest02.py を次に示す．

Python 側プログラム：ctypesTest02.py

```
1  import ctypes
2  import matplotlib.pyplot as plt
3
4  # 共有ライブラリの読込み
5  ex = ctypes.CDLL('./ctypesTest02.dll')
6
7  # 引数と戻り値の型の設定
8  ex.excgArray.argtype =  [ ctypes.POINTER(ctypes.c_double),
9      ctypes.c_int, ctypes.POINTER(ctypes.c_double) ]      # 引数の型を設定
10 ex.excgArray.restype = ctypes.c_int                      # 戻り値を double と設定
11
12 # 配列データの生成（リスト）
13 ax = [ x/100.0 for x in range(628) ]    # 定義域用
14
15 # リストをCの配列（ポインタ）に変換
16 n = len( ax )                          # 要素の個数（長さ）
17 ar_t = ctypes.c_double * n             # 配列の型の生成
18 ax2 = ctypes.byref( ar_t( *ax ) )      # Cに渡す定義域の配列（ポインタ）
19
20 # 得られた値域のデータを保持する配列の用意
21 ay2  = ar_t()                          # Cに渡す値域の配列（ポインタ）
22
23 # Cの関数の呼び出し
24 n2 = ex.excgArray( ax2, n, ay2 )
25 ay = list(ay2)                         # 値域の配列をリストに変換
26
27 print('Python側)\t戻り値:',n2)
28
29 # 可視化
30 plt.plot(ax,ay)
31 plt.show()
```

解説：

このプログラムでは，リスト ax に 0〜2π の範囲の数値を 0.01 刻みで作成し，それを C の関数に渡すことができるポインタ ax2 として変換している．計算結果の正弦関数の値を格納する配列のポインタを ay2 に受け取り，それを Python のリスト ay に変換している．定義域のリスト ax と値域のリスト ay から matplotlib を使って関数のグラフとして可視化している．

プログラムの実行の結果，標準出力に次のように表示される．

実行例.

```
C 側)        正弦関数の値の列を算出しました．:  628 個
Python 側)  戻り値:  628
```

また，プロットしたグラフが図 125 のような形で表示される．

【配列のポインタを C の関数の引数に与えるための処理】

呼び出す関数の引数の型を argtype 属性にリスト形式で設定する．

● 関数の引数の型の指定

CDLL オブジェクト. 共有ライブラリの関数名.argtype = **引数の型指定のリスト**

関数 excgArray は

```
int excgArray( double *a, int n, double *r )
```

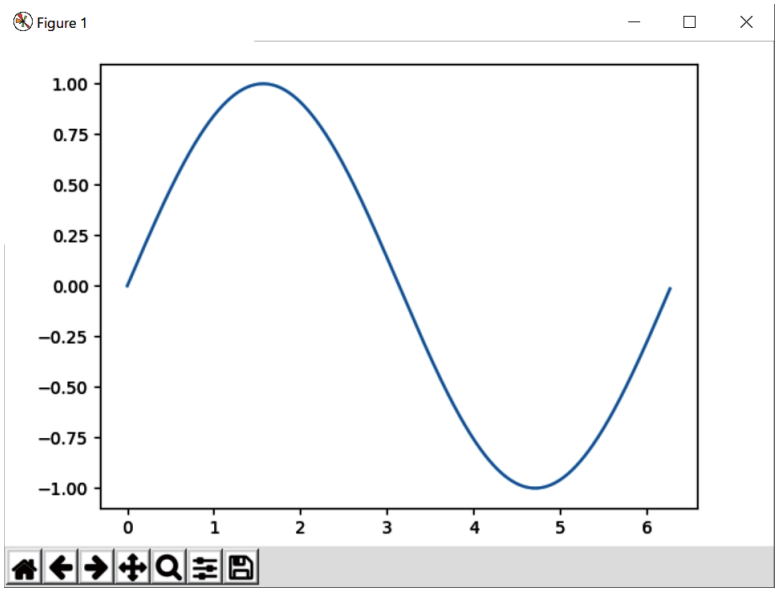


図 125: プロットの表示

のような型として定義されているので，引数の型に応じて次のように argtype 属性を設定する．

[ctypes.POINTER(ctypes.c_double), ctypes.c_int, ctypes.POINTER(ctypes.c_double)]

これを行っているのがプログラム ctypesTest02.py の 8〜9 行目である．また，配列へのポインタであることを指定するために POINTER(**型指定**) という表現を用いる．

プログラムの 17 行目では，C の関数の引数に与える配列の型をサイズを含めて ar_t として定義している．

● **配列とサイズの型の定義**

型指定 * サイズ

この型を用いて 18 行目では定義域のリスト ax から C に渡す配列 ax2 を生成し，21 行目では，計算結果を格納する配列 ay2 を生成している．

● **C 言語に渡す形式への変換**

byref(**型 (*リスト**))

24 行目では，ax2, ay2，それにデータの個数 n を引数に与えて C 言語の関数 excgArray を呼び出している．計算結果が格納されている配列 ay2 の内容を 25 行目で Python のリスト ay に変換している．

最後に 30〜31 行目でグラフをプロットしている．

5.4 PyInstaller

PyInstaller を用いると，Python のスクリプト（〜.py）から単独で動作するアプリケーションプログラム（Windows の場合は〜.exe）を生成（アプリケーションとしてビルド）することができる．

PyInstaller に関する情報は，公式インターネットサイト

https://www.pyinstaller.org/

から得られる．

5.4.1 簡単な使用例

サンプルスクリプト apptest00.py を用いて，アプリケーションをビルドする例を示す．

Python スクリプト：apptest00.py

```
1 from math import *      # 注）この形式の読込みは推奨されない
2
3 while True:
4     s = input('式（exitで終了）> ')
5     if s == 'exit':
6         break
7     else:
8         v = eval(s)
9         print( v )
```

このスクリプトは，標準入力（ターミナルウィンドウ）から入力された文字列を式として計算する「簡易電卓」のようなものであり，'exit' を入力すると終了する．通常のように，これを Python インタプリタで実行すると次のようになる．

例. Python インタプリタでの実行（Windows の場合）

```
C:¥Users¥katsu>py apptest00.py  Enter    ← OS のコマンドラインから Python を起動
式（exit で終了）> 1+2  Enter      ←式の入力
3          ←計算結果
式（exit で終了）> sqrt(2)  Enter    ←式の入力
1.4142135623730951  ←計算結果
式（exit で終了）> exit  Enter    ←終了
C:¥Users¥katsu>    ← OS のコマンドプロンプトに戻る．
```

次に，PyInstaller を用いて apptest00.py をビルドする例を示す．

例. PyInstaller によるビルド（Windows の場合）

```
C:¥Users¥katsu>py -m PyInstaller apptest00.py  Enter    ←アプリケーションのビルド開始
267 INFO: PyInstaller:  6.15.0, contrib hooks:  2025.8    ←ビルド処理に関するメッセージの表示が続く
267 INFO: Python:  3.13.5
289 INFO: Platform:  Windows-11-10.0.26100-SP0
          ⋮
        （途中省略）
          ⋮
12557 INFO: Building COLLECT because COLLECT-00.toc is non existent
12557 INFO: Building COLLECT COLLECT-00.toc
12772 INFO: Building COLLECT COLLECT-00.toc completed successfully.
12773 INFO: Build complete!  The results are available in:  C:\Users\katsu\TeX\Python\dist
C:¥Users¥katsu>    ← OS のコマンドプロンプトに戻る
```

この後，作業用のディレクトリ `build`（無ければ作成される）の下に，Python スクリプトと同名のディレクトリが作成され，その中に PyInstaller が作業に使用したファイル群が生成される．また，ディレクトリ `dist`（無ければ作成される）の下に，Python スクリプトと同名のディレクトリが作成され，その中にアプリケーションを構成するファイル群（DLL などを含む）が生成される．この `dist` には，スクリプト名と同じ名前の実行可能ファイル *.exe も作成される．

5.4.1.1　単一の実行ファイルとしてビルドする方法

単一の実行ファイルとしてアプリケーションをビルドするには，PyInstaller の処理を開始するコマンドラインにオプション '`--onefile`' を与える．この場合，`dist` ディレクトリ内の Python スクリプトと同名のディレクトリに，単一の実行ファイルが作成される．(例．図 126)

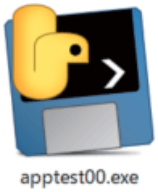


図 126: 単一の実行ファイルとしてビルドされた例（Windows の場合）

注意）Python スクリプトが使用するモジュールによっては，ビルドの段階で更なる作業が必要となることがある．詳しくは PyInstaller の公式インターネットサイトの情報や，使用するモジュールに関する情報を調べること．

■ GUI アプリケーション構築の事例

Python 標準の GUI ライブラリである Tkinter を用いて構築した GUI アプリケーションを PyInstaller で 1 つの実行形式ファイルにする事例を示す．次に示す calcTkinter01.py は簡単な電卓を実現したものである．

Python スクリプト：calcTkinter01.py

```
import tkinter as tk

#===== ボタンクリックに対応する処理 =====
def on_click(ch):
    if ch == 'C':           # クリアボタン
        var.set('')
    elif ch == '=':         # 「=」ボタンで計算実行
        expr = var.get()
        try:
            # ディスプレイの内容をevalで評価する
            result = eval(expr, {'__builtins__': None}, {})
            var.set(str(result))
        except Exception:
            var.set('Error')
    else:       # 数値と計算記号の入力
        var.set(var.get() + ch)

#===== GUI構築 =====
root = tk.Tk()
root.title('Tkinterの電卓')

# ディスプレイ部
var = tk.StringVar()
entry = tk.Entry(root, textvariable=var, justify='right',
                 font=('TkDefaultFont', 16), bd=5, relief='sunken')
entry.grid(row=0, column=0, columnspan=4, sticky='nsew', padx=5, pady=5)
entry.focus()
# ボタン配列
layout = [  ['7','8','9','/'],
            ['4','5','6','*'],
            ['1','2','3','-'],
            ['0','.', 'C','+'] ]
# ボタンの登録
for r, row in enumerate(layout, start=1):
    for c, ch in enumerate(row):
        tk.Button(root, text=ch, command=lambda s=ch: on_click(s),
                  font=('TkDefaultFont', 14)).grid(row=r, column=c,
                  sticky='nsew', padx=2, pady=2)
# ボタンがクリックされた際の処理の登録
tk.Button(root, text='=', command=lambda: on_click('='),
          font=('TkDefaultFont', 14)).grid(row=5, column=0, columnspan=4,
          sticky='nsew', padx=2, pady=2)

# リサイズに追従
```

```
45 for i in range(6):  root.grid_rowconfigure(i, weight=1)
46 for j in range(4):  root.grid_columnconfigure(j, weight=1)
47
48 # キーバインド（Enter=計算、Esc=クリア）
49 root.bind('<Return>', lambda e: on_click('='))
50 root.bind('<Escape>', lambda e: on_click('C'))
51
52 root.mainloop()     # イベントループの起動
```

これを通常の形で実行すると図 127 のような電卓アプリが表示される．

例. Windows 環境での calcTkinter01.py の実行

```
C:\Users\katsu\TeX\Python>py calcTkinter01.py
```

図 127: 電卓アプリ

このスクリプトを先に解説した方法でビルドして実行すると，図 127 のウィンドウが表示され，電卓アプリとして使用できるが，同時にコンソールのウィンドウ（図 128）も表示される．

図 128: GUI とは別に表示されるコンソールウィンドウ

GUI アプリケーションとしてビルドして使用する場合は，コンソールウィンドウの表示は不要である．ビルド処理の際に '--noconsole' オプションを与えると，出来上がったアプリケーションの実行時にコンソールウィンドウの表示が無くなる．

6　対話作業環境（JupyterLab）

Python を利用するには，プログラムを記述・編集するためのテキストエディタと，処理を実行するためのコマンドツール[108] が必要となる．これらツールの機能を統合して，プログラムの開発と実行の操作性を高めるための**対話作業環境**として JupyterLab がある．

6.1　JupyterLab と Python インタプリタ

Python 言語処理系（インタプリタ）は，OS のターミナル環境で起動して REPL 環境として使用することができるが，UI としての機能は豊富であるとは言えない．そこで，Python の REPL を強化する IPython が 2001 年に開発され，入力の補完や履歴管理の機能，独自の**マジックコマンド**などを備えた高度な対話シェルとなった．その後，コードの作成と実行，出力処理，文書の処理を一体化して扱う環境を目指して，Web ブラウザ上で操作できる IPython Notebook が 2011 年に開発された．

次に，Notebook から Python コードをネットワーク越しに安全に実行するため，ZeroMQ ベースのメッセージ通信モデル（Jupyter Messaging Protocol の原型）が導入され，これを実装するサーバ側コンポーネントとして IPython Kernel が開発された．その後，Notebook と Kernel を多言語対応の汎用基盤として再設計するプロジェクトが始まり，2014 年に Project Jupyter として独立した．これにより Notebook（UI）と Kernel（Python 実行エンジン）は完全に分離され，IPython Kernel は Jupyter の Python 実装として現在利用されている．

導入方法をはじめとする JupyterLab に関する詳しい情報は，Jupyter の公式 Web サイト[109]から得られる．

6.2　基礎事項

JupyterLab は **Web アプリケーション**として実装されている．Web アプリケーションは，

1. Web ブラウザ　ユーザと対話するフロントエンド　（クライアント）
2. Web サーバ　処理を実行するバックエンド　（サーバ）

という 2 つのシステム要素から成るもの（クライアント・サーバモデル）であり，JupyterLab も実行時にはクライアントの部分とサーバの部分が稼働する．具体的には，コマンドラインから JupyterLab を起動するとサーバ部分が稼働をはじめ，直後にフロントエンドとなる Web コンテンツが Web ブラウザ上に表示されるという流れとなる．

図 129: サーバとクライアントの連携 (通常の場合)

一般的な理解としては，クライアントシステムとサーバシステムは，別の場所に設置された別々のコンピュータで稼働するもの（図 129）[110] として考えられることが多いが，JupyterLab の使用においては，それぞれが同一のコンピュータ内で実行されるケース（図 130）となる．

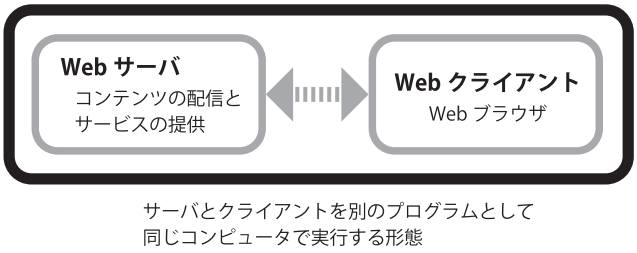


図 130: 同一のコンピュータ内でサーバとクライアントが連携するケース

[108] Windows の場合は「コマンドプロンプトウィンドウ」，Mac の場合は「ターミナル」などが代表的なコマンドツールである．

[109] https://jupyter.org/

[110] 通常の場合 Web サービスは，コンテンツを配信しているサーバシステム（管理団体が運営する Web サーバ）と，それを閲覧する Web ブラウザ（手元の端末装置）の連携によって実現されている．

