

Python3 ライブライブック

各種ライブラリの基本的な使用方法

OpenCV / Pillow / pygame / Eel / PyDub / NumPy / matplotlib / SciPy /
SymPy / gmpy2 / hashlib, passlib / Cython / Numba / ctypes / PyInstaller /
JupyterLab / json / psutil / urllib / zenhan / jaconv

Copyright © 2017-2025, Katsunori Nakamura

中村勝則

2025年8月18日

免責事項

本書の内容は参考的資料であり、掲載したプログラミ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
1.1 OpenCV	1
1.1.1 動画像の入力	2
1.1.2 ユーザインターフェース	2
1.1.2.1 動画ファイルからの入力	3
1.1.3 フレームのファイルへの保存	3
1.1.4 静止画像の読み込み	3
1.1.5 色空間とチャネル	4
1.1.5.1 画像フレームのチャネルの順序	5
1.1.5.2 色空間の変換	6
1.1.5.3 チャネルの分解と合成	7
1.1.5.4 色空間の選択に関するここと	8
1.1.5.5 参考) HSV の色相に関するここと	9
1.1.6 カラー画像からモノクロ画像への変換	10
1.1.6.1 モノクロ 2 値化 (1) : 一定の閾値で判定	10
1.1.6.2 モノクロ 2 値化 (2) : 閾値を適応的に判定	11
1.1.7 画像フレームに対する描画	13
1.1.7.1 基本的な考え方	13
1.1.7.2 線分, 矢印	13
1.1.7.3 矩形, 円, 楕円, 円弧	14
1.1.7.4 折れ線, 多角形	16
1.1.7.5 文字列	17
1.1.7.6 マーカー	18
1.2 Pillow	20
1.2.1 画像ファイルの読み込みと保存	20
1.2.1.1 EPS を読み込む際の解像度	21
1.2.1.2 画像ファイルの保存	21
1.2.2 Image オブジェクトの新規作成	21
1.2.3 画像の閲覧	22
1.2.4 画像の編集	22
1.2.4.1 画像の拡大と縮小	22
1.2.4.2 画像の部分の取り出し	22
1.2.4.3 画像の複製	23
1.2.4.4 画像の貼り付け	23
1.2.4.5 画像の回転	23
1.2.5 画像処理	23
1.2.5.1 色の分解と合成	23
1.2.5.2 カラー画像からモノクロ画像への変換	24
1.2.6 描画	25
1.2.7 アニメーション GIF の作成	26
1.2.8 Image オブジェクトから画素の数値配列への変換	27
2 GUI とマルチメディア	28
2.1 pygame	28
2.1.1 基礎事項	28
2.1.1.1 Surface オブジェクト	28

2.1.1.2 アプリケーションの実行ループ	28
2.1.2 描画機能	30
2.1.2.1 描画のサンプルプログラム	32
2.1.2.2 回転, 拡縮のサンプルプログラム	34
2.1.3 キーボードとマウスのハンドリング	36
2.1.4 音声の再生	37
2.1.4.1 Sound オブジェクトを用いる方法	38
2.1.4.2 Sound オブジェクトから NumPy の配列への変換	39
2.1.5 スプライトの利用	40
2.2 Eel	45
2.2.1 基礎事項	45
2.2.1.1 Chrome の起動モード	46
2.2.2 Python/JavaScript で記述した関数の呼出し	46
2.2.2.1 関数の戻り値について	48
2.3 メディアデータの変換: PyDub	49
2.3.1 音声ファイルの読み込み	49
2.3.2 音声データから NumPy の配列への変換	49
2.3.3 NumPy の配列から音声データへの変換	50
2.3.4 音声ファイルの保存	50
2.3.4.1 MP3 形式ファイルとして保存	50
2.3.4.2 WAV 形式ファイルとして保存	50
3 科学技術系	51
3.1 数値計算と可視化のためのライブラリ: NumPy / matplotlib	51
3.1.1 NumPy で扱うデータ型	51
3.1.2 配列オブジェクトの基本的な扱い方	52
3.1.2.1 配列の要素の型について	52
3.1.2.2 真理値の配列	53
3.1.2.3 基本的な計算処理	53
3.1.2.4 扱える値の範囲	54
3.1.2.5 特殊な値: 無限大と非数	55
3.1.2.6 データ列の生成 (数列の生成)	55
3.1.2.7 多次元配列の生成	56
3.1.2.8 配列の形状の調査	57
3.1.2.9 配列の要素へのアクセス	57
3.1.2.10 スライスにデータ列を与えた場合の動作	58
3.1.2.11 指定した行, 列へのアクセス	58
3.1.2.12 配列形状の変形	58
3.1.2.13 行, 列の転置	60
3.1.2.14 行, 列の反転と回転	60
3.1.2.15 型の変換	61
3.1.2.16 配列の複製 (コピー)	61
3.1.3 配列の連結と繰り返し	62
3.1.3.1 append, concatenate による連結	62
3.1.3.2 hstack, vstack による連結	63
3.1.3.3 tile による配列の繰り返し	64
3.1.4 配列への要素の挿入	64
3.1.4.1 行, 列の挿入	65

3.1.5	配列要素の部分的削除	66
3.1.5.1	区間を指定した削除	66
3.1.5.2	行, 列の削除	67
3.1.6	配列の次元の拡大	67
3.1.6.1	newaxis オブジェクトによる方法	67
3.1.6.2	expand_dims による方法	68
3.1.7	データの抽出	68
3.1.7.1	真理値列によるマスキング	68
3.1.7.2	条件式による要素の抽出	69
3.1.7.3	論理演算子による条件式の結合	69
3.1.7.4	where による要素の抽出と置換	69
3.1.7.5	最大値, 最小値, その位置の探索	70
3.1.8	データの整列 (ソート)	72
3.1.8.1	2 次元配列の整列	72
3.1.8.2	整列結果のインデックスを取得する方法	73
3.1.9	配列要素の差分の配列	73
3.1.10	配列に対する様々な処理	74
3.1.10.1	重複する要素の排除	74
3.1.10.2	要素の個数の集計	74
3.1.10.3	整数要素の集計	75
3.1.10.4	指定した条件を満たす要素の集計	75
3.1.11	配列に対する演算: 1 次元から 1 次元	75
3.1.12	データの可視化 (基本)	76
3.1.12.1	作図処理の基本的な手順	77
3.1.12.2	2 次元のプロット: 折れ線グラフ	77
3.1.12.3	グラフの目盛りの設定	80
3.1.12.4	対数軸のグラフの作成	81
3.1.12.5	複数のグラフの作成	81
3.1.12.6	matplotlib のグラフの構造	84
3.1.12.7	グラフの枠を非表示にする方法	87
3.1.12.8	極座標プロット	88
3.1.12.9	レーダーチャート (極座標の応用)	88
3.1.12.10	日本語の見出し・ラベルの表示	89
3.1.12.11	グラフを画像ファイルとして保存する方法	90
3.1.13	乱数の生成	91
3.1.13.1	一様乱数の生成	91
3.1.13.2	整数の乱数の生成	91
3.1.13.3	正規乱数の生成	91
3.1.13.4	乱数の seed について	91
3.1.13.5	RandomState オブジェクト	92
3.1.14	統計に関する処理	93
3.1.14.1	合計	93
3.1.14.2	最大値, 最小値	93
3.1.14.3	平均, 分散, 標準偏差	93
3.1.14.4	分位点, パーセント点	94
3.1.14.5	区間と集計 (階級と度数調査)	94
3.1.14.6	最頻値を求める方法	97
3.1.14.7	相関係数	100

3.1.14.8 データのシャッフル	101
3.1.15 データの可視化（2）	101
3.1.15.1 ヒストグラム	101
3.1.15.2 散布図	103
3.1.15.3 棒グラフ	103
3.1.15.4 円グラフ	105
3.1.15.5 箱ひげ図	107
3.1.16 データの可視化：3次元プロット	108
3.1.16.1 メッシュ（格子）の考え方	108
3.1.16.2 meshgrid 関数の働き	109
3.1.16.3 3次元プロットの準備	109
3.1.16.4 ワイヤフレーム	110
3.1.16.5 面プロット（surface plot）	111
3.1.16.6 3次元の棒グラフ	112
3.1.16.7 3次元の散布図	113
3.1.17 データの可視化：その他	114
3.1.17.1 ヒートマップ	114
3.1.17.2 表の作成	117
3.1.18 高速フーリエ変換（FFT）	119
3.1.18.1 時間領域から周波数領域への変換：フーリエ変換	119
3.1.18.2 周波数領域から時間領域への変換：フーリエ逆変換	119
3.1.18.3 プロットのアスペクト比の指定	122
3.1.18.4 フーリエ変換を使用する際の注意	122
3.1.19 複素数の計算	123
3.1.19.1 複素数の平方根	123
3.1.19.2 複素数のノルム	123
3.1.19.3 共役複素数	123
3.1.19.4 複素数の偏角（位相）	124
3.1.20 行列、ベクトルの計算（線形代数のための計算）	124
3.1.20.1 行列の和と積	124
3.1.20.2 単位行列、ゼロ行列、他	124
3.1.20.3 行列の要素を全て同じ値にする	125
3.1.20.4 ベクトルの内積	126
3.1.20.5 対角成分、対角行列	126
3.1.20.6 行列の転置	127
3.1.20.7 行列式と逆行列	127
3.1.20.8 固有値と固有ベクトル	127
3.1.20.9 行列のランク	128
3.1.20.10 複素共役行列	128
3.1.20.11 エルミート共役行列	128
3.1.20.12 ベクトルのノルム	128
3.1.21 入出力：配列オブジェクトのファイル I/O	129
3.1.21.1 テキストファイルへの保存	129
3.1.21.2 テキストファイルからの読み込み	131
3.1.21.3 バイナリファイルへの I/O	132
3.1.21.4 データの圧縮保存と読み込み	133
3.1.21.5 配列をバイトデータに変換して入出力に使用する	134
3.1.22 行列の検査	135

3.1.22.1	全て真, 少なくとも 1つは真: all, any	135
3.1.23	配列を処理するユーザ定義関数の実装について	136
3.1.24	行列の計算を応用した計算速度の改善の例	138
3.1.24.1	要素の型によって異なる計算速度	139
3.1.25	画像データの扱い	139
3.1.25.1	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)	141
3.1.25.2	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)	141
3.1.25.3	OpenCV における画素の配列	142
3.1.25.4	サンプルプログラム: 画像の三色分解	143
3.1.26	図形の描画: matplotlib.patches	145
3.1.26.1	四角形, 円, 楕円, 正多角形	145
3.1.26.2	円弧 (椭円弧)	146
3.1.26.3	ポリゴン	147
3.1.27	3 次元のポリゴン表示	147
3.1.28	日付, 時刻の扱い	149
3.1.28.1	datetime64 クラス	149
3.1.28.2	timedelta64 クラス	149
3.1.28.3	日付, 時刻の応用例	150
3.1.29	その他の機能	152
3.1.29.1	基準以上／以下のデータの抽出	152
3.1.29.2	指定した範囲のデータの抽出	154
3.2	科学技術計算用ライブラリ: SciPy	156
3.2.1	信号処理ツール: scipy.signal	156
3.2.1.1	基本的な波形の生成	156
3.2.2	WAV ファイル入出力ツール: scipy.io.wavfile	159
3.2.2.1	WAV 形式ファイル出力: write 関数	159
3.2.2.2	WAV 形式ファイル入力: read 関数	160
3.2.2.3	32-bit floating-point の WAV 形式サウンドデータ	160
3.3	数式処理ライブラリ: SymPy	162
3.3.1	モジュールの読み込みに関する注意	162
3.3.2	基礎事項	162
3.3.2.1	記号オブジェクトの生成	163
3.3.2.2	数式の簡単化 (評価)	163
3.3.2.3	評価なしの数式作成	164
3.3.2.4	数式からのオブジェクトの取り出し	164
3.3.2.5	式 $f(x, y, \dots)$ の構造 (頭部と引数列の取り出し)	165
3.3.2.6	定数	166
3.3.2.7	数式の表示に関すること	166
3.3.3	基本的な数式処理機能	166
3.3.3.1	式の展開	166
3.3.3.2	因数分解	167
3.3.3.3	指定した記号による整理	167
3.3.3.4	約分: 分数の簡単化 (1)	167
3.3.3.5	部分分数	167
3.3.3.6	分数の簡単化 (2)	168
3.3.3.7	代入 (記号の置換)	168
3.3.3.8	各種の数学関数	168
3.3.3.9	式の型	168

3.3.4	解析学的処理	169
3.3.4.1	極限	169
3.3.4.2	導関数	169
3.3.4.3	微分操作の遅延実行	170
3.3.4.4	原始関数	170
3.3.4.5	integrate の遅延実行	170
3.3.4.6	定積分	171
3.3.4.7	級数展開	171
3.3.5	各種方程式の求解	171
3.3.5.1	代数方程式の求解	171
3.3.5.2	微分方程式の求解	172
3.3.5.3	階差方程式の求解（差分方程式、漸化式）	174
3.3.6	線形代数	175
3.3.6.1	行列の連結	175
3.3.6.2	行列の形状	176
3.3.6.3	行列の要素へのアクセス	176
3.3.6.4	行列式	176
3.3.6.5	逆行列	176
3.3.6.6	行列の転置	177
3.3.6.7	ベクトル、内積	177
3.3.6.8	固有値、固有ベクトル	177
3.3.7	総和	178
3.3.8	パターンマッチ	178
3.3.9	数値計算	179
3.3.9.1	素因数分解	179
3.3.9.2	素数	179
3.3.9.3	近似値	180
3.3.9.4	数式を数値化する際の工夫	180
3.3.10	書式の変換出力	183
3.3.10.1	LATEX	183
3.3.10.2	MathML	184
3.3.11	グラフのプロット	184
3.3.11.1	グラフを画像ファイルに保存する方法	185
3.4	多倍長精度の数値演算用ライブラリ：gmpy2	186
3.4.1	基本的なデータ型	186
3.4.2	演算結果の型	190
3.4.3	数学関数	190
3.4.4	数論関連の演算	191
3.4.4.1	mod 演算	191
3.4.4.2	素数の生成	193
3.4.4.3	約数の検査	194
3.4.4.4	最大公約数、最小公倍数	194
3.4.5	乱数生成	195
3.4.6	性能の評価	195
3.4.6.1	整数の算術演算の比較	195
3.4.6.2	有理数の算術演算の比較	196
3.4.6.3	浮動小数点数による特殊関数の算出の比較	197
3.4.6.4	巨大素数の生成	198

3.4.6.5 亂数の品質と生成の速度	199
4 セキュリティ関連	201
4.1 hashlib	201
4.1.1 基本的な使用方法	201
4.2 passlib	201
4.2.1 使用できるアルゴリズム	201
5 プログラムの高速化／アプリケーション構築	203
5.1 Cython	203
5.1.1 使用例	203
5.1.2 高速化のための調整	204
5.2 Numba	205
5.2.1 基本的な使用方法	205
5.2.2 型指定による高速化	206
5.3 ctypes	207
5.3.1 C 言語による共有ライブラリ作成の例	207
5.3.2 共有ライブラリ内の関数を呼び出す例	207
5.3.3 引数と戻り値の扱いについて	208
5.3.3.1 配列データの受け渡し	210
5.4 PyInstaller	213
5.4.1 簡単な使用例	213
5.4.1.1 単一の実行ファイルとしてビルドする方法	214
6 対話作業環境（JupyterLab）	216
6.1 基礎事項	216
6.1.1 起動と終了	217
6.1.2 表示領域の構成と操作方法	217
6.1.2.1 ノートブックの使用例	218
6.1.2.2 カーネル（Kernel）について	220
6.1.3 Notebook での input 関数の実行	220
6.2 Markdown によるコメント表示	220
6.3 使用例	222
6.3.1 MathJax による SymPy の式の整形表示	222
6.3.2 IPython.display モジュールによるサウンドの再生	223
7 psutil ライブラリ	225
7.1 CPU 関連の情報の取得	225
7.1.1 CPU の構成：cpu_count 関数	225
7.1.2 CPU のクロック周波数：cpu_freq 関数	225
7.1.3 CPU 時間：cpu_times 関数	225
7.1.4 CPU 使用率：cpu_percent 関数	226
7.2 メモリ関連の情報の取得	226
7.2.1 仮想記憶の状況：virtual_memory 関数	226
7.2.2 スワップ領域の使用状況：swap_memory 関数	226
7.3 ディスク関連の情報の取得	227
7.3.1 パーティションの構成：disk_partitions 関数	227
7.3.2 ディスクの使用状況：disk_usage 関数	228
7.3.3 ディスクのI/O 状況：disk_io_counters 関数	228

7.4	ネットワーク関連の情報の取得	228
7.4.1	ネットワークの I/O 状況：net.io_counters 関数	228
7.4.2	現在の通信状況：net.connections 関数	229
7.5	プロセス関連の情報の取得	230
7.5.1	実行中のプロセス：pids 関数	230
7.5.2	Process クラス	231
7.5.2.1	メモリの使用状況の調査：memory.info	231
7.5.2.2	CPU 使用率：cpu_percent	232
7.5.2.3	CPU 時間：cpu_times	233
7.5.2.4	開いているファイルの調査：open_files	233
7.5.2.5	プロセスの通信状況：net_connections	233
7.5.2.6	プロセスの終了：terminate	234
7.5.3	サンプルプログラム：プロセスの監視	234
8	その他	236
8.1	json：データ交換フォーマット JSON の使用	236
8.1.1	JSON の表記	236
8.1.2	使用例	236
8.1.2.1	JSON データのファイル I/O	236
8.1.2.2	真理値, None の扱い	237
8.2	urllib：URL に関する処理	238
8.2.1	他バイト文字の扱い（‘%’ エンコーディング）	238
8.2.1.1	文字コード体系の指定	238
8.3	zenhan：全角↔半角変換	239
8.4	jaconv:日本語文字に関する各種の変換	239

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、現在は 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 <http://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、更に詳しい事柄につい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。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

```
1 # coding: utf-8
2 # モジュールのインポート
3 import cv2
4
5 # 動画像入力の開始
6 cap0 = cv2.VideoCapture(0)
7 # フレームレートの取得
8 fps = cap0.get(cv2.CAP_PROP_FPS)
9 # フレームサイズの取得
10 w = cap0.get(cv2.CAP_PROP_FRAME_WIDTH)
11 h = cap0.get(cv2.CAP_PROP_FRAME_HEIGHT)
12 print(fps,'(fps)'); print(w,'*',h)
13
14 # キャプチャと表示
15 while True:
16     # フレームの読み取り
17     (ret,frame) = cap0.read()
18     if ret:
19         # フレームの表示
20         cv2.imshow('Camera: 0',frame)
21         # キーボードの読み取り
22         k = cv2.waitKey(1)
23         if k == 27:      # ESCなら終了
24             break
25         elif k == 67 or k == 99:    # 'C' 'c' なら静止画保存
26             # JPEGのQualityは 0 - 100 : 大きいほど高画質
27             # cv2.imwrite('capture.jpg',frame,[cv2.IMWRITE_JPEG_QUALITY,100])
28             # PNGのQualityは 0 - 9 : 小さいほど高画質
29             cv2.imwrite('capture.png',frame,[cv2.IMWRITE_PNG_COMPRESSION,3])
30     else:
31         print('Camera is not ready.')
32         break
33
34 # 終了処理
35 cap0.release()          # 動画像入力の開放
36 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 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。

1.1.2 ユーザインターフェース

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 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。

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

1.1.3 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。

1.1.4 静止画像の読み込み

`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み込み》

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 # coding: utf-8
2
3 # モジュールのインポート
4 import cv2
5
6 # 画像ファイルの読み込み
7 frame = cv2.imread('Earth.jpg', cv2.IMREAD_UNCHANGED)      # そのまま
8 #frame = cv2.imread('Earth.jpg', cv2.IMREAD_COLOR )        # αチャンネル無視
9 #frame = cv2.imread('Earth.jpg', cv2.IMREAD_GRAYSCALE )     # グレースケールに変換
10
11 # リサイズ
12 fr2 = cv2.resize(frame, (640,640))
13
14 # 表示
15 cv2.imshow('Camera: 0',fr2)
16
17 # 待ち
18 while True:
19     # キーボードの読み取り
20     k = cv2.waitKey(1)
21     if k == 27:      # ESCなら終了
22         break
23
24 # 終了処理
25 cv2.destroyAllWindows() # ウィンドウの消去
```

このプログラムの 12 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
```

1.1.5 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

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。

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。 $0 \leq R \leq 255, 0 \leq G \leq 255, 0 \leq B \leq 255$ 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 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。 $0 \leq H \leq 180, 0 \leq S \leq 255, 0 \leq V \leq 255$ (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) $0 \leq L \leq 255, 1 \leq a \leq 255, 1 \leq b \leq 255$ (値の範囲は OpenCV 独自のもの)

* 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

1.1.5.1 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 matplotlib³ 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「3.1.25.3 OpenCV における画素の配列」(p.142) でも解説する。

例. チャネルの順序に関する確認

```
>>> import cv2 [Enter] ← OpenCV ライブラリの読み込み
>>> im = cv2.imread( 'couple.bmp' ) [Enter] ← 画像ファイル4 の読み込み
>>> 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

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。

⁴標準画像データベース SIDBA から引用。

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 ライブラリの読み込み  
>>> plt.imshow( im ) [Enter] ← matplotlib の機能を用いて画像を表示  
<matplotlib.image.AxesImage object at 0x000001CE791827C8>  
>>> 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  
>>> plt.imshow( im2 ) [Enter] ← matplotlib の機能を用いて画像を表示  
<matplotlib.image.AxesImage object at 0x000001CE7BF26488>  
>>> 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。

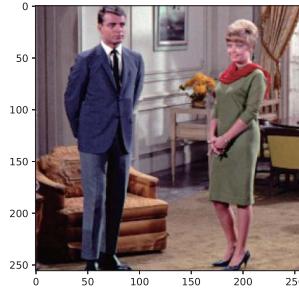


図 2: チャネルの順序を修正後、`matplotlib` で表示

BGR ⇔ RGB 間の変換は、次に説明する `cvtColor` でも可能である。

1.1.5.2 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)`

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。

▲▲ 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。

表 3: cvtColor の引数に与える「変換指定」の値（一部）

変換指定 (cv2 内での定義)	解説
COLOR_BGR2RGB	BGR から RGB への変換
COLOR_RGB2BGR	RGB から BGR への変換
COLOR_BGR2GRAY	BGR からグレースケールへの変換
COLOR_RGB2GRAY	RGB からグレースケールへの変換
COLOR_GRAY2BGR	グレースケールから BGR への変換
COLOR_GRAY2RGB	グレースケールから RGB への変換
COLOR_BGR2HSV	BGR から HSV への変換
COLOR_RGB2HSV	RGB から HSV への変換
COLOR_BGR2LAB	BGR から Lab への変換
COLOR_RGB2LAB	RGB から Lab への変換
COLOR_HSV2BGR	HSV から BGR への変換
COLOR_HSV2RGB	HSV から RGB への変換
COLOR_LAB2BGR	Lab から BGR への変換
COLOR_LAB2RGB	Lab から RGB への変換

1.1.5.3 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 # coding: utf-8
2
3 # モジュールのインポート
4 import cv2
5
6 # 画像ファイルの読み込み
7 frame = cv2.imread('Earth.jpg', cv2.IMREAD_UNCHANGED)
8
9 # リサイズ
10 f = cv2.resize(frame, (320, 320))
11
12 # 色分解
13 (f_b, f_g, f_r) = cv2.split(f)
14
15 # 表示
16 cv2.imshow('Red', f_r)          # 赤の成分
17 cv2.imshow('Green', f_g)        # 緑の成分
18 cv2.imshow('Blue', f_b)         # 青の成分
19
20 # 再度合成
21 f_bgr = cv2.merge((f_b, f_g, f_r))
22 # 表示
23 cv2.imshow("All", f_bgr)
24
25 # 待ち
26 while True:
27     # キー ボードの読み取り

```

```

28     k = cv2.waitKey(1)
29     if k == 27:      # ESCなら終了
30         break
31
32 # 終了処理
33 cv2.destroyAllWindows() # ウィンドウの消去

```

1.1.5.4 色空間の選択に関するこ

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. サンプル画像の読み込みと表示

```

>>> import cv2 [Enter] ← OpenCV ライブラリの読み込み
>>> import matplotlib.pyplot as plt [Enter] ← matplotlib ライブラリの読み込み
>>> imBGR = cv2.imread('Pepper.bmp') [Enter] ← サンプル画像5 の読み込み
>>> imRGB = cv2.cvtColor(imBGR, cv2.COLOR_BGR2RGB) [Enter] ← RGB に変換
>>> plt.imshow(imRGB) [Enter] ← matplotlib で画像を表示
<matplotlib.image.AxesImage object at 0x00000288FB0F5D88>
>>> 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。

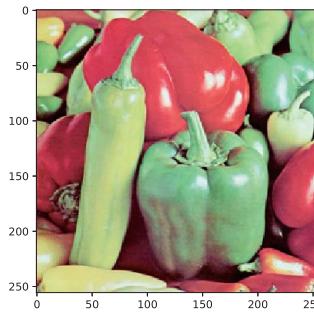


図3: サンプル画像 'Pepper.bmp'

次に、この画像の色空間をHSVに変換する。

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')
>>> a2 = ax[1].imshow(imS, cmap=plt.cm.gray) [Enter] ← S チャネルの作図
>>> t2 = ax[1].set_title('Saturation')
>>> a3 = ax[2].imshow(imV, cmap=plt.cm.gray) [Enter] ← V チャネルの作図
>>> t3 = ax[2].set_title('Value')
>>> 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。

次に、色空間をLabに変換する。

⁵標準画像データベース SIDBA から引用。

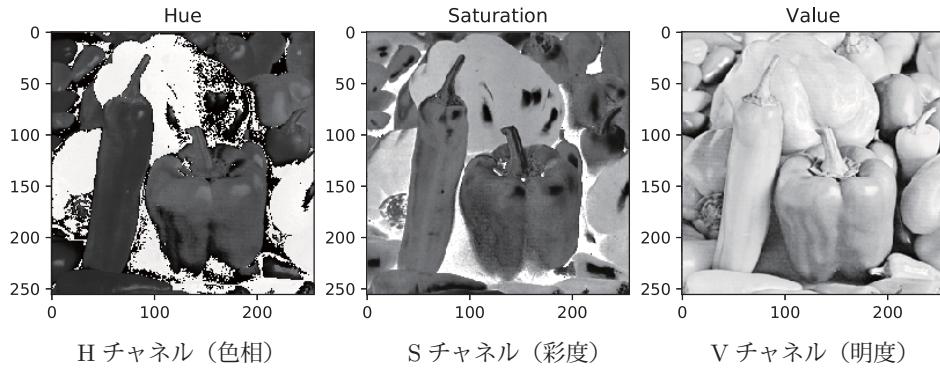


図 4: HSV に変換した後の各チャネル

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] ← HSV 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*')
>>> a2 = ax[1].imshow( ima, cmap=plt.cm.gray ) [Enter] ← a チャネルの作図
>>> t2 = ax[1].set_title('a*')
>>> a3 = ax[2].imshow( imb, cmap=plt.cm.gray ) [Enter] ← b チャネルの作図
>>> t3 = ax[2].set_title('b*')
>>> 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。

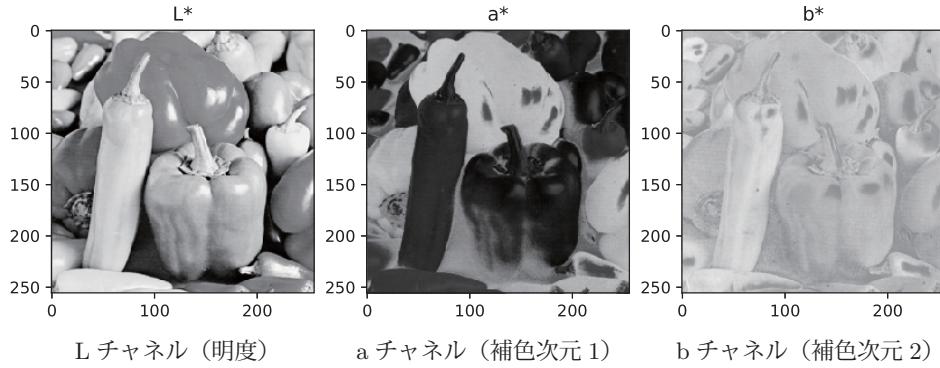


図 5: Lab に変換した後の各チャネル

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選択する。

1.1.5.5 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 $0^\circ \sim 360^\circ$ の円環構造となるが、OpenCV では $0 \sim 180$ の範囲の円環構造 (図 6) で表現する。

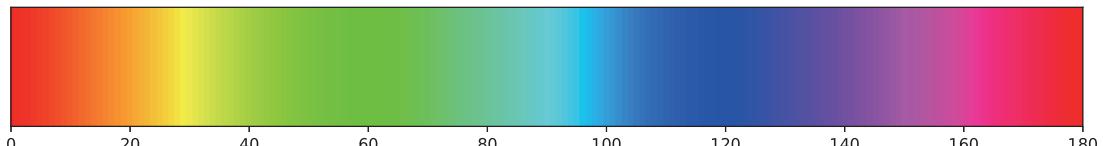


図 6: OpenCV における HSV 空間の色相 (H チャネル)

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 # coding: utf-8
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import cv2
5
6 # 0～180の色相を図示するための画像
7 H = np.array([list(range(0,181))*20]).astype('uint8')    # 0 <= H <= 180
8 S = np.asarray([[255]*181]*20).astype('uint8')           # 0～255
9 V = np.asarray([[255]*181]*20).astype('uint8')           # 0～255
10
11 imHSV = cv2.merge((H,S,V))                                # 上で作成した各チャネルを合成して
12 imRGB = cv2.cvtColor(imHSV, cv2.COLOR_HSV2RGB)            # 表示用にRGBに変換
13
14 plt.figure(figsize=(12,2))
15 plt.imshow(imRGB)
16 plt.xlim(0,180)
17 plt.yticks(ticks=[])
18 plt.show()
```

1.1.6 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 ライブラリの読み込み
>>> import matplotlib.pyplot as plt [Enter] ← matplotlib ライブラリの読み込み
>>> im = cv2.imread('couple.bmp') [Enter] ← 画像ファイル6 の読み込み
>>> imGr = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY) [Enter] ← モノクロ画像への変換
>>> plt.imshow(imGr, cmap=plt.cm.gray) [Enter] ← 画像の表示
<matplotlib.image.AxesImage object at 0x000002214219BD08>
>>> 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。

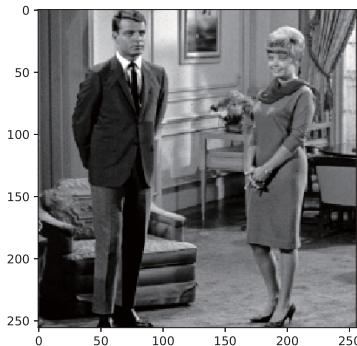


図 7: カラー画像からモノクロ画像への変換

1.1.6.1 モノクロ 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_OTSU)

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

⁶標準画像データベース SIDBA から引用。

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, 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')
>>> a2 = ax[1].imshow( imBW2, cmap=plt.cm.gray ) [Enter] ← 表示処理 (2)
>>> t2 = ax[1].set_title('THRESH_OTSU')
>>> 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。

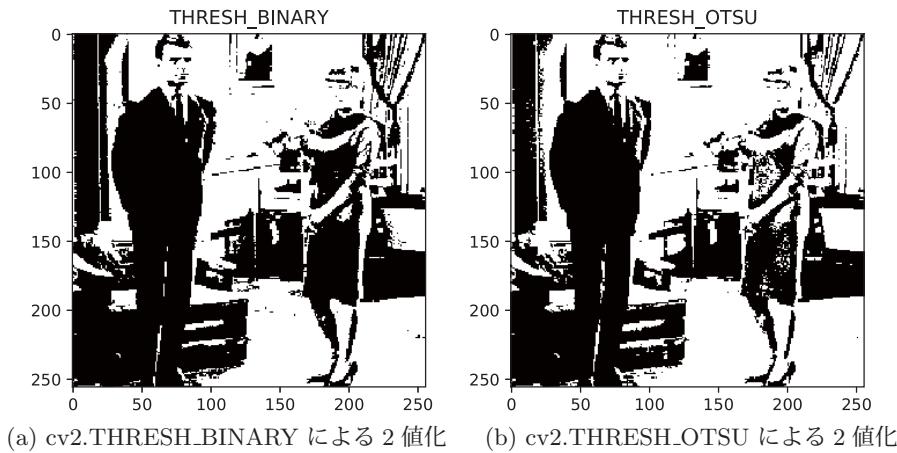


図 8: グレースケールを 2 値化した画像

1.1.6.2 モノクロ 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,
...                                     cv2.THRESH_BINARY, n, 15 )
...
...     a = ax[i].imshow( imAdpt, cmap=plt.cm.gray ) [Enter] ← 表示処理
...     t = ax[i].set_title('Block:'+str(n))
...
...     [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。

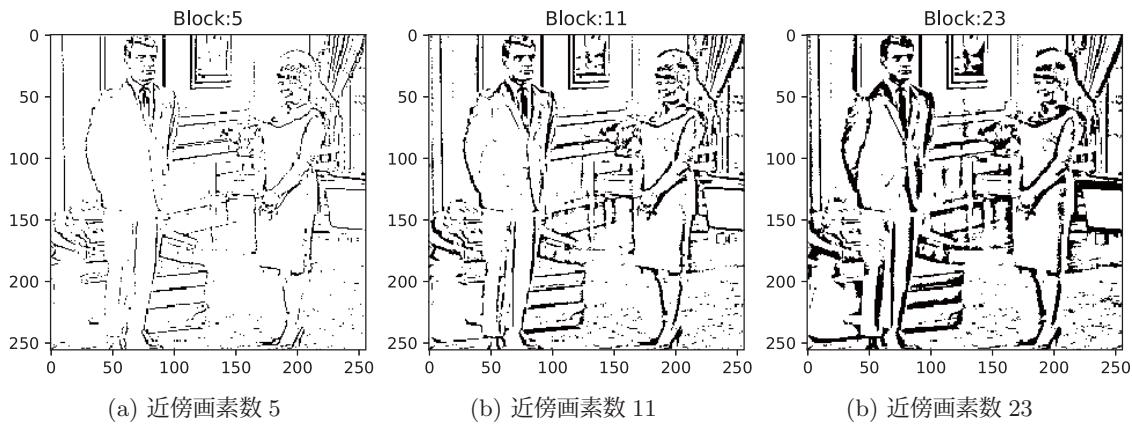


図 9: adaptiveThreshold による 2 値化

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み込み

```
>>> import cv2 [Enter] ← OpenCV の読み込み
>>> import matplotlib.pyplot as plt [Enter] ← matplotlib の読み込み
>>> imGr = cv2.imread( 'grayscale01.jpg', cv2.IMREAD_GRAYSCALE ) [Enter] ← 画像の読み込み
>>> plt.imshow( imGr, cmap=plt.cm.gray ) [Enter] ← 画像の表示処理
<matplotlib.image.AxesImage object at 0x0000029CAEF40E48>
>>> 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。

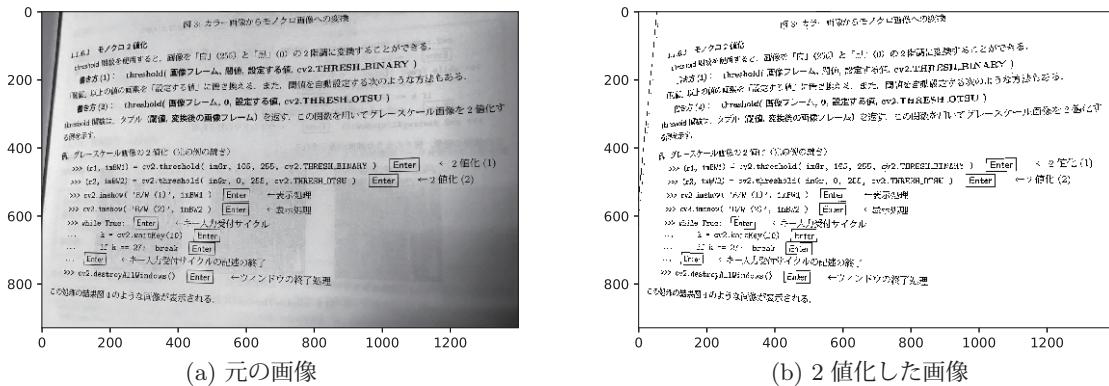


図 10: 影の写り込んだ文書画像の 2 値化

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,
...                                 cv2.THRESH_BINARY, 5, 19 ) [Enter] ← 2 値化の処理
>>> plt.imshow( imAdpt, cmap=plt.cm.gray ) [Enter] ← 画像の表示処理
<matplotlib.image.AxesImage object at 0x0000029CAEE1EE08>
>>> 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。

1.1.7 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。

1.1.7.1 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 の読み込み
>>> import matplotlib.pyplot as plt [Enter] ← matplotlib の読み込み
>>> import cv2 [Enter] ← OpenCV の読み込み
>>> im = np.full( (30,50,3), 255 ) [Enter] ← 全ての画素が白の画像フレームを作成
>>> plt.imshow( im ) [Enter] ← 画像の表示
<matplotlib.image.AxesImage object at 0x000001A32C93E488>
>>> plt.show() [Enter] ← 描画の実行
```

この結果、図11の(a)のような画像が表示される。

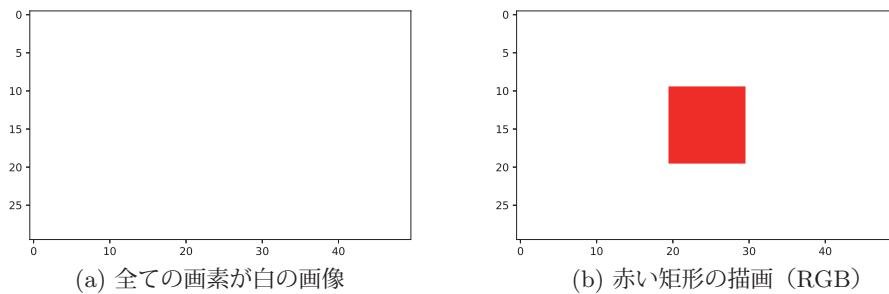


図11: 画素を直接設定して描画

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
>>> plt.imshow( im ) [Enter] ← 画像の表示
<matplotlib.image.AxesImage object at 0x000001A32E91EF08>
>>> 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)のような画像が表示される。

注意) 上に示した例はRGB色空間によるものである。

これを応用した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、それらについて解説する。

1.1.7.2 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。「描画手法」にはcv2.LINE_AAを指定することが推奨されている。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。

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 (cv2,plt は読み込み済みであるとする)

```
>>> imRGB = np.full( (360,640,3), 255 ) [Enter] ←白の画素の配列を作成
>>> 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。

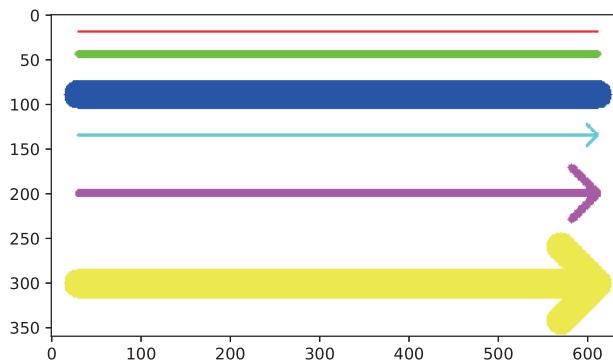


図 12: 線分と矢印の描画

1.1.7.3 矩形、円、橢円、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 (cv2,plt は読み込み済みであるとする)

```
>>> imRGB = np.full( (150,410,3), 255 ) [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。

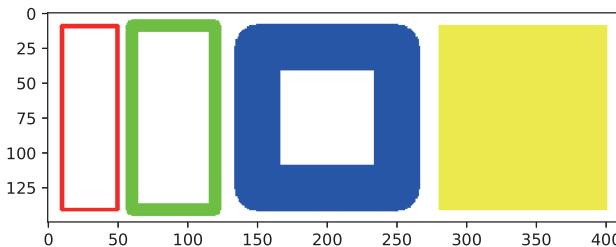


図 13: 矩形の描画

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 (cv2,plt は読み込み済みであるとする)

```
>>> imRGB = np.full( (100,200,3), 255 ) [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。

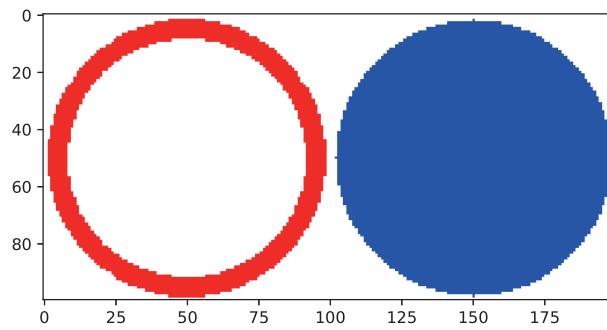


図 14: 円の描画

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 (cv2,plt は読み込み済みであるとする)

```
>>> imRGB = np.full( (100,200,3), 255 ) [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。

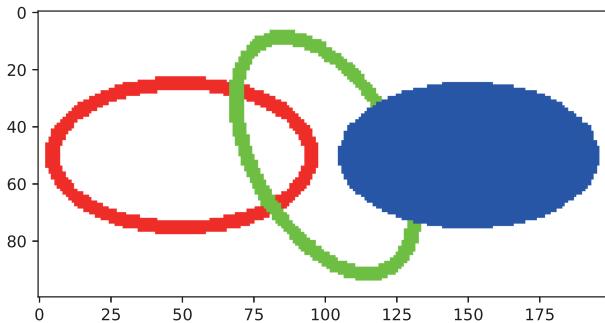


図 15: 楕円の描画

`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) に準じる。

例. 楕円弧の描画 (cv2, plt は読み込み済みであるとする)

```
>>> imRGB = np.full( (80,270,3), 255 ) [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] ←赤い楕円弧の描画
...           thickness=4, lineType=cv2.LINE_AA ) [Enter]
>>> im = cv2.ellipse( imRGB, (150,40), (50,30), 0, [Enter]
...           120, 330, (0,255,0), [Enter] ←緑の楕円弧の描画
...           thickness=4, lineType=cv2.LINE_AA ) [Enter]
>>> im = cv2.ellipse( imRGB, (232,40), (50,30), -30, [Enter]
...           120, 330, (0,0,255), [Enter] ←青い楕円弧の描画
...           thickness=4, lineType=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 16 のような画像フレームが表示される。

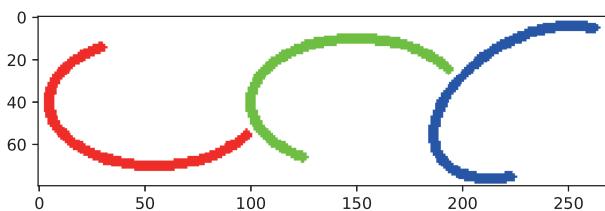


図 16: 楕円弧の描画

1.1.7.4 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 $[[x_1, y_1], [x_2, y_2], \dots, [x_n, y_n]]$ の形の配列であり、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折れ線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` に準じる。

例. 各種の折れ線の描画 (cv2,plt は読み込み済みであるとする)

```
>>> imRGB = np.full( (90,150,3), 255 ) [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] ←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] ←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],[110,70],[110,50],[80,70]]) [Enter] ←折れ線 3 の座標データ  
>>> im = cv2.fillPoly( imRGB, [pl3], (0,255,0) ) [Enter] ←折れ線 3 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。

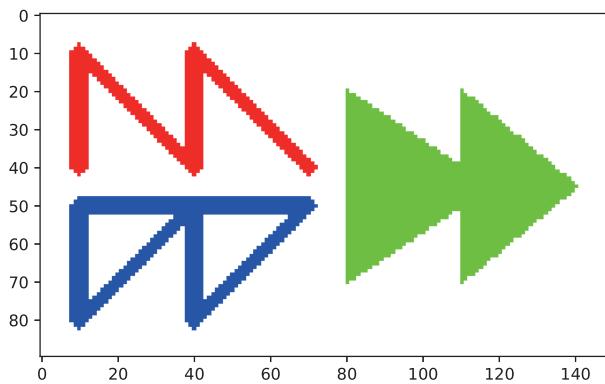


図 17: 折れ線の描画

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 に準じる。

例. 多角形の描画 (cv2,plt は読み込み済みであるとする)

```
>>> imRGB = np.full( (100,120,3), 255 ) [Enter] ←白の画素の配列を作成  
>>> pl1 = np.array([[60,10],[110,35],[110,65],[60,90],[10,65],[10,35]]) [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。

1.1.7.5 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。「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 4 に挙げる。

表 4 に挙げたフォントで文字列を描画する例を示す。

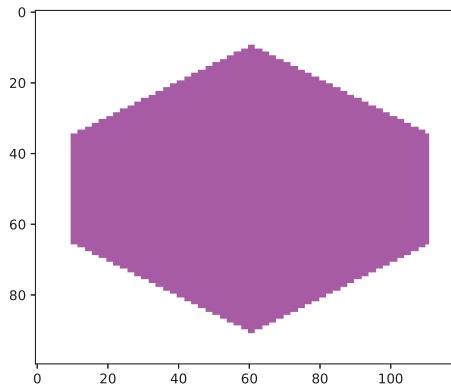


図 18: 多角形（凸）の描画

表 4: 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		

例. 文字列の描画 (cv2,plt は読み込み済みであるとする)

```
>>> imRGB = np.full( (240,540,3), 255 ) [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)
>>> 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)
>>> 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)
>>> 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)
>>> 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)
>>> 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)
>>> 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)
>>> 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。

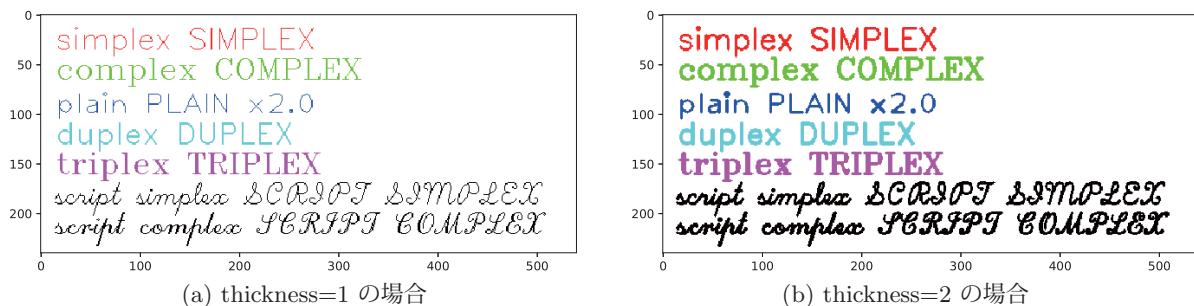


図 19: 文字列の描画 (thickness の値で太さを調整できる)

1.1.7.6 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, lineType=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。「マーカーの種類」に指定できるものを表 5 に挙げる。

表 5: 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 5 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 は読み込み済みであるとする)

```
>>> imRGB = np.full( (100,190,3), 127 ) [Enter] ←グレーの画素の配列を作成
>>> im = cv2.drawMarker( imRGB, (30,25), (255,0,0),
...     markerType=cv2.MARKER_CROSS, markerSize=20, [Enter] ←「+」の描画
...     thickness=1, line_type=cv2.LINE_AA )

>>> im = cv2.drawMarker( imRGB, (70,25), (0,255,0),
...     markerType=cv2.MARKER_TILTED_CROSS, markerSize=20, [Enter] ←「×」の描画
...     thickness=1, line_type=cv2.LINE_AA )

>>> im = cv2.drawMarker( imRGB, (110,25), (0,0,255),
...     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),
...     markerType=cv2.MARKER_DIAMOND, markerSize=20, [Enter] ←「◇」の描画
...     thickness=1, line_type=cv2.LINE_AA )

>>> im = cv2.drawMarker( imRGB, (30,65), (255,0,255),
...     markerType=cv2.MARKER_SQUARE, markerSize=20, [Enter] ←「□」の描画
...     thickness=1, line_type=cv2.LINE_AA )

>>> im = cv2.drawMarker( imRGB, (70,65), (255,255,0),
...     markerType=cv2.MARKER_TRIANGLE_UP, markerSize=20, [Enter] ←「△」の描画
...     thickness=1, line_type=cv2.LINE_AA )

>>> im = cv2.drawMarker( imRGB, (110,65), (255,255,255),
...     markerType=cv2.MARKER_TRIANGLE_DOWN, markerSize=20, [Enter] ←「▽」の描画
...     thickness=1, line_type=cv2.LINE_AA )

>>> 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。

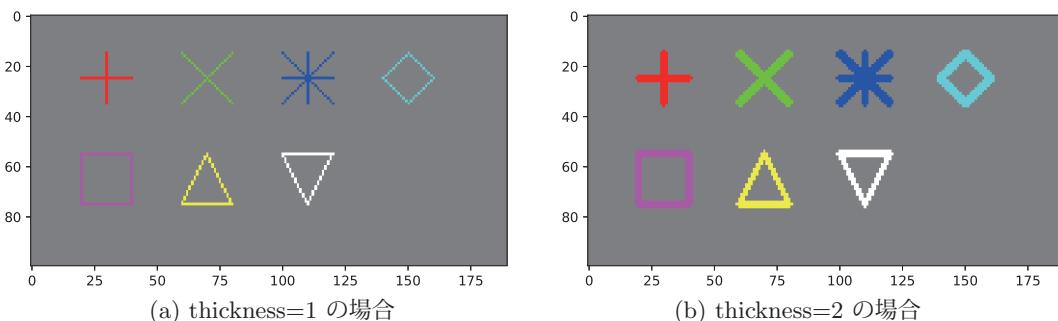


図 20: マーカーの描画 (thickness の値で太さを調整できる)

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
```

1.2.1 画像ファイルの読み込みと保存

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 の読み込み。

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

＜画像ファイルの読み込み＞

書き方： Image.open(ファイル名, モード)

モードの指定は省略可能であるが、指定するばあいは 'r' を与える。この結果、画像データが Image オブジェクトとして返される。ファイル名（パス名）は拡張子を伴う文字列で与える。

Pillow のバージョンが 4.1 の場合に読み書きがサポートされている画像フォーマットは次の通りである。

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

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。

画像の各種プロパティの調査

```
>>> from PIL import Image [Enter] ← パッケージの読み込み  
>>> im = Image.open('Earth.jpg') [Enter] ← 画像ファイルの読み込み  
>>> 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, 300) ← 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 6 のような種類がある。

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。

表 6: 画像モード

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

1.2.1.1 EPS を読み込む際の解像度

EPS (Encapsulated PostScript) はベクトル図形であり、Image オブジェクトとして読み込む際にラスタライズ⁷される。このときの解像度は暗黙で 72dpi 程度（標準的なディスプレイの解像度）となっている。更に高い解像度で EPS を読み込むには、open で EPS ファイルを読み込んだ直後に、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=倍率’ を与えてることで、暗黙の解像度に対する「倍率」を適用した形で再度ライタライズされる。（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)                  ← 4 倍の解像度で再度ラスタライズ
```

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(ファイル名, 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。ファイル名（パス名）は拡張子を伴う文字列で与える。

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 7: 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
	dpi=(x,y)	横方向、縦方向それぞれの解像度
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
	dpi=整数値	画像の解像度

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, サイズ, 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。

⁷ラスタライズ：画素に展開すること。

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。画像ビューワが開いている間は `show` メソッド実行部分で当該スレッドがブロックし、画像ビューワが終了するのを待つ。

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。

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 8）を指定することができる。

リサイズにおけるフィルタ指定の例

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

表 8: 各種フィルタ

フィルタ	効果
<code>Image.NEAREST</code>	画素の補間に近傍の画素を使用する（デフォルト）
<code>Image.BOX</code>	画素の補間にボックス近似を使用する
<code>Image.BILINEAR</code>	画素の補間に線形近似を使用する
<code>Image.HAMMING</code>	線形近似より若干鮮明な補間処理
<code>Image.BICUBIC</code>	画素の補間に 3 次補間を使用する
<code>Image.LANCZOS</code>	Lanczos フィルタによる補間処理

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。

```
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。

複製の例. 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 8 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 9 に示す。

表 9: 回転・反転の手法

method	処理
<code>Image.FLIP_LEFT_RIGHT</code>	左右反転
<code>Image.FLIP_TOP_BOTTOM</code>	上下反転
<code>Image.ROTATE_90</code>	反時計回り 90 度回転
<code>Image.ROTATE_180</code>	180 度回転
<code>Image.ROTATE_270</code>	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 6 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ものである。

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.21 の表 6）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. カラー画像の読み込みと表示

```
>>> from PIL import Image [Enter] ← Pillow ライブラリの読み込み  
>>> im = Image.open('couple.bmp') [Enter] ← 画像ファイルの読み込み  
>>> import numpy as np [Enter] ← NumPy ライブラリの読み込み  
>>> 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 ライブラリの読み込み  
>>> plt.imshow(imA) [Enter] ← 配列に変換した画像を表示  
<matplotlib.image.AxesImage object at 0x000001F730A04488>  
>>> 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。

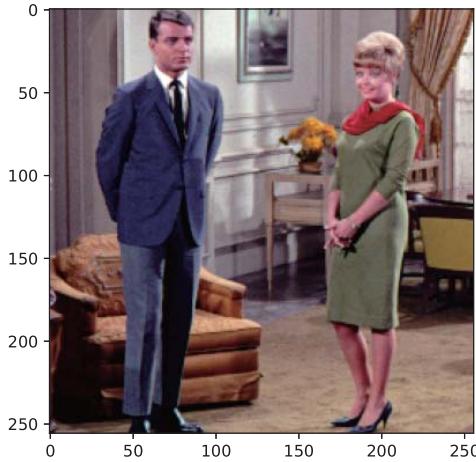


図 21: カラー画像

※ 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  
<matplotlib.image.AxesImage object at 0x000001F7344F1388>  
>>> 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。

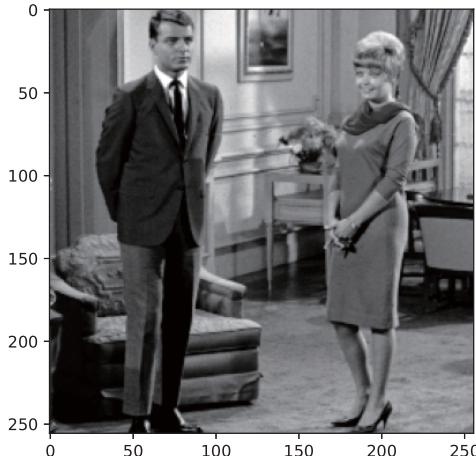


図 22: グレースケールに変換した画像

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 # coding: utf-8  
2 # モジュールの読み込み  
3 from PIL import Image  
4 from PIL import ImageDraw  
5  
6 # 白いImageオブジェクトの生成  
7 im = Image.new( 'RGB', (640,480), (255,255,255) )  
8 # Drawオブジェクトの取得  
9 drw = ImageDraw.Draw(im)  
10  
11 # 楕円の描画
```

```

12 |     drw.ellipse( [100,100,539,379], fill=(0,0,255) )
13 |     # 直線の描画
14 |     drw.line( [0,0,639,479], fill=(255,0,0), width=32 )
15 |
16 |     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おり、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。

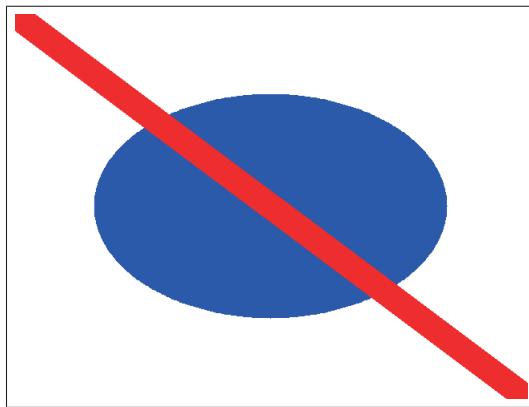


図 23: 実行結果

【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 10）する。

表 10: 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⁸ 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「3.1.25 画像データの扱い」(p.139) で解説する。

⁸ 「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、SDL⁹ 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
```

また、pygame 配下の各種モジュールを読み込むことが必要になることがある。

例. 終了イベントを意味する定数 QUIT を読み込む

```
from pygame.locals import QUIT
```

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。

初期化の例。

```
pygame.init()
```

2.1.1.1 Surface オブジェクト

pygame では、画像データなどのピットマップを **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.

```
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 ライブラリと共に通である。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¹⁰ 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。

⁹SDL (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), OSX(Apple), Linux, iOS(Apple), Android(Google) といった OS で利用できる。

¹⁰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。

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 タイムの不必要的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。

```
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 # coding: utf-8
2
3 # モジュールの読み込み
4 import sys
5 import pygame
6 from pygame.locals import QUIT
7
8 # pygameの初期化
9 pygame.init()
10
11 w = 400; h = 300
12 sf = pygame.display.set_mode( (w,h) )    # アプリケーションウィンドウ
13 pygame.display.set_caption('Application: pygame00.py')
14
15 fps = pygame.time.Clock()    # フレームレート制御のための Clock オブジェクト
16
17 # 画像の読み込み
18 im1 = pygame.image.load('ball01.jpg')
19 im1_w = im1.get_width()      # 画像の横幅の取得
20 im1_h = im1.get_height()    # 画像の高さの取得
21
22 # メインループ
23 x = 0; y = 0    # ボールの位置
24 dx = 3; dy = 2 # 移動量
25 while True:
26     sf.fill( (255,255,255) )    # 背景の色
27     sf.blit(im1, (x,y))        # ボールの描画
28     # イベントキューを処理するループ
29     for ev in pygame.event.get():
30         if ev.type == QUIT:      # 「終了」イベント

```

```

31         pygame.quit()
32         print('quitting...')
33         sys.exit()
34     # ディスプレイの更新
35     pygame.display.update()
36     # フレームレートの設定
37     fps.tick(30)      # 30FPSに設定
38     # ボール移動（位置変更）の処理
39     x += dx; y += dy    # 移動
40     if x + im1_w > w:    # ボールが右端に衝突した場合の処理
41         x = w - im1_w - 1
42         dx *= -1
43     elif x < 0:          # ボールが左端に衝突した場合の処理
44         x = 0
45         dx *= -1
46     if y + im1_h > h:    # ボールが床に衝突した場合の処理
47         y = h - im1_h - 1
48         dy *= -1
49     elif y < 0:          # ボールが天井に衝突した場合の処理
50         y = 0
51         dy *= -1

```

プログラムの 18~20 行目でボールの画像 ball01.jpg を load メソッドで読み込んで、その幅と高さを取得 (get_width メソッド, get_height メソッド) している。

26~27 行目でボールの画像をアプリケーションウィンドウに表示し、29~33 行目でイベントハンドリングを行っている。ここでは QUIT イベント（アプリケーションウィンドウの閉じるボタンをクリックしたときに発生）を検出してアプリケーションの終了処理をハンドリングするのみとしている。アプリケーションには短時間に複数のイベントが発生し、それらはイベントキューと呼ばれる待ち行列に蓄積される。29 行目の for 文は、イベントキューからイベントを 1 つずつ取り出し、それを順番にハンドリングするループとなっている。イベントキューのイベント列を取り出すには pygame.event.get() を実行する。イベントキューから取り出された個々のイベントはイベント種別をはじめとする各種のプロパティが保持されている。イベント種別は pygame.locals の中に定数として定義されており、プログラムの 6 行目にあるような形で読み込んで使用するのが一般的である。

35 行目では update メソッドを用いてウィンドウの更新を行い、37 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。（引数にフレームレートを与える）

画像の描画は 27 行目にあるように blit メソッドを用いる。（詳しくは後述）

13 行目の set_caption メソッドはウィンドウのタイトルを設定する。

30 行目にあるように、QUIT イベントを受けてこのアプリケーションは終了する。このイベントはアプリケーションウィンドウの閉じるボタンをクリックしたときに発生する。この際の pygame の終了処理として quit メソッドを実行する。この後、sys.exit() を呼び出してアプリケーションを終了させている。

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。

■ 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

■ 横円

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

■ 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

■ 画像

ファイルから読み込み : 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。

■ 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 # coding: utf-8
2
3 # モジュールの読み込み
4 import pygame
5
6 # pygameの初期化
7 pygame.init()
8
9 # フォントリストの取得と表示
10 fl = pygame.font.get_fonts()
11 for m in fl:
12     print(m)
```

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
```

2.1.2.1 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

```
1 # coding: utf-8
2
3 # モジュールの読み込み
4 import sys
5 import pygame
6 from pygame.locals import QUIT
7
```

```

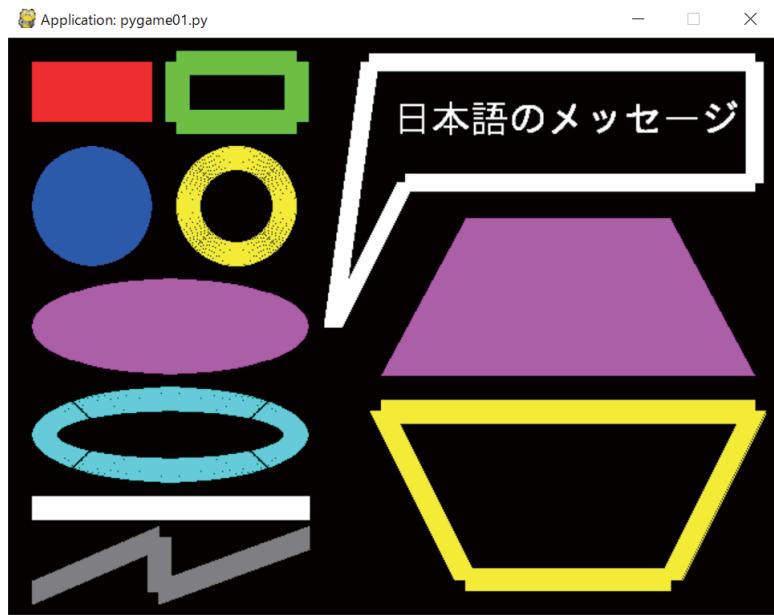
8 # pygameの初期化
9 pygame.init()
10
11 sf = pygame.display.set_mode( (640,480) )    # アプリケーションウィンドウ
12 pygame.display.set_caption('Application: pygame01.py')
13
14 fps = pygame.time.Clock()      # フレームレート制御のための Clock オブジェクト
15
16 # メインループ
17 while True:
18     # 四角形（塗りつぶし）の描画
19     pygame.draw.rect(sf, (255,0,0), (20,20,100,50))
20     # 四角形（枠）の描画
21     pygame.draw.rect(sf, (0,255,0), (140,20,100,50), 20 )
22
23     # 円（塗りつぶし）の描画
24     pygame.draw.circle(sf, (0,0,255), (70,140), 50 )
25     # 円（線による円周）の描画
26     pygame.draw.circle(sf, (255,255,0), (190,140), 50, 20 )
27
28     # 橋円（塗りつぶし）の描画
29     pygame.draw.ellipse(sf, (255,0,255), (20,200,230,80) )
30     # 橋円（線による周）の描画
31     pygame.draw.ellipse(sf, (0,255,255), (20,290,230,80), 20 )
32
33     # 線分の描画
34     pygame.draw.line(sf, (255,255,255), (20,390),(250,390), 20 )
35
36     # 折れ線の描画（開放端）
37     plst = [(20,460), (125,415), (125,460), (250,415)]
38     pygame.draw.lines(sf, (127,127,127), False, plst, 20 )
39     # 折れ線の描画（始点と終点を結ぶ）
40     plst = [(270,240), (300,20), (620,20), (620,120), (330,120)]
41     pygame.draw.lines(sf, (255,255,255), True, plst, 15 )
42
43     # 文字の描画（システムフォント）
44     fnt = pygame.font.SysFont('ipaゴシック',32)      # システムフォント
45     txt = fnt.render('日本語のメッセージ', True, (255,255,255) )
46     txt_rct = txt.get_rect()           # テキストオブジェクトの領域サイズ
47     sf.blit(txt,(320,50))
48
49     # ポリゴンの描画（塗りつぶし）
50     plst = [(310,280), (380,150), (550,150), (620,280)]
51     pygame.draw.polygon(sf, (255,0,255), plst )
52
53     # ポリゴンの描画（線による周）
54     buf = pygame.Surface( (330,160) )    # バッファ
55     plst = [(10,10), (80,150), (250,150), (320,10)] # バッファ上の座標
56     pygame.draw.polygon(buf, (255,255,0), plst, 20 )   # バッファに描画
57     sf.blit( buf, (300,300) )  # バッファをウィンドウへ
58
59     # イベントキューを処理するループ
60     for ev in pygame.event.get():
61         if ev.type == QUIT:      # 「終了」イベント
62             pygame.quit()
63             print('quitting... ')
64             sys.exit()
65
66     # ディスプレイの更新
67     pygame.display.update()
68     # フレームレートの設定
69     fps.tick(4) # 4FPSに設定

```

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。

■ 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作成して、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。



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

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.

2.1.2.2 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 # coding: utf-8
2
3 # モジュールの読み込み
4 import sys
5 import pygame
6 from pygame.locals import QUIT
7
8 # pygameの初期化
9 pygame.init()
10
11 sf = pygame.display.set_mode( (320,240) ) # アプリケーションウィンドウ
12 pygame.display.set_caption('Application: pygame02.py')
13
14 fps = pygame.time.Clock() # フレームレート制御のための Clock オブジェクト
15
16 # 画像の読み込み
17 im1 = pygame.image.load('pygame_logo.png')
18 agl = 0.0 # 初期角度

```

```

19
20 # メインループ
21 while True:
22     sf.fill( (0,0,0) ) # 毎フレームクリアする
23     # 画像の回転と表示
24     im2 = pygame.transform.rotate(im1, agl%360) # 回転処理
25     sf.blit(im2, (50,10) ) # ウィンドウに転送
26     agl += 2.4 # 次回の角度
27
28     # イベントキューを処理するループ
29     for ev in pygame.event.get():
30         if ev.type == QUIT: # 「終了」イベント
31             pygame.quit()
32             print('quitting...')
33             sys.exit()
34
35     # ディスプレイの更新
36     pygame.display.update()
37     # フレームレートの設定
38     fps.tick(30) # 30FPSに設定

```

プログラム：pygame04.py

```

1 # coding: utf-8
2
3 # モジュールの読み込み
4 import sys
5 import pygame
6 from pygame.locals import QUIT
7
8 # pygameの初期化
9 pygame.init()
10
11 sf = pygame.display.set_mode( (320,120) ) # アプリケーションウィンドウ
12 pygame.display.set_caption('Application: pygame04.py')
13
14 fps = pygame.time.Clock() # フレームレート制御のための Clock オブジェクト
15
16 # 画像の読み込み
17 im1 = pygame.image.load('pygame_logo.png')
18 im1_w = im1.get_width()
19 im1_h = im1.get_height()
20 rat = 0.0 # 初期比率
21 dr = 0.02
22
23 # メインループ
24 while True:
25     sf.fill( (0,0,0) ) # 每フレームクリアする
26     # 画像の回転と表示
27     im2 = pygame.transform.scale(im1,(int(rat*im1_w),int(rat*im1_h))) # 拡縮処理
28     sf.blit(im2, (10,10) ) # ウィンドウに転送
29     rat += dr # 次回の比率
30     if rat > 1.5:
31         rat = 1.5
32         dr *= -1
33     elif rat < 0.0:
34         rat = 0.0
35         dr *= -1
36
37     # イベントキューを処理するループ
38     for ev in pygame.event.get():
39         if ev.type == QUIT: # 「終了」イベント
40             pygame.quit()
41             print('quitting...')
42             sys.exit()
43
44     # ディスプレイの更新
45     pygame.display.update()
46     # フレームレートの設定
47     fps.tick(30) # 30FPSに設定

```

■ 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11に挙げる。これらイベントは `pygame.locals` の中に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 11: 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	<code>pos,button</code> : 位置とボタン
MOUSEBUTTONUP	マウスのボタンが放された	<code>pos,button</code> : 位置とボタン
MOUSEMOTION	マウスが動いた	<code>pos,buttons</code> : 位置とボタンタプル
KEYDOWN	キーボードが押された	<code>key,mod,unicode</code> : キー番号とモディファイア, 文字コード体系（エンコーディング）
KEYUP	キーボードが放された	<code>key,mod</code> : キー番号とモディファイア