6.2.1　起動と終了

JupyterLab を起動するには，コマンドシェル（コマンドのウィンドウ）で次のようなコマンドを発行する．

`jupyter lab` Enter

このコマンド操作で起動できない場合は，次のようなコマンド操作を試みること．

`py -m jupyter lab` Enter　　Windows 用 PSF 版 Python の場合
`python -m jupyter lab` Enter　　macOS，Linux の場合

これに続いて，次のような JupyterLab の起動メッセージが表示される．

```
[I 2025-10-11 14:42:46.050 ServerApp] jupyter_lsp  extension was successfully linked.|
[I 2025-10-11 14:42:46.060 ServerApp] jupyter_server_terminals  extension was successfully linked.|
[I 2025-10-11 14:42:46.060 ServerApp] jupyterlab  extension was successfully linked.|
        :
    （途中省略）
        :
[I 2025-10-11 14:42:48.867 LabApp] ‘sys_prefix‘ level settings are read-only, using ‘user‘ level
[I 2025-10-11 14:42:48.876 LabApp] Build is up to date    for migration to ‘lockedExtensions‘|
```

これで JupyterLab の Web サーバ部が起動し，更にこの後 Web ブラウザが起動して JupyterLab を使用するためのコンテンツが表示（クライアント部が実行）される．JupyterLab のサーバ部が起動している状態であれば，当該サーバプロセスの URL にアクセスすることで JupyterLab を使用することができる．例えば，起動メッセージの例の中に

`http://localhost:8888/lab?token=9ccaea5c819639a030bbd0e3d843aba23516d02f0ece33ab`

のような部分（毎回異なる）があれば，それがこの例におけるサーバプロセスの URL [111] である．

JupyterLab の使用を終了するには，Web ブラウザを終了するだけでなく，サーバプロセスを起動したコマンドウィンドウで CTRL + C を数回押下する必要がある．（この操作でサーバプロセスが終了する）

6.2.2　表示領域の構成と操作方法

Web ブラウザに表示された JupyterLab のウィンドウの例を図 131 に示す．

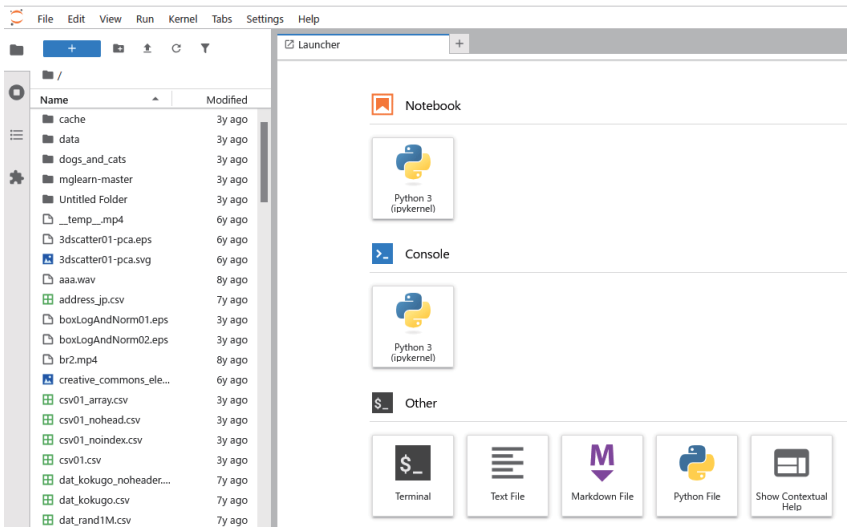


図 131: Web ブラウザで表示した JupyterLab のウィンドウ

JupyterLab のウィンドウは大きく分けて次のような 3 つの領域から成る．

● **メニューバー（Menu Bar）**

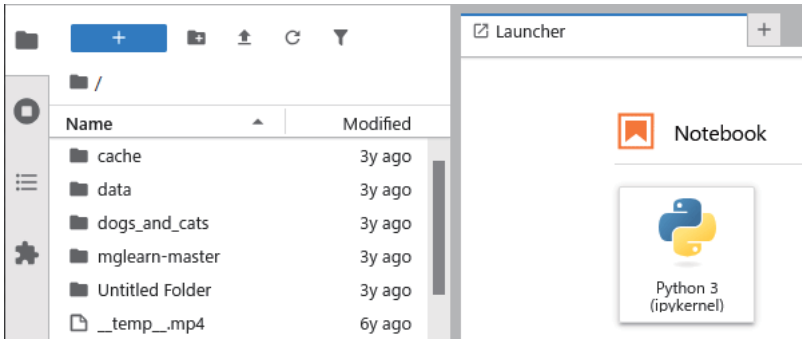
「File」,「Edit」,「View」,「Run」… とメニューが並んでいる横長の領域（図 132）．

図 132: メニューバー（Menu Bar）

[111] ローカルホストでサーバを稼働しているので，URL のホストアドレスの部分が `localhost` となっている．

● **左サイドバー（Left Sidebar）**
　各種の管理機能の領域.（図 133 (a)）

● **主作業域（Main Work Area）**
　主たる作業領域.（図 133 (b)）



(a) 左サイドバー（Left Sidebar）　　(b) 主作業域（Main Work Area）

図 133: サイドバーと作業域

この内，**主作業域**で主だった作業を行う.

JupyterLab を起動すると，主作業域には**ランチャー**（Launcher）が開かれ，そこから**ノートブック**（Notebook），**コンソール**（Console），**ターミナル**（Terminal），**テキストエディタ**（Text Editor）などを新規に作成できる．ノートブックは特に重要で，Python との対話をする機能であり，利用者と処理系のやりとり（対話過程）を**ノートブック**のデータファイルとして保存することができる．このノートブックは，再度開いて処理の再現や継続を可能とするものである．コンソールもノートブックと同様に，利用者と Python が対話するものであるが，より単純な Python のコマンドウィンドウであり，処理過程の保存といった管理機能は無い.

Terminal は OS のコマンドシェル[112] の機能であり，OS 上のコマンド作業を可能にする．また Text Editor はその名の通りテキストエディタであり，Python のプログラムを編集[113] することができる.

6.2.2.1　ノートブックの使用例

JupyterLab 起動時の状態では，主作業域（Main Work Area）には**ランチャー**（Launcher）のタブが表示されている．この中から「Notebook」のセクションにある「Python 3」のボタンをクリックすると図 134 のようなノートブックが表示される.

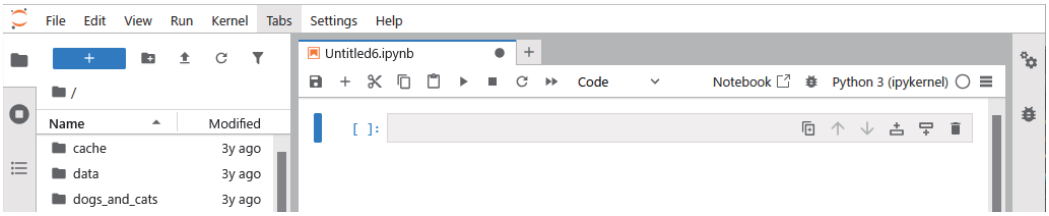


図 134: ノートブックの開始

この例では，主作業域に Untitled.ipynb というタイトルを持つノートブックが表示されている．これは，Untitled.ipynb というノートブックが作成されたことを意味しており，左サイドバーに同じ名前のファイルが作成されたことが表示されている．ノートブックには

```
[ ]:
```

と表示されており，これの右側に Python の文や式を入力することができる.

[112] macOS や Linux といった UNIX 系 OS ではコマンドシェルとして bash（sh） が起動する．Windows の場合は PowerShell などが起動する．ただし，実際の利用においてはどのようなコマンドシェルが起動するかを確かめること.

[113] コード体系や Tab の扱いに関する設定に注意すること.

例. 1 + 2 の計算

```
[1]: 1 + 2
[1]:   3
```

文や式を入力して Shift + Enter を押下すると，入力したものが実行される．
（Enter のみでは単なる改行となり，実行されない）
処理の結果として値が返されれば例のように

[番号]:（結果の値）

と表示される．

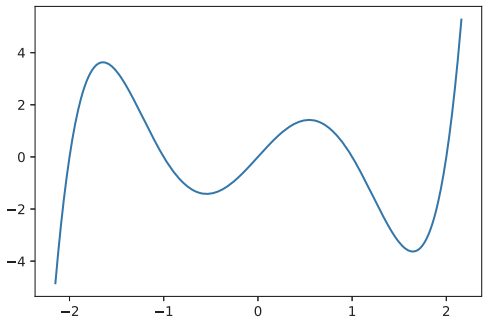
例. print 関数の実行

```
[2]: print('これは JupyterLab のテストです．')
     これは JupyterLab のテストです．
```

処理の結果が入力のすぐ下に表示されている．値は返されないので‘[番号]:’は表示されない．

例. matplotlib のグラフのインライン表示

```
[3]: import matplotlib.pyplot as plt
     import numpy as np

[4]: x = np.arange(-2.15,2.16,0.01)
     y = x*(x-2)*(x-1)*(x+1)*(x+2)

[5]: plt.plot(x,y)
[5]:  [<matplotlib.lines.Line2D at 0x2b552a70470>]
```



これは，numpy ライブラリで関数の値を算出した後，matplotlib で可視化した例である．この例の‘[3]’では必要なライブラリの読込みを行っている．‘[4]’では，numpy ライブラリを用いて関数 $y = x(x-2)(x-1)(x+1)(x+2)$ の定義域 x と値域 y を $-2.15 \leq x \leq 2.15$ の範囲で生成しており，‘[5]’では，matplotlib ライブラリを用いてそれを作図している．

ノートブックの入力セル‘[ ]:’の領域に入力する文や式の行数については利用者が自由に決めて良い．今回の例では，「パッケージの準備」や「値の生成」などを 1 つのまとまりとして入力しているが，文や式を個々別々に入力して実行しても問題はない．

ノートブックやコンソールに記録された‘[n]:’の領域は**セル**（Cell）と呼ばれ，再度の実行や評価が可能である．

6.2.2.2 カーネル（Kernel）について

ノートブックやコンソールでコードを実行すると，JupyterLab は Jupyter Server を介して Python カーネル（IPython kernel）に処理を送信し，その結果を受け取って表示する．すなわち，ノートブックやコンソールは，利用者との対話を保持するインターフェースであり，実際のコードの実行や評価は，Jupyter Server によって起動されたカーネルによって行われる．また，カーネルは Jupyter Server によって別プロセスとして起動され管理される．通常はノートブックごとに独立したカーネルが割り当てられるが，複数のノートブックやコンソールで同一カーネルを共有することも可能である．

ノートブックやコンソールにおける処理を請け負っているカーネルは，**停止**（Shutdown）や**再起動**（Restart）が

可能である．この性質を利用すると，ノートブックやコンソールにある全ての入力セル In[n]: を再度実行することができる．具体的には，メニューバーの「Kernel」メニューの中にある「Restart Kernel And Run All Cells...」を選択する．(「Run」メニュー配下にも様々な機能が提供されている)

6.2.3 Notebook での input 関数の実行

標準入力からの入力（ユーザからのキーボード入力）を input 関数で取得することができるが，Notebook 上で input() 関数を実行すると，ブラウザ上に簡易的な入力ダイアログ（1 行テキストボックス）が表示され，入力した文字列が標準入力としてプログラムに渡される．

Notebook 上で input 関数を実行して，取得した文字列を表示する例を次に示す．

例. Notebook 上での input 関数の実行

```
[*]: s = input('入力してください：')
     print('入力結果：'+s)
```

入力してください：

このように，実行したセルの直下に input 関数のプロンプトが表示され，そのすぐ右側に入力フィールドの UI が現れる．ここに文字列を入力することができる．(次の例を参照)

例. 入力フィールドへの入力（続き）

```
[*]: s = input('入力してください：')
     print('入力結果：'+s)
```

入力してください：入力した文字列

入力フィールドに“入力した文字列”と入力している．この直後に print 関数によってこれが表示される．(次の例を参照)

例. 先の例の続き

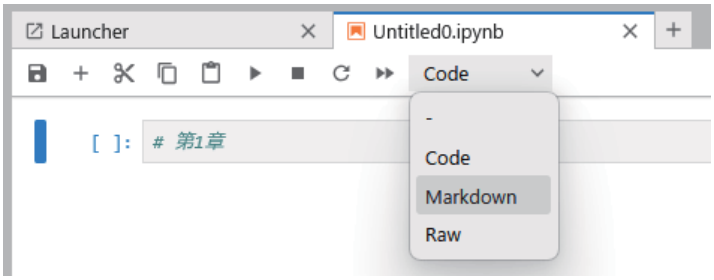
```
[6]: s = input('入力してください：')
     print('入力結果：'+s)
```

```
[6]: 入力してください： 入力した文字列
     入力結果：入力した文字列
```

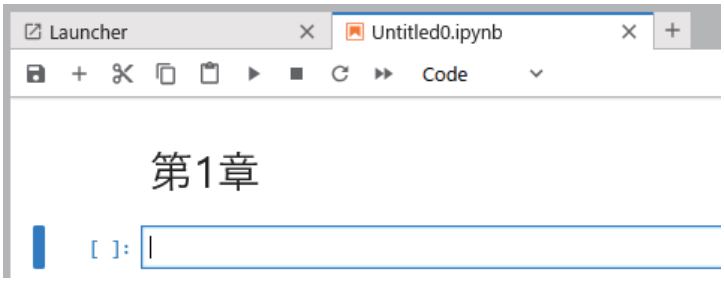
6.2.4 Markdown によるコメント表示

Jupyter Notebook では，セルの書式を次の例のように Markdown にすることで，そのセル自体を Python の実行対象としない「コメント」とすることができる．

例. 選択したセルを「Markdown」にする操作



書式を Markdown に設定したセルを評価すると「見出しセル」となる．(次の例)



Markdown のセル内では，先頭の「#」の個数によって階層的なレベルによる文書構造（章・節・項…）を表現できる．ノートブック内では，文書レベル毎に表示の縮小と展開ができる．

6.3 機能各論

6.3.1 matplotlib のための高度な表示機能

Jupyter Notebook のマジックコマンド **%matplotlib** を使用すると，matplotlib がグラフを作成する際の高度な機能が使える．

書き方： **%matplotlib widget**

これにより，ノートブック上に表示された matplotlib のグラフに対してマウスで様々な操作ができる．デフォルトでは **%matplotlib inline** であり，グラフに対するマウス操作はできない．