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`

```
1 # coding: utf-8
2
3 # モジュールの読み込み
4 import sys
5 import pygame
6 from pygame.locals import QUIT, MOUSEBUTTONDOWN, MOUSEBUTTONUP, \
7     MOUSEMOTION, KEYDOWN, KEYUP
8
9 # pygameの初期化
10 pygame.init()
11
12 sf = pygame.display.set_mode( (320,240) )    # アプリケーションウィンドウ
13 pygame.display.set_caption('Application: pygame05.py')
14
15 fps = pygame.time.Clock()      # フレームレート制御のための Clock オブジェクト
16
17 # メインループ
18 while True:
19     # イベントキューを処理するループ
20     for ev in pygame.event.get():
21         if ev.type == QUIT:      # 「終了」イベント
22             pygame.quit()
23             print('quitting...')
24             sys.exit()
25         elif ev.type == MOUSEBUTTONDOWN:
26             print('Mouse button was pressed:\t',ev.pos, ev.button)
27         elif ev.type == MOUSEBUTTONUP:
28             print('Mouse button was released:\t',ev.pos, ev.button)
29         elif ev.type == MOUSEMOTION:
30             print('Mouse is moving:\t\t',ev.pos, ev.buttons)
31         elif ev.type == KEYDOWN:
32             print('A key was pressed:\t\t',ev.key, ev.mod, ev.unicode)
33         elif ev.type == KEYUP:
34             print('The key was released:\t\t',ev.key, ev.mod)
35
36     # ディスプレイの更新
37     pygame.display.update()
38     # フレームレートの設定
39     fps.tick(12)      # 12 FPSに設定
```

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、MP3¹¹ や Ogg Vorbis¹² 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' を読み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 12 に挙げる。

表 12: 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
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 を示す。

プログラム：pygame06.py

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

¹¹MP3 (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。厳密には、特許に関連する事情のため、完全に自由に使える技術ではない。したがって、商用の利用には注意が必要である。

¹²Ogg 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。

```

30     if pygame.mixer.music.get_busy():
31         pass
32     else:
33         print('music has finished.')
34         pygame.quit()
35         sys.exit()
36
37     print('Playing:',pygame.mixer.music.get_pos(),'msec')
38
39     # ディスプレイの更新
40     pygame.display.update()
41     # フレームレートの設定
42     fps.tick(2) # 2FPSに設定

```

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。

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も音声の再生ができる。

Sound オブジェクトの生成： `pygame.mixer.Sound('音声ファイルのパス')`

この処理の結果として Sound オブジェクトが得られ、表 13 のようなメソッドが使用できる。

表 13: 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
<code>play()</code>	再生開始	<code>stop()</code>	再生終了
<code>get_volume()</code>	音量の取得	<code>set_volume()</code>	音量の設定
<code>get_length()</code>	再生時間の取得		

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 # coding: utf-8
2
3 # モジュールの読み込み
4 import sys
5 import pygame
6 from pygame.locals import QUIT
7
8 # pygameの初期化
9 pygame.init()
10
11 sf = pygame.display.set_mode((320,240)) # アプリケーションウィンドウ
12 pygame.display.set_caption('Application: pygame07.py')
13
14 fps = pygame.time.Clock() # フレームレート制御のための Clock オブジェクト
15
16 # サウンドデータの読み込みと再生
17 snd = pygame.mixer.Sound('sound02.ogg')
18 print('長さ:', snd.get_length())
19 print('音量:', snd.get_volume())
20 snd.play()
21
22 # メインループ
23 while True:
24     # イベントキューを処理するループ
25     for ev in pygame.event.get():
26         if ev.type == QUIT: # 「終了」イベント
27             pygame.quit()
28             print('quitting...')
29             sys.exit()
30

```

```

31 # 再生が終了したときの処理
32 if pygame.mixer.get_busy():
33     pass
34 else:
35     print('music has finished.')
36     pygame.quit()
37     sys.exit()
38
39 # ディスプレイの更新
40 pygame.display.update()
41 # フレームレートの設定
42 fps.tick(2) # 2FPSに設定

```

このプログラムの32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NumPy¹³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。

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 # coding: utf-8
2
3 # モジュールの読み込み
4 import pygame
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8 # pygameの初期化
9 pygame.mixer.pre_init(frequency=44100, size=-16, channels=2, buffer=4096)
10 pygame.init()
11
12 # サウンドデータの読み込み
13 snd = pygame.mixer.Sound('aaa.wav')
14 print('長さ:', snd.get_length())
15 print('音量:', snd.get_volume())
16
17 # NumPy配列への変換
18 ar = pygame.sndarray.array(snd)
19 print('配列の形状:', ar.shape)
20 print('配列の型:', ar.dtype)
21
22 # 左右の分離
23 arL = ar[:, 0] # 左チャンネル
24 arR = ar[:, 1] # 右チャンネル
25
26 # プロット
27 (fig, ax) = plt.subplots(2, 1, figsize=(8, 4))
28 plt.subplots_adjust(hspace=0.5)
29 ax[0].plot(arL)
30 ax[0].set_title('Left')
31 ax[1].plot(arR)
32 ax[1].set_title('Right')
33 plt.show()

```

このプログラムの9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。

¹³ 「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。

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=バッファーサイズ	通常は 4096 を設定しておく。

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 1.9.6
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。

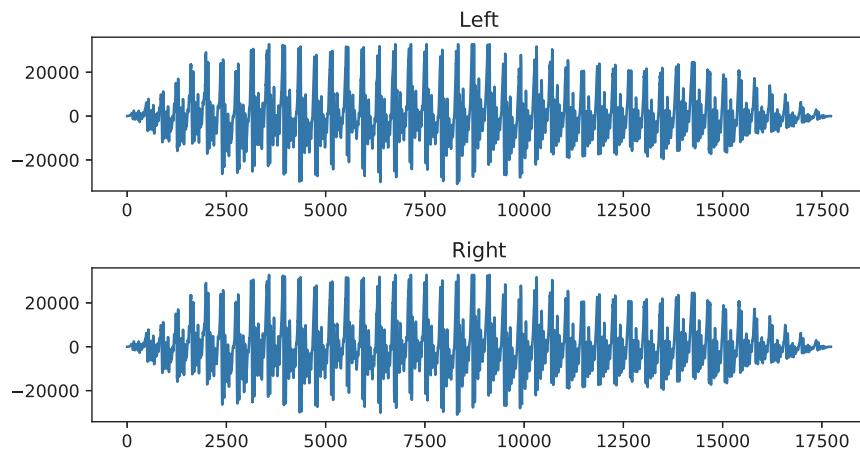


図 27: pygame08.py の実行で表示される波形のグラフ

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。

【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

プロ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 # coding: utf-8
2 # モジュールの読み込み
3 import sys
4 import pygame
5 from pygame.locals import QUIT      # 終了イベント
6 from pygame.locals import Rect     # 矩形クラス
7
8 pygame.init()    # pygameの初期化
9
10 ######
11 # スプライト用クラスの定義
12 # pygame.sprite.Sprite を継承して拡張する
13 # 基本プロパティ：
14 #   image - スプライト用画像
15 #   rect  - スプライトの位置とサイズ(矩形)
16 #   vx, vy - 移動時の差分(移動量)
17 #####
18 class MySprite( pygame.sprite.Sprite ):
19     def __init__(self, imgFname, x, y, vx, vy):
20         pygame.sprite.Sprite.__init__(self)
21         self.image = pygame.image.load(imgFname).convert_alpha()
22         width = self.image.get_width()
23         height = self.image.get_height()
24         self.rect = Rect(x, y, width, height)
25         self.vx = vx
26         self.vy = vy
27     def update(self):
28         global w, h      # ウィンドウサイズ(大域変数)
29         if self.rect.left < 0:
30             self.vx = -self.vx
31             self.rect.left = 0
32         if self.rect.right > w:
33             self.vx = -self.vx
34             self.rect.right = w
35         if self.rect.top < 0:
36             self.vy = -self.vy
37             self.rect.top = 0
38         if self.rect.bottom > h:
39             self.vy = -self.vy
40             self.rect.bottom = h
41         self.rect.move_ip(self.vx, self.vy)
42     def draw(self, screen):
43         screen.blit(self.image, self.rect)
44
45 #####
46 # ゲームフィールドの準備
47 #####
48 w = 400;    h = 300    # ウィンドウサイズ
49
50 # ウィンドウ(Surface)の生成
51 sf = pygame.display.set_mode( (w,h) )
52 pygame.display.set_caption('Sprite Test (1)')
53
54 # Clockオブジェクトの生成(フレームレート制御関連)
55 fps = pygame.time.Clock()
56
57 #####
58 # スプライトの生成
59 ######
```

```

60 spr1 = MySprite('ball.png', 0,0, 5,3 )
61
62 ######
63 # メインループ #
64 #####
65 while True:
66     # フレームレート設定(毎回)
67     fps.tick(30)
68     # ウィンドウ消去(毎回)
69     sf.fill((0,0,0))      # 毎回、真っ暗にする
70     # スプライト描画
71     spr1.update()        # スプライトの状態の更新
72     spr1.draw(sf)        # スプライトの表示
73     # イベント検出部
74     for ev in pygame.event.get():
75         if ev.type == QUIT:    # 終了処理
76             pygame.quit()
77             print('終了します。')
78             sys.exit()
79     # ウィンドウの更新
80     pygame.display.update()

```

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見定している。

このプログラムを実行した様子を図 28 に示す。

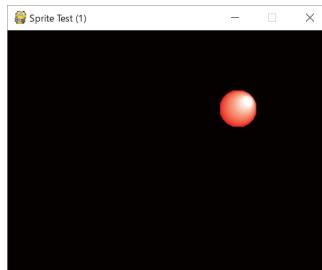


図 28: 1 個のボールがバウンドするアニメーション

■ 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

```

1 # coding: utf-8
2 # モジュールの読み込み

```

```

3 import sys
4 import pygame
5 from pygame.locals import QUIT      # 終了イベント
6 from pygame.locals import Rect     # 矩形クラス
7
8 pygame.init()    # pygameの初期化
9
10 ######
11 # スプライト用クラスの定義          #
12 # pygame.sprite.Sprite を継承して拡張する      #
13 # 基本プロパティ：                   #
14 #   image - スプライト用画像          #
15 #   rect  - スプライトの位置とサイズ（矩形）      #
16 #   vx, vy - 移動時の差分（移動量）          #
17 #####
18 class MySprite( pygame.sprite.Sprite ):
19     def __init__(self, imgFname, x,y,vx,vy):
20         pygame.sprite.Sprite.__init__(self)
21         self.image = pygame.image.load(imgFname).convert_alpha()
22         width = self.image.get_width()
23         height = self.image.get_height()
24         self.rect = Rect(x,y,width,height)
25         self.vx = vx
26         self.vy = vy
27     def update(self):
28         global w, h      # ウィンドウサイズ（大域変数）
29         if self.rect.left < 0:
30             self.vx = -self.vx
31             self.rect.left = 0
32         if self.rect.right > w:
33             self.vx = -self.vx
34             self.rect.right = w
35         if self.rect.top < 0:
36             self.vy = -self.vy
37             self.rect.top = 0
38         if self.rect.bottom > h:
39             self.vy = -self.vy
40             self.rect.bottom = h
41         self.rect.move_ip(self.vx, self.vy)
42     def draw(self, screen):
43         screen.blit(self.image, self.rect)
44
45 #####
46 # ゲームフィールドの準備          #
47 #####
48 w = 400;      h = 300      # ウィンドウサイズ
49
50 # ウィンドウ（Surface）の生成
51 sf = pygame.display.set_mode( (w,h) )
52 pygame.display.set_caption('Sprite Test (2)')
53
54 # Clockオブジェクトの生成（フレームレート制御関連）
55 fps = pygame.time.Clock()
56
57 #####
58 # スプライトの生成          #
59 #####
60 sg = pygame.sprite.Group()      # スプライトグループの生成
61 for i in range(5):
62     spr = MySprite('ball.png', i*80,i*60, 5,3 )
63     sg.add(spr)
64
65 #####
66 # メインループ          #
67 #####
68 while True:
69     # フレームレート設定（毎回）
70     fps.tick(30)
71     # ウィンドウ消去（毎回）
72     sf.fill( (0,0,0) )        # 毎回、真っ暗にする
73     # スプライト描画
74     sg.update()

```

```
75     sg.draw(sf)
76     # イベント検出部
77     for ev in pygame.event.get():
78         if ev.type == QUIT:      # 終了処理
79             pygame.quit()
80             print('終了します。')
81             sys.exit()
82     # ウィンドウの更新
83     pygame.display.update()
```

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。

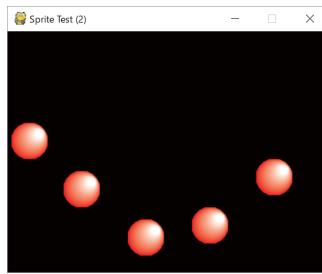


図 29: 5 個のボールがバウンドするアニメーション

2.2 Eel

Eel は HTML5 の枠組みで Python アプリケーションの GUI を実現するためのモジュールである。HTML5 でユーザインターフェースを構築するための基盤として Google Chrome ブラウザを使用する。このため、Google Chrome ブラウザから Python で記述した関数を呼び出す形のアプリケーション構築となる。また Google Chrome ブラウザの全機能が利用できるため、HTML5 による表現力の高いインターフェースが構築できるだけでなく、JavaScript と Python の両方の機能を組み合わせた形でアプリケーション全体を作り上げることができる。

本書では Eel に関する基本的な事柄について解説する。Eel に関する詳しい情報はインターネットサイト

<https://pypi.org/project/Eel/>

で公開されているのでそちらを参照のこと。

2.2.1 基礎事項

Eel を用いたアプリケーションは

- 1) Python で記述したスクリプト
- 2) GUI 構築用の HTML5 コンテンツ

から成る。更に 2) は HTML, JavaScript, CSS を記述したファイル群から成る。

次に、文字を表示するだけの簡単なサンプルプログラム（図 30）の実装を示す。

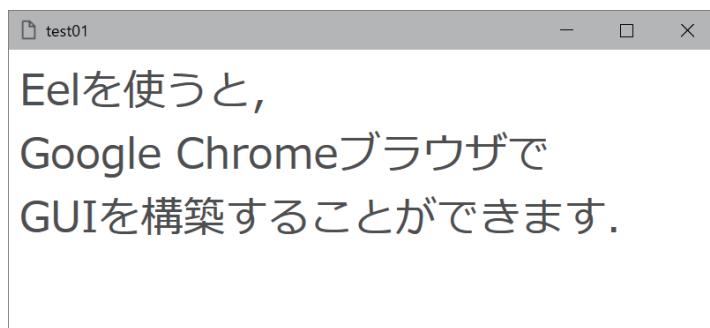


図 30: 最も簡単な Eel アプリケーション

このアプリケーションは Python で記述したプログラム本体 `test01.py` と、HTML で記述した UI コンテンツ `ui01.html` の 2 つから成る。（下記参照）

プログラム： `test01.py`

```
1 # coding: utf-8
2 # モジュールの読み込み
3 import eel
4
5 # Chrome起動オプション
6 op = {
7     'mode': 'chrome-app',      # 独自アプリケーションの形式
8     'mode': 'chrome',          # Chromeブラウザの形式
9     'chromeFlags': ['--window-size=550,250'] # GoogleChrome起動オプション
10 }
11
12 # UI初期化
13 eel.init('..')
14
15 # ChromeでUIを起動
16 eel.start('ui01.html', options=op )
```

UI 用 HTML ファイル： `ui01.html`

```
1 <html>
2
3 <head>
```

```

4 <meta charset='utf-8'>
5 <title>test01</title>
6 <!-- スタイル設定 -->
7 <style type="text/CSS">
8 p {
9     margin: 0pt;
10    font-size: 24pt;
11    color: #666666;
12    line-height: 1.5em;
13 }
14 </style>
15
16 <!-- Eel用に下記1行が必要 -->
17 <script type="text/javascript" src="/eel.js"></script>
18
19 </head>
20
21 <!-- UI本体 -->
22 <body>
23 <p>Eelを使うと、</p>
24 <p>Google Chrome ブラウザで </p>
25 <p>GUIを構築することができます。</p>
26 </body>
27
28 </html>

```

各種ファイル群の名前は任意に決めて良いが、構築するアプリケーション毎に1のフォルダにまとめるべきである。

test01.py の冒頭では

```
import eel
```

として Eel モジュールを読み込んでいる。また、GUI となる HTML コンテンツ内では Eel の機能を使用するために

```
<script type="text/javascript" src="/eel.js"></script>
```

を記述する。(ui01.html の 17 行目) この記述の中の `eel.js` という JavaScript のプログラムは Eel の動作によって自動的に与えられる。

Eel による GUI の起動は次の 2 つのステップから成る。

● ステップ1

プログラムの 13 行目にあるように、`init` メソッドを実行する。この際、引数には、当該アプリケーションを収めるディレクトリのパスを与える。

● ステップ2

プログラムの 16 行目にあるように、`start` メソッドを実行する。この際、第 1 引数には、UI となる HTML コンテンツのファイル名を文字列の型で与え、GoogleChrome ブラウザの起動時に与えるオプション列を辞書オブジェクトとしてキーワード引数 ‘option=’ に与える。

2.2.1.1 Chrome の起動モード

先の実行例では、Chrome の起動オプション ‘mode’ に ‘chrome-app’ を指定しているが、これを ‘chrome’ にする(8 行目のコメント)と、GUI のウィンドウが Web ブラウザの形式(図 31)となる。

2.2.2 Python/JavaScript で記述した関数の呼出し

Python スクリプトの中に定義を記述した関数を、Web ブラウザ側から呼び出して実行することができる。そのためには、Python 側の `def` 文(関数定義)の直前に

```
@eel.expose
```

というデコレータを記述する必要がある。また、Web ブラウザから使用する Python の関数の定義は、`init` メソッドの行以降、`start` メソッドの行までの間に記述すること。

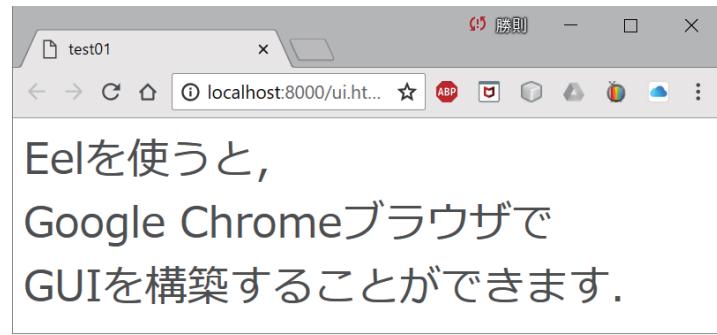


図 31: Web ブラウザ形式の UI

HTML から Python の関数を呼び出す、あるいは Python 側から HTML 中の JavaScript の関数を呼び出す方法について例を挙げて説明する。HTML で構成した UI への操作（ボタンのクリック）を受けて Python 側の関数を呼び出すプログラムについて考える。（図 32）

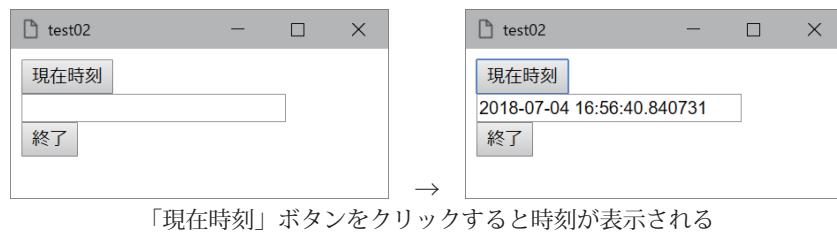


図 32: HTML から Python の関数を呼出して時刻を表示

これを実装した例を test02.py, ui02.html に示す。

プログラム：test02.py

```

1 # coding: utf-8
2 # モジュールの読み込み
3 from datetime import datetime
4 import eel
5
6 # Chrome起動オプション
7 op = {
8     'mode': 'chrome-app',      # 独自アプリケーションの形式
9     'chromeFlags': ['--window-size=300,150'] # GoogleChrome起動オプション
10 }
11
12 # UI初期化
13 eel.init('..')
14
15 # UIから呼び出す関数
16 @eel.expose
17 def pyFun1():
18     t = str(datetime.now())
19     eel.jsFun2(t)
20
21 # ChromeでUIを起動
22 eel.start('ui02.html', options=op )

```

UI用 HTML ファイル：ui02.html

```

1 <html>
2
3 <head>
4 <meta charset='utf-8'>
5 <title>test02</title>
6
7 <style type="text/css">
8 #t1 {

```

```

9     width: 200px;
10 }
11 </style>
12
13 <!-- Eel用に下記1行が必要 -->
14 <script type="text/javascript" src="/eel.js"></script>
15
16 <!-- UI制御用関数 -->
17 <script type="text/javascript">
18 // Pythonから呼び出す関数
19 eel.expose(jsFun2);
20 function jsFun2( v ) {
21     t1.value = v;
22 }
23 </script>
24
25 </head>
26
27 <!-- UI本体 -->
28 <body>
29 <input type="button" value="現在時刻" onClick="eel.pyFun1()"><br>
30 <input type="text" value="" id="t1"><br>
31 <input type="button" value="終了" onClick="window.close()">
32 </body>
33
34 </html>

```

`test02.py` の 17~19 行目に定義されている関数 `pyFun1` は現在時刻を求めて、UI 側のテキストフィールドにそれを表示するためのものである。この関数は HTML 側から呼び出すために `@eel.expose` デコレータが直前（16 行目）に記述されている。HTML 側からこれを呼び出すには、「`eel.`」を関数呼出しの直前に記述する。これにより JavaScript の関数と同じように呼び出すことができる。

JavaScript の関数を Python 側から呼び出すこともできる。この場合、JavaScript の関数の定義と共に、`eel.expose` 関数を用いて（`ui02.html` の 19 行目）Python 側から利用できるように設定しておく必要がある。

このアプリケーションでは、`ui02.html` の「現在時刻」ボタンをクリックすると、Python 側の関数 `pyFun1` が呼び出される。この関数では、`datetime` モジュールにより現在時刻を求め、それを JavaScript の関数 `jsFun2` に渡して `ui02.html` のテキストフィールド（`id="t1"`）の `value` 属性に設定する。これにより UI のテキストフィールドに時刻が表示される。

2.2.2.1 関数の戻り値について

Python と JavaScript の間で関数を呼び出す際、引数に値を渡すことはできるが、通常の `return` による戻り値の受け渡しはできない。これは、Python 処理系と Web ブラウザが異なるプロセスとして実行されているため¹⁴ である。今回のプログラムでは、Python 側の関数 `pyFun1` が JavaScript 側の関数 `jsFun2` を呼び出すことで、Python から JavaScript に値を渡して、`jsFun2` の動作としてテキストフィールド `t1` に値を設定している。

¹⁴参考) 異なるプロセスの間で値を受け渡しするには特別な方法が必要となる。

2.3 メディアデータの変換： PyDub

PyDub を用いることで、各種のメディアデータ（音声、動画）のフォーマットを変換することができる。PyDub はに関する情報は公式インターネットサイト

<http://pydub.com/>

から入手することができる。PyDub はメディアデータの変換に FFmpeg を用いており、利用に先立って FFmpeg¹⁵ をインストールしておく必要がある。

ここでは、WAV 形式の音声データと MP3 形式の音声データの間の変換を例に挙げて、PyDub の基本的な使用方法について説明する。

2.3.1 音声ファイルの読み込み

PyDub では音声データを AudioSegment クラスのオブジェクトとして扱う。このクラスを扱うためには次のようにして必要なものを Python 処理系に読み込む。

```
from pydub import AudioSegment
```

例. WAV 形式音声ファイル aaa.wav の読み込み

```
>>> from pydub import AudioSegment [Enter] ← AudioSegment クラスの読み込み  
>>> a = AudioSegment.from_wav('aaa.wav') [Enter] ← 音声ファイルの読み込み
```

この例では WAV 形式音声ファイル ‘aaa.wav’ の内容を from_wav メソッドで読み込み、それを AudioSegment クラスのオブジェクト a にしている。MP3 形式の音声ファイルを読み込むには from_mp3 メソッドで同様の処理を行う。

AudioSegment オブジェクトの重要なプロパティを表 14 に示す。

表 14: AudioSegment オブジェクトの重要なプロパティ

プロパティ	説明
sample_width	量子化ビット長（バイト単位）
channels	チャンネル数
frame_rate	サンプリング周波数（サンプリングレート：単位は Hz）
duration_seconds	音声データの長さ（秒）

例. AudioSegment オブジェクトのプロパティ（先の例の続き）

```
>>> a.sample_width [Enter] ← 量子化ビット長（バイト単位）  
2 [Enter] ← 2 バイト（16 ビット）  
>>> a.channels [Enter] ← チャンネル数  
2 [Enter] ← 2 チャンネル（通常のステレオ音声）  
>>> a.frame_rate [Enter] ← サンプリング周波数（サンプリングレート）  
44100 [Enter] ← 44.1KHz  
>>> a.duration_seconds [Enter] ← 長さ  
0.40199546485260773 [Enter] ← 音声の再生時間（秒）
```

2.3.2 音声データから 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、AudioSegment オブジェクトが持つ音声のデータ列を NumPy¹⁶ の配列データに変換する必要がある。そのためには次のような手順を踏む。

1. AudioSegment の音声データを get_array_of_samples メソッドで取り出す。
2. それを NumPy の配列（ndarray）に変換する。

実際の処理の例を次に示す。

¹⁵FFmpeg: メディアデータの変換のためのソフトウェア (<https://ffmpeg.org/>)

¹⁶「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. AudioSegment オブジェクトから NumPy の配列への変換（先の例の続き）

```
>>> import numpy as np [Enter] ← NumPy の読み込み
>>> ar = np.array(a.get_array_of_samples()) [Enter] ← AudioSegment から NumPy 配列への変換
>>> ar.dtype [Enter] ← データ形の調査
dtype('int16') ← 2 バイト符号付き整数
>>> ar.shape [Enter] ← 配列の形状の調査
(35456,) ← 35456 個の数値列
```

この例では、AudioSegment オブジェクト a が持つデータ列を、NumPy の配列オブジェクト ar に変換している。データ列は 1 次元であるが、チャンネル数が 2（左右のステレオ音声）の場合は、各チャンネルのデータの格納順は「左, 右, 左, 右, …」となる。

2.3.3 NumPy の配列から音声データへの変換

NumPy ライブラリを用いると、数値としての音声データの列に対して様々な処理を施すことができる。あるいは任意の音声波形を合成することも容易である。NumPy の配列データを AudioSegment オブジェクトに変換することで、それを音声のデータファイルとして保存することができる。

NumPy の配列を AudioSegment オブジェクトに変換するには AudioSegment クラスのコンストラクタに各種のキーワード引数を指定して実行する。

例. NumPy の配列から AudioSegment オブジェクトを生成する（先の例の続き）

```
>>> a2 = AudioSegment( data=ar, sample_width=2, frame_rate=44100, channels=2 ) [Enter]
```

この例のように、NumPy の配列をキーワード引数 ‘data’ に指定する。また、表 14 に示した各種プロパティと同名のキーワード引数があり、それらの値を設定することができる。

2.3.4 音声ファイルの保存

2.3.4.1 MP3 形式ファイルとして保存

AudioSegment オブジェクトの内容を音声データファイルとして保存するには export メソッドを使用する。このメソッドの第一引数に保存先のファイル名（パス名）を与え、フォーマット名をキーワード引数 ‘format’ に、再生ビットレートを ‘bitrate’ に、変換処理プログラムに渡す引数並びを ‘parameters’ に指定する。

例. AudioSegment オブジェクト a2 を MP3 形式ファイル ‘aaa.mp3’ として保存（先の例の続き）

```
>>> a2.export('aaa.mp3', format='mp3', bitrate='128k', parameters=[ '-cbr_quality', '10' ]) [Enter]
<_io.BufferedRandom name='aaa.mp3'> ← 保存処理中のメッセージ
```

この例では FFmpeg プログラムにエンコーディングの品質を指定する引数 ‘-cbr_quality 10’ を渡しているが、‘parameters’ は省略することができる。

2.3.4.2 WAV 形式ファイルとして保存

AudioSegment オブジェクトの内容を WAV 形式の音声ファイルとして保存する場合も export メソッドを使用する。この場合はキーワード引数 ‘format’ に ‘wav’ を与える。

例. AudioSegment オブジェクト a2 を WAV 形式ファイル ‘aaa2.wav’ として保存（先の例の続き）

```
>>> a2.export('aaa2.wav', format='wav') [Enter]
<_io.BufferedRandom name='aaa2.wav'> ← 保存処理中のメッセージ
```

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 <http://www.numpy.org/> で公開されている。

NumPy を使用するには次のように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)

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、扱うデータ列（配列）の要素の型は主として表 15 に挙げるようなもの¹⁷ である。

表 15: 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

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 15 のどれか 1 種類に限定される。表 15 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 15 に挙げたデータ型の配列に比べると処理が遅く¹⁸ なる。

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 15 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 の数値は

`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。

¹⁷ 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。

¹⁸ これに関しては「3.1.24.1 要素の型によって異なる計算速度」(p.139) で具体例を示す。

例. NumPy 独自の int64 の生成

```
>>> import numpy as np [Enter] ← NumPy の読み込み  
>>> 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。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
```

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
```

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。本書では以降、ndarray オブジェクトを array オブジェクトと呼ぶ。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、後の「3.1.20.2 単位行列、ゼロ行列、他」(p.124) で示すような方法（ゼロ行列の生成）で配列を生成することができる。

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('int32') ← 要素の型は'int32'であることがわかる
```

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 15 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 は複素数を扱うことができる。これに関しては「3.1.19 複素数の計算」(p.123) 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。

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「3.1.22 行列の検査」(p.135) でも説明する。

3.1.2.3 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

```
>>> 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
>>> 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] ← 究乗
array([ 1, 81], dtype=int32) ← 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（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
>>> 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], dtype=int32) ← 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¹⁹ 結果となる。

線形代数の演算に関しては後の「3.1.20 行列、ベクトルの計算（線形代数のための計算）」(p.124) 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²⁰ している。

例. pi, sin の値

```
>>> np.pi [Enter] ← π (円周率)
3.141592653589793 ← 値
>>> np.sin( np.pi / 2 ) [Enter] ← sin(  $\frac{\pi}{2}$  )
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([0.0000000e+00, 7.07106781e-01, 1.0000000e+00, 7.07106781e-01, 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「3.1.11 配列に対する演算：1 次元から 1 次元」(p.75) 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] ←  $\sqrt{-2}$  を求める試み
__main__:1: RuntimeWarning: invalid value encountered in sqrt ← 警告メッセージ「不正な値」
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  $\sqrt{-2}$  を求める
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 での複素数の扱いについては後の「3.1.19 複素数の計算」(p.123) で更に詳しく解説する。

3.1.2.4 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 16)

表 16: 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
```

¹⁹ この機能はブロードキャストの一部。

²⁰ 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。

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。

3.1.2.5 特殊な値：無限大と非数

NumPy では表 17 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 17: 特殊な数

記号	意味	記号	意味	記号	意味
inf	正の無限大	NINF	負の無限大 表示の際は ‘-inf’	nan	非数

※ $-np.inf == np.NINF$

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.NINF < -10**10000 [Enter] ← np.NINF と非常に小さな数  $-10^{10000}$  の比較
True ← np.NINF の方が小さい
>>> np.inf + np.inf [Enter] ← np.inf 同士の和
inf ← 正の無限大
>>> np.NINF + np.NINF [Enter] ← np.NINF 同士の和
-inf ← 負の無限大
>>> np.inf + np.NINF [Enter] ← 正負の異なる無限大同士の和
nan ← 非数
>>> 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. 無限大かどうかの判定

```
>>> np.isinf( np.inf ) [Enter] ← np.inf を判定
True ← 正の無限大
>>> np.isinf( np.NINF ) [Enter] ← np.NINF を判定
True ← 負の無限大
>>> np.isinf( 2 ) [Enter] ← 2 を判定
False ← 無限大ではない
```

3.1.2.6 データ列の生成（数列の生成）

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。

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。

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,  
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1. ]) ← 得られた数列
```

このように最後の1を含んで10個のデータを得るために、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
```

`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。対数スケールで関数の定義域のデータ列を生成するには `logspace` を使うのが良い。

例. $2^0 \sim 2^{10}$ 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([ 1.0000000e+00, 4.0000000e+00, 1.6000000e+01,  
       6.4000000e+01, 2.5600000e+02, 1.0240000e+03]) ← 得られた数列
```

キーワード引数 `'base='` を省略すると 10 が基底となる。

例. $10^0 \sim 10^{10}$ 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([ 1.0000000e+00, 1.0000000e+02, 1.0000000e+04,  
       1.0000000e+06, 1.0000000e+08, 1.0000000e+10]) ← 得られた数列
```

3.1.2.7 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],  
       [5, 6, 7, 8]]) ← 生成結果
```

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],  
       [3, 4]],  
       [[5, 6],  
       [7, 8]]]) ← 生成結果
```

3.1.2.8 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 ( $2 \times 2 \times 2$ ) 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

```
>>> 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
```

3.1.2.9 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、リストの場合と同様にスライス ‘[]’ が使用できる。また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できる。

例. 3×3 の配列の各要素へのアクセス

```
>>> import numpy as np [Enter] ← NumPy モジュールの読み込み  
>>> a = np.zeros( (3,3,3) ) [Enter] ←  $3 \times 3$  の配列の生成21  
>>> 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)  
>>> 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.]])
```

²¹p.124 「3.1.20.2 単位行列、ゼロ行列、他」で説明する。

3.1.2.10 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、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] ← 2n 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
```

3.1.2.11 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], [21, 22, 23], [31, 32, 33]]) ← 結果表示
>>> 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
```

特定の行や列に対して 1 度に値を与えることができる。

例. 指定した行や列に 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], [101, 102, 103], [ 31, 32, 33]]) ← 結果表示
>>> 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], [101, -2, 103], [ 31, -3, 33]]) ← 結果表示
```

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。

3.1.2.12 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], [4, 5, 6, 7]]) ← 結果表示
```

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（ $2 \times 2 \times 2$ ）に変形  
>>> a3 [Enter] ←内容確認  
array([[[0, 1],  
       [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],  
      [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],  
      [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 # coding: utf-8  
2 import numpy as np  
3 import time  
4  
5 sz = 10000  
6 # 2次元配列の作成  
7 m0 = np.arange(0,sz**2)  
8 m = m0.reshape((sz,-1))  
9 print('2次元配列の形状:',m.shape)  
10  
11 #--- flattenによる平坦化 ---  
12 t1 = time.time()  
13 m1 = m.flatten()  
14 t2 = time.time()  
15 b = (m0==m1).all()  
16 print('flattenの所要時間:\t',t2-t1,'t検証結果:',b)  
17  
18 #--- ravelによる平坦化 ---  
19 t1 = time.time()  
20 m2 = m.ravel()  
21 t2 = time.time()  
22 b = (m0==m2).all()  
23 print('ravelの所要時間:\t',t2-t1,'t検証結果:',b)  
24  
25 #--- reshapeによる平坦化 ---  
26 t1 = time.time()  
27 m3 = m.reshape(-1)  
28 t2 = time.time()  
29 b = (m0==m3).all()  
30 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。

```

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

```

(使用した計算機環境: CPU Intel Core i7-6770HQ 2.6GHz, RAM 16GB, Windows 10)

ravel, reshape による平坦化の処理にはほとんど時間がかかることがある。

3.1.2.13 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], [4, 5, 6]]) ←結果表示
>>> a.T [Enter] ←行と列の転置
array([[1, 4], [2, 5], [3, 6]]) ←結果表示

```

この処理はプロパティの参照なので、元の配列には一切変更がない。

3.1.2.14 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], [4, 5, 6], [7, 8, 9]]) ←結果表示
>>> np.flip(a, axis=0) [Enter] ←縦方向（上下）の反転
array([[7, 8, 9], [4, 5, 6], [1, 2, 3]]) ←結果表示
>>> np.flip(a, axis=1) [Enter] ←横方向（左右）の反転
array([[3, 2, 1], [6, 5, 4], [9, 8, 7]]) ←結果表示

```

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]])
```

3.1.2.15 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
```

スカラー値の型の変換は

`np. 型名 (値)`

とする。

例. 整数值を NumPy の int32 型に変換する

```
>>> a = np.int32( 5 ) [Enter] ←Python の通常の整数を変換  
>>> a [Enter] ←内容確認  
5 ←Python の通常の整数に見えるが…  
>>> type( a ) [Enter] ←型を調べると  
<class 'numpy.int32'> ←NumPy の int32 型である
```

例. NumPy の int32 型を float64 型に変換する（先の例の続き）

```
>>> b = np.float64( a ) [Enter] ←先の int32 型の数を変換  
>>> b [Enter] ←内容確認  
5.0 ←Python の通常の浮動小数点数（float）に見えるが…  
>>> type( b ) [Enter] ←型を調べると  
<class 'numpy.float64'> ←NumPy の float64 型である
```

3.1.2.16 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], [3, 4]]) ←変数 a の内容
>>> b = a [Enter] ←変数 a が指示する配列を変数 b に割り当てる
>>> b [Enter] ←変数 b の値の確認
array([[1, 2], [3, 4]]) ←変数 b の内容
```

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

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], [5, 6]]) ←変数 a と同じ内容
>>> 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], [3, 4]]) ←変数 a の内容
>>> c [Enter] ←変数 c の内容は…
array([[1, 2], [5, 6]]) ←変数 a に対する編集の影響がない
```

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]
                                         ↑3つの1次元配列を用意
>>> ac = np.concatenate( [a1,a2,a3] ) [Enter] ←全て連結
>>> 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]])
```

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], [2, 3]]) ←連結結果(2次元になる)
```

参考) 配列の連結にはこれ等の他にも‘c_’, ‘r_’といった関数も存在するが、本書では割愛する。

3.1.3.3 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], [3, 4]]) ←結果表示

>>> 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], [3, 4, 3, 4, 3, 4], [1, 2, 1, 2, 1, 2], [3, 4, 3, 4, 3, 4]]) ←結果表示
```

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],  
... [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（整数值）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²² 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(n1, n2))`

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。

²²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。

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.

例. 行の削除

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

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],  
       [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] ←モジュールの読み込み  
>>> 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] [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。

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] ←条件式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 18 のような記述による。

表 18: 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配列（あるいはそのタプル）を返す。

書き方 1: where(条件式)

例. where による要素の抽出

```
>>> a = np.array( range(10) ) [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
>>> np.where( a > 5 ) [Enter] ← 5 より大きな要素を抽出する
(array([6, 7, 8, 9], dtype=int64),) ← 結果 (配列のタプル)
>>> np.where( a > 5 )[0] [Enter] ← 5 より大きな要素の抽出 (タプルの先頭部分)
array([6, 7, 8, 9], dtype=int64) ← 結果 (配列)
```

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<5, a, 10*a ) [Enter] ← 5 以上の要素のみ 10 倍する処理
array([ 0, 1, 2, 3, 4, 50, 60, 70, 80, 90]) ← 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 (先の例の続き)

```
>>> a = np.identity( 3 ) [Enter] ← 3 次元の単位行列23
>>> a [Enter] ← 内容確認
array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]) ← 結果
>>> 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], [2, 4, 2], [2, 2, 4]]) ← 結果
```

3.1.7.5 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
10 ← 結果
>>> a.min() [Enter] ← 最小値を求める
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
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
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。

²³ 「3.1.20.2 単位行列, ゼロ行列, 他」(p.124) で解説する。

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  
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  
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], dtype=int64) ← 結果
```

この例の動作の解釈を図 33 に示す。



図 33: 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], dtype=int64) ← 結果
```

この例の動作の解釈を図 34 に示す。



図 34: 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 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）
```

3.1.8.1 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。

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。

3.1.8.2 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], dtype=int64) ←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], dtype=int64) ←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 35 に示す。

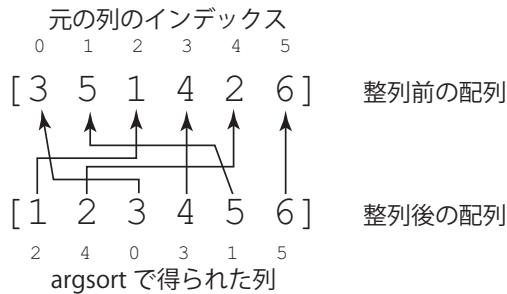


図 35: argsort の処理

`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],
       [300, 3, 100],
       [ 30, 200, 20]])
>>> a.argsort(axis=0) [Enter] ← 縦方向の整列
array([[ 0,  0,  0],
       [2,  1,  2],
       [1,  2,  1]], dtype=int64)
>>> a.argsort(axis=1) [Enter] ← 横方向の整列
array([[ 0,  1,  2],
       [1,  2,  0],
       [2,  0,  1]], dtype=int64)
```

3.1.9 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
>>> np.diff(a3,3) [Enter] ←3階の差分を求める
array([ 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 36 に示す.

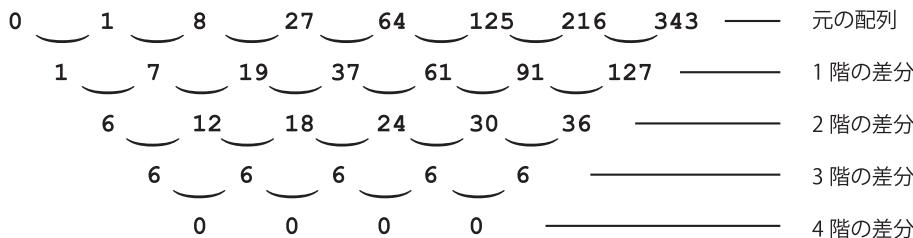


図 36: 配列の差分

3.1.10 配列に対する様々な処理

3.1.10.1 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
```

処理結果は整列済みの形で得られる。また、元の配列の内容は変更されない。

3.1.10.2 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], dtype=int64) ←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「唯一の要素」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。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  
>>> (u,c) = np.unique(a,return_counts=True) [Enter] ←要素数の集計  
>>> u,c [Enter] ←戻り値の確認  
(array(['a', 'b', 'c'], dtype='|<U1'), array([2, 2, 1], dtype=int64)) ←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  
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。

3.1.10.3 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], dtype=int64) ←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。

3.1.10.4 指定した条件を満たす要素の集計

count_nonzero 関数を使用すると、指定した条件を満たす要素の個数を得ることができる。

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
```

3.1.11 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. (つづき) 正弦関数の配列のプロット

```
>>> import matplotlib.pyplot as plt [Enter] ←matplotlibパッケージを'plt'として読み込んでいる  
>>> plt.plot(lx,ly) [Enter] ←プロットオブジェクトの生成  
[<matplotlib.lines.Line2D object at 0x0000026B4AF4B828>] ←生成結果  
>>> plt.show() [Enter] ←プロットを表示
```

この結果、図 37 に示すようなプロットが表示される。

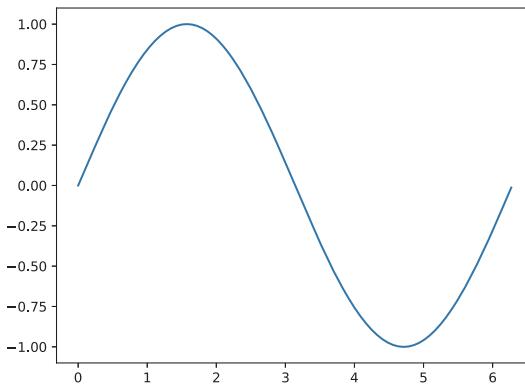


図 37: プロットの表示

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「3.1.12 データの可視化」で説明する。

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 19 に示す²⁴。詳しくは Numpy の公式サイトを参照のこと。

表 19: NumPy が提供する関数の一部

関数	説明	関数	説明
<code>sum</code>	配列要素の合計を求める	<code>mean</code>	配列要素の平均を求める
<code>var</code>	配列要素の分散を求める	<code>std</code>	配列要素の標準偏差を求める
<code>max</code>	配列要素の最大値を求める	<code>min</code>	配列要素の最小値を求める

3.1.12 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 <http://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)

²⁴ 使用方法は「3.1.14 統計に関する処理」(p.93) で説明する。

3.1.12.1 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²⁵ 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。多くの場合においてこの処理は省略できる。

2) 作図に関する処理

描画するグラフのサイズやタイトルの設定、各種グラフの描画、描画したグラフのファイルへの保存といった各種の処理を行う。

3) 表示に関する処理

`show` 関数を呼び出して、作成した図を実際にウィンドウに表示する。この段階で作図の流れは完了する。

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。

3.1.12.2 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 # coding: utf-8
2 # ライブラリの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 # データ列の生成
7 lx = np.arange(0.0, 6.28, 0.01)          # 定義域の生成
8 ly1 = np.sin(lx)                          # 正弦関数の列
9 ly2 = np.cos(lx)                          # 余弦関数の列
10
11 # データ列のプロット
12 plt.figure( figsize=(6,3) )               # 作図処理の開始（省略可）
13 plt.plot(lx,ly1, label='sin(x)')         # プロット(1)
14 plt.plot(lx,ly2, label='cos(x)')         # プロット(2)
15 plt.hlines([-1,0,1],                      # 水平の線
16             0,np.pi*2,ls='--',lw=1.5,color='#b0b0b0')
17 plt.vlines([0,np.pi/2,np.pi,np.pi*3/2,np.pi*2],
18            -1,1,ls='--',lw=1.5,color='#b0b0b0')    # 垂直の線
19 plt.xlabel('x')                         # 横軸ラベル
20 plt.ylabel('y')                         # 縦軸ラベル
21 plt.legend()                           # 凡例の表示
22 plt.title('trigonometric functions: sin, cos') # タイトルの表示
23 plt.show()                             # プロットの表示
24 plt.close()                            # 作図処理の終了（省略可）
```

プログラムの説明：

7~9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 13,14 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 15~18 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 19,20 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。21 行目では関数 `legend` によってグラフに凡例を付与している。22 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。（表 21 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。

²⁵ 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。

このプログラムを実行すると図 38 のようなプロットが表示される。

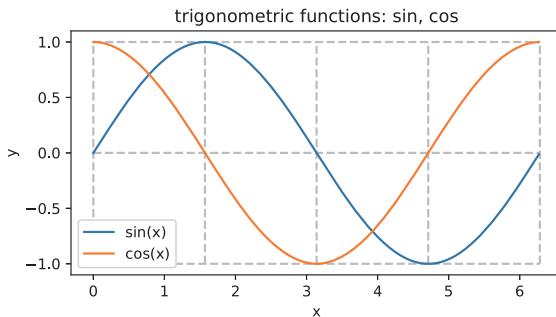


図 38: プロットの表示

【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 20）がある。

表 20: 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。False を与えると凡例を表示しない。
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。

【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 21 に示す。

表 21: 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

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

$$-\frac{\pi}{2} \leq x \leq \frac{\pi}{2}, \quad \frac{\pi}{2} \leq x \leq \frac{3\pi}{2}, \quad \frac{3\pi}{2} \leq x \leq \frac{5\pi}{2}$$

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 # coding: utf-8
2 # モジュールの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 # データ列の生成
7 p1 = -1.0 * np.pi / 2.0
8 p2 = np.pi / 2.0
9 p3 = 3.0 * np.pi / 2.0
10 p4 = 5.0 * np.pi / 2.0
11
12 lx1 = np.arange(p1+0.01, p2, 0.01) # 定義域の生成(1)
13 ly1 = np.tan(lx1) # 正接関数の列(1)
14 lx2 = np.arange(p2+0.01, p3, 0.01) # 定義域の生成(2)
15 ly2 = np.tan(lx2) # 正接関数の列(2)
16 lx3 = np.arange(p3+0.01, p4, 0.01) # 定義域の生成(3)
17 ly3 = np.tan(lx3) # 正接関数の列(3)
18
19 # データ列のプロット
20 plt.plot(lx1,ly1, color='red', lw=3, ls=':' ) # プロット(1)
21 plt.plot(lx2,ly2, color='black', lw=2, ls='--' ) # プロット(2)
22 plt.plot(lx3,ly3, color='blue', lw=1, ls='-' ) # プロット(3)
23 plt.grid(ls='--', lw=0.8, alpha=1.0) # グリッド表示
24 plt.ylim(-100,100)
25 plt.xlabel('x') # 横軸ラベル
26 plt.ylabel('y') # 縦軸ラベル
27 plt.title('trigonometric functions: tan(x)') # タイトルの表示
28 plt.show() # プロットの表示

```

このプログラムを実行すると図 39 のようなプロットが表示される.

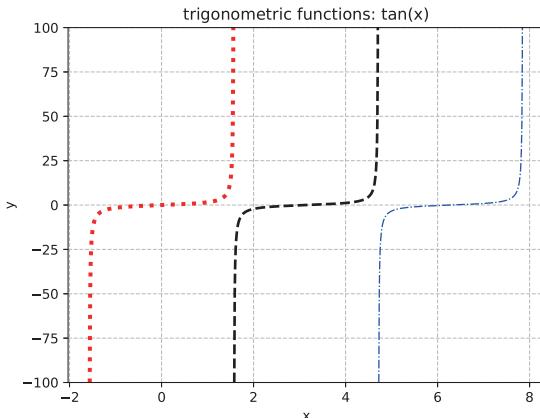


図 39: プロットの表示

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に通するものがある.

3.1.12.3 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 の読み込み
>>> import matplotlib.pyplot as plt [Enter] ← matplotlib の読み込み
>>> 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
>>> plt.figure( figsize=(4,2) ) [Enter] ← 描画処理の開始
<Figure size 400x200 with 0 Axes>
>>> plt.plot( x, y ) [Enter] ← グラフのプロット
[<matplotlib.lines.Line2D object at 0x0000015DC85DADC8>]
>>> 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 40 の (a) のような目盛りが表示される。

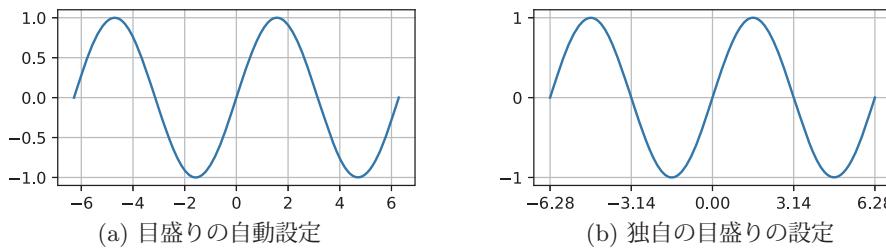


図 40: プロットの表示

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] )
```

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）

```
>>> fig = 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 40 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 41 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）

```
>>> fig = 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] ← 横軸目盛りの数値非表示
>>> t2 = plt.yticks(ticks=[-1,0,1], visible=False) [Enter] ← 縦軸目盛りの数値非表示
>>> plt.grid() [Enter]
>>> plt.show() [Enter]
```

また、`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。

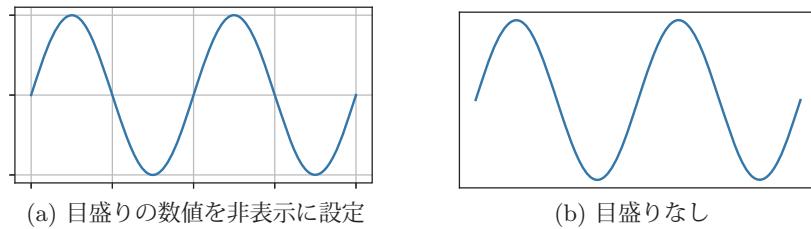


図 41: プロットの表示

例. 目盛りを非表示にする (先の例の続き)

```
>>> fig = 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
>>> plt.show() Enter
```

これを実行すると、図 41 の (b) のようなグラフが表示される。

3.1.12.4 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 42 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 42 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 42 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
```

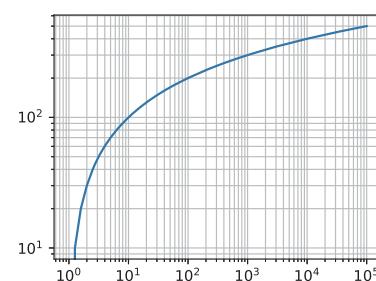
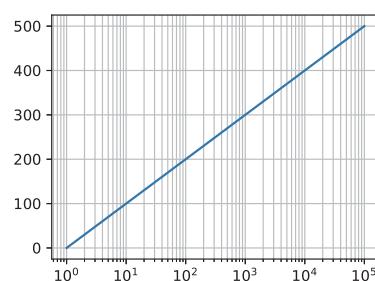
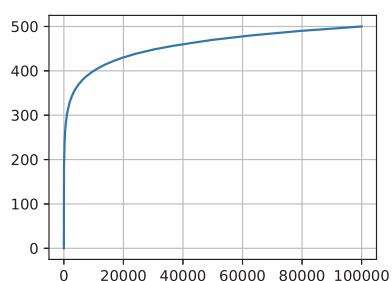


図 42: `xscale`, `yscale` 関数による軸の設定

3.1.12.5 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` メソッドを使用する。`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 # coding: utf-8
2
3 # モジュールの読み込み
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7 # データ列の生成
8 x = np.arange(-6.3, 6.3, 0.01) # 定義域の生成
9 y1 = np.sin(x) # 値域の生成(1)
10 y2 = np.cos(x) # 値域の生成(2)
11
12 # matplotlibによるプロット
13 (fig, ax) = plt.subplots(2, 1, figsize=(5,3))
14 plt.subplots_adjust(hspace=1.0)
15
16 ax[0].plot(x, y1, linewidth=1, color='red')
17 ax[0].set_title('sin(x)')
18 ax[0].set_ylabel('y')
19 ax[0].set_xlabel('x')
20 ax[0].grid(True)
21
22 ax[1].plot(x, y2, linewidth=1, color='green')
23 ax[1].set_title('cos(x)')
24 ax[1].set_ylabel('y')
25 ax[1].set_xlabel('x')
26 ax[1].grid(True)
27
28 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 43 に示す。

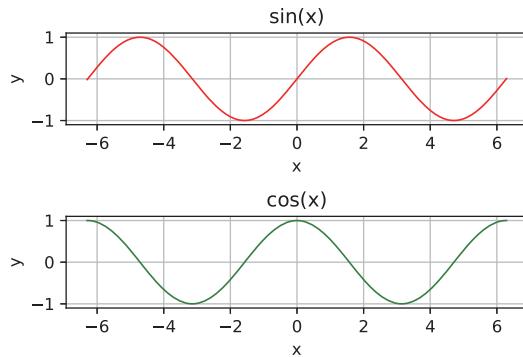


図 43: 2つのグラフを表示した例（縦に2つ）

プログラムの解説 :

nplot02-2.py の 8~10 行目で定義域と値域（正弦関数、余弦関数）を生成している。13 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ます。subplots の実行後、matplotlib.figure.Figure オブジェクトと matplotlib.axes.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。

14 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。

プログラム nplot02-2.py の 13~14 行目を,

```
(fig, ax) = plt.subplots(1, 2, figsize=(7,3))
plt.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 44 参照)

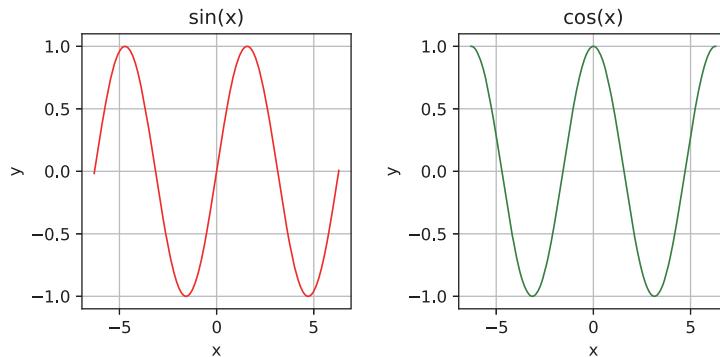


図 44: 2 つのグラフを表示した例（横に 2 つ）

■ 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 # coding: utf-8
2
3 # モジュールの読み込み
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7 # データ列の生成
8 x1 = np.arange(-6.3, 6.3, 0.01) # 定義域の生成(1)
9 y1 = np.sin(x1) # 値域の生成(1)
10 y2 = np.cos(x1) # 値域の生成(2)
11 y3 = np.exp(x1) # 値域の生成(3)
12
13 x2 = np.arange(0.01, 10, 0.01) # 定義域の生成(2)
14 y4 = np.log(x2) # 値域の生成(4)
15
16 # matplotlibによるプロット
17 (fig, ax) = plt.subplots(2, 2, figsize=(8,4))
18 plt.subplots_adjust(wspace=0.3, hspace=0.7)
19
20 ax[0,0].plot(x1, y1, linewidth=1, color='red')
21 ax[0,0].set_title('sin(x)')
22 ax[0,0].set_ylabel('y')
23 ax[0,0].set_xlabel('x')
24 ax[0,0].grid(True)
25
26 ax[1,0].plot(x1, y2, linewidth=1, color='green')
27 ax[1,0].set_title('cos(x)')
28 ax[1,0].set_ylabel('y')
29 ax[1,0].set_xlabel('x')
30 ax[1,0].grid(True)
31
32 ax[0,1].plot(x1, y3, linewidth=1, color='blue')
33 ax[0,1].set_title('exp(x)')
34 ax[0,1].set_ylabel('y')
35 ax[0,1].set_xlabel('x')
36 ax[0,1].grid(True)
37
38 ax[1,1].plot(x2, y4, linewidth=1, color='black')
39 ax[1,1].set_title('log(x)')
40 ax[1,1].set_ylabel('y')
41 ax[1,1].set_xlabel('x')
42 ax[1,1].grid(True)
43
44 plt.show()
```

プログラムの解説：

`nplot02-4.py` の 8~14 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。17 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 45 に示す。

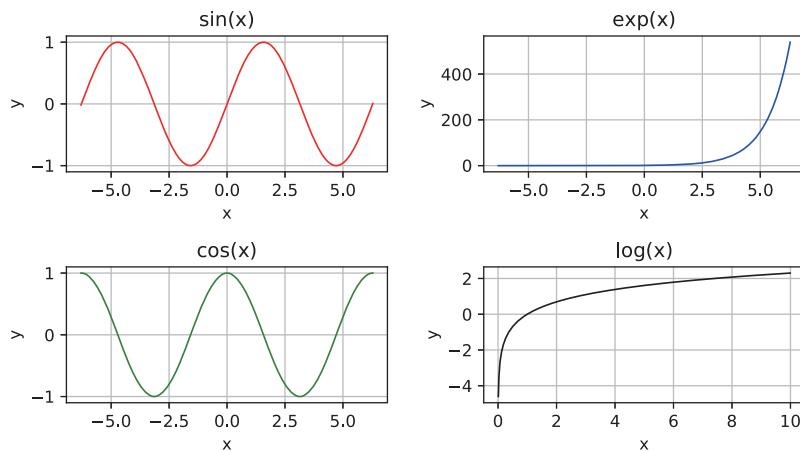


図 45: 縦横に 4 つのグラフを表示した例

3.1.12.6 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

```
1 # coding: utf-8
2 # ライブライリの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5 # データ1
6 x1 = np.linspace(-2.2, 2.2, 100)
7 y1 = x1**5-5*x1**3+4*x1
8 # データ2
9 x2 = np.linspace(-2, 2, 100)
10 y2 = x2**2-1
11 ######
12 # 2つの図
13 ######
14 (fig, axs) = plt.subplots(1, 2, figsize=(9, 4))
15 plt.subplots_adjust(wspace=0.32)
16 # 1つ目の図
17 axs[0].set_xlim(x1.min(), x1.max())
18 axs[0].set_ylim(y1.min(), y1.max())
19 axs[0].plot(x1, y1)
20 axs[0].vlines(0, y1.min(), y1.max(), lw=1.5, color='gray')
21 axs[0].hlines(0, x1.min(), x1.max(), lw=1.5, color='gray')
22 axs[0].grid(True)
23 axs[0].set_title('x**5-5*x**3+4*x')
24 axs[0].set_xlabel('x')
25 axs[0].set_ylabel('y')
26 # 2つ目の図
27 axs[1].set_xlim(x2.min(), x2.max())
28 axs[1].set_ylim(y2.min(), y2.max())
29 axs[1].plot(x2, y2)
30 axs[1].vlines(0, y2.min(), y2.max(), lw=1.5, color='gray')
31 axs[1].hlines(0, x2.min(), x2.max(), lw=1.5, color='gray')
32 axs[1].grid(True)
33 axs[1].set_title('x**2-1')
```

```

34 |     axs[1].set_xlabel('x')
35 |     axs[1].set_ylabel('y')
36 |     fig.suptitle('Multiple plots')
37 |     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46のようなグラフが表示される。

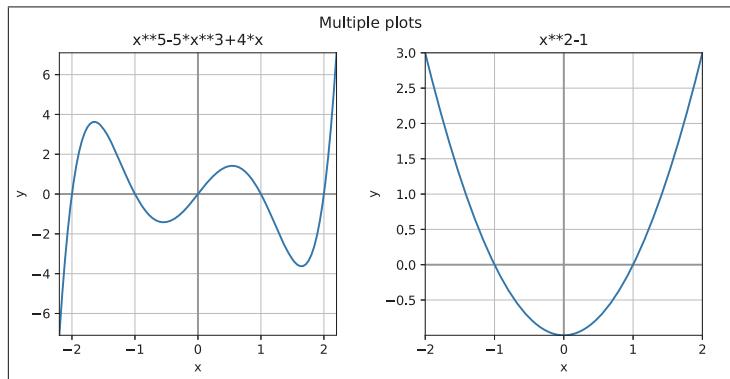


図 46: nplot02-6.py の実行結果

このグラフを図解すると図47のような部分から成ることがわかる。

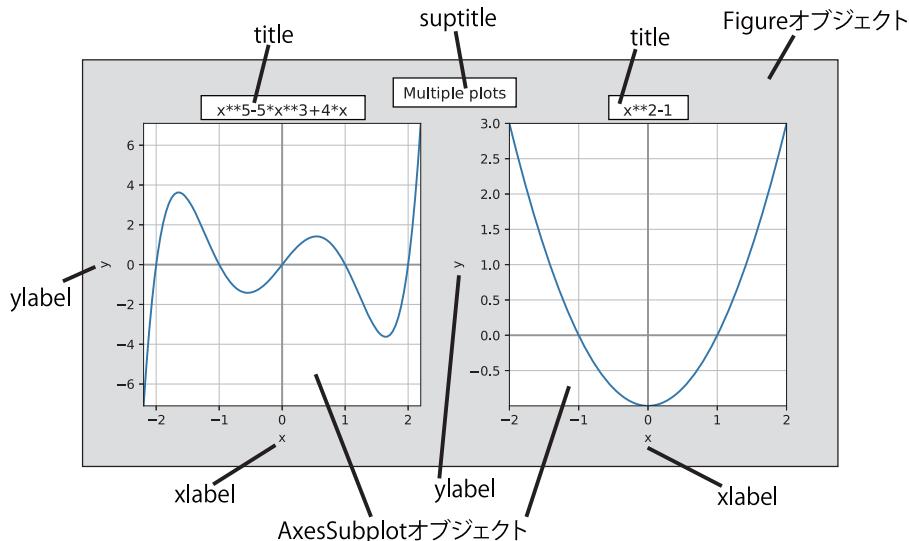


図 47: matplotlib のグラフの構成

【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AxesSubplotオブジェクトを持ち、これが具体的に描画されるグラフである。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AxesSubplotを配列の形で保持する。

● AxesSubplot オブジェクト

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 AxesSubplot オブジェクトのみを持つ場合

描画に関する各種のメソッドは最上位の Figure オブジェクトに対して行う。実際の処理は Figure オブジェクト配下の AxesSubplot 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 AxesSubplot 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 AxesSubplot オブジェクトに対して行う。

以上のことと踏まえると、1 つのみ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 AxesSubplot 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 # coding: utf-8
2 # ライブラリの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5 # データ1
6 x1 = np.linspace(-2.2, 2.2, 100)
7 y1 = x1**5 - 5*x1**3 + 4*x1
8 ######
9 # 1つの図
10 #####
11 plt.figure(figsize=(4,4))
12 # 描画範囲の指定
13 plt.xlim(x1.min(), x1.max())
14 plt.ylim(y1.min(), y1.max())
15 plt.plot(x1, y1)
16 plt.vlines(0, y1.min(), y1.max(),
17             lw=1.5, color='gray')
18 plt.hlines(0, x1.min(), x1.max(),
19             lw=1.5, color='gray')
20 plt.grid(True)
21 plt.xlabel('x')
22 plt.ylabel('y')
23 plt.title('x**5-5*x**3+4*x')
24 plt.suptitle('Single plot')
25 plt.show()

```

プログラム：nplot02-7.py

```

1 # coding: utf-8
2 # ライブラリの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5 # データ1
6 x1 = np.linspace(-2.2, 2.2, 100)
7 y1 = x1**5 - 5*x1**3 + 4*x1
8 #####
9 # 1つの図：別 の 方法
10 #####
11 plt.figure(figsize=(4,4))
12 fig = plt.gcf(); ax = plt.gca()
13 ax.set_xlim(x1.min(), x1.max())
14 ax.set_ylim(y1.min(), y1.max())
15 ax.plot(x1, y1)
16 ax.vlines(0, y1.min(), y1.max(),
17             lw=1.5, color='gray')
18 ax.hlines(0, x1.min(), x1.max(),
19             lw=1.5, color='gray')
20 ax.grid(True)
21 ax.set_xlabel('x')
22 ax.set_ylabel('y')
23 ax.set_title('x**5-5*x**3+4*x')
24 fig.suptitle('Single plot')
25 plt.show()

```

nplot02-7.py の 12 行目では、gcf(), gca() によって作図対象の Figure オブジェクトと AxesSubplot を、fig と ax にそれぞれ取得している。

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 22 に示す。

表 22: 各種の設定のための関数の違い

説明	Figure 用メソッド	AxesSubplot 用メソッド
横軸の描画範囲の指定 縦軸の描画範囲の指定	xlim(下限, 上限) ylim(下限, 上限)	set_xlim(下限, 上限) set_ylim(下限, 上限)
横軸の設定 縦軸の設定	xticks(各種の引数) yticks(各種の引数)	set_xticks(各種の引数) set_yticks(各種の引数)
横軸のラベルの設定 縦軸のラベルの設定	xlabel(横軸ラベル) ylabel(縦軸ラベル)	set_xlabel(横軸ラベル) 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 48 のようなグラフを描画する。

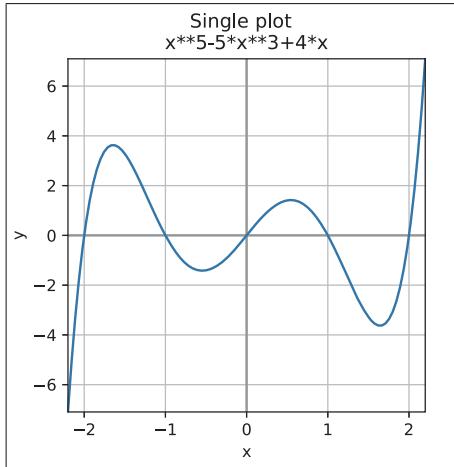


図 48: nplot02-5.py, nplot02-7.py の実行結果

3.1.12.7 グラフの枠を非表示にする方法

グラフの枠は AxesSubplot 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 AxesSubplot 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 AxesSubplot 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 # coding: utf-8
2 import numpy as np
3 import matplotlib.pyplot as plt
4 # サンプルデータ
5 x = np.linspace(0, 2*np.pi, 360)
6 y = np.sin(3*x)
7
8 # プロット
9 (fig, axs) = plt.subplots(1, 4, figsize=(8, 2))
10 for (i, ax) in enumerate(axs):          # グラフを4枚描画
11     ax.plot(x, y)
12     ax.set_xticks(ticks=[]);      ax.set_yticks(ticks[])
13 # 枠の消去
14 axs[0].set_title('no top')
15 axs[0].spines['top'].set_visible(False)    # 上の枠を消去
16 axs[1].set_title('no bottom')
17 axs[1].spines['bottom'].set_visible(False)  # 下の枠を消去
18 axs[2].set_title('no left')
19 axs[2].spines['left'].set_visible(False)    # 左の枠を消去
20 axs[3].set_title('no right')
21 axs[3].spines['right'].set_visible(False)   # 右の枠を消去
22 plt.show()

```

このプログラムでは、10~12 行目で同じグラフを 4 つ描画し、15 行目、17 行目、19 行目、21 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 49 のように表示される。

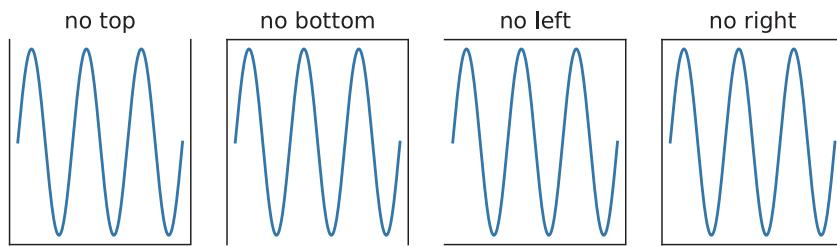


図 49: 枠の非表示

3.1.12.8 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 AxesSubplot 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 # coding: utf-8
2 import numpy as np
3 import matplotlib.pyplot as plt
4 # データの作成
5 x = np.linspace(0, 6*np.pi, 1080)
6 y = 2*x
7 # プロット
8 plt.figure(figsize=(5,5))
9 ax = plt.subplot(projection='polar')
10 ax.plot(x, y)
11 plt.show()