注意）JupyterLab でこの機能を使用するには ipympl モジュールを予めインストールしておく必要がある．

例. Windows 用 PSF 版 Python の場合

```
py -m pip install ipympl
```

このコマンドの使用例を以下に示す．

例. モジュールの読込みとプロット用データの作成

入力：

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-2*np.pi,2*np.pi,101)
y = np.sin(x)
```

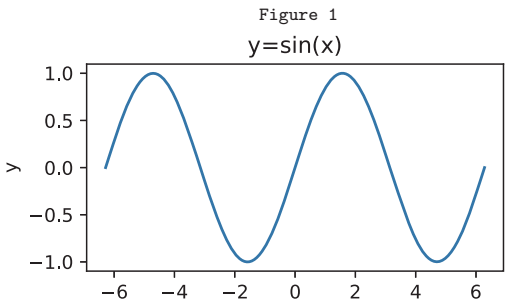
この処理により，NumPy，matplotlib が読み込まれ，x，y にプロット用のデータ（正弦関数）ができた．これをプロットしたものを Notebook 用の UI で表示する例を次に示す．

例. 上記データのグラフ表示（先の例の続き）

入力：

```
%matplotlib widget                                    ←マジックコマンド
plt.figure(figsize=(4,2))
plt.plot(x,y)
plt.xlabel('x'); plt.ylabel('y'); plt.title('y=sin(x)')
plt.show()
```

出力：



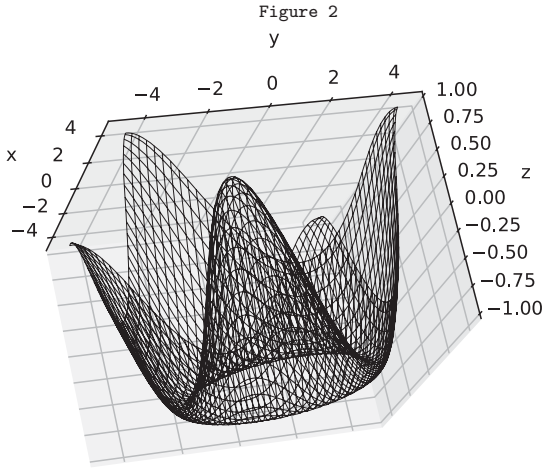
表示されたグラフの上にマウスポインタを重ねると ⌂，←，→，✥，□，💾 のようなボタンが図の端に表示さる．□ はグラフの指定領域を拡大する機能，✥ は表示部分の平行移動，💾 はグラフを画像データとして保存する機能であ

る．またこの機能を使って 3 次元グラフの回転などの操作も可能である．（次の例）

例. 3 次元グラフの回転操作（先の例の続き）

入力：
```
# meshgrid の作成
x = np.arange(-4.5, 4.5, 0.1)
y = np.arange(-4.5, 4.5, 0.1)
(X,Y) = np.meshgrid(x,y)
# 3 次元データ
Z = np.cos(np.sqrt(X**2+Y**2)) # 関数の算出
# 関数のプロット
%matplotlib widget
fig, ax = plt.subplots(subplot_kw='projection':'3d')
ax.plot_wireframe(X,Y,Z, lw=0.5, color='black')
ax.set_xlabel('x'); ax.set_ylabel('y'); ax.set_zlabel('z')
plt.show()
```

出力：



■ ローカル環境のノートブックでのグラフ表示

JupyterLab の環境がローカルの計算機環境で閉じている場合は，マジックコマンド

%matplotlib tk

で，グラフが独自ウィンドウ上で表示される．これは，matplotlib をローカル計算機のコマンドウィンドウ（ターミナル環境）で使用する場合と同じである．

6.3.2 MathJax による SymPy の式の整形表示

数式処理パッケージ SymPy（p.190「3.3 SymPy」参照のこと）は MathJax [114] の機能を利用して，Jupyter Notebook 上で数式を整形表示することができる．この機能を使用するには SymPy の init_printing 関数によって，当該機能の使用を設定しておく必要がある．具体的には

```
import sympy as sp
sp.init_printing(use_latex='mathjax')
```

とする．SymPy の式を整形表示する例を示す．

例. モジュールの読込みと，数式整形表示の設定

入力：
```
import sympy as sp
sp.init_printing(use_latex='mathjax')
```

[114] Web ブラウザ上で数式を整形表示するための JavaScript ライブラリ.（公式サイトは https://www.mathjax.org/）

例. 不定積分（先の例の続き）

入力：
```
sp.simplify( 'Integral(f(x),x)' )
```

出力：
$$\int f(x)\,dx$$

例. 行列（先の例の続き）

入力：
```
m = sp.simplify( 'Matrix([[a,b],[c,d]])' )
m
```

出力：
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

入力：
```
m.inv()     # 逆行列
```

出力：
$$\begin{bmatrix} \frac{d}{ad-bc} & -\frac{b}{ad-bc} \\ -\frac{c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}$$

例. 総和（先の例の続き）

入力：
```
sp.simplify( 'Sum(f(x),(x,0,n))' )
```

出力：
$$\sum_{x=0}^{n} f(x)$$

例. 極限（先の例の続き）

入力：
```
sp.simplify( 'Limit(1/f(x),x,oo)' )
```

出力：
$$\lim_{x\to\infty} \frac{1}{f(x)}$$

例. 導関数（先の例の続き）

入力：
```
sp.simplify( '(f(x)*g(x)).diff(x)' )
```

出力：
$$f(x)\frac{d}{dx}g(x) + g(x)\frac{d}{dx}f(x)$$

6.3.3 IPython.display モジュールによるサウンドの再生

Jupyter Notebook では WAV，MP3，AAC（M4A コンテナ）といったサウンドデータを読み込んで再生することができる．ここでは特に

- WAV 形式サウンドファイルの読込みと再生
- NumPy を用いて生成した波形データ配列の再生

に関する基本的な機能を紹介する．

■ モジュールの読込み

NumPy ライブラリと IPython.display モジュールの Audio クラスを読み込む例を次に示す．

例. モジュールの読込み

入力：
```
import numpy as np
from IPython.display import Audio
```

次に，WAV 形式サウンドのファイル 'aaa.wav' を読み込む例を示す．

例. aaa.wav の読込み（先の例の続き）

入力：
```
a1 = Audio('aaa.wav')
print(' データタイプ :',type(a1))
```

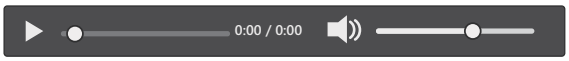
出力： データタイプ : <class 'IPython.lib.display.Audio'>

WAV 形式ファイルを読み込んで，それを Audio クラスのオブジェクト a1 にしている．このオブジェクトの再生作業の例を次に示す．

例. Audio オブジェクト a1（先の例の続き）

入力：
```
a1
```

出力：



Audio オブジェクトの値を出力した時にこのようなインターフェースが表示される．▶ の部分をクリックするとサウンドが再生される．

ノートブック上でサウンド波形を合成する例を次に示す．

例. NumPy の機能を用いてサウンドを合成する例（先の例の続き）

入力：
```
r = 44100                    # サンプリング周波数
t = np.arange(0,3,1/r)       # 時間軸 0〜3 秒，1/r 刻み
f = 440                      # 音の周波数
s = np.sin( 2*np.pi*f*t ) # 波形データ：sin(2 π ft)
```

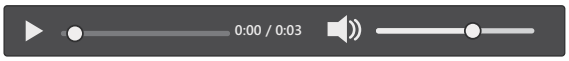
これは 440Hz の正弦波形 $\sin(440 \times 2\pi \times t)$ を 3 秒の長さで生成した例である．サウンドをデータとして扱うには，サンプリング周波数，量子化ビット数，チャンネル数の 3 つの意識する必要がある．サンプリング周波数は `r = 44100` としているが，チャンネル数に関してはデータ配列の行数から自動的に決まる[115]．また，量子化ビット数も，NumPy 配列の型を元に Audio オブジェクト生成時に決定[116] される．

この例で生成された波形の配列オブジェクト s を再生する例を次に示す．

例. 生成した正弦波形の再生（先の例の続き）

入力：
```
a2 = Audio( s, rate=r )
a2
```

出力：



この例でも ▶ で再生できる．

[115] モノラルの場合は単純な一次元配列，ステレオの場合は左右 2 列の 2 次元配列（行方向が時間）の形式で与える．

[116] 自動的にスケーリングされることもある．

7 表示制御

7.1 curses

OS のコマンド操作を行うターミナルウィンドウなどの**キャラクタ端末**では，キーボードによる文字の入力と，ディスプレイへの文字の出力を基本とする CUI [117]（Character User Interface）である．この形式のユーザインターフェースでは，文字は左側から右側にかけて順に表示され，改行処理によって次の（下の）行の左端に表示位置が移動し，キーボードからの入力位置もこの順序に従う．標準入出力も基本的にこの形式に従っており，通常の CUI では，ターミナルウィンドウ上の指定した任意の位置に文字を出力することはできない．

UNIX 系 OS [118] では，キャラクタ端末における入出力のための curses ライブラリが C 言語のために用意されている．またこのライブラリを応用した，Python 言語上で使える同名のライブラリが標準ライブラリとして提供されている．curses ライブラリを使用することで，通常の CLI の制約を受けない画面入出力が実現できる．例えば次の例 curses01.py の実行の様子について考える．

プログラム：curses01.py

```python
# coding: utf-8
import curses
import time

stdscr = curses.initscr()   # 初期化処理：画面を指すwindowsオブジェクトが得られる.
curses.curs_set(0)          # カーソル表示を無効化
stdscr.clear()              # 画面消去

for y in range(9,-1,-1):
    for x in range(29,-1,-1):
        stdscr.addch(y, x, 'X')     # 指定された位置（y行x列）に文字を表示
        stdscr.refresh()            # 画面を更新
        time.sleep(0.02)            # 少し待つ

curses.curs_set(1)          # カーソル表示を有効化

stdscr.addstr(11,0,'何かキーを押してください.')    # 終了待機メッセージ
stdscr.getch()      # ユーザーの入力を待つ
curses.endwin()     # 終了処理
```

このプログラムをターミナルウィンドウで実行すると，画面の表示が一旦消去され，右のような表示が刻々と出来上がる．その際，画面の右下から左上にかけて 'X' が表示されることが確認できる．

最後に「何かキーを押してください.」と表示され，何かキーを押すとプログラムが終了し，ターミナルウィンドウは，プログラム開始前の状態に戻る．

（プログラムの詳細は後に解説する）

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

何かキーを押してください.
```

curses01.py を実行したところ

このプログラムの実行で確認できるように，curses の機能によって，指定した任意の画面位置に文字を表示することができる．また，プログラム実行中の画面状態は通常のターミナルウィンドウとは別の状態（**curses モード**）であり，プログラム終了後は開始前の画面の状態が復元される．

注意）

Windows 環境の Python には curses が標準添付されていないが，サードパーティ[119] が，ほぼ同じ機能のライブラリを windows-curses と名付けて開発して公開しており，それを pip などで Python 処理系にインストールすると，他のプラットフォームと同様の形で curses を import して利用することができる．

[117]**CLI**（Command Line Interface）とも呼ばれる．

[118]Linux や Apple 社の macOS などが代表的.

[119]https://pypi.org/project/windows-curses/

Python の curses ライブラリに関して，本書では最も基本的な使用方法について解説する．詳細に関しては，Python の公式インターネットサイトや curses の技術資料などを参照のこと．

7.1.1 curses 使用の流れ

curses による画面制御（curses モード）は初期化関数 initscr の実行から始まる．

書き方： initscr()

この関数は window クラスのオブジェクトを返す．以後そのオブジェクトを用いて画面に対する入出力を行う．

curses モードを終了するには endwin 関数を実行する．

書き方： endwin()

この関数を実行すると，画面は通常の状態に戻る．

wrapper 関数を用いると，curses モードの初期化と終了の処理を自動化できる．

書き方： wrapper(コールバック関数)

initscr 関数を自動的に実行して，得られた window オブジェクトを「コールバック関数」の引数に渡して実行する．「コールバック関数」が終了する際に endwin 関数を自動的に実行する．「コールバック関数」の戻り値は wrapper 関数の戻り値となる．

画面のサイズ（行，列）は，window オブジェクトに対して getmaxyx メソッドを実行することで取得できる．

書き方： window オブジェクト.getmaxyx()

「window オブジェクト」の画面のサイズを，**(行, 列)** の形式のタプルとして返す．

7.1.2 画面に対する出力，制御

画面上の指定した位置に 1 つの文字を表示するには，window オブジェクト. に対して addch メソッドを実行する．

書き方： window オブジェクト.addch(y, x, 文字)

画面の「y」行「x」列の位置に「文字」を表示する．この場合の「文字」は 1 バイトで表現される 1 文字（ASCII 文字など）であり，マルチバイト文字や文字列を表示する場合は次に示す addstr メソッドを使用する．

書き方： window オブジェクト.addstr(y, x, 文字列)

addch，addstr には 4 番目の引数として色の設定を与えることができる．これに関しては後に解説する．

注意）curses はマルチバイト文字には十分に対応できていないので可能な限り ASCII 文字を使用すること．

注意）画面の最も右下の位置に文字を描画しようとするとエラー（curses.error）が発生する．（試されたい）

7.1.2.1 色の設定と適用

curses では**色ペア**という概念で文字に着色する．これは前景色と背景色を組にした情報であり，色ペアは番号を与えて管理する．また curses では色の管理と適用をするにあたり，start_color 関数で初期化しておく必要[120] がある．

書き方： start_color()

この後，init_pair 関数によって，色ペアを設定する．

書き方： init_pair(番号, 前景色, 背景色)

「番号」を付けて「前景色」と「背景色」の組を登録する．「番号」は 1〜255 の整数である．「前景色」,「背景色」に指定する値は curses ライブラリに定数（表 45）として定義されている．

登録した色ペアは color_pair 関数で呼び出す．

書き方： color_pair(番号)

登録された「番号」の色ペアの情報を取り出し，それが意味する整数値を返す．この値を addch，addstr 関数の第 4 引数に与えることで表示する文字に着色することができる．

[120] 端末環境によってはこの処理が必要ない場合もある．

表 45: curses の色の定数（一部）

色	定　義	値	色	定　義	値
黒	COLOR_BLACK	0	白	COLOR_WHITE	7
赤	COLOR_RED	4	シアン	COLOR_CYAN	3
緑	COLOR_GREEN	2	マゼンタ	COLOR_MAGENTA	5
青	COLOR_BLUE	1	黄	COLOR_YELLOW	6

画面全体の基本的な前景色と背景色を設定するには window オブジェクトに対して bkgd メソッドを実行する．

書き方：　window オブジェクト.bkgd(背景文字, color_pair(番号))