```

プログラム : nplotPol02.py

```

1 # coding: utf-8
2 import numpy as np
3 import matplotlib.pyplot as plt
4 # データの作成
5 x = np.linspace(0, 2*np.pi, 360)
6 y = np.sin(10*x)+1
7 # プロット
8 plt.figure(figsize=(5,5))
9 ax = plt.gca(projection='polar')
10 ax.plot(x, y)
11 plt.show()

```

上のプログラム nplotPol01.py, nplotPol02.py では 9 行目で AxesSubplot オブジェクトを取得している。具体的には subplot メソッドや gca 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。

これらプログラムを実行して得られるグラフを図 50 に示す。

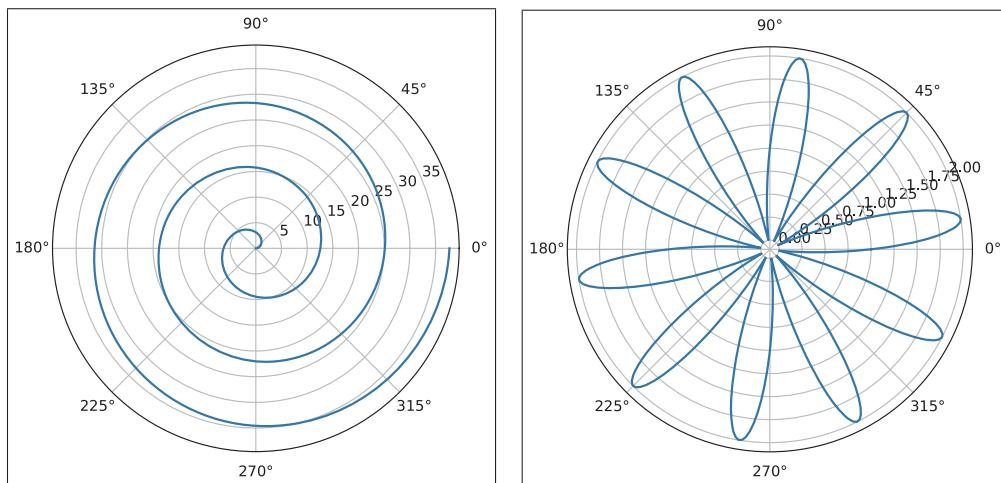


図 50: nplotPol01.py, nplotPol02.py の実行結果

3.1.12.9 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。

プログラム : nplotPol03.py

```

1 # coding: utf-8
2 import numpy as np
3 import matplotlib.pyplot as plt
4

```

```

5 # データの作成
6 lbl = ['A', 'B', 'C', 'D', 'E', 'F']          # ラベル
7 x = np.linspace(0, 2*np.pi, len(lbl)+1)      # 各ラベルの角度
8 # レーダーチャートにする値
9 # 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
10 d1 = [1, 0.4, 0.2, 0.7, 0.5, 1, 1]        # データ1
11 d2 = [0.3, 0.5, 1, 0.2, 0.8, 0.5, 0.3]    # データ2
12
13 # プロット
14 plt.figure(figsize=(5,5))
15 ax = plt.gca(projection='polar')
16 ax.plot(x, d1, label='d1')
17 ax.plot(x, d2, label='d2')
18 ax.set_xticks(ticks[])
19 ax.set_thetagrids(x[:-1]*180/np.pi, lbl)
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 $0^\circ \sim 360^\circ$ ($0 \sim 2\pi$ 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 19 行目にあるように `set_thetagrids` メソッドを `AxesSubplot` オブジェクトに対して実行する。

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)`

また、極座標グラフの外周の枠線を非表示にするには `AxesSubplot` 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 51 のようなレーダーチャートが表示される。

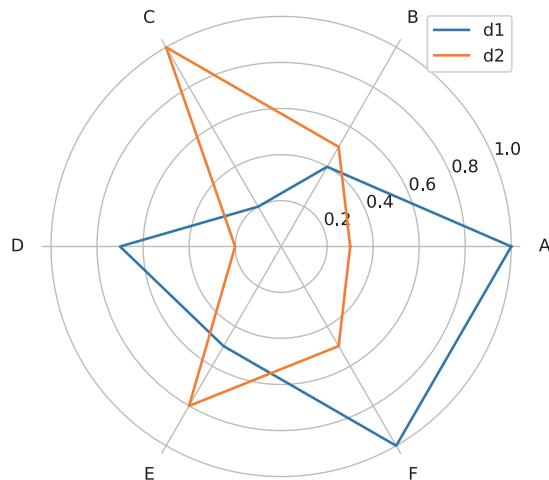


図 51: レーダーチャート

3.1.12.10 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

```
1 # coding: utf-8
2 # モジュールの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5 from matplotlib.font_manager import FontProperties
6
7 # グラフで使用する日本語フォント
8 fp = FontProperties(fname=r'C:\WINDOWS\Fonts\msgothic.ttc', size=11)
9
10 # データ列の生成
11 lx = np.arange(-3.15, 3.15, 0.01) # 定義域の生成
12 ly1 = np.sin(lx) # 正弦関数の列
13 ly2 = np.cos(lx) # 余弦関数の列
14
15 # データ列のプロット
16 plt.figure(figsize=(6,3)) # 作図処理の開始（省略可）
17 plt.plot(lx, ly1, label='正弦関数') # プロット(1)
18 plt.plot(lx, ly2, label='余弦関数') # プロット(2)
19 plt.xlabel('定義域', fontproperties=fp) # 横軸ラベル
20 plt.ylabel('値域', fontproperties=fp) # 縦軸ラベル
21 plt.legend(prop=fp) # 凡例の表示
22 # タイトルの表示
23 plt.title('正弦関数, 余弦関数のプロット', fontproperties=fp)
24 plt.grid(True)
25 # 出力
26 plt.savefig('nplot01_out.ps') # 画像ファイル出力
27 plt.show() # プロットの表示
```

5行目で `FontProperties` クラスを読み込み、8行目でフォントを読み込んで `fp` に与えている。19,20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 21 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 52 に示す。

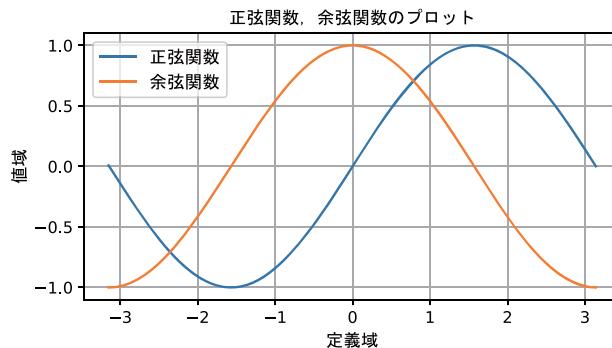


図 52: 見出しおとラベルを日本語にした例

参考) `matplotlib` で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として `japanize-matplotlib` が存在している。

3.1.12.11 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²⁶ 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。

²⁶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。

3.1.13 亂数の生成

3.1.13.1 一様乱数の生成

random 関数は 0 以上 1 未満の乱数を生成する.

書き方 :

- 1) np.random.rand() 亂数を 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] ← 亂数を 1 つ生成  
0.9002721968823484 ← 成績結果  
>>> np.random.rand(5) [Enter] ← 亂数を 5 個生成  
array([ 0.32644595, 0.20630809, 0.98017323, 0.09793674, 0.39467418]) ← 成績結果  
>>> 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], ← 成績結果  
      [ 0.2018859 , 0.88586831, 0.88083754],  
      [ 0.73630688, 0.78485615, 0.98865664],  
      [ 0.58109305, 0.75149191, 0.26337745],  
      [ 0.67083326, 0.91048167, 0.9451317 ]])
```

3.1.13.2 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.

```
>>> 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]) ← 成績結果
```

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.

3.1.13.3 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.

例.

```
>>> 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])  
>>> 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]])
```

3.1.13.4 亂数の 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、特定の統計的特徴に沿った形の乱数を生成するための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) を起点として、決められた並びの乱数列を生成する。関数 random.seed は、引数に与えられた数値を乱数生成過程の初期状態として設定する。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, 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.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)

>>> 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)
```

3.1.13.5 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。(次の例参照)

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 オブジェクトを用いることで、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 統計に関する処理

基本的な統計量を算出する方法を示す。

3.1.14.1 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 = np.random.normal(4,2,100000) Enter ← $\mu=4, \sigma=2$  の正規乱数を  $10^5$  個生成
>>> r.sum() Enter ←合計を求める
399973.1259455481 ←計算結果
```

3.1.14.2 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
12.936924874373553 ←計算結果
>>> r.min() Enter ←最小値を求める
-5.673905593171707 ←計算結果
```

3.1.14.3 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
3.9997312594554812 ←計算結果
>>> r.var() Enter ←標本分散を求める
4.017122067277002 ←計算結果
>>> r.std() Enter ←標本標準偏差を求める
2.004275945890935 ←計算結果
```

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、標本標準偏差をそれぞれ求める。不偏分散、不偏標準偏差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  
4.017162238899392 ←計算結果  
>>> r.std(ddof=1) [Enter] ←不偏標準偏差を求める  
2.0042859673458255 ←計算結果
```

3.1.14.4 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。

例. 分位点 (先の例の続き)

```
>>> np.quantile( r, 0 ) [Enter] ←四分位数 0 の点の値 (最小値) を求める  
-5.673905593171707 ←計算結果  
>>> np.quantile( r, 0.25 ) [Enter] ←四分位数 25% の点の値を求める  
2.6465003734023607 ←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  
3.999836531264145 ←計算結果  
>>> np.quantile( r, 0.75 ) [Enter] ←四分位数 75% の点の値を求める  
5.368313237300546 ←計算結果  
>>> np.quantile( r, 1 ) [Enter] ←四分位数 100% の点の値 (最大値) を求める  
12.936924874373553 ←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 引数には要素数の百分率を 0~100 の範囲の値で与える。

3.1.14.5 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 つのデータにする  
>>> c = np.digitize( a, bins=b ) [Enter] ←区間集計の実行  
>>> 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] ←集計結果の確認  
[1 2 2 4] ←集計結果
```

この例で示した集計処理を図 53 で図解する。

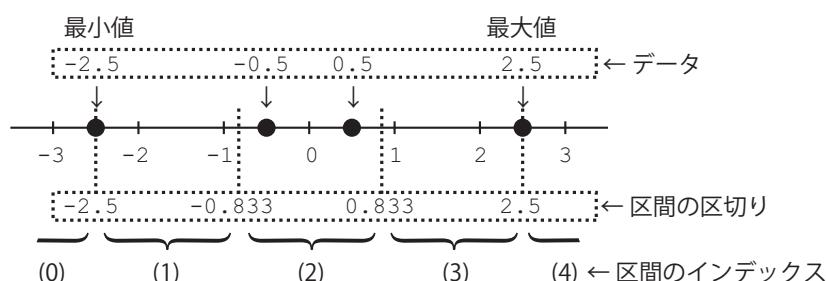


図 53: `digitize` による区間集計のイメージ
最小値未満の区間のインデックスが 0 となっている

表 23: `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

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 23）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。

次に示す例は、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ものである。

例. 正規乱数のデータ列を区間毎にラベル付け

```
>>> np.random.seed(0) [Enter] ←乱数の初期化27
>>> a = np.random.normal(50,10,100000) [Enter] ← $\mu = 50, \sigma = 10$  の正規乱数を  $10^5$  個生成
>>> b = np.linspace(a.min(), a.max(), 40) [Enter] ←集計用の区間データ列
>>> 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
[67.64052346 54.00157208 59.78737984 72.40893199 68.6755799] ←元のデータ
[29 23 26 31 29] ←集計結果：各要素が属する区間のインデックス
```

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。更に `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として `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`²⁸ 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）

例. `s` の内容（先の例の続き）

```
>>> print(s) [Enter] ←s の内容表示
[ 0    2    1    0    2    5   20   38   78   170   321   496   872 1393
 2168 3140 4183 5632 6900 8014 8899 9369 9264 8555 7624 6530 5165 3852
 2677 1855 1164  751  407  227  115   61   28   15     5     1     1] ←s の内容：
                                                               先頭要素が 0
                                                               になっている。
```

²⁷ 実行例の再現性のために行った。

²⁸ 「3.1.10.3 整数要素の集計」(p.75) 参照のこと。

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
plt.figure(figsize=(6,2))
plt.bar( b,s[1:], width=2.0)
plt.xlabel('class'); plt.ylabel('frequency')
plt.show()
```

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 54 に示すようなグラフが表示される。

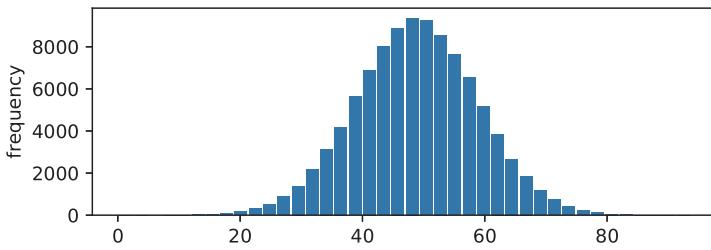


図 54: 度数分布のプロット

【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
(1.4788234681988328, 92.41771912903697) ←最小値と最大値
```

これらの値から判断し、0~90までの5刻みの区間で度数を集計することを考える。

例. 0~90までの区間を5刻みで集計（先の例の続き）

```
>>> b2 = np.linspace( 0, 90, 19 ) [Enter] ←0~90までを18等分して19個のデータ配列を作成
>>> print( b2 ) [Enter] ←内容確認
[ 0.  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
(20,) ←長さ20の一次元配列
>>> print( s2 ) [Enter] ←内容確認
[0      2      1      24     114     528    1588    4362    9027   15030   19257
 19214   15095   9142   4315   1682    478     116     23      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)

```
>>> plt.figure(figsize=(4.5,2)) [Enter] ←描画サイズの設定
<Figure size 450x200 with 0 Axes> ←戻り値
>>> plt.bar( b2,s2[1:], width=4.0) [Enter] ←棒グラフの描画
<BarContainer object of 19 artists> ←戻り値
>>> plt.xlabel('class'); plt.ylabel('frequency') [Enter] ←軸ラベルの設定
Text(0.5, 0, 'class') ←戻り値
Text(0, 0.5, 'frequency')
>>> plt.show() [Enter] ←描画実行
```

これによって図 55 のようなグラフが表示される。

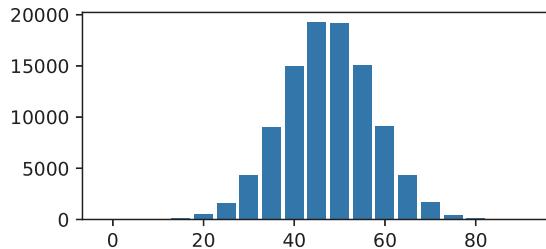


図 55: 度数分布のプロット (5 刻みに区間調整後)

- 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。

3.1.14.6 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に従う乱数を作成する。そのための例を次に示す。(numpy, matplotlib.pyplot は読み込み済みである前提)

例. $\mu = 0, \sigma = 0.5$ の対数正規分布に従う乱数の作成

```
>>> np.random.seed(0) [Enter] ←乱数の初期化29
>>> dat = np.random.lognormal( 0, 0.5, 1000000 ) [Enter] ←106 個のデータを生成
>>> plt.figure( figsize=(6,2) ) [Enter] ←作図の開始
<Figure size 600x200 with 0 Axes>
>>> g = plt.hist(dat,bins=80) [Enter] ←階級数 80 でヒストグラムを作成
>>> plt.xlim(0,4)
(0.0, 4.0)
>>> plt.show() [Enter] ←作図の実行
```

この結果、図 56 のようなヒストグラムが表示される。

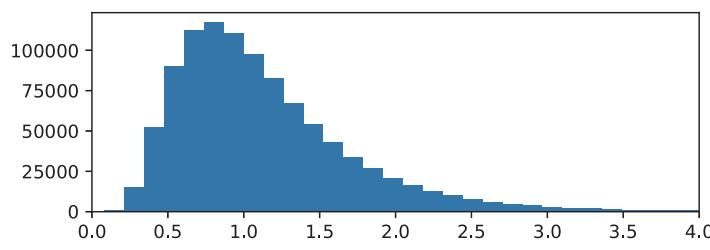


図 56: 対数正規分布に従う乱数のヒストグラム

²⁹ 実行例の再現性のために行った。

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: [0.08199071 0.21313196 0.34427321]  
区間の末尾: [10.31100821 10.44214946 10.57329071]
```

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  
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 $g[1][6]$ の中間に最頻値であると結論する。

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: 0.8032675852230751 ← 結果
```

この例から最頻値は 0.8032675852230751 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）

```
>>> plt.figure(figsize=(6,2)) [Enter] ← 作図の開始  
<Figure size 600x200 with 0 Axes>  
>>> g = plt.hist(dat, bins=80) [Enter] ← ヒストグラムを作成  
>>> plt.xlim(0,4)  
(0.0, 4.0)  
>>> 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  
<matplotlib.collections.LineCollection object at 0x00000190ECD0BC88>  
>>> plt.show() [Enter] ← 作図の実行
```

この結果、図 57 のようにヒストグラム中に最頻値が図示される。

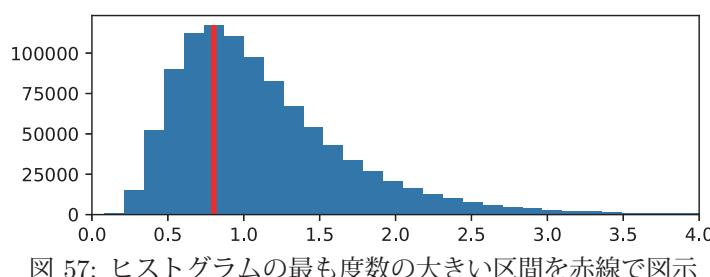


図 57: ヒストグラムの最も度数の大きい区間を赤線で図示

【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
>>> plt.figure( figsize=(6,2) ) [Enter] ← 作図の開始
<Figure size 600x200 with 0 Axes>
>>> 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
<BarContainer object of 81 artists>
>>> plt.xlim(0,4)
(0.0, 4.0)
>>> 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 58 のようなヒストグラムが表示される。

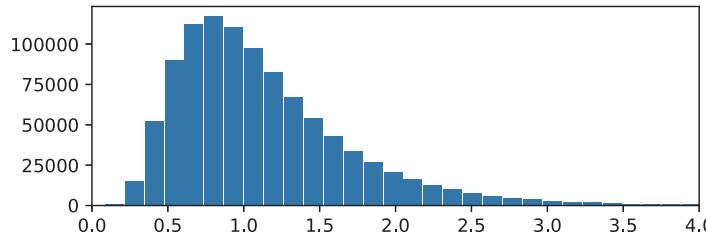


図 58: bar 関数で描画したヒストグラム

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
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。

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: 0.8032675852230751 ← 結果
```

この例から最頻値は 0.8032675852230751 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）

```
>>> plt.figure( figsize=(6,2) ) [Enter] ← 作図の開始
<Figure size 600x200 with 0 Axes>
>>> 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
<BarContainer object of 81 artists>
>>> plt.xlim(0,4)
(0.0, 4.0)
>>> plt.vlines([md2], 0, s.max(), lw=3, color='red') [Enter] ← 最頻値の位置に垂直線を描画
<matplotlib.collections.LineCollection object at 0x00000190E8473C48>
>>> plt.show() [Enter] ← 作図の実行
```

この結果、図 59 のようにヒストグラム中に最頻値が図示される。

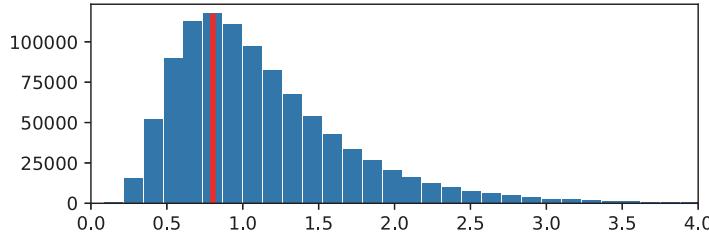


図 59: ヒストグラムの最も度数の大きい区間を赤線で図示

3.1.14.7 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

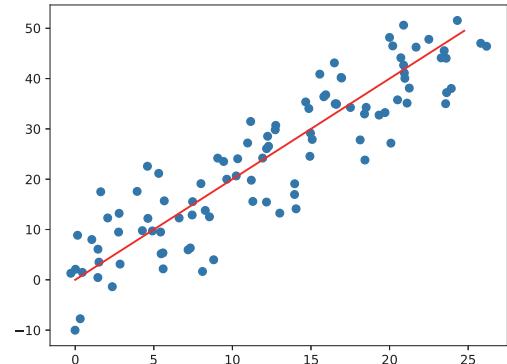
```
>>> x = np.arange(0,25,0.25); y = 2*x      Enter    ← y = 2x を満たすデータ列 x, y を作成する。
>>> xr = x + np.random.normal(0,1,100)   Enter    ← x を乱数で搅乱したデータ列
>>> yr = y + np.random.normal(0,6,100)   Enter    ← y を乱数で搅乱したデータ列
```

これによりノイズを含んだ2つのデータ列 `xr`, `yr` ができた。この散布図（p.103 「3.1.15.2 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）

```
>>> f = plt.figure()      Enter
>>> g1 = plt.plot(x,y,color='red') Enter
>>> g2 = plt.scatter(xr,yr)   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。搅乱する前のデータ `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.          , 0.90157731],
       [0.90157731, 1.          ]])
```

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

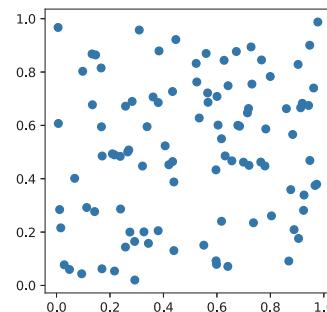
```
>>> xr2 = np.random.rand(100)  Enter    ← 100 個の一様乱数
>>> yr2 = np.random.rand(100)  Enter    ← 100 個の一様乱数
```

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
>>> 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.          , 0.15953472],
       [0.15953472, 1.        ]]) ←相関行列
```

`xr2`, `yr2` の間に相関はほぼ見られないことがわかる.

3.1.14.8 データのシャッフル

`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
```

`permutation`, `shuffle` は, 先に説明した `RandomState` オブジェクトに対して実行することもできる.

3.1.15 データの可視化 (2)

3.1.15.1 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.

例. 正規乱数のヒストグラム

```
>>> np.random.seed(0) [Enter] ←乱数の初期化30
>>> a = np.random.normal(50,10,100000) [Enter] ← $\mu = 50, \sigma = 10$  の正規乱数を  $10^5$  個生成
>>> plt.figure( figsize=(5,2) ) [Enter] ←描画サイズの設定
<Figure size 500x200 with 0 Axes> ←戻り値
>>> 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 60 のようなヒストグラムが表示される.

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.

³⁰ 実行例の再現性のために行った.

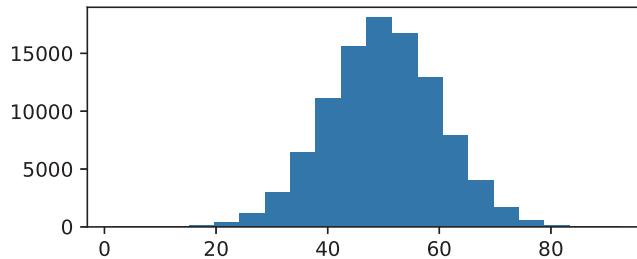


図 60: ヒストグラムの表示

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
[3.0000e+00 2.0000e+00 2.3000e+01 9.9000e+01 4.0900e+02 1.1490e+03
3.0180e+03 6.4130e+03 1.1081e+04 1.5631e+04 1.8079e+04 1.6767e+04
1.2937e+04 7.9430e+03 4.0020e+03 1.6830e+03 5.5600e+02 1.5800e+02
4.0000e+01 7.0000e+00]

>>> print( d[1] ) [Enter] ← 戻り値の 2 番目の要素の確認
[ 1.47882347 6.02576825 10.57271303 15.11965782 19.6666026 24.21354738
28.76049217 33.30743695 37.85438173 42.40132652 46.9482713 51.49521608
56.04216086 60.58910565 65.13605043 69.68299521 74.22994 78.77688478
83.32382956 87.87077435 92.41771913]
```

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} 以上 b_n 未満」である。ただし、最終の区間のみ「 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.96 「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）

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, 95, 20 ) [Enter] ← 集計区間の境界値の配列を作成
>>> plt.figure( figsize=(5,2) ) [Enter] ← 描画サイズの設定
<Figure size 500x200 with 0 Axes> [Enter] ← 戻り値
>>> 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 61 のようなヒストグラムが表示される。

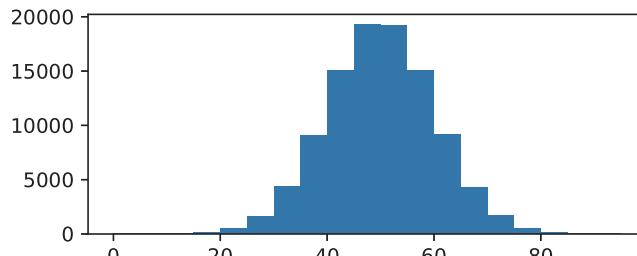


図 61: ヒストグラムの表示（階級指定）

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  
[2.0000e+00 1.0000e+00 2.4000e+01 1.1400e+02 5.2800e+02 1.5880e+03 ← 階級毎の度数の  
4.3620e+03 9.0270e+03 1.5030e+04 1.9257e+04 1.9214e+04 1.5095e+04 配列  
9.1420e+03 4.3150e+03 1.6820e+03 4.7800e+02 1.1600e+02 2.3000e+01  
2.0000e+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. 60. ← 階級の境界値  
65. 70. 75. 80. 85. 90. 95.] の配列
```

注意) NumPy の digitize 関数と matplotlib の hist 関数では、区間の考え方と同じではないので注意すること。

3.1.15.2 散布図

散布図のプロットには scatter 関数を使用する。サンプルプログラムを nplot04.py に示す。

プログラム : nplot04.py

```
1 # coding: utf-8  
2 # モジュールの読み込み  
3 import numpy as np  
4 import matplotlib.pyplot as plt  
5  
6 # データ列の生成  
7 datx = np.random.normal(50, 10, 4000)  
8 daty = np.random.normal(50, 10, 4000)  
9  
10 # 散布図の表示  
11 plt.scatter(datx, daty, alpha=0.2)  
12 plt.xlabel('x')  
13 plt.ylabel('y')  
14 plt.grid(ls='--', lw=0.8, alpha=1.0)  
15 plt.title('mean:50, SD:10, total: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、それぞれの軸の平均が 50 ($\mu = 50$)、標準偏差が 10 ($\sigma = 10$) となる場合の例である。

このプログラムを実行すると図 62 のようなプロットが表示される。

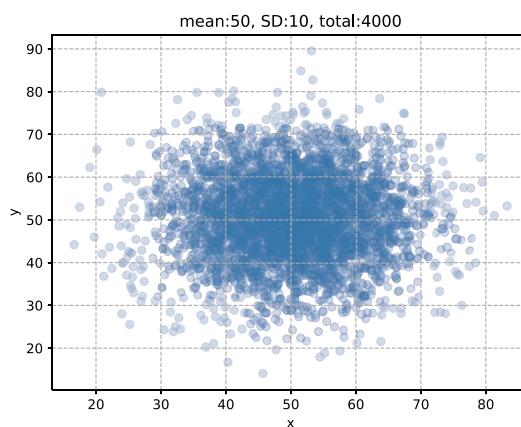


図 62: プロットの表示

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。

3.1.15.3 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。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、任意に作成した定義域に対する値域のデータ列をプロットするものである。

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 # coding: utf-8
2 import numpy as np          # NumPyの読み込み
3 import matplotlib.pyplot as plt # matplotlibの読み込み
4
5 # データ列の生成
6 datx = np.arange(-1.0, 1.2, 0.2)
7 daty = -datx**2 + 1
8
9 # 棒グラフの表示
10 (fig, ax) = plt.subplots(1, 2, figsize=(9, 4))
11 plt.subplots_adjust(wspace=0.4)
12
13 ax[0].bar(datx, daty, width=0.17)           # 縦の棒グラフ
14 ax[0].set_xlabel('x'); ax[0].set_ylabel('y')
15 ax[0].set_xlim(-1, 1)
16 ax[0].set_title('-x^2+1')
17
18 ax[1].barh(datx, daty, height=0.17)         # 横の棒グラフ
19 ax[1].set_xlabel('y'); ax[1].set_ylabel('x')
20 ax[1].set_ylim(-1, 1)
21 ax[1].set_title('-x^2+1')
22
23 plt.show()

```

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 63 のようなプロットが表示される。

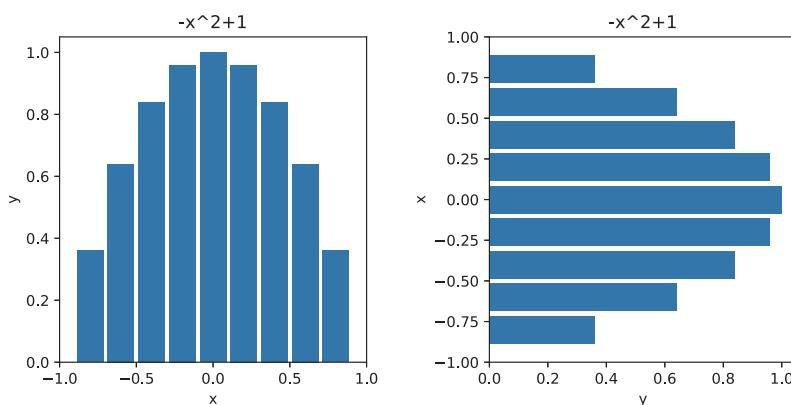


図 63: 棒グラフの表示

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 # coding: utf-8
2 import numpy as np          # NumPyの読み込み
3 import matplotlib.pyplot as plt # matplotlibの読み込み
4
5 # データ列の生成
6 daty = [13515271, 8839469, 7483128, 2610353]
7 lbl = ['Tokyo', 'Osaka', 'Aichi', 'Kyoto']
8
9 # 棒グラフの表示
10 (fig, ax) = plt.subplots(1, 2, figsize=(8, 4))
11 plt.subplots_adjust(wspace=0.4)
12 fig.suptitle('Populations 2015')
13
14 ax[0].bar(lbl, daty)           # 縦の棒グラフ

```

```

15 ax[0].set_xlabel('prefecture')
16 ax[0].set_ylabel('population')
17
18 ax[1].barh(lbl, daty) # 横の棒グラフ
19 ax[1].set_xlabel('population')
20 ax[1].set_ylabel('prefecture')
21
22 plt.show()

```

このプログラムを実行すると図 64 のようなプロットが表示される。

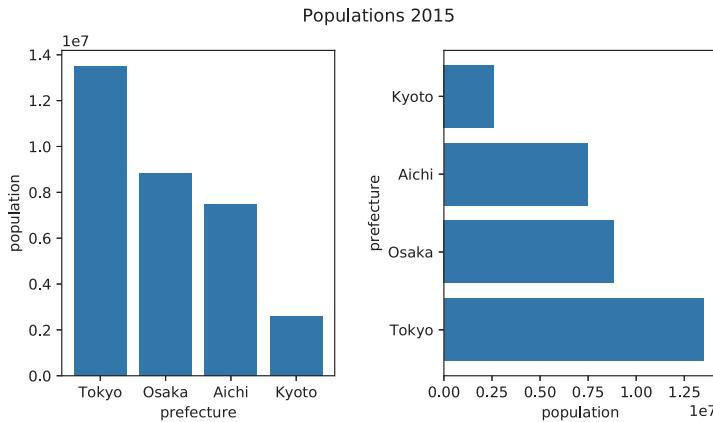


図 64: 定義域にラベルを取る棒グラフ

3.1.15.4 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 360 進法の度数で与える。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 # coding: utf-8
2 import numpy as np                      # NumPyの読み込み
3 import matplotlib.pyplot as plt          # matplotlibの読み込み
4
5 # データの作成
6 x = np.array( [1,3,5,7,9] )             # 値の配列
7
8 #--- 作図 ---
9 (fig,ax) = plt.subplots( 2, 3, figsize=(12,8) )
10 # データのみ（各種引数なし）の作図
11 ax[0,0].pie( x )
12 ax[0,0].set_title('"(a) no args"')
13
14 # 各種引数を指定して作図
15 lbl = ['1st','2nd','3rd','4th','5th']    # ラベルのリスト
16 ax[0,1].pie( x, labels=lbl, startangle=90, counterclock=False,
17               autopct='%.1f%%' )
18 ax[0,1].set_title('"(b) with args"')
19
20 # explodeの例
21 expd = [ 0, 0, 0.1, 0, 0 ]
22 ax[0,2].pie( x, labels=lbl, startangle=90, counterclock=False,
23               autopct='%.1f%%', explode=expd )
24 ax[0,2].set_title('"(c) with explode"')
25
26 # 色指定の例
27 clr = [ '#00ff00', '#0000ff', '#00ffff', '#ff00ff', '#ffff00' ]
28 ax[1,0].pie( x, labels=lbl, startangle=90, counterclock=False,
29               autopct='%.1f%%', explode=expd, colors=clr )
30 ax[1,0].set_title('"(d) color change"')
31
32 # 濃淡の例
33 clr = [ '0.4', '0.5', '0.6', '0.7', '0.8' ]      # 明るさ（暗0～1.0明）
34 ax[1,1].pie( x, labels=lbl, startangle=90, counterclock=False,
35               autopct='%.1f%%', explode=expd, colors=clr )
36 ax[1,1].set_title('"(e) gray scale"')
37
38 # ラベルのスタイル指定の例
39 ax[1,2].pie( x, labels=lbl, startangle=90, counterclock=False,
40               autopct='%.1f%%', explode=expd,
41               textprops={'color': 'white', 'weight': 'bold' } )
42 ax[1,2].set_title('"(f) label style"')
43
44 plt.show()
```

このプログラムを実行すると図 65 のような円グラフが作成される。

図 65 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。

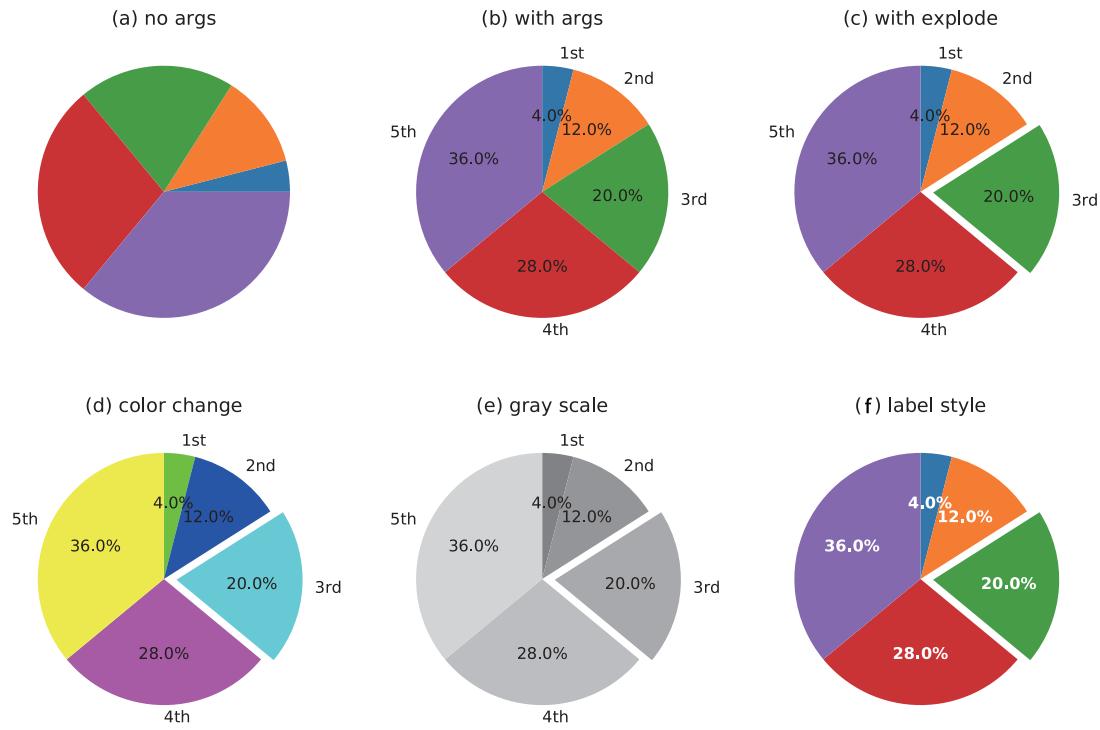


図 65: 円グラフ

3.1.15.5 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], 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`

```

1 # coding: utf-8
2 # モジュールの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 # データの作成
7 d1 = np.random.normal(0, 1, 1000)      # N[0, 1]
8 d2 = np.random.chisquare(4, 1000)       # 自由度4の  $\chi^2$  分布
9 d3 = np.random.gamma(2, 2, 1000)        # 形状母数2, 尺度母数2のガンマ分布
10
11 # プロット
12 plt.figure( figsize=(6, 3) )
13 plt.boxplot( [d1, d2, d3], labels=['N[0, 1]', ' $\chi^2(k=4)$ ', ' $\Gamma(k=2, \theta=2)$ ' ] )
14 plt.ylim(-4, 15)
15 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 66(a) のようなプロットが表示される。

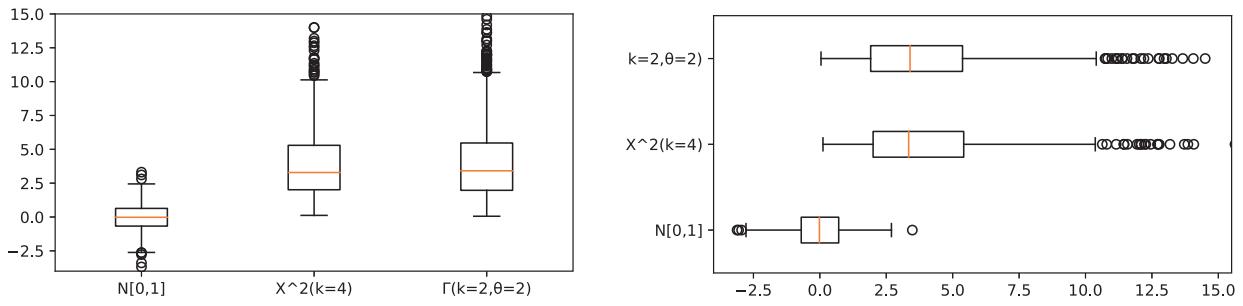
`boxplot` 関数に引数「`vert=False`」を与えると横方向の箱ひげ図が描画される。プログラム `nplot04-3.py` の 13~14 行目を

```

plt.boxplot( [d1, d2, d3], labels=['N[0, 1]', ' $\chi^2(k=4)$ ', ' $\Gamma(k=2, \theta=2)$ ' ], vert=False )
plt.xlim(-4, 15.5)

```

と書き換えて実行すると図 66(b) のようなグラフがプロットされる。



(a) デフォルト
図 66: 箱ひげ図の表示

(b) `vert=False` でプロット

【解説】箱ひげ図

箱ひげ図はデータの度数の分布を表現するものであり、25パーセント点 ($Q_{1/4}$)、50パーセント点 ($Q_{2/4}$: 中央値)、75パーセント点 ($Q_{3/4}$) を「箱」で表示する。また、箱の長さ $Q_{3/4} - Q_{1/4}$ を IQR (interquartile range) として、有効なデータの範囲を $Q_{1/4} - IQR \sim Q_{3/4} + IQR$ であると考え、その範囲内にないデータは「外れ値」とみなす。(図 67)

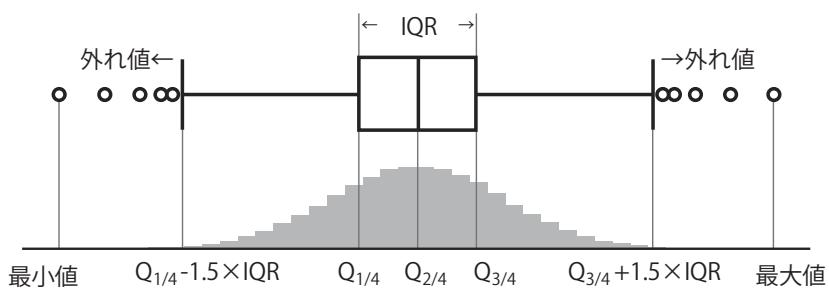


図 67: 箱ひげ図

3.1.16 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。

3.1.16.1 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 68 に示す。

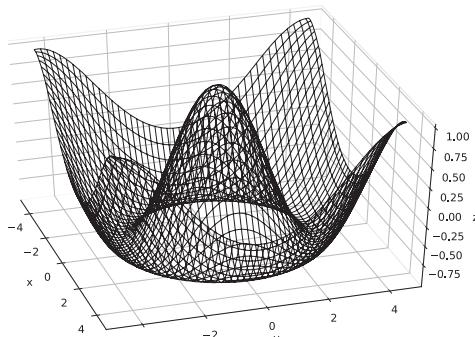


図 68: $z = \cos(\sqrt{x^2 + y^2})$ の概形

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 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 ( $90 \times 90$ ) 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 ( $90 \times 90$ ) 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 ( $90 \times 90$ ) の配列であることがわかる
```

3.1.16.2 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 69 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 69 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 69 に図解する。

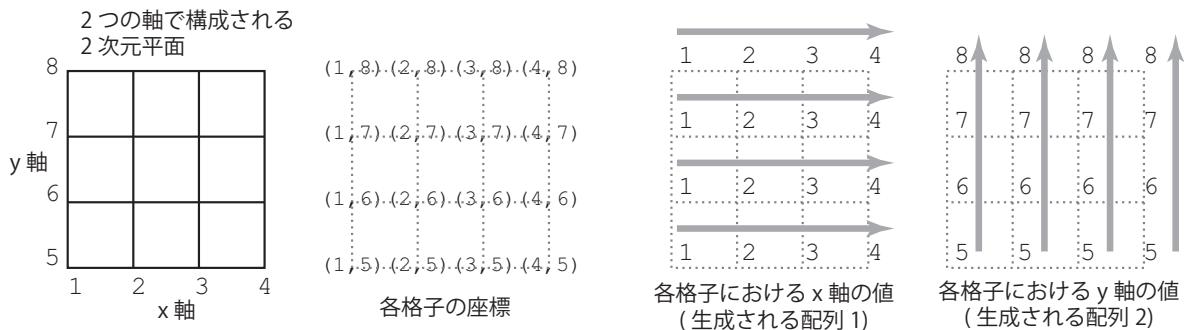


図 69: meshgrid が作成する配列

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 69 の右端のものと比べると逆になっている。

3.1.16.3 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。

3.1.16.4 ワイヤフレーム

先に挙げた関数 $z = \cos(\sqrt{x^2 + y^2})$ のワイヤフレームプロットを描画するプログラムを 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 plt.show()
```

13 行目にある plot_wireframe メソッドでワイヤフレームを生成している。このメソッドの第 1~第 3 引数に x, y, z それぞれの軸のデータを格納する配列を与える。キーワード引数には p.78 の表 20 に挙げるようなものが指定できる。軸のラベルを表示するには、14~16 行目にあるように set_xlabel, set_ylabel, set_zlabel メソッドを使用する。

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 70 のようなプロットが表示される。

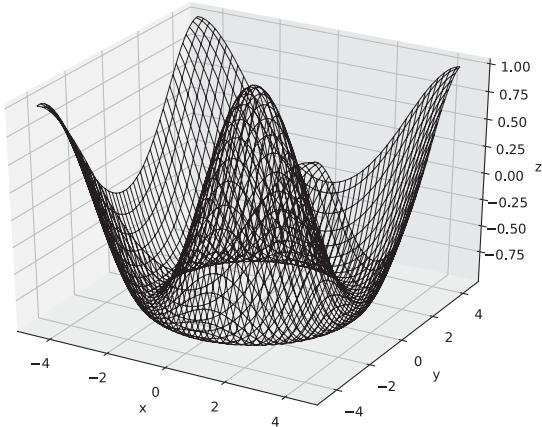


図 70: プロットの表示

3.1.16.5 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 71 に示す。

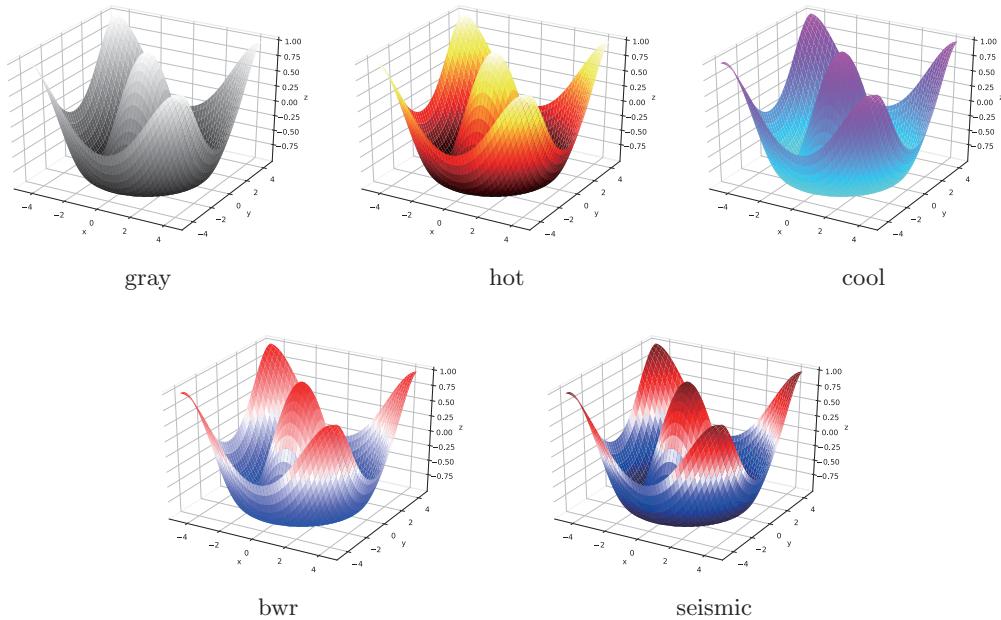


図 71: いくつかのカラーマップの例

単純で便利なカラーマップを表 24 に示す。

表 24: 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

これらカラーマップを使用した例を「3.1.25.4 サンプルプログラム：画像の三色分解」(p.143) で示す。

3.1.16.6 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。先に説明したワイヤフレームや面プロットの作成とは方法が異なり、描画対象は `Axes3DSubplot` オブジェクトで、これは `Figure` オブジェクトに対して `add_subplot(projection='3d')` メソッドを実行して作成する。得られた `Axes3DSubplot` オブジェクトに対して `bar3d` メソッドを実行することで 3 次元の棒グラフを描画する。

次に、 $z = \cos(\sqrt{x^2 + y^2}) + 1$ 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.sqrt(X**2+Y**2)) + 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 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。これを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72に示す。

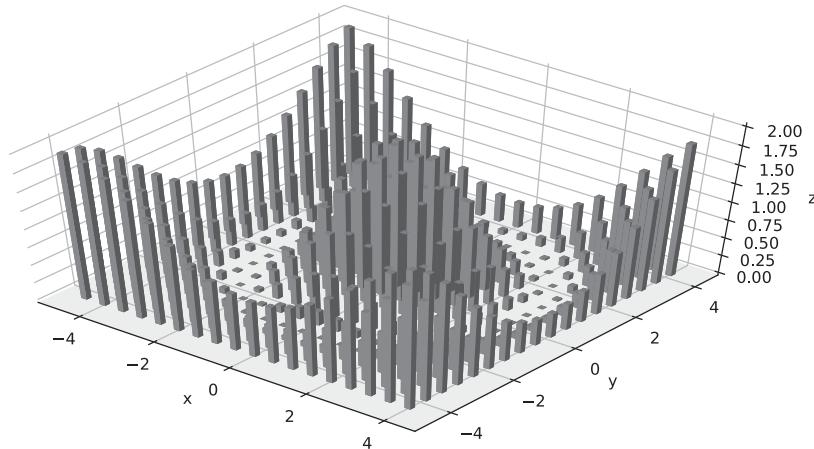


図 72: $z = \cos(\sqrt{x^2 + y^2}) + 1$ の棒グラフ

3.1.16.7 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。これを応用すると3次元の散布図を作成することができる。それを行うサンプルプログラムを scatter3d01.py に示す。

プログラム：scatter3d01.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # データの作成
5 x1 = np.random.normal(0,0.2,100)
6 y1 = np.random.normal(0,0.2,100)
7 z1 = np.random.normal(0,0.2,100)
8 #
9 x2 = np.random.normal(1,0.2,100)
10 y2 = np.random.normal(1,0.2,100)
11 z2 = np.random.normal(0,0.2,100)

```

```

12
13 # 3次元散布図
14 fig, ax = plt.subplots(subplot_kw={'projection': '3d'})
15 ax.scatter(x1,y1,z1)
16 ax.scatter(x2,y2,z2)
17 ax.set_xlabel('x')
18 ax.set_ylabel('y')
19 ax.set_zlabel('z')
20 plt.show()

```

このプログラムは乱数で構成される 2つのグループの点の集合を散布するもので、実行すると図 73 のようなグラフが表示される。

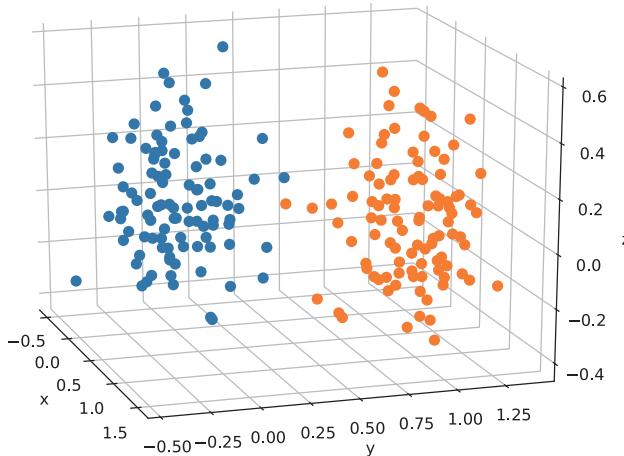


図 73: 3 次元の散布図の例

3.1.17 データの可視化：その他

3.1.17.1 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。

次に示すサンプルプログラム heatmap01.py は 2 次元のリストから、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

```

1 # coding: utf-8
2 # モジュールの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 w = 0.5      # 幅
7 ax = np.arange(-3,3+w,w)      # 横軸の値
8 ay = np.arange(-3,3+w,w)      # 縦軸の値
9 # 2次元データの作成
10 a = []
11 for y in ay:
12     a.append( 1 / (np.sqrt(ax**2+y**2)+1) )
13
14 # ヒートマップの描画
15 plt.figure( figsize=(5,5) )
16 plt.pcolor( a, cmap=plt.cm.hot )
17 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 74 の (a) である。

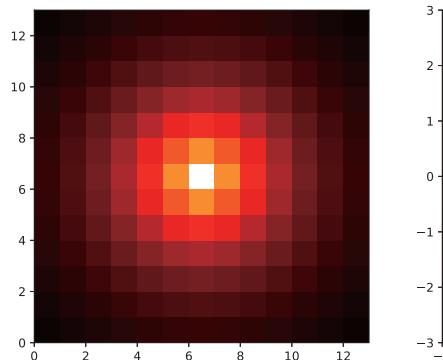
プログラム：heatmap02.py

```

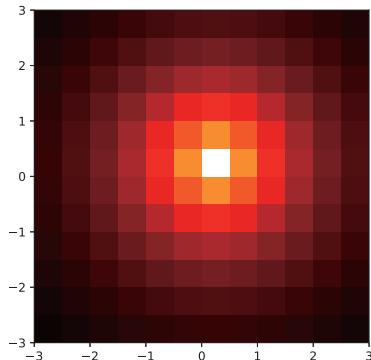
1 # coding: utf-8
2 # モジュールの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 w = 0.5      # 幅
7 ax = np.arange(-3,3+w,w)      # 横軸の値
8 ay = np.arange(-3,3+w,w)      # 縦軸の値
9 # 3次元データの作成
10 (X,Y) = np.meshgrid(ax,ay)
11 Z = 1 / (np.sqrt(X**2+Y**2)+1)
12
13 # ヒートマップの描画
14 plt.figure( figsize=(5,5) )
15 plt.pcolor( X,Y,Z, cmap=plt.cm.hot )
16 plt.show()

```

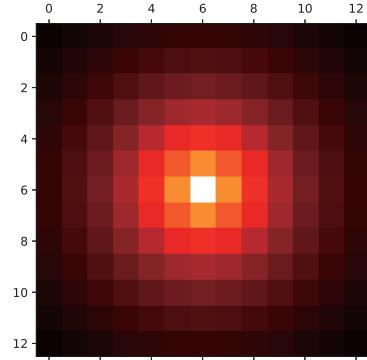
このプログラムを実行して作成したヒートマップが図 74 の (b) である。



(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 74: 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

```

1 # coding: utf-8
2 # モジュールの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 w = 0.5      # 幅
7 ax = np.arange(-3,3+w,w)      # 横軸の値
8 ay = np.arange(-3,3+w,w)      # 縦軸の値
9 # 2次元データの作成
10 a = []
11 for y in ay:
12     a.append( 1 / (np.sqrt(ax**2+y**2)+1) )
13
14 # ヒートマップの描画
15 (fig,ax) = plt.subplots( figsize=(5,5) )
16 ax.matshow( a, cmap=plt.cm.hot )
17 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 74 の (c) である。`matshow` における作図の大きさは `subplots` の引数に指定（15 行目）し、`matshow` は `subplots` が返す要素の第 2 要素（プログラム中の `ax`）に対して実行する。

matshow による作図では、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。

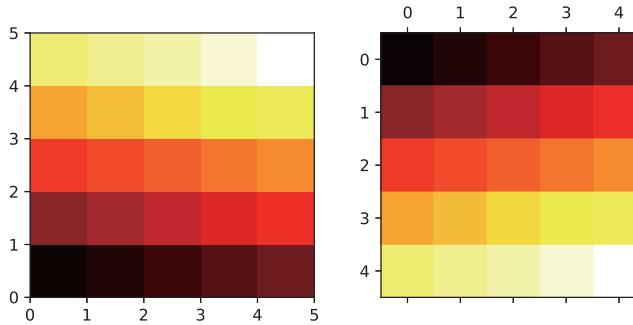
プログラム：heatmap04.py

```

1 # coding: utf-8
2 # モジュールの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5 # データの作成
6 a = np.arange(0,25,1).reshape( (5,5) )
7 # ヒートマップの描画
8 (fig,ax) = plt.subplots(1,2,figsize=(7,3))
9 ax[0].pcolor( a, cmap=plt.cm.hot )
10 ax[1].matshow( a, cmap=plt.cm.hot )
11 plt.show()

```

このプログラムを実行すると図 75 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 75: 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。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 heatmap05.py を次に示す。

プログラム：heatmap05.py

```

1 # coding: utf-8
2 # モジュールの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5 # データの作成
6 a = np.arange(0,25,1).reshape( (5,5) )
7 # ヒートマップの描画
8 (fig,ax) = plt.subplots(1,2,figsize=(7,3))
9 a0 = ax[0].pcolor( a, cmap=plt.cm.hot )      # 作図結果の戻り値を取得
10 print( 'type of pcolor: ', type(a0) )
11 a1 = ax[1].matshow( a, cmap=plt.cm.cool )    # 作図結果の戻り値を取得
12 print( 'type of matshow: ', type(a1) )
13 fig.colorbar( a0, ax=ax[0] )      # pcolor にカラーバーを添える
14 fig.colorbar( a1, ax=ax[1] )      # matshow にカラーバーを添える
15 plt.show()

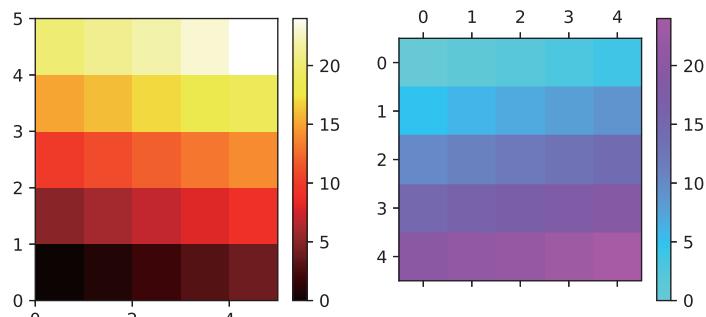
```

プログラムの 9, 11 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 (13, 14 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=カラーを添える描画面)

「カラーを添える描画面」は対象の AxesSubplot オブジェクトである。

heatmap05.py を実行すると図 76 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 76: カラーバーの表示

3.1.17.2 表の作成

table 関数を使用すると、2次元の配列を表の形で描くことができる。

書き方： `table(cellText=配列, bbox=[x1, y1, x2, y2])`

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 plttable01.py に示す。

プログラム：plttable01.py

```
1 # coding: utf-8
2 # ライブドリの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 #--- テーブル（2次元配列）の作成 ---
7 rtbl = np.random.randint(0,100,(3,3))
8 print(rtbl)      # 表示処理（文字：標準出力）
9
10 #--- 表の作成(1)：最も素朴な表示 ---
11 plt.figure(figsize=(3,2))
12 plt.table( cellText=rtbl, bbox=[0,0,1,1] )
13 plt.gca().axis('off')    # 軸の目盛りを非表示
14 plt.title('Simple table')
15 plt.show()
16
17 #--- 表の作成(2)：見出し、色、水平位置の指定 ---
18 # 色の定義
19 Ccell = [[ '#dddddd', '#ffffff', '#ffffff' ],
20           [ '#ffffff', '#dddddd', '#ffffff' ],
21           [ '#ffffff', '#ffffff', '#dddddd' ]]
22 Ccol = [ '#ffbbbb', '#bbffbb', '#bbbbff' ]      # 列見出しの色
23 Crow = [ '#bbbbff', '#fffbff', '#ffffbb' ]      # 行見出しの色
24 # プロット
25 plt.figure(figsize=(3,2))
26 plt.table(
27     cellText=rtbl,
28     bbox=[0,0,1,1],
29     cellLoc='center',
30     colLabels=['A', 'B', 'C'],
31     rowLabels=['X', 'Y', 'Z'],
32     cellColours=Ccell,
33     colColours= Ccol,
34     rowColours= Crow
35   )
36 plt.gca().axis('off')    # 軸の目盛りを非表示
37 plt.title('row/col label, colours, location')
38 plt.show()
```

プログラムの 7 行目で乱数配列を生成し、内容確認のため 8 行目でそれをターミナルウィンドウに表示する。11～15 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 $(0,0) \sim (1,1)$ 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。

19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 関数のキーワード引数 ‘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 29 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 関数のキーワード引数 ‘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 30～31 行目）

■ セルの背景色の設定

table 関数のキーワード引数 ‘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 19～23 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

```
[[16 77 51]
 [32 39 46]
 [50 61 41]]
```

と表示され、図 77 の (a), (b) の表が順番に表示される。

Simple table		
16	77	51
32	39	46
50	61	41

(a) 素朴な形の表

row/col label, colours, location			
	A	B	C
X	16	77	51
Y	32	39	46
Z	50	61	41

(b) 行、カラムの見出しと着色

図 77: 表の描画

3.1.18 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。

3.1.18.1 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のスケール（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=時間領域の最大値)`

データ個数は `fft` 関数に与えたデータ列の長さであり、そのデータの最終要素の時刻を時間領域の最大値として与える。`fftfreq` 関数は周波数スケールのデータ列を返す。この値を横軸として、`fftfreq` 関数で得られた周波数成分のデータ列を縦軸としてプロットすると、周波数領域のプロットができる。ただし、フーリエ変換の結果として得られるデータは複素数である³¹ ことに注意すること。

3.1.18.2 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)`

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`

```
1 # coding: utf-8
2 # ライブライアリの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import pylab
6
7 ######
8 # 波形データの生成 #
9 #####
10 ----- 時間軸データ -----
11 d_x = np.array([i/400.0 for i in range(400)])
12
13 ----- 正弦波 -----
14 d_sin = np.sin(4.0 * np.pi * d_x)
15
16 # 波形のプロット
```

³¹ NumPy での複素数の扱いについては後の「3.1.19 複素数の計算」(p.123) を参照のこと。

```

17 pylab.figure(figsize=(8,2))
18 plt.plot(d_x, d_sin, lw=1.5, color='black')
19 plt.xlabel('time') # 横軸ラベル
20 plt.ylabel('y') # 縦軸ラベル
21 plt.title('sin')
22 plt.show()
23
24 -----鋸歯状波（ノコギリ波）-----
25 d_saw = np.array([i/100.0-1.0 for i in range(200)]*2)
26
27 # 波形のプロット
28 pylab.figure(figsize=(8,2))
29 plt.plot(d_x, d_saw, lw=1.5, color='black')
30 plt.xlabel('time') # 横軸ラベル
31 plt.ylabel('y') # 縦軸ラベル
32 plt.title('Saw')
33 plt.show()
34
35 -----三角波 -----
36 tmp = [i/50.0-1.0 for i in range(100)]
37 d_tri = np.array((tmp+tmp[::-1])*2)
38
39 # 波形のプロット
40 pylab.figure(figsize=(8,2))
41 plt.plot(d_x, d_tri, lw=1.5, color='black')
42 plt.xlabel('time') # 横軸ラベル
43 plt.ylabel('y') # 縦軸ラベル
44 plt.title('Triangle')
45 plt.show()
46
47 -----方形波 -----
48 d_rct = np.array((-1.0)*100+[1.0]*100)*2
49
50 # 波形のプロット
51 pylab.figure(figsize=(8,2))
52 plt.plot(d_x, d_rct, lw=1.5, color='black')
53 plt.xlabel('time') # 横軸ラベル
54 plt.ylabel('y') # 縦軸ラベル
55 plt.title('Rect')
56 plt.show()
57
58 #####フーリエ変換#####
59 # 周波数軸データ
60 # データ長
61 n = len(d_x) # データ長
62 frq = n * np.fft.fftfreq(n, d=1.0)
63
64 -----正弦波の解析-----
65 f_sin = np.fft.fft(d_sin)
66 f_sin_n = np.sqrt(f_sin.real**2 + f_sin.imag**2)
67
68 # 振幅スペクトルのプロット
69 pylab.figure(figsize=(8,2))
70 plt.bar(frq, f_sin_n, color='black')
71 plt.xlim(-8,8)
72 plt.xlabel('Frequency (Hz)') # 横軸ラベル
73 plt.ylabel('Amplitude') # 縦軸ラベル
74 plt.title('Amplitude spectrum of sin')
75 plt.show()
76
77 -----鋸歯状波（ノコギリ波）の解析-----
78 f_saw = np.fft.fft(d_saw)
79 f_saw_n = np.sqrt(f_saw.real**2 + f_saw.imag**2)
80
81 # 振幅スペクトルのプロット
82 pylab.figure(figsize=(8,2))
83 plt.bar(frq, f_saw_n, color='black')
84 plt.xlim(-60,60)
85 plt.xlabel('Frequency (Hz)') # 横軸ラベル
86 plt.ylabel('Amplitude') # 縦軸ラベル
87 plt.title('Amplitude spectrum of Saw')
88
```

```

89 plt.show()
90
91 #----- 三角波の解析 -----
92 f_tri = np.fft.fft(d_tri)
93 f_tri_n = np.sqrt( f_tri.real**2 + f_tri.imag**2 )
94
95 # 振幅スペクトルのプロット
96 pylab.figure(figsize=(8,2))
97 plt.bar(frq, f_tri_n, color='black')
98 plt.xlim(-30,30)
99 plt.xlabel('Frequency (Hz)')      # 横軸ラベル
100 plt.ylabel('Amplitude')          # 縦軸ラベル
101 plt.title('Amplitude spectrum of Triangle')
102 plt.show()
103
104 #----- 方形波の解析 -----
105 f_rct = np.fft.fft(d_rct)
106 f_rct_n = np.sqrt( f_rct.real**2 + f_rct.imag**2 )
107
108 # 振幅スペクトルのプロット
109 pylab.figure(figsize=(8,2))
110 plt.bar(frq, f_rct_n, color='black')
111 plt.xlim(-60,60)
112 plt.xlabel('Frequency (Hz)')      # 横軸ラベル
113 plt.ylabel('Amplitude')          # 縦軸ラベル
114 plt.title('Amplitude spectrum of Rect')
115 plt.show()
116
117 ######
118 # フーリエ逆変換
119 ######
120 #----- 方形波の逆変換 -----
121 i_rct = np.fft.ifft(f_rct)
122
123 # 波形のプロット
124 pylab.figure(figsize=(8,2))
125 plt.plot(d_x, i_rct.real, lw=1.5, color='black')
126 plt.xlabel('time')    # 横軸ラベル
127 plt.ylabel('y')       # 縦軸ラベル
128 plt.title('Rect (Fourier inverse transform)')
129 plt.show()

```

解説：

11~56 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 78 に示す。

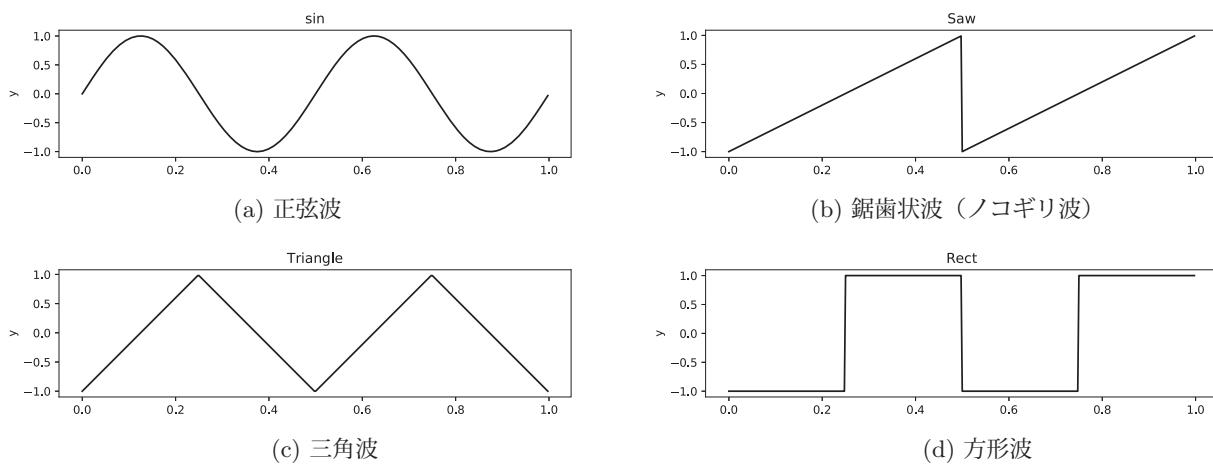


図 78: 波形データ（時間の関数）

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 62~115 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 62~63 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 79 に示す。（棒グラフの描画には matplotlib の bar 関数を使用して

いる)