画面の背景を「背景文字」で満たすが，必要ない場合は半角の空白文字（ASCII コード 32）を 1 つを与える．

7.1.2.2　装飾の適用

表 46 に curses における装飾の定義を示す．

表 46: curses の文字装飾の定数（一部）

定　義	装　飾	定　義	装　飾
A_BOLD	太字	A_UNDERLINE	下線
A_REVERSE	反転表示	A_BLINK	点滅
A_DIM	薄暗い表示	A_STANDOUT	強調表示

表 46 の定義と color_pair 関数の戻り値とのビット論理和を取ったものを addch，addstr 関数の第 4 引数に与えることで，表示する文字に着色と装飾を施すことができる．

7.1.2.3　テキストカーソルの設定

文字入力の際に入力位置を示す**テキストカーソル**[121] の状態を設定するには curs_set 関数を用いる．

書き方：　curs_set(状態)

「状態」は整数値（下記）で与える．

0: 非表示，　　1: 通常の（標準的な）カーソル，　　2: 強く目立つカーソル

7.1.2.4　画面の消去，更新

window オブジェクトに対して clear メソッドを実行することで，画面の表示内容を消去できる．

書き方：　window オブジェクト.clear()

window オブジェクトに対して refresh メソッドを実行することで，画面の表示内容を更新できる．

書き方：　window オブジェクト.refresh()

この処理は，画面の初期化や，ユーザによる入力処理の後に実行すると良い．また，addch，addstr を多数実行した後に実行すると良い．このメソッドは画面更新の必要に応じて実行する．

7.1.2.5　警告表現

flash 関数で画面を瞬かせることができる．

書き方：　flash()

beep 関数で警告音を出すことができる．

書き方：　beep()

文字に着色と装飾を施して表示し，終了時に警告表現を出す例を curses02.py に示す．

プログラム：curses02.py

```
1 # coding: utf-8
2 import curses
3
```

[121] **キャレット**ともいう．

```
4  # コールバック関数
5  def cursesMode( stdscr ):
6      H,W = stdscr.getmaxyx()      # スクリーンサイズの取得
7      curses.curs_set(0)           # カーソル表示を無効化
8      curses.start_color()         # 色制御機能を初期化
9
10     # 色ペアの設定  番号     前景色                    背景色
11     curses.init_pair( 1, curses.COLOR_BLACK,    curses.COLOR_WHITE )
12     curses.init_pair( 2, curses.COLOR_RED,      curses.COLOR_WHITE )
13     curses.init_pair( 3, curses.COLOR_GREEN,    curses.COLOR_WHITE )
14     curses.init_pair( 4, curses.COLOR_BLUE,     curses.COLOR_WHITE )
15     curses.init_pair( 5, curses.COLOR_WHITE,    curses.COLOR_BLACK )
16     curses.init_pair( 6, curses.COLOR_CYAN,     curses.COLOR_WHITE )
17     curses.init_pair( 7, curses.COLOR_MAGENTA,  curses.COLOR_WHITE )
18     curses.init_pair( 8, curses.COLOR_YELLOW,   curses.COLOR_WHITE )
19
20     stdscr.clear()       # 画面消去
21     stdscr.refresh()     # 画面更新（念の為）
22
23     # 文字列描画（着色）
24     stdscr.addstr( 1,  1, f'ScreenHeight{H}, ScreenWidth{W}' )
25     stdscr.addstr( 3,  1, 'BLACK  ', curses.color_pair(1) )
26     stdscr.addstr( 3,  9, 'RED    ', curses.color_pair(2) )
27     stdscr.addstr( 3, 17, 'GREEN  ', curses.color_pair(3) )
28     stdscr.addstr( 3, 25, 'BLUE   ', curses.color_pair(4) )
29     stdscr.addstr( 5,  1, 'WHITE  ', curses.color_pair(5) )
30     stdscr.addstr( 5,  9, 'CYAN   ', curses.color_pair(6) )
31     stdscr.addstr( 5, 17, 'MAGENTA', curses.color_pair(7) )
32     stdscr.addstr( 5, 25, 'YELLOW ', curses.color_pair(8) )
33     # 文字列描画（装飾）
34     stdscr.addstr( 7,  1, 'BOLD     ', curses.A_BOLD )
35     stdscr.addstr( 7, 11, 'UNDERLINE', curses.A_UNDERLINE )
36     stdscr.addstr( 7, 21, 'REVERSE  ', curses.A_REVERSE )
37     stdscr.addstr( 9,  1, 'BLINK    ', curses.A_BLINK )
38     stdscr.addstr( 9, 11, 'DIM      ', curses.A_DIM )
39     stdscr.addstr( 9, 21, 'STANDOUT ', curses.A_STANDOUT )
40     stdscr.refresh()     # 画面更新（念の為）
41
42     # 終了処理
43     curses.curs_set(2)       # カーソル表示を強調
44     stdscr.addstr( 11, 1, 'Press any key to end:' )
45     stdscr.getch()           # キーの入力待ち
46     curses.flash()           # 画面を瞬かせる
47     curses.beep()            # ビープ音
48
49 curses.wrapper( cursesMode )    # コールバック関数を実行
```

このプログラムを実行した際のコンソールの表示を図 135 に例示する．

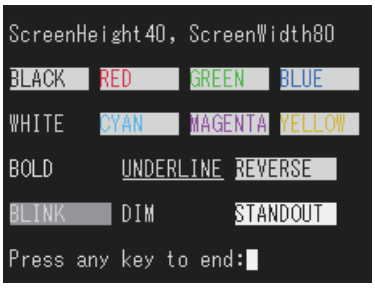


図 135: curses02.py を実行したところ

この後，何かキーを押すと，警告表現を出してプログラムが終了する．

注意）着色や装飾の効果，警告表現は，実行する環境によって異なることに注意すること．

7.1.3　キーボード入力

window オブジェクトに対して getch メソッドを実行することでキーボードの入力を受け付けることができる．

書き方：　window オブジェクト.getch()

キーが押されるまで待ち，押されたキーのコード（整数値）を返す．このメソッドの動作はデフォルトでは**同期入力**（ブロッキングモード）であるが，

window オブジェクト.nodelay(True)

を実行することで**非同期入力**[122]（ノンブロッキングモード）となる．非同期入力の状態で何もキーが押されていないとき，getch メソッドは -1 を返す．非同期入力の getch メソッドを同期入力に戻すには，nodelay メソッドの引数に False を渡して実行する．

getch メソッドが返すキーコードは，基本的にはその文字の ASCII コードであるが，カーソルキーやファンクションキーなどの制御用のキーが押された場合も様々な値を返す．それらの多くは curses の定数として定義されており，その一部を表 47 に示す．詳しくは Python の公式インターネットサイトなどを参照のこと．

表 47: curses のキーコードの定義（一部）

定　義	キー	値	定　義	キー	値
KEY_HOME	HOME	262	KEY_PPAGE	PageUp	339
KEY_END	END	358	KEY_NPAGE	PageDown	338
KEY_DOWN	↓	258	KEY_UP	↑	259
KEY_LEFT	←	260	KEY_RIGHT	→	261
KEY_DC	Delete	330	KEY_IC	Insert	331
KEY_Fn	ファンクションキー n．F1〜F12				

getch メソッドと同様の機能を持つ getkey メソッドもある．

書き方：　window オブジェクト.getkey()

キーの押下を受け付け，そのキーに対応する値を str 型で返す．

7.1.3.1　文字列の入力

window オブジェクトに対して getstr メソッドを実行することで，文字列の入力を受け付けることができる．このメソッドによる入力は Enter キーで終了する．

書き方：　window オブジェクト.getstr()

入力された文字列を bytes 型で返す．このメソッドには，入力中の編集機能はなく，BackSpace などの入力もそのままデータとして戻り値の bytes オブジェクトの中に含まれる．

注意）getstr はマルチバイト文字の入力に完全には対応していないので注意すること．

getstr メソッドの入力中は，押下したキーの文字は画面にはエコー（表示）されない．入力時のエコーを有効（エコーモード）にするには echo 関数を実行する．

書き方：　echo()

エコーモードを解除するには noecho 関数を実行する．

書き方：　noecho()

7.1.3.2　raw モード入力

raw 関数を実行すると，ウィンドウが **raw モード**になり，CTRL キーと他のキーの組み合わせなどによる制御が無効になる[123]．押下されたキーを純粋に受け取る際に raw モードが有用である．

書き方：　raw()

raw モードに移行する．この後で通常の状態に戻すには noraw 関数を実行する．

書き方：　noraw()

[122]キーが押されているかどうかに関わらず，getch メソッドは値を返して終了することを意味する．

[123]無効にならない制御もあるので注意すること．例えば，多くの環境で F11 の機能は無効にならない．

参考）raw モード以外にも様々な入力モードがある．詳しくは Python の公式インターネットサイトや，curses に関する技術資料を参照のこと．

7.1.4 ウィンドウのリサイズ

curses モードでプログラムを実行中，ウィンドウサイズが変えられると getch メソッドが KEY_RESIZE（整数値:546）を返す．getch メソッドを応用したサンプルプログラム curses03.py を示す．このプログラムは，カーソルキー（矢印キー）や HOME キーで画面に軌跡を描くものである．また，画面のリサイズにも対応している．

プログラム：curses03.py

```
# coding: utf-8
import curses, time

# コールバック関数
def cursesMode( stdscr ):
    curses.raw()                # rawモードに入る
    H,W = stdscr.getmaxyx()     # スクリーンサイズの取得
    curses.curs_set(0)          # カーソル表示を無効化
    curses.start_color()        # 色制御機能を初期化

    # 色ペアの設定  番号     前景色                  背景色
    curses.init_pair( 1, curses.COLOR_BLACK,    curses.COLOR_WHITE )
    curses.init_pair( 2, curses.COLOR_WHITE,    curses.COLOR_BLACK )

    stdscr.bkgd( ' ', curses.color_pair(1) )    # 画面の背景の設定
    stdscr.clear()      # 画面消去
    stdscr.refresh()    # 画面更新（念の為）

    y = H // 2;     x = W // 2      # 位置をホームに初期化
    stdscr.addch( y, x,' ',curses.color_pair(2))
    stdscr.addstr( H-1, 1, f'({y},{x}),Press [ESC] key to end  ' )
    while True:
        k = stdscr.getch()
        if k == 27:                 # ESC:終了
            break
        elif k == curses.KEY_HOME:  # 初期位置に戻る
            y = H // 2;     x = W // 2
        elif k == curses.KEY_DOWN:  # ↓
            y += 1
            if y >= H:
                curses.flash();     curses.beep()
                y = H - 1
        elif k == curses.KEY_UP:    # ↑
            y -= 1
            if y < 0:
                curses.flash();     curses.beep()
                y = 0
        elif k == curses.KEY_RIGHT: # →
            x += 1
            if x >= W:
                curses.flash();     curses.beep()
                x = W - 1
        elif k == curses.KEY_LEFT:  # ←
            x -= 1
            if x < 0:
                curses.flash();     curses.beep()
                x = 0
        elif k == curses.KEY_RESIZE:    # 画面リサイズ
            stdscr.clear()
            H,W = stdscr.getmaxyx()
            stdscr.addstr(1,1,f'Window size is changing.({k})')
            stdscr.addstr(2,1,f'Height:{H}, Width:{W}')
            stdscr.refresh()
            y = H // 2;     x = W // 2
        if y == H-1 and x == W-1:   # 描いてはいけない座標
            curses.flash();     curses.beep();
            stdscr.addstr( H-2, 1, f'That position is untouchable!.' )
            stdscr.refresh()
            time.sleep(2)
```

```
60              stdscr.clear();      stdscr.refresh()
61              y = H // 2;      x = W // 2
62          stdscr.addch( y, x,' ',curses.color_pair(2))
63          stdscr.addstr( H-1, 1, f'({y},{x}),Press [ESC] key to end  ' )
64
65      # 終了処理
66      curses.curs_set(1)  # カーソル表示
67      curses.noraw()      # rawモード解除
68
69  curses.wrapper( cursesMode )    # コールバック関数を実行
```

このプログラムを起動すると画面が白地になり，中央にスペース（黒）が表示され，それがカーソルキー ↑,↓,←,→ に合わせて移動して軌跡を描く．ウィンドウサイズを変更すると画面がクリアされ，画面左上に変更状況が表示（図136）される．

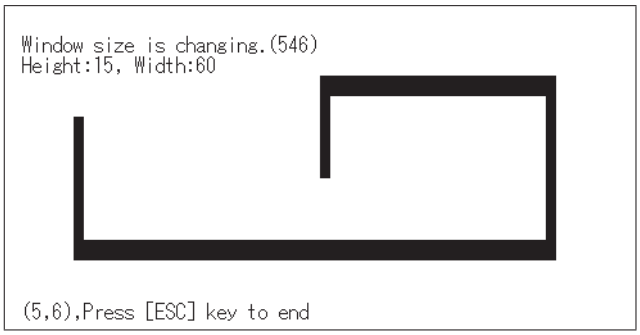


図 136: curses03.py を実行したところ

getstr メソッドで文字列を bytes 型のデータとして取得する例を curses04.py に示す．

プログラム：curses04.py

```
1   # coding: utf-8
2   import curses
3
4   # コールバック関数
5   def cursesMode( stdscr ):
6       curses.echo()       # エコーを有効化
7       stdscr.clear()      # 画面消去
8       stdscr.refresh()    # 画面更新（念の為）
9
10      while True:
11          stdscr.addstr(1,1,"Input String('exit' to end): ")
12          stdscr.clrtoeol()                       # 右側をクリア
13          s = stdscr.getstr().decode('utf-8')
14          if s == 'exit': # 終了
15              break
16          else:
17              stdscr.addstr(2,1,' ');     stdscr.clrtoeol()   # 行をクリア
18              stdscr.addstr(2,1,f"> '{s}'")
19              stdscr.refresh()
20
21      curses.noecho() # エコーを無効にする
22
23  curses.wrapper( cursesMode )    # コールバック関数を実行
```

このプログラム中にある clrtoeol メソッドは，現在位置からスクリーンの右端までの行内容をクリア（消去）するものである．

書き方：　window オブジェクト.clrtoeol()

curses04.py を実行すると，

```
Input String('exit' to end):
```

と表示され，文字列入力を受け付ける．これに対して

```
Input String('exit' to end):  Hello!
```
Enter

と入力すると，画面の表示が

```
Input String('exit' to end):
> 'Hello!'
```

となる．また，exit Enter と入力するとプログラムが終了する．

7.1.5 画面の内容の採取

画面上の指定した位置の 1 文字分の情報を取得するには，window オブジェクトに対して inch メソッドを実行する．

書き方： window オブジェクト.inch(y, x)