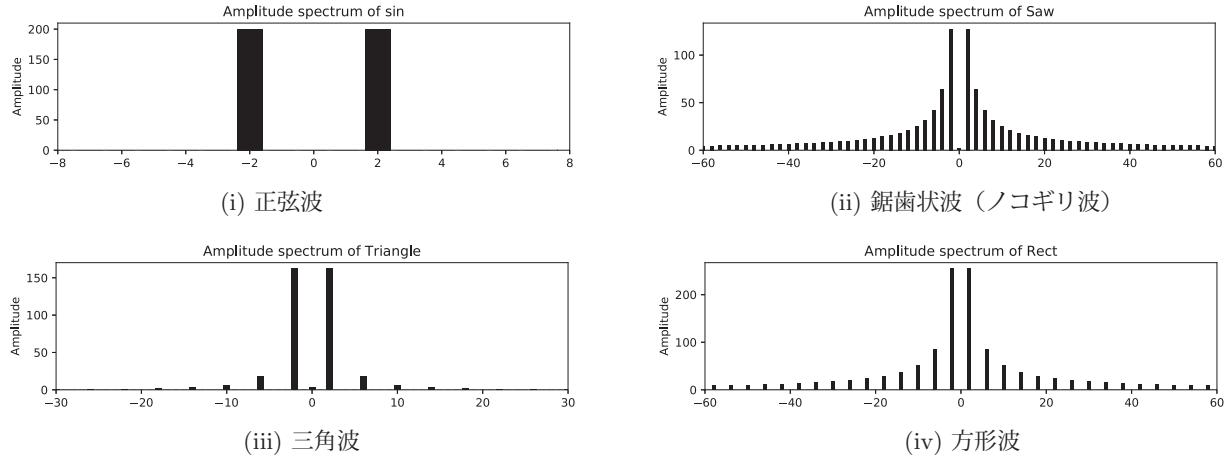


図 79: 振幅スペクトル（周波数の関数）

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。

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。
(121 行目) それをプロットしているのが 124~129 行目の部分であり、プロット結果を図 80 に示す。

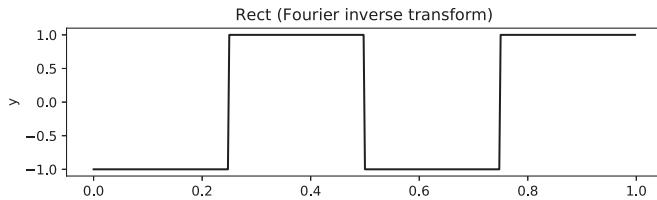


図 80: フーリエ逆変換で再構成した方形波

3.1.18.3 プロットのアスペクト比の指定

matplotlib によるプロットにおいて、作図時のアスペクト比（縦横比）を指定するには pylab モジュールを読み込んで（プログラム npfft01.py の 5 行目）関数 figure を使用する。

書き方： pylab.figure(figsize=(横, 縦))

3.1.18.4 フーリエ変換を使用する際の注意

フーリエ変換で時間軸上の波形を周波数成分に変換する際には注意しなければならないことがある。離散的で有限な長さのデータをフーリエ変換すると、与えられたデータ列が繰り返されるものと見做した上での周波数成分が得られる。先のサンプルプログラムは、2Hz の単純な波形がそのまま繰り返されるという前提の信号解析であり、実際の信号解析ではそのようなことはあり得ない。すなわち、波形データを無思慮に切り出してフーリエ変換を実行すると、元の波形データには含まれない周波数成分が現れてしまう。これは、データ列の時間軸の両端を繋いだ形の波形が繰り返されていると見做した結果である。このような問題を解決するには、波形データを適切に切り出すための窓関数³²を推定しながら、それを適用する形でフーリエ変換を実行しなければならない。

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。

³² 実際の周波数成分のみを取り出すために、適切な時間領域のデータを切り出す関数。

3.1.19 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
```

3.1.19.1 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
array([nan, 2.]) ←実数とならない要素が nan (非数) になっている
```

参考。配列が実数か複素数化を判定すには `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
True ←複素数を含む（複素数の配列である）
```

3.1.19.2 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
```

3.1.19.3 共役複素数

共役複素数を求めるには `conj` 関数（あるいはメソッド）を使用する。

例. 共役複素数

```
>>> 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）
```

3.1.19.4 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

```
>>> np.pi/4, np.pi/2 [Enter] ← $\pi/4, \pi/2$  を求めてみる  
(0.7853981633974483, 1.5707963267948966) ←計算結果
```

3.1.20 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。

3.1.20.1 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。

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]])
```

3.1.20.2 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.],  
       [ 0., 0., 0.]])           ←処理結果
```

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.],  
       [ 0., 0., 0., 0.]])        ←2 行 4 列のゼロ行列
```

例. 全て 1 の配列の生成: ones 関数

```
>>> 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.],  
       [ 1., 1., 1.]])           ←処理結果
```

例. 単位行列の生成: identity 関数

```
>>> np.identity( 5 ) [Enter]    ← $5 \times 5$  の単位行列  
array([[ 1., 0., 0., 0., 0.],  
       [ 0., 1., 0., 0., 0.],  
       [ 0., 0., 1., 0., 0.],  
       [ 0., 0., 0., 1., 0.],  
       [ 0., 0., 0., 0., 1.]])    ←処理結果
```

3.1.20.3 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.],  
       [ 1., 1., 1., 1.],  
       [ 1., 1., 1., 1.]])           ←結果表示  
>>> 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],  
       [ 3.1416, 3.1416, 3.1416, 3.1416],  
       [ 3.1416, 3.1416, 3.1416, 3.1416]])    ←結果表示
```

参考) 疎な行列の作成

疎な行列（スパース行列：sparse matrix）³³ 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。

³³ 大部分の要素が 0 であるような行列。

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],  
       [7, 7, 7, 7, 7, 7],  
       [7, 7, 7, 7, 7, 7]]) ← 結果表示
```

3.1.20.4 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  $v_1 = (1, 2, 3)$  の作成  
>>> v2 = np.array( [4,5,6] ) [Enter] ← ベクトル  $v_2 = (4, 5, 6)$  の作成  
>>> np.dot(v1,v2) [Enter] ← 内積  $v_1 \cdot v_2$  の計算  
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  $v_1 \cdot v_2$  の計算  
32 ← 内積の値
```

3.1.20.5 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],  
       [4, 5, 6],  
       [7, 8, 9]]) ← 結果表示  
>>> 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）
```

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],  
       [0, 22, 0],  
       [0, 0, 33]]) ← 結果表示
```

3.1.20.6 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],
       [ 5,  2, -3],
       [ 2, -1,  3]])
>>> np.transpose(a) [Enter] ← 転置の実行(1)
array([[ 1,  5,  2],
       [ 0,  2, -1],
       [-2, -3,  3]])
>>> a.T [Enter] ← 転置の実行(2)
array([[ 1,  5,  2],
       [ 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。

3.1.20.7 行列式と逆行列

例. 行列式を求める (先のつづき): det 関数

```
>>> 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 (つづき): inv 関数

```
>>> np.linalg.inv(a) [Enter] ← 逆行列を求める
array([[ 0.14285714,  0.0952381 ,  0.19047619],
       [-1.          ,  0.33333333, -0.33333333],
       [-0.42857143,  0.04761905,  0.0952381 ]]) ← 結果表示
```

3.1.20.8 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 (つづき): eig 関数

```
>>> (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 ],
       [-0.85880917+0.j           , -0.85880917-0.j           , -0.90931800+0.j ],
       [-0.36590772-0.01350281j, -0.36590772+0.01350281j,  0.35731146+0.j ]]) ← 結果表示
```

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を配列にして返す。個々の固有値にはそれに対応する固有ベクトルがあり、それを行列（配列）にしたものを作成する。

eig 関数は固有値と固有ベクトルを次のような形式のタプルで返す。

(固有値の配列, 固有ベクトルの配列)

「固有ベクトルの配列」の各列が固有ベクトルとなっている。

注) eig 関数が返したタプルにおいて、固有値と固有ベクトルは順番に対応する。すなわち、**固有値の配列**の第 n 番目の固有値に対する固有ベクトルは、**固有ベクトルの配列**の第 n 番目の列である。従って、指定した固有値に対応する固有ベクトルを取り出すには、**固有ベクトルの配列**を転置した後に添え字を指定して取り出すといった作業が必要となる。

3.1.20.9 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],
       [-3,  0,  7],
       [ 9,  4, -6]])
>>> np.linalg.matrix_rank(a1) [Enter] ←ランクの算出
3 ←結果表示
>>> 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],
       [-3,  0,  7],
       [-4,  2, -10]])
>>> np.linalg.matrix_rank(a2) [Enter] ←ランクの算出
2 ←結果表示
```

3.1.20.10 複素共役行列

複素共役行列を求めるには conjugate 関数を使用する。

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],
       [ 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],
       [ 0.-0.j, -3.-1.j]])
```

3.1.20.11 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],
       [ 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],
       [-5.-7.j, -3.-1.j]])
```

3.1.20.12 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
1.4142135623730951 ←結果表示
```

3.1.21 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のサイズが大きい場合が多く、適宜ファイルに出力保存したり、それを読み込んで処理をするという流れになる。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。

3.1.21.1 テキストファイルへの保存

random パッケージの rand 関数を使用して大きな乱数表を作成して、それを CSV 形式³⁴ 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

```
>>> rnd = np.random.rand( 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(出力先ファイル名, 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 81 に示す。

L2	A	B	C	D	E	F	G
1	7.0586177E-01	9.8161076E-01	9.6601103E-01	8.9160988E-02	1.7624552E-01	6.6354880E-01	4.2147287E-01
2	7.0944301E-01	4.8800316E-01	4.3856332E-01	5.9453042E-01	1.1948329E-01	8.5148735E-01	4.1622966E-01
3	7.5425629E-01	1.9923005E-01	8.0172316E-01	2.4649451E-01	3.6265127E-01	1.1033137E-01	7.9105902E-01
4	5.9106786E-02	4.0207431E-01	7.8047742E-01	1.9629748E-01	4.0758277E-01	9.6288520E-01	3.7309372E-01
5	3.9081352E-01	6.8698652E-01	4.5855603E-01	1.1644451E-01	8.6609573E-01	2.1652566E-01	3.8099760E-01
6	1.6489224E-01	6.9020775E-01	4.6385445E-01	9.7552768E-01	3.8880095E-02	2.9494056E-03	3.2352230E-02
7	7.2729904E-01	8.4540898E-01	6.3104417E-01	4.0700006E-01	2.9498669E-01	3.2818641E-01	2.6988482E-01
8	5.9819136E-01	7.6903394E-01	4.3795073E-01	5.2329515E-01	8.2189522E-01	9.7185035E-01	6.5961091E-01
9	4.2947333E-01	6.7136186E-01	3.5284120E-01	6.2424114E-02	7.86895739E-01	9.8900315E-01	4.0814013E-02
10	8.2624744E-01	2.6035833E-01	4.2931562E-01	3.6102405E-01	9.2454230E-01	2.5976500E-01	7.5668125E-01
11	7.42040853E-01	5.3672794E-01	4.1130967E-01	1.3676242E-03	2.0832216E-01	3.8416228E-01	3.8210064E-01
12	9.5327874E-01	9.6415344E-01	4.4619234E-01	3.9782717E-01	8.6448164E-01	9.2265489E-01	9.8670874E-01
13	7.8811375E-01	4.8073127E-01	5.0499849E-01	4.8085178E-01	5.0479560E-01	9.9246350E-01	7.6266953E-01
14	2.9278840E-01	5.14395589E-01	3.2649005E-01	1.7486252E-01	3.1071240E-01	2.6462944E-01	9.1237733E-01
15	5.7602735E-01	4.4285908E-01	6.0423864E-01	1.3168516E-01	2.0861528E-02	9.3103036E-01	7.1858372E-01
16	2.9736532E-01	2.9200560E-01	1.2145600E-01	2.3980909E-01	2.1420448E-01	2.4229163E-01	3.0452012E-01

図 81: 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（つづき）

```
>>> np.random.seed(8) [Enter] ← seed の設定35  
>>> r = np.random.normal( 50, 30, (3,3) ) [Enter] ← 3 行 3 列の乱数配列  
>>> print( r ) [Enter] ← 内容確認  
[[ 52.7361415 82.738482 -8.40910927]  
 [ 8.40951403 -18.89474723 122.2950291 ] ← 内容表示  
 [101.83508506 116.13668854 73.84482918]]
```

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。

³⁴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。

³⁵p.91 「3.1.13.4 亂数の seed について」を参照のこと。

例. 書式付き保存 (つづき)

```
>>> np.savetxt('test01.csv', r, delimiter=',',  
...     fmt=['%3d', '%5.1f', '%.2f'])
```

出力結果のファイル : test01.csv

1	52, 82.7,-8.41
2	8,-18.9,122.30
3	101,116.1,73.84

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’

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 (つづき)

```
>>> np.savetxt('test02.csv', r, delimiter=',',  
...     encoding='utf-8',)  
...     fmt=['%3d', '%5.1f', '%.2f'],  
...     header='1列目,2列目,3列目')
```

出力結果のファイル : test02.csv

1	# 1列目,2列目,3列目
2	52, 82.7,-8.41
3	8,-18.9,122.30
4	101,116.1,73.84

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 (つづき)

```
>>> np.savetxt('test03.csv', r, delimiter=',',  
...     encoding='utf-8',)  
...     fmt=['%3d', '%5.1f', '%.2f'],  
...     header='1列目,2列目,3列目',  
...     comments='')
```

出力結果のファイル : test03.csv

1	1列目,2列目,3列目
2	52, 82.7,-8.41
3	8,-18.9,122.30
4	101,116.1,73.84

■ 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³⁶ 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

```
>>> catd = np.array([['みかん', 'orange'], ['りんご', 'apple'],  
...     ['バナナ', 'banana']])  
>>> print(catd)
```

←文字列の配列

←内容確認

```
[['みかん', 'orange'],  
 ['りんご', 'apple'],  
 ['バナナ', 'banana']]
```

←配列の内容

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='%',  
...     header='Japanese,English', comments='')
```

←ファイルへの出力

³⁶ 「カテゴリカルデータ」(categorical data) とも呼ばれる。

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

3.1.21.2 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。

例. CSV 形式のデータを読み込む（つづき）

```
>>> rnd2 = np.loadtxt('random.csv', delimiter=',') [Enter] ← CSV データの読み込み
>>> rnd == rnd2 [Enter] ← 元の配列との比較を試みる
array([[ True, True, True, True, True, True, True, True, True], ← 比較結果
       [ True, True, True, True, True, True, True, True, True],
       [ True, True, True, True, True, True, True, True, True], ← 中省略
       [ True, True, True, True, True, True, True, True, True]], dtype=bool)
```

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「3.1.22 行列の検査」を参照のこと）

`loadtxt` 関数の使用方法は次の通りである。

書き方 (1)： `np.loadtxt(入力元ファイル名, delimiter=区切り文字)`

この関数は、読み込んだデータを配列オブジェクトとして返す。

■ 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み込み

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。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 ファイルの読み込み

```
>>> 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）の読み込み

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。

例. サンプルデータの作成

```
>>> rdat = np.zeros( (5,10) ) [Enter] ←ここからサンプルデータの作成
>>> for i in range(10): [Enter]
...     rdat[:,i] = np.arange(i*10,i*10+5) [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.] ←5行10列の配列
 [ 3.  13.  23.  33.  43.  53.  63.  73.  83.  93.]
 [ 4.  14.  24.  34.  44.  54.  64.  74.  84.  94.]]
>>> 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
5	4, 14, 24, 34, 44, 54, 64, 74, 84, 94

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] ←列指定読み込み
>>> print( r1 ) [Enter] ←読み取り結果の確認
[10.  11.  12.  13.  14.] ←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。また、このキーワード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] ←複数列の読み込み
>>> print( r135 ) [Enter] ←読み取り結果の確認
[[10.  30.  50.]
 [11.  31.  51.]
 [12.  32.  52.]
 [13.  33.  53.]
 [14.  34.  54.]] ←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
```

この例のように、loadtxt 関数にキーワード引数

usecols=[n₁, n₂, n₃, ...]

を与えると、n₁, n₂, n₃, ... 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。

3.1.21.3 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。

例. バイナリファイルに対する I/O (つづき)

```
>>> 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  
>>> rnd == rnd2 [Enter] ←元の配列との比較を試みる  
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],  
           . . .  
       [ True, True, True, True, True, True, True, True, True, True]], dtype=bool)
```

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。(行列同士の比較に関しては「3.1.22 行列の検査」を参照のこと)

save と load の使用方法 :

```
np.save(出力先ファイル名, 配列オブジェクト)  
np.load(入力元ファイル名)
```

load 関数は読み込んだデータを配列オブジェクトとして返す。

3.1.21.4 データの圧縮保存と読み込み

サイズの大きな配列オブジェクトを複数取り扱う場合、それら配列をまとめて圧縮処理³⁷ して、1つのファイルとして保存することができる。

例. 配列オブジェクトの圧縮保存と読み込み (つづき)

```
>>> np.savez('random.npz',name1=rnd) [Enter] ←配列 rnd をファイル 'random.npz' に圧縮保存  
>>> arc = np.load('random.npz') [Enter] ←ファイル 'random.npz' から読み込み  
>>> arc['name1']==rnd [Enter] ←元の配列との比較を試みる  
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],  
           . . .  
       [ True, True, True, True, True, True, True, True, True, True]], dtype=bool)
```

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。(行列同士の比較に関しては「3.1.22 行列の検査」を参照のこと)

圧縮保存の使用方法 :

```
np.savez(出力先ファイル名, データ名=配列オブジェクト, …)  
np.load(入力元ファイル名)
```

圧縮データを load 関数で読み込むと NpzFile オブジェクトが返される。この NpzFile オブジェクトに添え字として [データ名] を付けることで、配列オブジェクトを参照することができる。先の例ではデータ名として 'name1' というものを与えて配列 rnd を 'random.npz' というファイルに保存している。それを読み込んで arc という NpzFile オブジェクトを得た後、データ名を付加して arc['name1'] として配列オブジェクトを参照している。このような方法で、複数の配列オブジェクトに異なるデータ名を付けて圧縮保存することができる。

³⁷ZIP フォーマットで圧縮している。複数のデータをまとめて管理するアーカイブの機能も備えている。

3.1.21.5 配列をバイトデータに変換して入出力に使用する

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。

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³⁸ ことができる。

³⁸ これに関しては拙書「Python3入門」で解説しています。

3.1.22 行列の検査

先に示した例のように、2つの行列（配列）の要素が全て等しいかどうかを調べる場合、比較演算子'=='を用いると、各要素の比較結果である真理値の行列が得られる。この比較結果の行列（真理値の行列）に対して all メソッドを使用すると全て True かどうかを更に判定することができる。

例. 行列の要素が全て等しいかどうかの判定（つづき）

```
>>> (rnd == rnd2).all() [Enter] ←全ての要素が True か判定  
True 全ての要素が True である
```

all の他にも、「少なくとも 1 つの要素が True である」ことを検査する any メソッドもある。

3.1.22.1 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  
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  
False ←検査結果
```

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  
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  
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
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
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
False                  ←等しい要素がないことがわかる
```

3.1.23 配列を処理するユーザ定義関数の実装について

NumPy は配列に対する処理を実行する便利な関数やメソッドを提供しているが、ユーザ独自の関数の実装が求められることも多い。素朴な方法としては、NumPy の配列の要素 1 つ 1 つに対して施す処理を for などの文で繰り返し実行するというものが挙げられる。ただし for 文による繰り返し処理は実行時間が大きくなることも多く、またプログラムの可読性が低下する原因にもなる。ここでは、ユーザ定義関数の実装において実行時間と可読性の問題を小さくするための方法について紹介する。

■ for, map, vectorize の比較

次のような関数を考える。

$$\sin(x) + \frac{1}{2} \sin(2x) + \frac{1}{3} \sin(3x) + \frac{1}{4} \sin(4x) + \frac{1}{5} \sin(5x) + \frac{1}{6} \sin(6x) + \frac{1}{7} \sin(7x) + \frac{1}{8} \sin(8x)$$

これはノコギリ波（鋸歯状波）の波形を近似的に表現した関数である。この関数を定義域となる配列データに対して実行することを考える。次に示すサンプルプログラム npfunctest01.py は $[-20, 20]$ の範囲の 0.0004 刻みのデータ列 x に対して、上に示した関数を fun として定義して適用するものである。

プログラム：npfunctest01.py

```
1 # coding: utf-8
2 # モジュールの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import time
6
7 x = np.arange(-20, 20, 0.0004)      # 定義域データの配列
8 n = len(x)                          # 要素の数
9 print('要素数：', n)
10
11 ##### 求める #####
12 # sin(x) + 1/2*sin(2*x) + 1/3*sin(3*x) 求める #
13 #####
14
15 # 要素に対する計算を実行する関数
16 def fun(x):
17     return np.sin(x) + 1.0/2.0*np.sin(x*2.0) + \
18         1.0/3.0*np.sin(x*3.0) + 1.0/4.0*np.sin(x*4.0) + \
19         1.0/5.0*np.sin(x*5.0) + 1.0/6.0*np.sin(x*6.0) + \
20         1.0/7.0*np.sin(x*7.0) + 1.0/8.0*np.sin(x*8.0)
21
22 ----- 実行時間テスト(1) -----
23 print('方法1：forによる繰り返し')
24 t1 = time.time()
25 ly1 = []
26 for i in range(n):
27     ly1.append(fun(x[i]))
28 y1 = np.array(ly1)
29 t = time.time() - t1
30 print(t, '秒\n')
31
32 plt.plot(x, y1)
```

```

33 plt.show()
34
35 #---- 実行時間テスト(2) -----
36 print('方法2：map関数による方法')
37 t1 = time.time()
38 ly2 = map(fun,x)
39 y2 = np.array(list(ly2))      # この時に計算が実行される
40 t = time.time() - t1
41 print(t,'秒\n')
42
43 plt.plot(x,y2)
44 plt.show()
45
46 #---- 実行時間テスト(3) -----
47 print('方法3：np.vectorizeによる方法')
48 t1 = time.time()
49 vfun = np.vectorize(fun)      # 関数が「ベクトル化」される
50 y3 = vfun(x)                 # 計算実行
51 t = time.time() - t1
52 print(t,'秒')
53
54 plt.plot(x,y3)
55 plt.show()

```

解説：

7行目で定義域のデータを生成している。16～20行目で関数 `fun` を定義しており、この関数を後の行で定義域の全要素に対して実行する。

25～28行目では `for` 文を用いて計算を行い、同様の計算を38～39行目では `map` 関数を使用して実行している。49行目では NumPy の `vectorize` 関数を使用して、関数 `fun` を `vfun` に変換している。この結果得られた `vfun` は NumPy の配列の全要素に対して一度に処理を施し、結果の値を要素として持つ配列を返す。

このプログラムを実行した結果図 82 のようなグラフが表示される。(3回表示される)

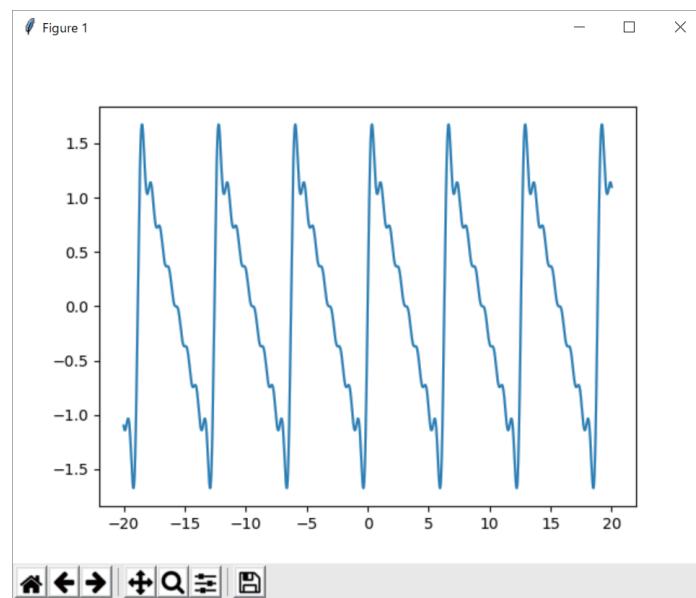


図 82: グラフの表示（3回表示される）

プログラムの実行に伴って次のように標準出力に出力され、`for` による方法、`map` による方法、`vectorize` による各方法での実行時間がわかる。

例. 実行結果の出力例

```
要素数： 100000
方法 1：for による繰り返し
2.2479827404022217 秒
方法 2：map 関数による方法
1.8539953231811523 秒
方法 3：np.vectorize による方法
1.2290003299713135 秒
```

for による実行が最も時間がかかり、 map 関数による実行はそれよりも若干早いことがわかる。NumPy の vectorize 関数でベクトル化された関数による実行が最も早い（for と比較して約 2 倍の速度である）ことがわかる。ただし、計算対象のデータの要素の数や型、実行する関数の定義、さらには計算機環境によって実行時間は変わるので注意が必要である。

3.1.24 行列の計算を応用した計算速度の改善の例

多くの場合において NumPy の行列計算の実行速度は非常に早い。計算量が多くなる処理に関しては、行列の計算が応用できるように極力工夫すべきである。先に示したプログラム npfunctest01.py は行列同士の積の計算を応用できる例であり、応用したプログラムを npfunctest02.py に示す。

プログラム：npfunctest02.py

```
1 # coding: utf-8
2 # ライブライアリの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import time
6
7 x = np.arange(-20, 20, 0.0004)      # 定義域データの配列
8 n = len(x)                         # 要素の数
9 print('要素数：', n)
10
11 #####行列の計算を応用したプログラム#####
12 # 行列の計算を応用したプログラム
13 ######
14 t1 = time.time()                  # 時間計測開始
15 ##### ここから #####
16 # sin(n*x) の配列を n=1～8 について作成して束ねる処理と
17 # 1/n の配列を作成する処理
18 #####
19 M = []; r = []
20 for n in range(1, 9):
21     y = np.sin(n*x)      # 各 n について sin(n*x) の配列を作成
22     M.append(y)
23     r.append(1/n)        # 1/n を作成
24
25 R = np.array(r)                  # 1/n の配列を NumPy の配列に変換
26 A = np.vstack(M)                # sin(n*x) の配列を 2 次元配列に合成
27 ##### ここまで #####
28
29 V = np.dot(A.T, R)             # 行列の積を利用して波形合成
30 t2 = time.time()               # 時間計測終了
31 print(t2-t1, '秒')
32
33 plt.plot(x, V.T)              # グラフをプロット
34 plt.show()
```

このプログラムを実行すると図 82 と同じグラフが表示されるが、同一の計算機環境（CPU:Intel Core i7 5500U 2.4GHz）における実行時間が 0.0312 秒であった。これは npfunctest01.py の np.vectorize による方法に比べても 40 倍以上の計算速度である。

3.1.24.1 要素の型によって異なる計算速度

配列の要素の型によって計算速度が異なることがあるので注意が必要である。ここでは、 500×500 のサイズの行列同士の積を求めるのに要する時間について例を挙げて考える。各種データ型毎に行列の積を計算する時間を測定する。

例. int64 の場合の計算時間

```
>>> a1i = np.arange( 0, 500000, 2, dtype='int64' ).reshape( (500,500) ) [Enter] ←行列 a1i
>>> a2i = np.arange( 1, 500000, 2, dtype='int64' ).reshape( (500,500) ) [Enter] ←行列 a2i
>>> import time [Enter] ←時間計測用に time モジュールを読み込む
>>> t1 = time.time() [Enter] ←開始時間
>>> a3i = np.dot(a1i,a2i) [Enter] ←行列の積の計算
>>> t2 = time.time() [Enter] ←終了時間
>>> print(t2-t1) [Enter] ←経過時間
0.169266939163208 ←単位は秒
```

例. float64 の場合の計算時間（先の例の続き）

```
>>> a1f = a1i.astype('float64') [Enter] ←float64 の行列 a1f を作成
>>> a2f = a2i.astype('float64') [Enter] ←float64 の行列 a2f を作成
>>> t1 = time.time() [Enter] ←開始時間
>>> a3f = np.dot(a1f,a2f) [Enter] ←行列の積の計算
>>> t2 = time.time() [Enter] ←終了時間
>>> print(t2-t1) [Enter] ←経過時間
0.06248879432678223 ←単位は秒
```

例. object の場合の計算時間（先の例の続き）

```
>>> a1oi = a1i.astype('object') [Enter] ←object の行列 a1oi を作成
>>> a2oi = a2i.astype('object') [Enter] ←object の行列 a2oi を作成
>>> t1 = time.time() [Enter] ←開始時間
>>> a3oi = np.dot(a1oi,a2oi) [Enter] ←行列の積の計算
>>> t2 = time.time() [Enter] ←終了時間
>>> print(t2-t1) [Enter] ←経過時間
7.531817436218262 ←単位は秒
```

これは CPU Intel Core i7-6770HQ 2.59GHz, RAM 16GB, OS Windows 10 Pro, NumPy with Intel MKL の環境下での実行結果である。上記の時間計測を 3 回実行して計算時間の平均を求めたものを表 25 に示す。

表 25: 型毎に異なる計算時間

型	平均計算時間（秒）	int64 の場合に対する比率
int64	0.165572	1
float64	0.059437	0.358981
object	7.665909	46.299683

上に示した計算機環境では float64 型の場合の計算が最も早いことがわかる。また、object 型の場合では計算速度が非常に小さくなることもわかる。

3.1.25 画像データの扱い

NumPy では画素の RGB 値の配列（図 83 参照）を画像として表示することができる。

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 84 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。

画素 $\blacksquare = [R, G, B, \alpha]$ R,G,B, α の 4 要素のリスト(α が無い形式もある)
グレースケールの場合はリストではなくスカラー値

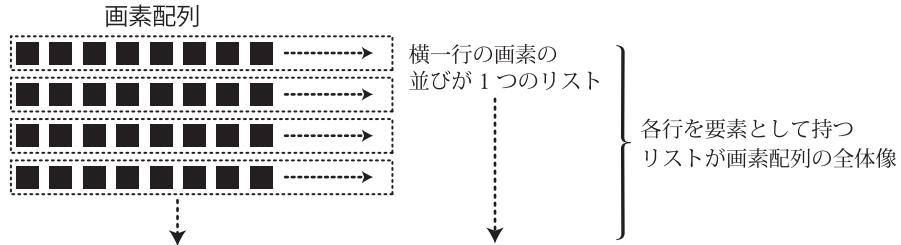


図 83: NumPy における画素の配列



図 84: 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 84 を画素の配列として作成する

```
>>> import numpy as np [Enter] ← NumPy の読み込み
>>> 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 [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。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 の読み込み
>>> plt.figure() [Enter] ← 描画処理の開始
<Figure size 640x480 with 0 Axes> ← 戻り値
>>> plt.imshow( imA ) [Enter] ← 画素の配列を表示するメソッド
<matplotlib.image.AxesImage object at 0x0000022B5C136208> ← 戻り値
>>> plt.show() [Enter] ← 描画の実行
```

この結果、imshow メソッドによって図 85 のように表示される。

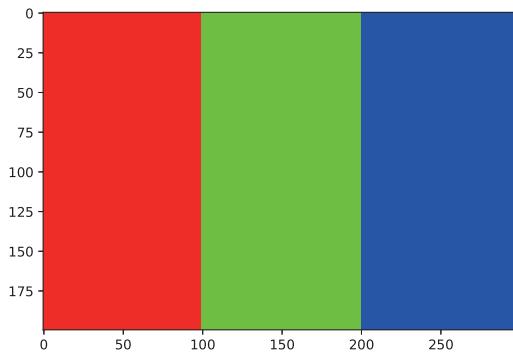


図 85: 画素の配列を表示した例

3.1.25.1 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.20) 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( np.uint8(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 84 のような画像が表示される)

3.1.25.2 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
>>> plt.figure() [Enter] ←描画処理の開始
<Figure size 640x480 with 0 Axes> ←戻り値
>>> plt.imshow( imA2 ) [Enter] ←画素の配列を表示
<matplotlib.image.AxesImage object at 0x0000022B5AF479B0> ←戻り値
>>> plt.show() [Enter] ←描画の実行
```

この結果、図 85 のように表示される。

課題) 図 86 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。

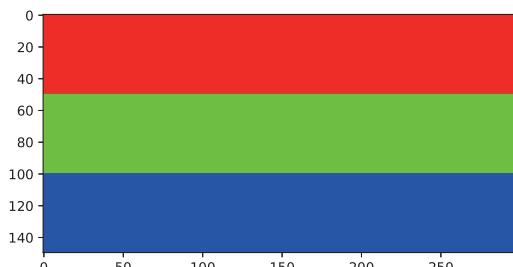


図 86: 作成する画像

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.20) を参照のこと。

3.1.25.3 OpenCVにおける画素の配列

OpenCV ライブラリ³⁹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み込み（先の例の続き）

```
>>> import cv2 [Enter] ←OpenCV ライブラリの読み込み
>>> imcv = cv2.imread( 'rgb01.png', cv2.IMREAD_UNCHANGED ) [Enter] ←画像ファイルの読み込み
>>> 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 85）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）

```
>>> plt.imshow( imcv ) [Enter] ←画像を表示
<matplotlib.image.AxesImage object at 0x000001788527D708>
>>> plt.show() [Enter] ←作図の実行
```

この結果、図 87 のように表示されてしまう。

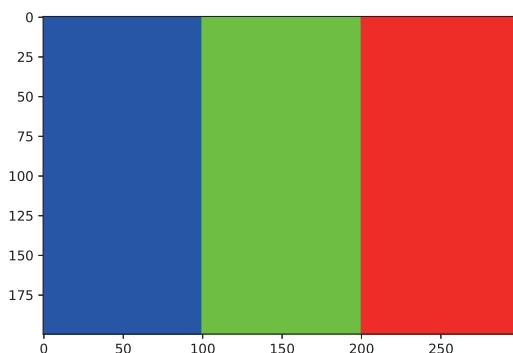


図 87: OpenCV の imread メソッドで読み込んだ配列を表示

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）

```
>>> plt.imshow( imcv2 ) [Enter] ←修正した画像を表示
<matplotlib.image.AxesImage object at 0x0000017885688F88>
>>> plt.show() [Enter] ←作図の実行
```

この結果、図 88 のように表示される。

³⁹p.1 「1.1 OpenCV」で解説している。

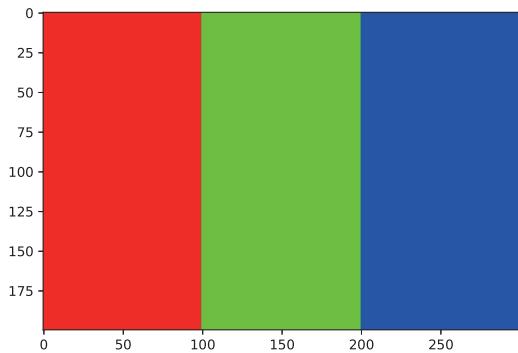


図 88: 色成分の順序を修正した画像

3.1.25.4 サンプルプログラム：画像の三色分解

PIL ライブライア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 # coding: utf-8
2 # ライブライアリの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5 from PIL import Image
6
7 # Image from https://pixabay.com/ja/
8 IM = Image.open('ImgCat.jpg')      # 読込み
9 (R,G,B) = IM.split()            # 3色分解
10 # 配列データに変換
11 im = np.asarray(IM)
12 r = np.asarray(R)
13 g = np.asarray(G)
14 b = np.asarray(B)
15
16 # 表示
17 (fig,axs) = plt.subplots(1,2,figsize=(8,2))
18 axs[0].imshow(im);                  axs[0].set_title('Original')
19 axs[1].imshow(r,cmap='Greys_r');    axs[1].set_title('Grayscale')
20 #plt.savefig('imshow01_exe1.eps')
21 plt.show()
22 (fig,axs) = plt.subplots(1,3,figsize=(12,2))
23 axs[0].imshow(r,cmap='Reds_r');     axs[0].set_title('Red')
24 axs[1].imshow(r,cmap='Greens_r');   axs[1].set_title('Green')
25 axs[2].imshow(r,cmap='Blues_r');    axs[2].set_title('Blue')
26 #plt.savefig('imshow01_exe2.eps')
27 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 (9 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 (11~14 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 89 のように表示される。