「y」行「x」列の位置の 1 文字分の情報を，文字コード，色，装飾などを含んだ**属性付き文字データ**として返す．これは 32 ビットの整数値であり，下位 8 ビットが文字コード，それより上位が，色や装飾などを表す**属性データ**である．

7.1.5.1 文字コードや属性情報の採取

属性付き文字データ（inch メソッドの戻り値）から文字コードのみを取得するには

属性付き文字データ & 0xFF

を算出し，属性データのみを取得するには

属性付き文字データ & ~0xFF

を算出する．

上の方法で得られた属性データの中の，色ペア番号の部分を意味するビットマスクとして A_COLOR が定義されている．従って，これと属性データの論理積を取った後，右に 24 ビットシフトすると色ペアの番号が得られる．

書き方： (属性データ & curses.A_COLOR) >> 24

また，強調，下線，反転の属性部分を意味するビットマスクが A_BOLD，A_UNDERLINE，A_REVERSE として定義されており，次のような記述によって，各種装飾の状態を真理値（True/False）の形で得ることができる．

強調の状態： bool(属性データ & curses.A_BOLD)
下線の状態： bool(属性データ & curses.A_UNDERLINE)
反転の状態： bool(属性データ & curses.A_REVERSE)

画面上の指定した位置の**属性付き文字データ**を取得して，その内訳を表示するサンプルプログラムを curses05.py に示す．

プログラム：curses05.py

```
 1 # coding: utf-8
 2 import curses
 3
 4 # コールバック関数
 5 def cursesMode( stdscr ):
 6     curses.curs_set(0)       # カーソル表示を無効化
 7     stdscr.clear()           # 画面消去
 8     stdscr.refresh()         # 画面更新（念の為）
 9     curses.start_color()     # 色制御機能を初期化
10
11     # 色ペアの設定  番号     前景色                    背景色
12     curses.init_pair( 1, curses.COLOR_RED,       curses.COLOR_BLACK )
13     curses.init_pair( 2, curses.COLOR_GREEN,     curses.COLOR_BLACK )
14     curses.init_pair( 3, curses.COLOR_BLUE,      curses.COLOR_BLACK )
15
16     # 目盛り表示
17     stdscr.addstr(0,0,'0----+----1');        stdscr.addstr(1,0,'1')
18
19     # 色，装飾付き文字出力
20     stdscr.addch(1,2,'N')
21     stdscr.addch(1,3,'R',curses.color_pair(1))
22     stdscr.addch(1,4,'G',curses.color_pair(2))
23     stdscr.addch(1,5,'B',curses.color_pair(3))
24     stdscr.addch(1,7,'R',curses.color_pair(1) | curses.A_BOLD)
25     stdscr.addch(1,8,'G',curses.color_pair(2) | curses.A_UNDERLINE)
26     stdscr.addch(1,9,'B',curses.color_pair(3) | curses.A_STANDOUT)
27
```

```
28      while True:
29          k = stdscr.getch()  # 採取位置の入力
30          if k == 27: break   # ESCなら終了
31          if ord('1') < k < ord(':'):     # '2'～'9'の範囲の場合
32              stdscr.addch(0,12,' ');     stdscr.clrtoeol()   # 0行目クリア
33              stdscr.addch(1,12,' ');     stdscr.clrtoeol()   # 1行目クリア
34              stdscr.addch(2,0,' ');      stdscr.clrtoeol()   # 2行目クリア
35              dat = stdscr.inch(1,k-48)   # その画面位置の1文字分のデータを採取
36              ch = chr(dat & 0xFF)        # 文字そのものを取得
37              attr = dat & ~0xFF          # 属性を取得
38              cp = (attr & curses.A_COLOR) >> 24          # 色ペア番号を取得
39              is_bold = attr & curses.A_BOLD              # 強調の状態を取得
40              is_underline = attr & curses.A_UNDERLINE    # 下線の状態を取得
41              is_reverse = attr & curses.A_REVERSE        # 反転の状態を取得
42              # 採取情報の表示
43              stdscr.addstr(0,12,f'row:{chr(k)}')         # 採取位置（横位置）
44              stdscr.addstr(1,12,f'char:{ch}')            # 文字
45              stdscr.addstr(0,19,'color_pair')
46              stdscr.addstr(1,19,f'{cp:-10d}')            # 色ペア番号
47              stdscr.addstr(0,30,'BOLD')
48              stdscr.addstr(1,30,f'{bool(is_bold)}')      # 強調の状態
49              stdscr.addstr(0,36,'UNDERLINE')
50              stdscr.addstr(1,36,f'{bool(is_underline)}') # 下線の状態
51              stdscr.addstr(0,46,'REVERSE')
52              stdscr.addstr(1,46,f'{bool(is_reverse)}')   # 反転の状態
53              stdscr.addch(2,k-48,'^')                    # ポインタ表示
54
55      curses.curs_set(1)      # カーソル表示
56
57  curses.wrapper( cursesMode )    # コールバック関数を実行
```

このプログラムの起動直後は 

と表示される．この画面の中の「NRGB RGB」の各文字の位置の情報を取得するには，横（カラム）位置の値 2～9 のキーを押す．実行例を次に示す.（試されたい）

curses05.py の実行例

キー	表　　示
2	0----+----1 row:2 color_pair BOLD UNDERLINE REVERSE 1 NRGB RGB char:N 0 False False False
3	0----+----1 row:3 color_pair BOLD UNDERLINE REVERSE 1 NRGB RGB char:R 1 False False False
4	0----+----1 row:4 color_pair BOLD UNDERLINE REVERSE 1 NRGB RGB char:G 2 False False False
5	0----+----1 row:5 color_pair BOLD UNDERLINE REVERSE 1 NRGB RGB char:B 3 False False False
7	0----+----1 row:7 color_pair BOLD UNDERLINE REVERSE 1 NRGB RGB char:R 1 True False False
8	0----+----1 row:8 color_pair BOLD UNDERLINE REVERSE 1 NRGB RGB char:G 2 False True False
9	0----+----1 row:9 color_pair BOLD UNDERLINE REVERSE 1 NRGB RGB char:B 3 True False True

7.1.5.2　画面の中の文字列の採取

window オブジェクトに対して instr メソッドを実行することで，画面の中の指定した位置から右側にある文字列を取得することができる.

書き方：　window オブジェクト.instr(y, x)

「y」行「x」列の位置から画面右端までの内容を bytes 型で返す．取得する長さを 3 つ目の引数として与えることもできる.

instr メソッドを使用するサンプルプログラムを curses06.py に示す.

プログラム：curses06.py

```
# coding: utf-8
import curses

# コールバック関数
def cursesMode( stdscr ):
    curses.echo()           # エコーモード
    stdscr.clear()          # 画面消去
    stdscr.refresh()        # 画面更新（念の為）

    # 目盛りと文字列の表示
    stdscr.addstr(0,0,'0----+----1----+----2----+----3----+----4');
    stdscr.addstr(1,0,'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNO')

    while True:
        stdscr.addstr(3,0,'Input position to pick (exit to end): ')
        stdscr.clrtoeol()
        s = stdscr.getstr().decode('utf-8')
        if s == 'exit': break   # 終了
        s2 = s.split(',')
        p1 = int( s2[0] )
        lngth = None
        if len(s2) > 1:
            lngth = int( s2[1] )
        if 0 <= p1 <= 40:
            if lngth:
                b = stdscr.instr(1,p1,lngth)    # 画面上の文字列の採取
            else:
                b = stdscr.instr(1,p1)          # 画面上の文字列の採取
            stdscr.addch(4,0,' ');      stdscr.clrtoeol()
            stdscr.addstr(4,0,f"> '{b.decode('utf-8').rstrip()}'")

    curses.noecho()         # エコーモード解除

curses.wrapper( cursesMode )    # コールバック関数を実行
```

このプログラムを実行すると，画面に

```
0----+----1----+----2----+----3----+----4
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNO

Input position to pick (exit to end):
```

と表示される．これに対して例えば 26 Enter と入力すると，アルファベットの 26 カラム目以降が採取され，次の行に

```
> 'ABCDEFGHIJKLMNO'
```

と表示される．位置の入力の直後にコンマ「,」で区切って長さを与えることもできる．例えば 26,4 Enter と入力すると，次の行に

```
> 'ABCD'
```

と表示される．

7.2 進捗表示： tqdm

Python 処理系で対話的なプログラミングを行う際，時間のかかる処理を実行している間は処理の進捗を表示することが望ましい．進捗表示のためのライブラリとして tqdm がある．これは，Python 処理系の標準ライブラリではなく，サードパーティ（tqdm developers）が開発して公開している[124] ライブラリである．tqdm はコマンド対話のためのターミナルウィンドウの環境で進捗表示を実現するだけでなく，Web ブラウザ上で Jupyter Notebook [125] を使用する際の進捗表示機能も提供する．このライブラリを使用するには，pip や conda などの管理ツールであらかじめ Python プログラミング環境にインストールしておく必要がある．

pip によるインストール作業の例： `pip install tqdm`
conda によるインストール作業の例： `conda install -c conda-forge tqdm`

注意）Anaconda ディストリビューションの場合は tqdm は予めインストールされている．また pip は，使用する環境によっては「`python -m pip`」，あるいは「`py -m pip`」とする場合がある．

7.2.1 基本的な使用方法

tqdm ライブラリの tqdm クラスを使用して進捗表示を行うので，これを次のようにして読み込むと良い．

```
from tqdm import tqdm
```

7.2.1.1 for 文による反復処理での使用方法

最も簡単な事例を挙げて tqdm の使用方法を解説する．次の例は平方数 $0, 1, 4, 9, \cdots$ を 0.3 秒間隔で 20 個作成してリストにするものである．具体的には，反復処理用のイテラブルを tqdm インスタンスにする．

書き方：　tqdm(反復処理用イテラブル)

例. 20 回の反復ループ

```
>>> import time   Enter
>>> from tqdm import tqdm   Enter      ← tqdm クラスの読込み
>>> r = []   Enter      ←平方数を格納するリストを初期化
>>> for x in tqdm(range(20)):   Enter      ←反復処理（ここで tqdm を使う）
...       r.append(x**2)   Enter      ←平方数の作成と格納
...       time.sleep(0.3)   Enter      ← 0.3 秒待機
...   Enter        ←反復処理の記述の終了
```

この直後に for 文による反復処理が始まり，

```
20%|■■■                          | 4/20 [00:02<00:40, 3.32it/s]
                ↓進捗表示が徐々に進む↓
100%|■■■■■■■■■■■■■■■■■■■■| 20/20 [00:06<00:00, 3.32it/s]
```

のように進捗表示が進む．進捗表示の基準は反復回数であり，

百分率　進捗ゲージ　実行中の回数/総実行回数　[経過時間<残り時間の予想, 毎秒の反復回数]

が表示される．

この例では，20 回の繰り返しのためのイテラブル `range(20)` を tqdm インスタンスにしている．

上の例を実行した後，r の内容を確認する例を次に示す．

例. 処理後に r の値を確認（先の例の続き）

```
>>> r   Enter      ←値の確認              ↓値がリストとして得られている
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361]
```

注意）tqdm コンストラクタに与えるイテラブルの長さが予めわからない場合は，進捗ゲージ（■■■…）の部分と，残り時間の予想は表示されない．

[124] https://github.com/tqdm/tqdm

[125] Anaconda ディストリビューションや Google Colaboratory などのノートブック機能を支えている．

■ 進捗表示部の長さ

進捗表示部の長さは使用する表示デバイス（ターミナルウィンドウなど）の横幅から自動的に設定されるが，tqdm に引数 'ncols=幅' を与えることで「幅」を設定できる.「幅」に 0 を与えると進捗ゲージの表示がなくなり，文字情報だけで進捗が表示される.

7.2.1.2 イテラブルなデータによらない反復処理での使用方法

tqdm コンストラクタの引数にイテラブルなデータではなく，反復処理の回数だけを与えるという使用方法もある.

書き方： tqdm(total=反復回数)

この形で tqdm インスタンスを作成して，それに対して update メソッドを発行する.

書き方： tqdm インスタンス.update(増分)

tqdm インスタンスは，進捗情報（反復回数など）を保持しており，それが保持する反復回数を，与えた「増分」だけ大きくする.

反復処理が全て終了して tqdm インスタンスの使用を終了するには close メソッドを使用する.

書き方： tqdm インスタンス.close()

参考）close 処理を失念しないように，with 構文を使用すると安全である.

この方法による進捗表示では，インスタンス生成と update の実行を明示的に行うので，表示に乱れが起こることがある．従って，tqdm インスタンス生成と，update メソッドを含む反復処理を，関数定義のスコープ内に収めると安全である.（次の例）

例. tqdm インスタンスの生成と update を明示的に行う関数の定義（先の例の続き）

```
>>> def exe():  Enter      ←関数定義の記述の開始
...       r = []   Enter
...       pb = tqdm( total=20 )   Enter
...       for x in range(20):   Enter
...           r.append(x**2)   Enter
...           pb.update(1)   Enter
...           time.sleep(0.3)   Enter
...       pb.close()   Enter
...       return r   Enter
...   Enter      ←関数定義の記述の終了
>>>          ← Python の REPL に戻った
```

この後，定義した関数 exe を実行する.（次の例）

例. 上の関数 exe の実行（先の例の続き）

```
>>> exe()   Enter      ←処理の開始
100%|■■■■■■■■■■■■■■■■■■■■| 20/20 [00:06<00:00, 3.32it/s]   ←進捗表示   ↓戻り値
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361]
```

実際の処理では，反復回数が予めわからないことが多い．その場合は，tqdm インスタンス生成時に引数 'total=' 与えない．それによって，進捗ゲージなしの進捗表示となる.

※ ここで説明した方法は，反復処理の回数が予めわからない場合の進捗表示に有効なものとなる.

7.2.1.3 Jupyter Notebook のための tqdm

Jupyter Notebook（Anaconda，Google Colab. などのノートブックシステム）専用の tqdm クラスも存在する．これは，同ライブラリに同梱されており，

```
from tqdm.notebook import tqdm
```

として読み込む．基本的な使い方は先に説明した tqdm とほぼ同じである.

8　psutil ライブラリ

psutil は Python の標準ライブラリではなく，サードパーティ[126] が開発して公開するライブラリであり，システムリソース（CPU，メモリ，ディスク，ネットワーク，プロセスなど）に関する情報を取得する機能を提供する．このライブラリは利用に先立って pip などでインストールしておく必要がある．

例. `pip install psutil`

また利用に際しては次のようにしてライブラリを読み込む．

```
import psutil
```

8.1　CPU 関連の情報の取得

8.1.1　CPU の構成：cpu_count 関数

書き方：　cpu_count( )

計算機環境の CPU（もしくはそのコア）の個数を返す．引数に 'logical=False' を与えると物理 CPU（物理コア）の数を返し，'logical=True' を与えると論理 CPU の数[127] を返す．（デフォルトは True）

例. CPU の個数を調べる

```
>>> psutil.cpu_count(logical=False)  Enter    ←物理コア数を調べる
4            ←物理コア数
>>> psutil.cpu_count(logical=True)  Enter    ←論理コア数を調べる
8            ←論理コア数
```

8.1.2　CPU のクロック周波数：cpu_freq 関数

書き方：　cpu_freq( )

CPU のクロック周波数に関する情報を scpufreq オブジェクトの形で返す．

例. クロック周波数の情報を調べる（先の例の続き）

```
>>> psutil.cpu_freq()  Enter    ←クロック周波数情報の取得
scpufreq(current=2918.0, min=0.0, max=2918.0)    ←得られた scpufreq オブジェクト
```

得られた scpufreq オブジェクトの属性 min，max からはクロック周波数の下限と上限の値が得られる．計算機環境が下限の情報を与えない場合は min の値は 0 となる．また，現在実行中の状態におけるクロック周波数は current 属性から得られる．

cpu_freq 関数に引数 'percpu=True' を与えると，各論理 CPU に対応する scpufreq オブジェクトがリストの形で返される．（デフォルトは False）ただし，計算機環境によっては論理 CPU ごとの情報が得られないケースもあり，その場合は 1 個の scpufreq オブジェクトのみを持つリストが返される．

8.1.3　CPU 時間：cpu_times 関数

書き方：　cpu_times( )

システム起動時を起点とする CPU の稼働時間に関する情報を scputimes オブジェクトの形で返す．

例. CPU 稼働時間の情報を調べる（先の例の続き）

```
>>> psutil.cpu_times()  Enter    ← CPU 稼働時間情報の取得
scputimes(user=16968.796875, system=12453.015625,    ←↓得られた scputimes オブジェクト
        idle=4276172.390625, interrupt=925.4375, dpc=336.453125)
```

scputimes オブジェクトの各属性（表 48）の値の単位は秒である．

cpu_times 関数に引数 'percpu=True' を与えると，各論理 CPU に対応する scputimes オブジェクトがリストの形で返される．（デフォルトは False）

[126] https://github.com/giampaolo/psutil

[127] **ハイパースレッディング**（HT）などの並列処理技術を応用した CPU では，1 つの物理コアで複数の処理を同時に実行することができ，物理コアの総数よりも多くの論理 CPU が存在することになる．

表 48: scputimes オブジェクトの属性（一部）

属性	解説
user	ユーザーモードで実行された通常のプロセスが費やした時間
system	カーネルモードで実行されたプロセスが費やした時間
idle	システムがアイドル状態であった時間
interrupt	ハードウェア割り込みを処理するために費やした時間
dpc	遅延プロシージャコール（DPC）を処理するために費やした時間. （DPC は，標準の割り込みよりも低い優先度で実行される割り込み）

論理 CPU がハイパースレッディングなどの並列処理技術によって実現されている場合，全論理 CPU の時間の合計は実時間とはならないことに注意すること．また，引数なし（デフォルト）で実行された場合，得られる scputimes オブジェクトの各属性は全論理 CPU のものの合計となる．

8.1.4 CPU 使用率：cpu_percent 関数

書き方： cpu_percent( )

直近の CPU 使用率（CPU 時間から算出）を float で返す.

例. 直近の CPU 使用率を調べる（先の例の続き）

```
>>> psutil.cpu_percent()  Enter      ←直近の CPU 使用率を調べる
0.2            ← 20%
```

cpu_percent 関数に引数 'interval=秒数' を与えると「秒数」の間待機して，その前後の CPU 時間から CPU 使用率を算出する.

例. 1 秒間の CPU 使用率を調べる（先の例の続き）

```
>>> psutil.cpu_percent(interval=1)  Enter      ← 1 秒間の CPU 使用率を調べる
0.4            ← 40%
>>> for _ in range(3):  print(psutil.cpu_percent(interval=1))  Enter      ← 3 回実行
...  Enter      ← for 文の完結
0.6
0.0            ← CPU 使用率が 1 秒おきに 3 回出力される
0.6
```

8.2 メモリ関連の情報の取得

8.2.1 仮想記憶の状況：virtual_memory 関数

書き方： virtual_memory( )

計算機環境の仮想記憶の状況を svmem オブジェクトとして返す.

例. 仮想記憶の状況の調査（先の例の続き）

```
>>> psutil.virtual_memory()  Enter      ←仮想記憶の状況を調べる
svmem(total=34031902720, available=21871828992, percent=35.7,
                used=12160073728, free=21871828992)
```

svmem オブジェクトの各属性を表 49 に示す.

表 49 の available 属性の値は，実際に使用可能なメモリのサイズを意味しており，free 属性とは異なる．free 属性の値は，調査時点で全く使用されていないメモリの大きさを示している.

8.2.2 スワップ領域の使用状況：swap_memory 関数

書き方： swap_memory( )

仮想記憶における**スワップ領域**[128] の使用状況を sswap オブジェクトとして返す.

[128]仮想記憶の内容を退避するための，補助記憶装置（ディスクなどのストレージ）上の領域.

表 49: svmem オブジェクトの属性（一部）

属性	解説
total	総物理メモリ（単位：バイト）．スワップ領域は含まない．
available	使用可能なメモリ（単位：バイト）．
percent	使用中のメモリの割合（百分率）
used	使用中のメモリ（単位：バイト）
free	空きメモリ（単位：バイト）　※上記 available とは異なる．

例. スワップ領域の使用状況の調査（先の例の続き）

```
>>> psutil.swap_memory()  Enter      ←スワップ領域の使用状況を調べる
sswap(total=5100273664, used=77336576, free=5022937088, percent=1.5, sin=0, sout=0)
```

sswap オブジェクトが得られている．このオブジェクトの各属性を表 50 に示す．

表 50: sswap オブジェクトの属性

属性	解説
total	スワップ領域のサイズ（単位：バイト）
used	使用中のスワップ領域のサイズ（単位：バイト）
free	未使用のスワップ領域のサイズ（単位：バイト）
percent	スワップ領域の使用率（百分率）
sin	スワップインの量（単位：バイト）．ディスクからメモリへの移動量
sout	スワップアウトの量（単位：バイト）．メモリからディスクへの移動量

8.3　ディスク関連の情報の取得

8.3.1　パーティションの構成：disk_partitions 関数

書き方：　disk_partitions( )

計算機環境のディスクのパーティションの構成を sdiskpart オブジェクトのリストとして返す．

例. パーティションの構成の調査（Windows での実行例：先の例の続き）

```
>>> psutil.disk_partitions()  Enter      ←パーティションの構成を調べる
[sdiskpart(device='C:¥¥', mountpoint='C:¥¥', fstype='NTFS', opts='rw,fixed'),
 sdiskpart(device='D:¥¥', mountpoint='D:¥¥', fstype='NTFS', opts='rw,fixed')]
```

sdiskpart オブジェクトのリストが得られている．このオブジェクトの各属性を表 51 に示す．

表 51: sdiskpart オブジェクトの属性

属性	解説
device	デバイスの名前
mountpoint	パーティションがマウントされている場所
fstype	ファイルシステムの種類
opts	パーティションのオプション．コンマ区切りの文字列． 'rw'：読み書き可能．　'ro'：読み取り専用， 'fixed'：固定ディスク，　'removable'：リムーバブル， 'noexec'：実行不可，　他

disk_partitions 関数は，仮想ディスクやリムーバブルディスクは調査の対象にしないことがある，その場合，引数 'all=True' を与えると，それらも含め全てのパーティションの情報を取得する．

8.3.2 ディスクの使用状況：disk_usage 関数

書き方：　disk_usage(パス)

「パス」が示すディレクトリが存在するパーティションの使用状況を sdiskusage オブジェクトとして返す.「パス」には，ファイルではなくディレクトリを示すものを与える.

例. ディスクの使用状況の調査（先の例の続き）

```
>>> psutil.disk_usage('.')  Enter      ←パーティションの使用状況を調べる
sdiskusage(total=511085375488, used=456475533312, free=54609842176, percent=89.3)
```

この例では，カレントディレクトリが属しているパーティションの使用状況を調べ，sdiskusage オブジェクトを得ている．このオブジェクトの total 属性に当該パーティションの総容量，used 属性に使用中の容量，free 属性に未使用の容量，percent 属性にディスクの使用率（百分率）が得られる.

8.3.3 ディスクの I/O 状況：disk_io_counters 関数

書き方：　disk_io_counters( )

システム起動時からのディスクの I/O の状況（累積）を sdiskio オブジェクトとして返す.

例. ディスクの I/O 状況の調査（先の例の続き）

```
>>> psutil.disk_io_counters()  Enter      ←ディスクの I/O 状況を調べる
sdiskio(read_count=4101552, write_count=8761204, read_bytes=156201640448,
  write_bytes=274797508608, read_time=1413, write_time=3273)
```

システム起動時からの，全物理ディスクの統計情報が sdiskio オブジェクトとして得られている．このオブジェクトの各属性を表 52 に示す.

表 52: sdiskio オブジェクトの属性（一部）

属性	解説
read_count	ディスクの読み取り操作の回数
write_count	ディスクの書き込み操作の回数
read_bytes	読み取られたデータの総量（単位：バイト）
write_bytes	書き込まれたデータの総量（単位：バイト）
read_time	ディスクの読み取り操作に要した総時間（単位：ミリ秒）
write_time	ディスクの書き込み操作に要した総時間（単位：ミリ秒）

disk_io_counters 関数に引数 'perdisk=True'（デフォルトは False）を与えると，物理ディスクごとの sdiskio オブジェクトが辞書の形で得られる.

例. 物理ディスクごとの I/O 状況の調査（先の例の続き）

```
>>> psutil.disk_io_counters( perdisk=True )  Enter
{'PhysicalDrive0':  sdiskio(read_count=4031974, write_count=8728921,
                            read_bytes=150887646208, write_bytes=245865650176,
                            read_time=1151, write_time=2548),
 'PhysicalDrive1':  sdiskio(read_count=69678, write_count=34687,
                            read_bytes=5316595200, write_bytes=28971454464,
                            read_time=262, write_time=729)}
```

この例では，'PhysicalDrive0'，'PhysicalDrive1' が示す 2 つの物理ディスクの統計情報が得られている．辞書のキーはシステムが適切な形で与える.

8.4 ネットワーク関連の情報の取得

8.4.1 ネットワークの I/O 状況：net_io_counters 関数

書き方：　net_io_counters( )

システム起動時からのネットワークの I/O 状況（累積）を snetio オブジェクトとして返す.

例. ネットワークの I/O 状況の調査（先の例の続き）

```
>>> psutil.net_io_counters()  Enter      ←ネットワーク I/O 状況の調査
snetio(bytes_sent=3889675079, bytes_recv=48781973057, packets_sent=19719232,
  packets_recv=39328744, errin=0, errout=0, dropin=0, dropout=0)
```

システム起動時からの，ネットワーク I/O の統計情報が snetio オブジェクトとして得られている．このオブジェクトの各属性を表 53 に示す．

表 53: snetio オブジェクトの属性

属性	解説
bytes_sent	送信されたデータの総量（単位：バイト）
bytes_recv	受信されたデータの総量（単位：バイト）
packets_sent	送信されたパケットの総数
packets_recv	受信されたパケットの総数
errin	受信エラーの総数
errout	送信エラーの総数
dropin	受信ドロップの総数
dropout	送信ドロップの総数

net_io_counters 関数に引数 'pernic=True'（デフォルトは False）を与えると，ネットワークインターフェースごとの snetio オブジェクトが辞書の形で得られる．

例. ネットワークインターフェースごとの I/O 状況の調査（先の例の続き）

```
>>> psutil.net_io_counters( pernic=True )  Enter
{'Wi-Fi':  snetio(bytes_sent=0, bytes_recv=0, packets_sent=0, packets_recv=0, errin=0,
                  errout=0, dropin=0, dropout=0),
 'ローカル エリア接続* 1':  snetio(bytes_sent=0, bytes_recv=0, packets_sent=0, packets_recv=0,
                  errin=0, errout=0, dropin=0, dropout=0),
 'ローカル エリア接続* 2':  snetio(bytes_sent=0, bytes_recv=0, packets_sent=0,
                  packets_recv=0, errin=0, errout=0, dropin=0, dropout=0),
 'イーサネット':  snetio(bytes_sent=3889877658, bytes_recv=48783394042, packets_sent=19720289,
                  packets_recv=39330396, errin=0, errout=0, dropin=0, dropout=0),
 'Loopback Pseudo-Interface 1':  snetio(bytes_sent=0, bytes_recv=0, packets_sent=0,
                  packets_recv=0, errin=0, errout=0, dropin=0, dropout=0)}
```

8.4.2 現在の通信状況：net_connections 関数

書き方：　net_connections( )

現在のアクティブなネットワーク通信の状況を sconn オブジェクトのリストとして返す．sconn オブジェクトは，当該計算機上で実行中のプロセスの，アクティブな通信ソケット[129] に関するものである．

例. 現在の通信状況の調査（先の例の続き）

```
>>> psutil.net_connections()  Enter      ←現在の通信状況の調査
[sconn(fd=-1, family=<AddressFamily.AF_INET: 2>, type=<SocketKind.SOCK_DGRAM: 2>,
           laddr=addr(ip='192.168.0.4', port=1900), raddr=(), status='NONE', pid=7216),
 sconn(fd=-1, family=<AddressFamily.AF_INET: 2>, type=<SocketKind.SOCK_STREAM: 1>,
           laddr=addr(ip='0.0.0.0', port=49667), raddr=(), status='LISTEN', pid=2504),
 …]
```

複数の sconn オブジェクトのリストが得られている．個々の sconn オブジェクトは，net_connections 関数実行時点におけるアクティブな通信ソケットに対応している．sconn オブジェクトの各属性を表 54 に示す．

net_connections 関数には引数 'kind=' を与えて，調査対象を選ぶことができる．調査対象の種別を表 56 に挙げる．

[129] これに関しては拙書「Python3 入門」でも解説しています．

表 54: sconn オブジェクトの属性

属性	解説
fd	ソケットのファイルディスクリプタ．0 以上の整数値． -1 の場合は特定のファイルディスクリプタが無いことを意味する．
family	アドレスファミリ．　<AddressFamily.AF_INET: 2>：IPv4 アドレスファミリ， <AddressFamily.AF_INET6:10>：IPv6 アドレスファミリ， <AddressFamily.AF_UNIX: 1>：UNIX ドメインソケット
type	ソケットの種類．　<SocketKind.SOCK_STREAM: 1>：TCP ソケット， <SocketKind.SOCK_DGRAM: 2>：UDP ソケット， <SocketKind.SOCK_RAW: 3>：生のソケット
laddr	ローカルアドレス．IPv4 の場合は addr(ip=IP アドレス,port=ポート番号)，IPv6 の場合は addr(ip=IP アドレス,port=ポート番号,flowinfo=データフロー識別子,scopeid=スコープ識別子)， UNIX ドメインソケットの場合は addr(path=パス)．
raddr	リモートアドレス．値の形式は laddr と同じ．リモートアドレスがない場合は 空のタプル () となる．
status	接続状態．表 55 の接続状態を参照のこと．
pid	プロセス ID．この接続を所有するプロセスの ID

表 55: sconn オブジェクトの接続状態

属性	解説
ESTABLISHED	接続が確立され，データの送受信が可能な状態
SYN_SENT	接続要求（SYN パケット）が送信され，応答を待っている状態
SYN_RECV	接続要求（SYN パケット）が受信され，応答を送信した状態
FIN_WAIT1	接続終了要求（FIN パケット）が送信され，応答を待っている状態
FIN_WAIT2	接続終了要求（FIN パケット）に対する応答が受信され，接続が終了するのを待っている状態
TIME_WAIT	接続が終了し，一定時間待機している状態
CLOSE	接続が閉じられた状態
CLOSE_WAIT	リモート側が接続を閉じ，ローカル側がそれに応答するのを待っている状態
LAST_ACK	接続終了要求（FIN パケット）に対する応答を送信し，最終的な ACK を待っている状態
LISTEN	ソケットが接続要求を待機している状態
CLOSING	両側が接続終了要求（FIN パケット）を送信し，まだ全てのパケットが受信されていない状態
NONE	特定の状態がない場合

表 56: net_connections 関数の引数 kind=種別

種別	解　　説	種別	解　　説
'inet'	IPv4 および IPv6 のソケット接続	'inet4'	IPv4 のソケット接続
'inet6'	IPv6 のソケット接続	'tcp'	TCP ソケット接続
'tcp4'	IPv4 の TCP ソケット接続	'tcp6'	IPv6 の TCP ソケット接続
'udp'	UDP ソケット接続	'udp4'	IPv4 の UDP ソケット接続
'udp6'	IPv6 の UDP ソケット接続	'unix'	UNIX ドメインソケット接続
'all'	すべての種類のソケット接続	（デフォルトは 'inet）	

8.5　プロセス関連の情報の取得

8.5.1　実行中のプロセス：pids 関数

書き方：　pids( )

実行中の全てのプロセスの**プロセス ID**（PID）のリストを返す．

例. 実行中のプロセスの調査（先の例の続き）

```
>>> psutil.pids()  Enter      ←現在実行中のプロセスの調査
[0, 4, 156, 556, 608, 640, 668, 756, 776, 848, 948, 968, … ]      ← PID のリスト
```

8.5.2 Process クラス

Process クラスは，実行中のプロセスの情報を取り扱うためのものである．

書き方： Process(プロセス ID)

これは Process クラスのコンストラクタであり，指定した「プロセス ID」（PID）のプロセスに関する情報を保持するインスタンスを生成する．

注意）この Process クラスは multiprocessing モジュールが提供する Process クラスとは別のものである．
psutil ライブラリと multiprocessing モジュールを併用する場合は名前の衝突に注意すること．

実行中の Python 処理系のプロセスを用いる例を挙げて，このクラスについて解説する．

例. 実行中の Python 処理系を指す Process オブジェクトの作成

```
>>> import os   Enter      ←必要なモジュールの
>>> import psutil   Enter      ←必要なモジュールの
>>> p = os.getpid(); print(p)   Enter      ← Python 処理系の PID を取得
20728          ← PID（システムが設定した値）
>>> pr = psutil.Process( p )   Enter      ← Python 処理系を指す Process オブジェクトを作成
>>> pr   Enter      ←内容確認
psutil.Process(pid=20728, name='python.exe', status='running', started='18:24:07')
```

Process インスタンスが得られていることがわかる．このオブジェクトの pid 属性にはプロセス ID が，name 属性にはそのプログラムの名前（この場合は Python 処理系である'python.exe'）が，status 属性には実行状態が，started 属性には開始時刻が保持されている．

Process オブジェクトに対して使える各種メソッドを以下に挙げる．

8.5.2.1 メモリの使用状況の調査：memory_info

memory_info メソッドを用いると，プロセスのメモリの使用状況を詳細に知ることができる．

書き方： Process オブジェクト.memory_info( )

「Process オブジェクト」のメモリ使用状況を示す pmem オブジェクトを返す．

例. プロセスのメモリの使用状況の調査（先の例の続き）

```
>>> pm = pr.memory_info()   Enter      ←調査実行
>>> pm   Enter      ←結果の確認
pmem(rss=100777984, vms=1138958336, shared=26755072, text=2818048, lib=0,
     data=247259136, dirty=0)
```

pmem オブジェクトが得られている．このオブジェクトの各属性を表 57 に示す．

表 57: pmem オブジェクトの属性（一部）

属性	解　　説
rss	プロセスが実際に使用している物理メモリのサイズ.（Resident Set Size）
vms	プロセスに割り当てられた仮想メモリの総量.（Virtual Memory Size）
shared	他のプロセスと共有されているメモリのサイズ
text	プロセスの実行可能なコード（テキストセグメント）のサイズ
lib	共有ライブラリのサイズ
data	プロセスのデータセグメントとヒープのサイズ
dirty	変更されたがまだディスクに書き込まれていないメモリのサイズ

メモリサイズの単位はバイト

pmem オブジェクトの属性の種類は，Python 言語処理系を実行している計算機環境によって（OS の種類によって）異なることがあるので，詳細に関しては公式インターネットサイトなどの資料を参照のこと．

pmem オブジェクトの属性のうち，特に重要なものとして rss と vms がある．vms は当該プロセスが確保してい

る仮想記憶の全サイズ，rss は当該プロセスがアクティブに使用している物理メモリのサイズである．

参考）先の例を Windows 環境で実行すると次のような pmem オブジェクトが得られることがある．

```
pmem(rss=17948672, vms=11739136, num_page_faults=4756, peak_wset=18313216,
     wset=17948672, peak_paged_pool=129120, paged_pool=128944, peak_nonpaged_pool=11848,
     nonpaged_pool=11672, pagefile=11739136, peak_pagefile=12124160, private=11739136)
```

通常は，アクティブに使用している物理記憶のサイズは，全仮想記憶のサイズより小さいが，この例では，rss の値が vms の値を上回っている．

■ 応用例

memory_info メソッドを応用して，プロセスのメモリ使用状況を調べるツール mstat.py を実装する例を示す．

プログラム：mstat.py

```
1  # coding: utf-8
2  import psutil, os   # 必要なモジュールの読み込み
3
4  # プロセスのメモリの使用状況を調べる関数
5  def pstat( pid=None ):
6      if pid == None:
7          # Python処理系のpmemオブジェクト
8          minf = psutil.Process( os.getpid() ).memory_info()
9      else:
10         # pidのプロセスのpmemオブジェクト
11         minf = psutil.Process( pid ).memory_info()
12     return (minf.rss, minf.vms)
13
14 # システムの
15
16 if __name__ == '__main__':
17     m = pstat()
18     print('Python処理系のメモリ:')
19     print('  物理メモリ(Bytes):',m[0])
20     print('  仮想メモリ(Bytes):',m[1])
```

上記スクリプトに定義されている pstat 関数は，引数に与えた PID のプロセスのメモリ使用状況を調べ，物理記憶のサイズと仮想記憶のサイズのタプルを返す．引数を省略すると，Python 処理系のアクティブな物理記憶のサイズと，仮想記憶の全サイズのタプルを返す．

例. Python 言語処理系のメモリ使用状況を調べる

```
>>> import mstat   Enter      ←スクリプトをモジュールとして読み込む
>>> mstat.pstat()  Enter      ←メモリ使用状況調査
(12140544, 19988480)      ←タプル (物理記憶, 仮想記憶)
```

このスクリプトを直接実行するとそれらの値を標準出力に出力する．

例. スクリプトを直接実行した際の出力

```
Python 処理系のメモリ:
  物理記憶 (Bytes):  12140544
  仮想記憶 (Bytes):  19988480
```

8.5.2.2　CPU 使用率：cpu_percent

cpu_percent メソッドを用いると，プロセスの CPU 使用率を知ることができる．

書き方：　Process オブジェクト.cpu_percent( )

「Process オブジェクト」が示すプロセスの CPU 使用率を返す．これは psutil.cpu_percent 関数（p.266 で解説）と同様の考え方で，プロセスを対象に CPU 使用率を算出するものである..

8.5.2.3 CPU 時間：cpu_times

cpu_times メソッドを用いると，プロセスの CPU 時間を知ることができる．

書き方： Process オブジェクト.cpu_times( )

「Process オブジェクト」が示すプロセスの CPU 時間を示す pcputimes オブジェクトを返す．

例. プロセスの CPU 時間の調査（先の例の続き）

```
>>> pr.cpu_times()  Enter      ←調査実行
pcputimes(user=0.015625, system=0.0, children_user=0.0, children_system=0.0)
```

pcputimes オブジェクトが得られているのがわかる．このオブジェクトの user 属性には当該プロセスのユーザーモードの CPU 時間，system 属性にはカーネルモードの CPU 時間が保持されている．対象プロセスの全ての子プロセスの CPU 時間（合計）も得られており，それは children_user 属性（ユーザーモード），children_system 属性（カーネルモード）が保持している．

8.5.2.4 開いているファイルの調査：open_files

open_files メソッドを用いると，プロセスが開いているファイルを調べることができる．

書き方： Process オブジェクト.open_files( )

「Process オブジェクト」が示すプロセスが開いているファイルを，popenfile オブジェクトのリストの形で返す．popenfile オブジェクトは，指定したプロセスが開いている個々のファイルを表すもので，表 58 に示すような属性を持つ．

表 58: popenfile オブジェクトの属性

属性	解　　説
path	ファイルの絶対パス
fd	ファイルディスクリプタ（番号）
position	ファイル内の現在の位置.（seek された現在位置）
mode	ファイルのオープンモード.（'r'，'w' など）
flags	ファイルのフラグ．C 言語の open 関数（システムコールの 1 つ）の第 2 引数に与えるフラグと同じ形式.

例. プロセスが開いているファイルの調査（Linux での実行例：先の例の続き）

```
>>> f1=open('dummy1','w'); f2=open('dummy2','w')  Enter    ← 2 つのファイルを開く
>>> pr.open_files()  Enter    ←ファイルの使用状況を調査
[popenfile(path='/usr/home/katsu/dummy1', fd=42, position=0, mode='w', flags=557057),
 popenfile(path='/usr/home/katsu/dummy2', fd=43, position=0, mode='w', flags=557057)]
>>> f1.close(); f2.close()  Enter    ← 2 つのファイルを閉じる
```

これは，Python 処理系が 2 つのファイル dummy1，dummy2 を開いた（作成した）状態で open_files メソッドを実行した例であり，それらファイルに対応する popenfile オブジェクトがリストの形で得られていることがわかる．

注意）psutil は Windows 環境に十分対応できていない部分があり，この例と同じ処理を Windows 環境で実行すると，open_files メソッドの実行結果は

```
[popenfile(path='C:¥¥dummy2', fd=-1), popenfile(path='C:¥¥dummy1', fd=-1)]
```

などとなり，特定の属性が得られないことがある．

8.5.2.5 プロセスの通信状況：net_connections

net_connections メソッドを用いると，プロセスの通信状況を調べることができる．

書き方： Process オブジェクト.net_connections( )

「Process オブジェクト」が示すプロセスの，アクティブなネットワーク通信の状況を pconn オブジェクトのリストとして返す．pconn オブジェクトは，プロセスのアクティブな通信ソケット[130] に関するものである．

[130] これに関しては拙書「Python3 入門」でも解説しています．

例. プロセスの通信状況の調査（先の例の続き）

```
>>> pr.net_connections()  Enter      ←通信状況の調査
[pconn(fd=21, family=<AddressFamily.AF_INET: 2>, type=<SocketKind.SOCK_STREAM: 1>,
       laddr=addr(ip='127.0.0.1', port=39893), raddr=(), status='LISTEN'),
 pconn(fd=34, family=<AddressFamily.AF_INET: 2>, type=<SocketKind.SOCK_STREAM: 1>,
       laddr=addr(ip='127.0.0.1', port=54812), raddr=addr(ip='127.0.0.1', port=58761),
       status='ESTABLISHED')]
```

これはプロセスが 2 つのソケット通信をしている場合の例である．通信しているソケットがない場合は空リスト `[]` が返される．

pconn オブジェクトの属性に関しては，先の「現在の通信状況：net_connections 関数」（p.269）で解説した sconn オブジェクトに準ずる．

8.5.2.6 プロセスの終了：terminate

terminate メソッドを用いると，指定したプロセスを終了させることができる．

書き方：　Process オブジェクト.terminate( )

「Process オブジェクト」が示すプロセスを終了する．

例. プロセスの終了（先の例の続き）

```
>>> pr.terminate()  Enter      ←プロセスの終了
C:¥Users¥katsu>     ← OS のプロンプトに戻った（Windows の場合）
```

注意）この例では，Python 言語処理系を terminate メソッドで終了しているが，この目的のためには exit 関数を使用するべきである．

8.5.3 サンプルプログラム：プロセスの監視

multiprocessing モジュールによるマルチプロセスの実行状態を psutil ライブラリの機能で監視するサンプルを mproc02.py に示す．このサンプルでは，multiprocessing モジュールが提供する Process クラスのインスタンス p1，p2，p3 を生成して実行する．各プロセスでは，longTask 関数（時間がかかる計算処理）を実行する．また，それらプロセスの状態を調べるために，psutil が提供する Process クラスのオブジェクト pr1，pr2，pr3 に変換して各種メソッド（先に解説したもの）を実行する．

プログラム：mproc02.py

```
1  # coding: utf-8
2  import time
3  import multiprocessing as mlpr
4  import psutil
5
6  # 実行時間がかかる処理
7  def longTask():
8      n = 1
9      for _ in range(500000):      # 回数は適宜調整すること
10         n *= 2
11
12 if __name__ == '__main__':
13     # プロセス作成
14     p1 = mlpr.Process(target=longTask, args=())
15     p2 = mlpr.Process(target=longTask, args=())
16     p3 = mlpr.Process(target=longTask, args=())
17
18     # プロセス実行
19     p1.start();     p2.start();     p3.start()
20     # psutil.Processオブジェクトに変換
21     pr1 = psutil.Process(p1.pid)
22     pr2 = psutil.Process(p2.pid)
23     pr3 = psutil.Process(p3.pid)
24
25     # プロセス監視：全プロセスが終了するまで監視を繰り返す（1秒間隔）
26     t1 = time.time()
27     while any( [ p1.is_alive(), p2.is_alive(), p3.is_alive() ] ):
28         # メモリの状態を調べる
29         mi1 = pr1.memory_info();  mi2 = pr2.memory_info();  mi3 = pr3.memory_info()
30         # CPU使用率を調べる
```

```
31          cp1 = pr1.cpu_percent();  cp2 = pr2.cpu_percent();  cp3 = pr3.cpu_percent();
32          # CPU時間を調べる
33          ct1 = pr1.cpu_times();    ct2 = pr2.cpu_times();    ct3 = pr3.cpu_times();
34          # プロセスの状態を出力する
35          print('\t\t   [p1]\t\t   [p2]\t\t   [p3]')
36          print(f'memory_info:rss\t{mi1.rss:9d}\t{mi2.rss:9d}\t{mi3.rss:9d}')
37          print(f'memory_info:vms\t{mi1.vms:9d}\t{mi2.vms:9d}\t{mi3.vms:9d}')
38          print(f'cpu_percent\t{cp1:9.2f}\t{cp2:9.2f}\t{cp3:9.2f}')
39          print(f'cpu_times:user\t{ct1.user:9.2f}\t{ct2.user:9.2f}\t{ct3.user:9.2f}')
40          print(f'cpu_times:sys\t{ct1.system:9.2f}\t{ct2.system:9.2f}\t{ct3.system:9.2f}')
41          print()
42          time.sleep(1)   # 1秒待機
43      t2 = time.time()
44      p1.join();  p2.join();  p3.join()   # 安全のため
45      print(f'elapsed time\t{t2-t1:9.2f}')
```

プログラム中の while 文でプロセスの状態監視と出力を繰り返す．繰り返しの間隔は 1 秒で，全プロセスが終了するまで行う．プロセスで実行する関数 longTask 内の反復回数は適宜調整すること．

このサンプルを実行した際の出力例を次に示す．

実行例：mproc02.py を実行した際のコンソールの出力

```
                    [p1]           [p2]           [p3]
memory_info:rss   7544832        3715072        1769472
memory_info:vms   4190208        2392064         425984      ←状態出力：1回目
cpu_percent          0.00           0.00           0.00
cpu_times:user       0.02           0.00           0.00
cpu_times:sys        0.00           0.00           0.00

                    [p1]           [p2]           [p3]
memory_info:rss  18456576       18092032       18329600
memory_info:vms  12255232       11874304       12128256      ←状態出力：2回目
cpu_percent         92.20          89.10          82.80
cpu_times:user       0.91           0.89           0.83
cpu_times:sys        0.03           0.00           0.00

                    [p1]           [p2]           [p3]
memory_info:rss  18575360       18108416       18370560
memory_info:vms  12378112       11874304       12169216      ←状態出力：3回目
cpu_percent         93.80          84.40          96.90
cpu_times:user       1.84           1.73           1.80
cpu_times:sys        0.03           0.00           0.00

                    [p1]           [p2]           [p3]
memory_info:rss  18636800       18173952       18444288
memory_info:vms  12435456       11935744       12242944      ←状態出力：4回目
cpu_percent        101.60          95.30          92.20
cpu_times:user       2.86           2.69           2.72
cpu_times:sys        0.03           0.00           0.00

elapsed time         4.01                                    ←処理全体の経過時間
```

9　その他

9.1　json：データ交換フォーマット JSON の使用

JSON（JavaScript Object Notation）は，**キー**と**値**を対応させるデータ構造の表記[131] である．JSON は JavaScript 言語で使用するオブジェクトの構造に由来するが，異なるアプリケーションソフトや言語処理系の間でデータを受け渡すためのデータフォーマットとして普及している．Python にはキーと値を対応させるデータ構造である**辞書**（dict 型）があるが，JSON と親和性が高く，Python 処理系と他のソフトウェアの間でデータの受け渡しをする際に JSON の表記に依ることが多い．

Python には JSON を扱うためのモジュールが標準的に提供されており，

```
import json
```

として読み込んで使用することができる．

9.1.1　JSON の表記

JSON では ‘[…]’ で括った**配列**と，‘{…}’ で括った**オブジェクト**というデータ構造が表現でき，それらは Python の**リスト**と**辞書**とほぼ同じ表記法を用いる．また JSON のデータ構造に含めることができる要素としては，**数値**（整数，浮動小数点数），**真理値**（true/false），**空値**（null），**文字列**がある．文字列の引用符にはダブルクオーテーション「”」を使用し，「¥」によるエスケープシーケンスが使用できる．真理値や空値の表記は Python のものとは異なるが，json モジュールを用いることで，JSON と Python 処理系の間で適切に変換することができる．

9.1.2　使用例

例. Python のデータ構造から JSON への変換

```
>>> import json   Enter      ←モジュールの読み込み
>>> d = {'りんご':'apple', 'みかん':'orange', 'ぶどう':'grape'}   Enter   ←辞書を用意
>>> js = json.dumps( d )   Enter      ← dumps によって JSON に変換
>>> print( js )   Enter      ←内容確認
{"¥u308a¥u3093¥u3054":"apple", "¥u307f¥u304b¥u3093":"orange",
"¥u3076¥u3069¥u3046":"grape"}      ←結果表示（エスケープされた UNICODE 文字列）
```

この例のように dumps 関数によって Python のオブジェクトが JSON の表記に従った文字列に変換される．この際，多バイト文字は「¥」でエスケープされた UNICODE 列となる．dumps 関数に引数 `ensure_ascii=False` を与える（デフォルトは True）と多バイト文字がそのままの形式となる．

例. 多バイト文字をエスケープしない例（先の例の続き）

```
>>> js2 = json.dumps( d, ensure_ascii=False )   Enter      ← dumps によって JSON に変換
>>> print( js2 )   Enter      ←内容確認
{"りんご":  "apple", "みかん":  "orange", "ぶどう":  "grape"}   ←日本語はそのまま
```

JSON 表記の文字列を Python のオブジェクトに変換するには loads 関数を用いる．（次の例）

例. JSON から Python のデータ構造への変換（先の例の続き）

```
>>> json.loads( js )   Enter      ← loads によって JSON を Python のオブジェクトに変換
{'りんご':'apple', 'みかん':'orange', 'ぶどう':'grape'}      ←結果表示
```

多バイト文字がエスケープされていない文字列（先の例の js2）も同様に loads 関数で辞書に変換することができる．

9.1.2.1　JSON データのファイル I/O

dump 関数（dumps と混同しないように注意）を使用すると Python のデータ構造を JSON に変換したものをファイルに出力することができる．

書き方：　dump(Python のデータ構造, ファイル)

[131] RFC 8259

例. JSON のファイルへの出力（先の例の続き）

```
>>> f = open( 'dic01.json', 'w' )  Enter    ←ファイル 'dic01.json' を出力モードで作成
>>> json.dump( d, f )  Enter        ←そのファイルに辞書 d の内容を出力
>>> f.close()  Enter        ←ファイルを閉じる
```

この処理で，上の js の内容と同じものがファイル 'dic01.json' に出力される．

dump 関数も `ensure_ascii=False` 引数を与えることができ，多バイト文字をそのまま出力できる．ただしその場合は，出力ファイルのエンコーディングを正しく指定すること．

例. utf-8 エンコーディングで JSON をファイルへ出力する（先の例の続き）

```
>>> f = open( 'dic01-2.json', 'w', encoding='utf-8' ) Enter ← utf-8 の出力用ファイル 'dic01-2.json'
>>> json.dump( d, f, ensure_ascii=False )  Enter        ←辞書 d の内容を出力
>>> f.close()  Enter        ←ファイルを閉じる
```

ファイルに保存されている JSON データを読み込むには load 関数（loads と混同しないように注意）を使用する．

書き方：　load(ファイル)

読み取った JSON データを Python 形式に変換したものを返す．

例. JSON のファイルからの入力（先の例の続き）

```
>>> f = open( 'dic01.json', 'r' )  Enter    ←ファイル 'dic01.json' を入力モードで開く
>>> d2 = json.load( f )  Enter        ←そのファイルから JSON を読み込んで Python 形式に変換
>>> f.close()  Enter        ←ファイルを閉じる
>>> d2                ←読み取った内容を確認
{'りんご':  'apple', 'みかん':  'orange', 'ぶどう':  'grape'}        ←内容表示
```

多バイト文字がエスケープされていないファイル（先の例の 'dic01-2.json'）も同様に load 関数で読み込むことができる．その場合は，ファイルのオープン時に適切にエンコーディングを指定すること．

9.1.2.2　真理値，None の扱い

空値や真理値も dumps, loads 関数で適切に変換される．（次の例参照）

例. 空値や真理値の変換（先の例の続き）

```
>>> L = [1,2,3,None,True,False]  Enter  ←リストを用意
>>> js = json.dumps( L )  Enter     ← dumps によって JSON に変換
>>> js  Enter     ←内容確認
'[1, 2, 3, null, true, false]'     ←結果表示（JSON 表記の文字列になっている）
>>> json.loads( js )  Enter     ← loads によって JSON を Python のオブジェクトに変換
[1, 2, 3, None, True, False]     ←結果表示
```

9.2 urllib：URL に関する処理

URL の処理に関する機能を提供する urllib が Python 標準のライブラリとして利用できる．

9.2.1 多バイト文字の扱い（‘%’ エンコーディング）

URL に記述する多バイト文字（日本語など）は ‘%’ を用いた文字コードとして表記される．例えば「ウィキペディア日本語版」の URL には日本語の文字列が含まれており，Web ブラウザ上での表示は

‘https://ja.wikipedia.org/wiki/メインページ’

となるが，これは実際には

‘https://ja.wikipedia.org/wiki/%E3%83%A1%E3%82%A4%E3%83%B3%E3%83%9A%E3%83%BC%E3%82%B8’

という文字列として扱われる．

多バイト文字列を文字コード列にエンコードするには urllib.parse.quote 関数を用いる．この機能を読み込むには

import urllib.parse

とする．

例. 多バイト文字列を文字コード列にエンコードする

```
>>> import urllib.parse  Enter          ←必要な機能の読込み
>>> e = urllib.parse.quote( '日本語' )  Enter  ←文字コード列に変換
>>> e  Enter                ←内容確認
'%E6%97%A5%E6%9C%AC%E8%AA%9E'  ←結果表示
```

文字列 ’日本語’ が ‘%’ 表記の文字コード列に変換されていることがわかる．このような文字コード列を元の文字列に戻す（逆変換）には urllib.parse.unquote を用いる．

例. 文字コード列を多バイト文字列に戻す（先の例の続き）

```
>>> s = urllib.parse.unquote( e )  Enter  ←元の文字列に戻す
>>> s  Enter                ←内容確認
’日本語’  ←結果表示
```

9.2.1.1 エンコーディングの指定

quote，unquote 共に，エンコーディングを明に指定する場合はキーワード引数 ‘encoding=**エンコーディング**’ を与える．

例. エンコーディング（Shift-JIS）を指定しての変換と逆変換

```
>>> e = urllib.parse.quote( '日本語', encoding='shift-jis' )  Enter  ←文字コード列に変換
>>> e  Enter                ←内容確認
'%93%FA%96%7B%8C%EA'  ←結果表示
>>> s = urllib.parse.unquote( e, encoding='shift-jis' )  Enter  ←元の文字列に戻す
>>> s  Enter                ←内容確認
’日本語’  ←結果表示
```

9.3 jaconv：日本語文字に関する各種の変換

jaconv モジュールは日本語文字に関する様々な変換機能を提供する．jaconv に関する情報についてはインターネットサイト

https://pypi.org/project/jaconv/
https://github.com/ikegami-yukino/jaconv

を参照のこと．

■ ひらがな⇔カタカナ変換

ひらがなをカタカナに変換するには hira2kata 関数を，カタカナをひらがなに変換するには kata2hira 関数を用いる．（次の例参照）

例. ひらがな⇔カタカナ変換

```
>>> import jaconv  Enter      ←モジュールの読み込み
>>> s = jaconv.hira2kata('この文はひらがなを含んでいます．')  Enter  ←ひらがなをカタカナに変換
>>> s  Enter      ←内容確認
'コノ文ハヒラガナヲ含ンデイマス．'      ←ひらがなの部分がカタカナに変換されている
>>> jaconv.kata2hira(s)  Enter  ←カタカナをひらがなに変換
'この文はひらがなを含んでいます．'      ←変換されている
```

■ 全角⇔半角の変換

全角⇔半角変換の例を次に示す．

例. 全角ひらがな→半角カタカナ（先の例の続き）

```
>>> s = jaconv.hira2hkata('この文はひらがなを含んでいます．') Enter ←全角ひらがなを半角カタカナに変換
>>> s  Enter      ←内容確認
'ｺﾉ文ﾊﾋﾗｶﾞﾅﾗ含ﾝﾃﾞｲﾏｽ．'      ←結果表示
```

この例のように，全角ひらがなを半角カタカナに変換するには関数 hira2hkata を用いる．

例. 半角→全角（先の例の続き）

```
>>> jaconv.h2z(s)  Enter  ←半角を全角に変換
'コノ文ハヒラガナヲ含ンデイマス．'      ←結果表示
```

この例のように，半角文字を全角文字に変換するには関数 h2z を用いる．また，逆の変換を行う関数 z2h も使用できる．これらの関数はカタカナを変換対象としているが，英数記号も変換対象とする場合は表 59 のようなキーワード引数を与える．

表 59: 全角⇔半角の変換対象とする文字の指定

対象文字	カタカナ	数字	数字以外の ascii 文字
キーワード引数	kana=**True**/False	digit=True/**False**	ascii=True/**False**

太字はデフォルト

例. 全角→半角

```
>>> jaconv.z2h('ａｂｃｄ１２３４＃＄％＆アイウエ')  Enter  ←全角を半角に変換
'ａｂｃｄ１２３４＃＄％＆ｱｲｳｴ'      ←カタカナのみが変換されている
>>> jaconv.z2h('ａｂｃｄ１２３４＃＄％＆アイウエ',digit=True,ascii=True)  Enter
                                  対象の文字を指定して変換 ↑
'abcd1234#$%&ｱｲｳｴ'      ←変換されている
```

索 引

謝辞

本書の内容に関して，インターネット（電子メール，SNS など）を介して多くの方々から有効な助言やリクエストをいただきました．本書の執筆と維持のために大きな貢献となっております．特に，誤った記述に対する厳しいご指摘は大変にありがたいものです．ここにお礼申し上げます．今後もご協力いただけましたら幸いです．

「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

著者：中村勝則

発行：2026 年 3 月 2 日

本書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/b465b0cf05b1b7fd4975

上記 URL の QR コード

● 連絡先

nkatsu2012@gmail.com

中村勝則