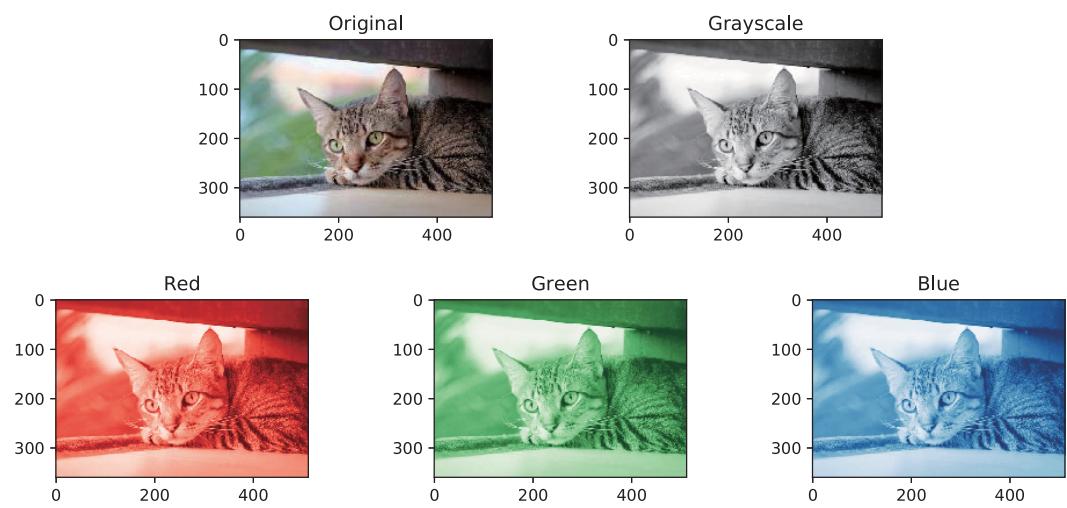


図 89: グレースケール表示と 3 色分解した画像の表示

3.1.26 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⁴⁰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 mat-
plotlib.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 AxesSubplot (p.85) に対して add_patch
メソッドで描画する。

3.1.26.1 四角形、円、橢円、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) indexCircle

■ 橢円： 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 # coding: utf-8
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import matplotlib.patches as pat
5
6 # 四角形
7 p1 = pat.Rectangle( xy=(0,0), width=1.5, height=1, fc='blue', ec='red', lw=4 )
8 p2 = pat.Rectangle( xy=(0,1), width=1.5, height=1, fc='blue', ec='red', lw=4,
9                     angle=30 )
10
11 # 円
12 p3 = pat.Circle( xy=(2.2,0.5), radius=0.5, fc='yellow', ec='blue', lw=8 )
13
14 # 橢円
15 p4 = pat.Ellipse( xy=(3.7,0.5), width=1.5, height=1,
16                   fc='cyan', ec='magenta', lw=6 )
17 p5 = pat.Ellipse( xy=(3.7,1.8), width=1.5, height=1,
18                   fc='cyan', ec='magenta', lw=6, angle=30 )
```

⁴⁰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。

```

20 # 正多角形
21 p6 = pat.RegularPolygon( xy=(5.2,0.5), radius=0.5, numVertices=5,
22                         fc='magenta', ec='green', lw=8 )
23 p7 = pat.RegularPolygon( xy=(5.2,1.8), radius=0.5, numVertices=5,
24                         fc='magenta', ec='green', lw=8, orientation=np.pi/2 )
25
26 # 描画
27 plt.figure( figsize=(8,3.53) )
28 ax = plt.gca()
29 ax.add_patch(p1); ax.add_patch(p2); ax.add_patch(p3); ax.add_patch(p4)
30 ax.add_patch(p5); ax.add_patch(p6); ax.add_patch(p7)
31 plt.xlim(-0.8,6.0); plt.ylim(-0.2,2.8)
32 plt.show()

```

このプログラムを実行すると図 90 のように表示される。

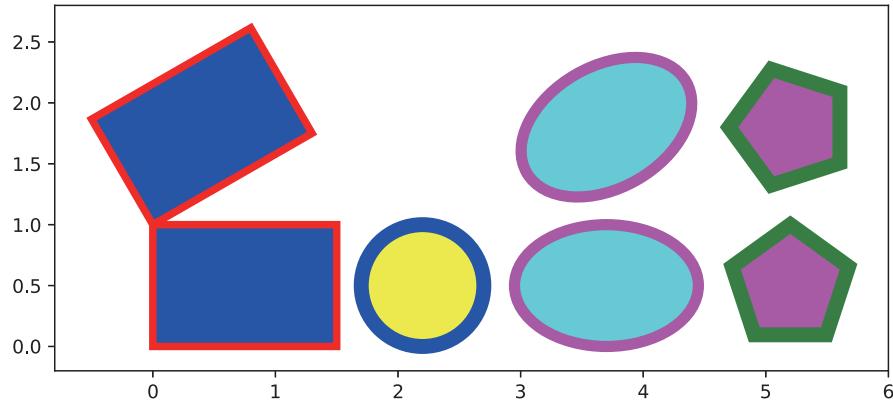


図 90: mPolygon2D01.py の実行結果

3.1.26.2 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 # coding: utf-8
2 import matplotlib.pyplot as plt
3 import matplotlib.patches as pat
4
5 # 円弧
6 p1 = pat.Arc( xy=(0,0.5), width=0.8, height=0.8,
7                 ec='red', lw=12, theta1=30, theta2=330 )
8 p2 = pat.Arc( xy=(0.3,0.5), width=0.8, height=0.8,
9                 ec='red', lw=12, theta1=330, theta2=30 )
10 p3 = pat.Arc( xy=(2,0.5), width=1.8, height=0.8,
11                ec='blue', lw=12, theta1=290, theta2=250 )
12 p4 = pat.Arc( xy=(4,0.5), width=1.8, height=0.8,
13                ec='blue', lw=12, theta1=290, theta2=250, angle=30 )
14
15 # 描画
16 plt.figure( figsize=(14,3.2) )
17 ax = plt.gca()
18 ax.add_patch(p1); ax.add_patch(p2)
19 ax.add_patch(p3); ax.add_patch(p4)
20 plt.xlim(-0.8,5.2); plt.ylim(-0.2,1.3)
21 plt.show()

```

このプログラムを実行すると図 91 のように 4 つの円弧が表示される。

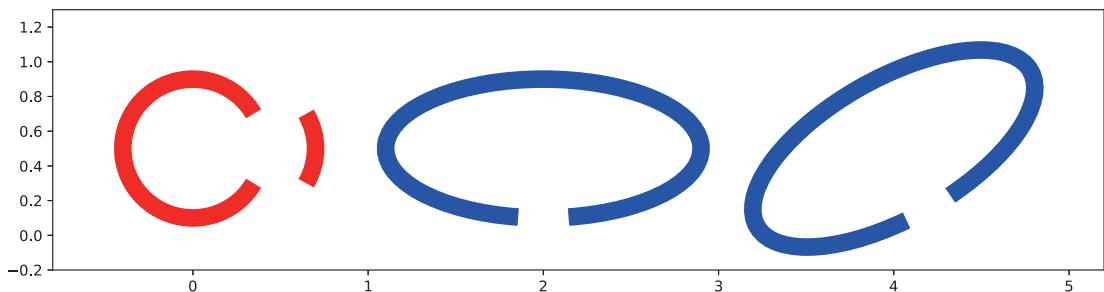


図 91: mPolygon2D02.py の実行結果

3.1.26.3 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。

`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 # coding: utf-8
2 import matplotlib.pyplot as plt
3 import matplotlib.patches as pat
4
5 # 円弧
6 # ポリゴン
7 p1 = pat.Polygon( xy=[(0,0),(1,0),(0,1)], fc='yellow', ec='green', lw=8 )
8
9 # 描画
10 plt.figure( figsize=(4,4) )
11 ax = plt.gca()
12 ax.add_patch(p1)
13 plt.xlim(-0.2,1.2); plt.ylim(-0.2,1.2)
14 plt.show()
```

このプログラムを実行すると図 92 のようなポリゴンが表示される。

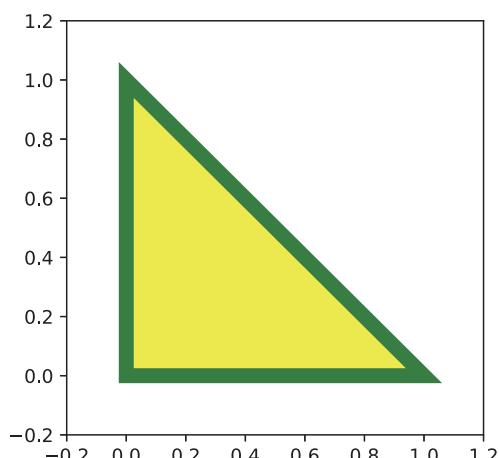


図 92: mPolygon2D03.py の実行結果

3.1.27 3 次元のポリゴン表示

先の「3.1.16 データの可視化：3 次元プロット」(p.108) 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⁴¹ について説明する。

⁴¹ 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。

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
import matplotlib.patches as pat
from mpl_toolkits.mplot3d import Axes3D
import mpl_toolkits.mplot3d.art3d as art3d
```

■ 3次元ポリゴン： `Poly3DCollection(3次元座標リスト, color=色)`

作成した `Poly3DCollection` オブジェクトは3次元プロット空間 (`Axes3D`) 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 # coding: utf-8
2 import matplotlib.pyplot as plt
3 import matplotlib.patches as pat
4 from mpl_toolkits.mplot3d import Axes3D
5 import mpl_toolkits.mplot3d.art3d as art3d
6
7 # ポリゴン
8 p1 = art3d.Poly3DCollection([(0,0,0),(0,2,0),(1,1,0)],color='red')
9 p2 = art3d.Poly3DCollection([(0,0,0),(1,1,1),(1,1,0)],color='green')
10 p3 = art3d.Poly3DCollection([(0,2,0),(1,1,1),(1,1,0)],color='blue')
11 p4 = art3d.Poly3DCollection([(0,0,0),(1,1,1),(0,2,0)],color='yellow')
12
13 # 描画
14 ax = Axes3D(plt.figure())
15 ax.add_collection3d(p1);    ax.add_collection3d(p2)
16 ax.add_collection3d(p3);    ax.add_collection3d(p4)
17 ax.set_xlim(-0.2,1.2);    ax.set_ylim(-0.2,2.5);    ax.set_zlim(-0.2,1.0)
18 ax.set_xlabel('x');    ax.set_ylabel('y');    ax.set_zlabel('z')
19 plt.show()
```

このプログラムを実行すると図 93 のようなポリゴンが表示される。

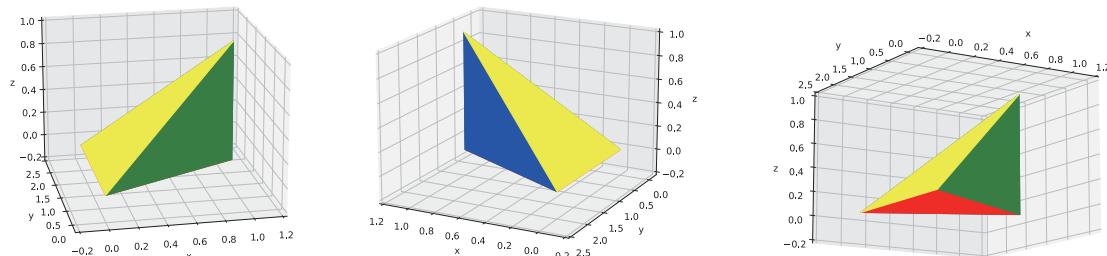


図 93: `mPolygon3D00.py` の実行結果（同一のグラフを様々な角度で表示した例）

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 94）ことに注意すること。

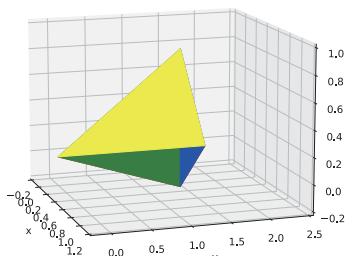


図 94: 不自然なポリゴンの配置の例

3.1.28 日付、時刻の扱い

3.1.28.1 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 の読み込み  
>>> d1 = np.datetime64('2022-01-01T00:00:00') [Enter] ← datetime64 オブジェクトの生成  
>>> d1 [Enter] ← 確認  
numpy.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。ただし、古い版の NumPy との互換性を保つために当面はタイムゾーン情報が扱えるが、将来においてタイムゾーン情報を扱う機能が廃止される可能性があるため、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
```

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(整数值, 単位)`

この形を取る場合、「1970-01-01T00:00:00」を基準とした `datetime64` オブジェクトが作成される。またその際「整数值」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 26 に示す。

表 26: 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				

3.1.28.2 timedelta64 クラス

`datetime64` オブ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。

3.1.28.3 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
```

3.1.29 その他の機能

3.1.29.1 基準以上／以下のデータの抽出

ある基準と比較して、それより大きい／小さいデータを抽出する関数に maximum , minimum がある。

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」から抽出して返す。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」から抽出する。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。

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 # coding: utf-8
2 # ライブドリの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 # データの作成
7 x = np.linspace(0, 20, 1000)
8 y = np.sin(x) / 2.0          # 基準列
9 r = np.random.rand(1000)*2 - 1 # データ列
10
11 # データとベースラインのプロット
12 plt.figure(figsize=(4, 4))
13 plt.plot(x, r, label='data', ls='None', marker='.')
14 plt.plot(x, y, label='base line', lw=2.5, c='red')
15 plt.ylim(-1.1, 1.1)
16 plt.legend()
17 plt.title('Data and baseline')
18 plt.show()
19
20 # maximumによる抽出
21 r1 = np.maximum(r, y)        # 基準列以上のデータを取り出す
22 plt.figure(figsize=(4, 4))
23 plt.plot(x, r1, ls='None', marker='.')
24 plt.ylim(-1.1, 1.1)
25 plt.title('Upper side')
26 plt.show()
27
28 # minimumによる抽出
29 r2 = np.minimum(r, y)        # 基準列以下のデータを取り出す
30 plt.figure(figsize=(4, 4))
31 plt.plot(x, r2, ls='None', marker='.')
32 plt.ylim(-1.1, 1.1)
33 plt.title('Lower side')
34 plt.show()
```

このプログラムは、データ配列 r を基準値の配列 y と比較して抽出する例である。7～18 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 95 の a）している。21～26 行目で基準値以上のデータ（図 95 の b）を、29～34 行目で基準値以下のデータ（図 95 の c）を抽出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 # coding: utf-8
2 # ライブドリの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
```

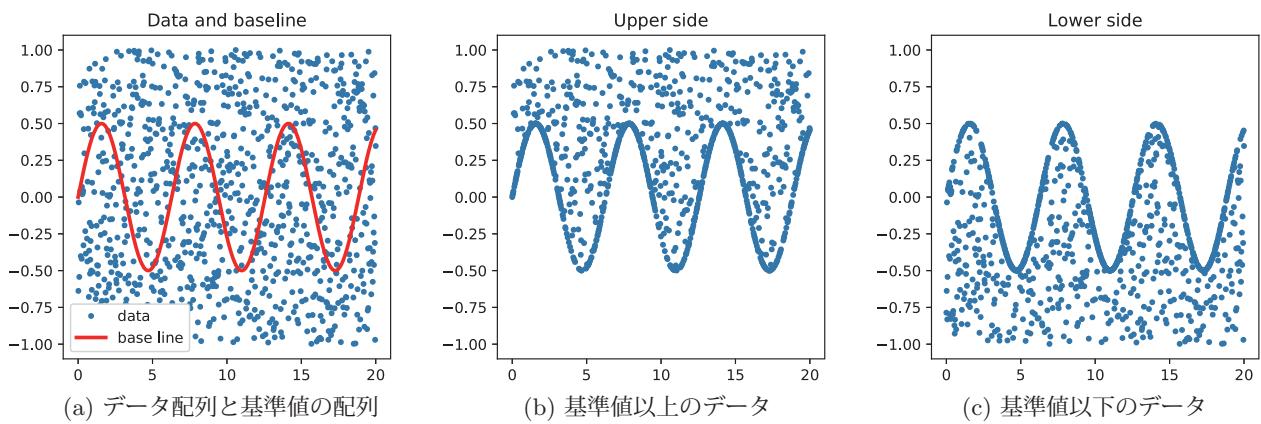


図 95: 上側, 下側のデータの抽出

```

5
6 # データの作成
7 x = np.linspace(0,20,1000)
8 y = np.sin(x)
9
10 # 元のデータのプロット
11 plt.figure(figsize=(4,4))
12 plt.plot(x,y,lw=2.5,c='red')
13 plt.ylim(-1.1,1.1)
14 plt.title('Original')
15 plt.show()
16
17 # スカラーによる抽出（上側）
18 y1 = np.maximum(y,0.5)
19 plt.figure(figsize=(4,4))
20 plt.plot(x,y1,label='base line',lw=2.5,c='red')
21 plt.ylim(-1.1,1.1)
22 plt.title('Upper side')
23 plt.show()
24
25 # スカラーによる抽出（下側）
26 y1 = np.minimum(y,0.5)
27 plt.figure(figsize=(4,4))
28 plt.plot(x,y1,label='base line',lw=2.5,c='red')
29 plt.ylim(-1.1,1.1)
30 plt.title('Lower side')
31 plt.show()

```

このプログラムを実行すると、図 96 のようなグラフが順番に表示される。

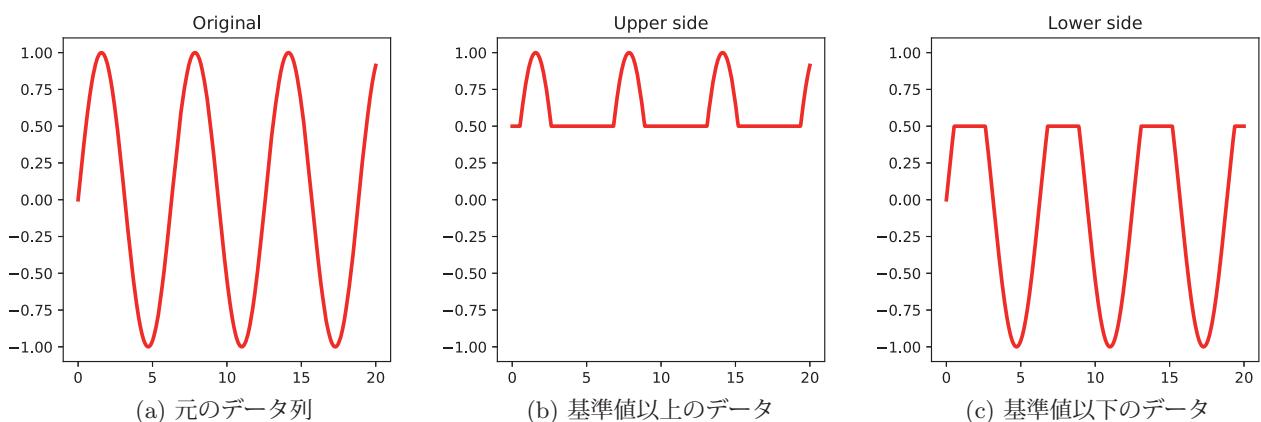


図 96: スカラー値を基準にして上側, 下側データを抽出

このように、スカラーの基準値を与えると単純な切り取りができる。

3.1.29.2 指定した範囲のデータの抽出

`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制限することができる。

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 $-0.5 \leq y \leq 0.5$ の範囲に制限してプロットするものである。

プログラム：`npclip01.py`

```

1 # coding: utf-8
2 # ライブドリの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 # データの生成
7 x = np.linspace(0, 30, 200)
8 y = np.sin(x)
9
10 # プロット：clipなし
11 plt.figure(figsize=(6,4))
12 plt.plot(x, y)
13 plt.xlabel('x'); plt.ylabel('y')
14 plt.ylim(-1.1, 1.1)
15 plt.title('y=sin(x)')
16 plt.show()
17 print('min:', y.min(), ' max:', y.max())
18
19 # プロット：clipあり
20 yc = np.clip(y, -0.5, 0.5)
21 plt.figure(figsize=(6,4))
22 plt.plot(x, yc)
23 plt.xlabel('x'); plt.ylabel('yc')
24 plt.ylim(-1.1, 1.1)
25 plt.title('clipped: yc=np.clip(y,-0.5,0.5)')
26 plt.show()
27 print('min:', yc.min(), ' max:', yc.max())

```

このプログラムを実行すると、図 97 のようなグラフが順番に表示される。

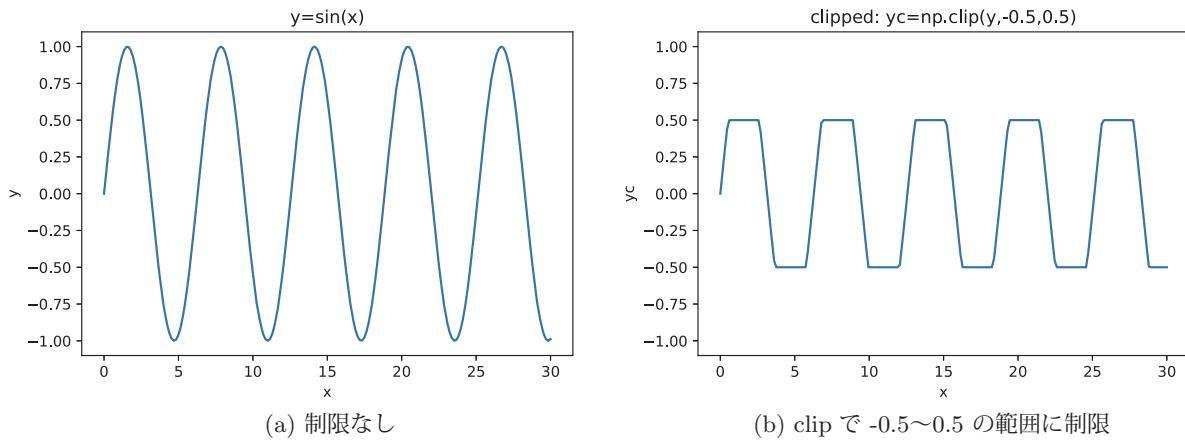


図 97: `clip` 関数による値の制限

y の値を $-0.5 \leq y \leq 0.5$ の範囲に制限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 # coding: utf-8
```

```

2 # ライブライアリの読み込み
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 # データの作成
7 x = np.linspace(0, 20, 1000)
8 y1 = np.sin(x) / 4.0 + 0.75      # 上限
9 y2 = np.sin(x) / 4.0 - 0.75      # 下限
10 r = np.random.rand(1000)*2 - 1   # データ列
11
12 # データ, 下限, 上限のプロット
13 plt.figure(figsize=(6,4))
14 plt.plot(x,r,label='data',ls='None',marker='.')
15 plt.plot(x,y1,label='upper bound',lw=2)
16 plt.plot(x,y2,label='lower bound',lw=2)
17 plt.ylim(-1.1,1.1)
18 plt.legend()
19 plt.title('Data and bounds')
20 plt.show()
21
22 # プロット: clipによる制限
23 rc = np.clip(r, y2, y1)
24 plt.figure(figsize=(6,4))
25 plt.plot(x, rc, ls='None', marker='.')
26 plt.ylim(-1.1,1.1)
27 plt.title('clipped data')
28 plt.show()

```

このプログラムを実行すると、図 98 のようなグラフが順番に表示される。

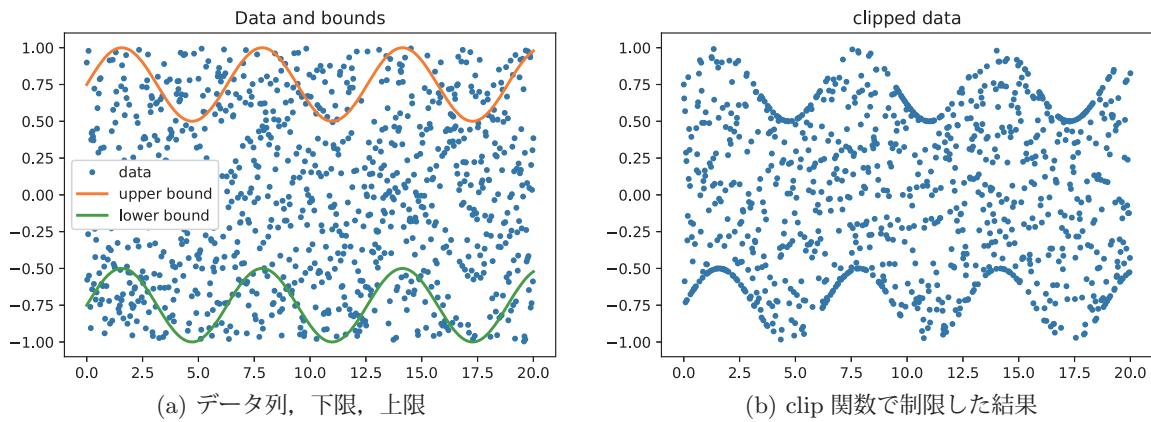


図 98: 下限, 上限の列を clip 関数に与える例

指定した下限, 上限の範囲にデータが制限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 27）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 27: 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
fftpack	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 $t = 0 \sim 1$ に対する周波数 $f = 1 \text{ Hz}$ の矩形波 $\text{square}(2\pi ft)$ 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み込み
>>> import numpy as np [Enter] ← NumPy の読み込み
>>> from scipy import signal [Enter] ← scipy.signal の読み込み
>>> 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 99 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。

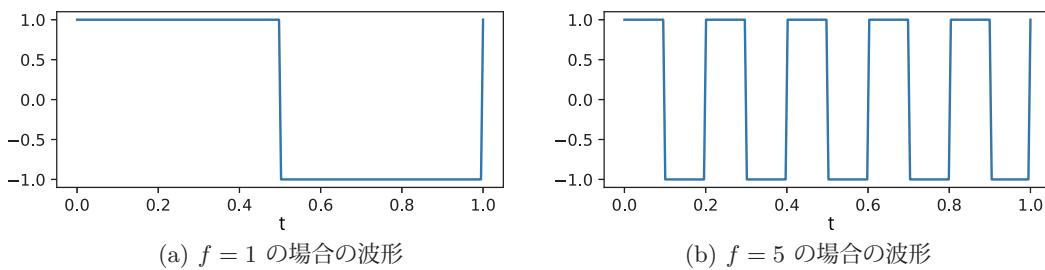


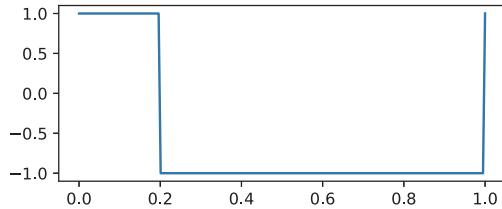
図 99: 矩形波の生成

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。

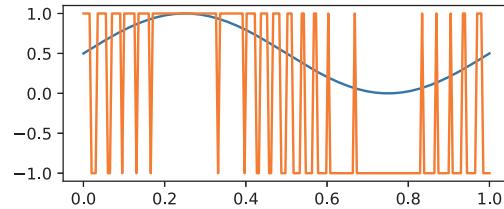
例. $duty=0.2$ として実行した例 (先の例の続き)

```
>>> 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 100 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 100: 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=’ を与える。

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  $(\sin(2\pi t) + 1)/2$  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 100 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。

■ 鋸歎状波 (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」に対する鋸歎状波の波形 (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 である。

次の例は、時間軸 $t = 0 \sim 1$ に対する周波数 $f = 1$ Hz の鋸歎状波 $\text{sawtooth}(2\pi ft)$ を生成して波形をプロットするものである。

例. 周期 1 つ分の鋸歎状波 (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] ←鋸歎状波の生成
>>> 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 101 の (a) が表示される。

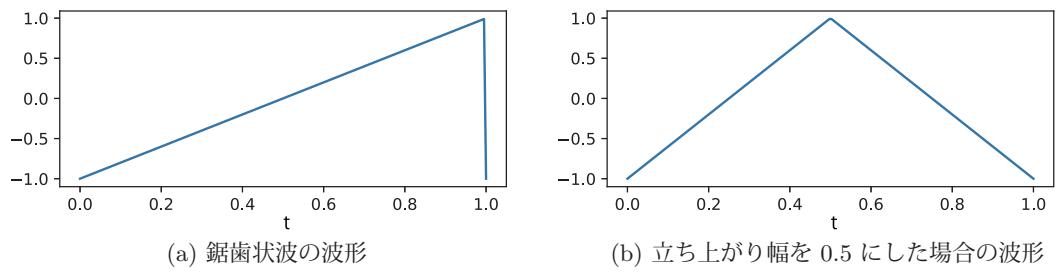


図 101: 鋸歎状波の生成

先の例において、鋸歎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 101 の (b) が表示される。

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] ←鋸歎状波の生成
>>> 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 102 が表示される。

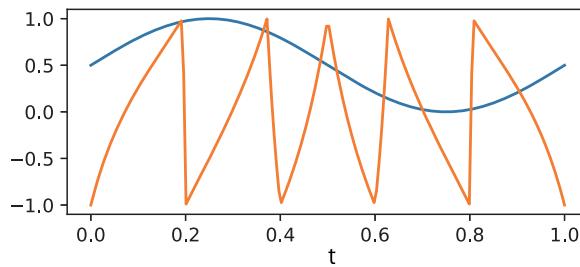


図 102: 鋸歎状波の立ち上がり幅の変調

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歎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**⁴² 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⁴³ 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」⁴⁴ 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**⁴⁵ 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 ) [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`⁴⁶ 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)、音質を求める場合は 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。

⁴²Microsoft 社と IBM 社によって開発されたデータフォーマット

⁴³これに関しては拙書「Python3 入門」で解説しています。

⁴⁴1 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。

⁴⁵音声波形の 1 つの値を表現するビット長

⁴⁶「3.1.2.4 扱える値の範囲」(p.54) 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 ファイルの読み込み
>>> 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 の読み込み
>>> (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 103 のような波形のプロットが表示される。

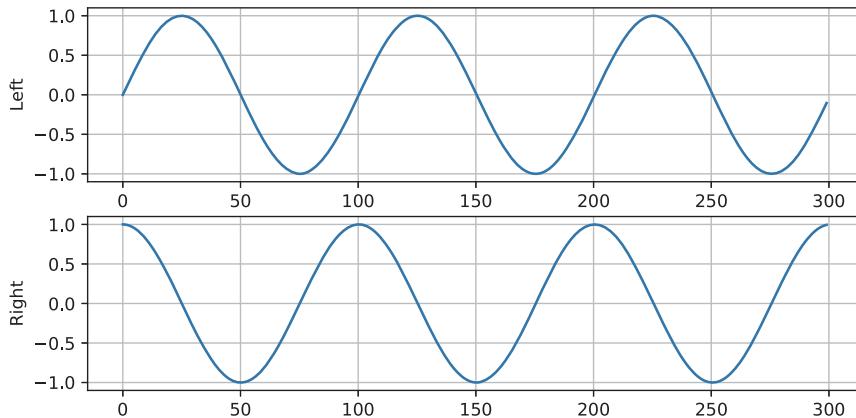


図 103: ステレオサウンドの波形のプロット

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 Audacity⁴⁷ で開いた様子を図 104 に示す。

⁴⁷公式インターネットサイト : <https://www.audacityteam.org/>



図 104: 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。このことを図 105 に示す。

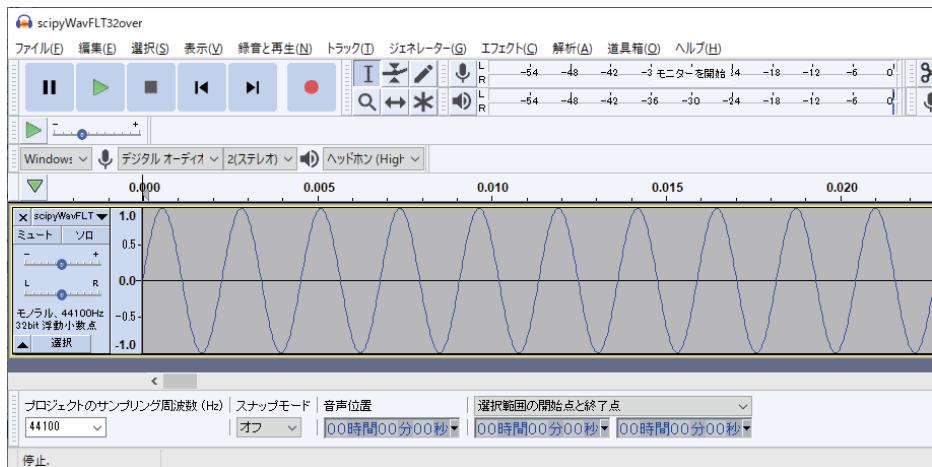


図 105: 振幅を半分に縮小 (-6db 増幅) したところ：元の波形が再現されている

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⁴⁸ 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。

⁴⁸IEEE 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⁴⁹ 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⁵⁰ 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 モジュールの読み込みに関する注意

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。また幕乗は '**' 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 の読み込み
>>> 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。

⁴⁹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*

⁵⁰ウルフラン・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き、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 関数の場合に準じる。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⁵¹ 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
```

⁵¹ 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 $f(x,y,\dots)$ の構造（頭部と引数列の取り出し）

$f(x,y,\dots)$ 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 28 に示す.

表 28: 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⁵²

```
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

$$\int f(x) dx$$

のように文字の 2 次元配置⁵³ 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⁵⁴ 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
```

⁵²後の「3.3.4.5 integrate の遅延実行」(p.170) で解説する。

⁵³キャラクターアート

⁵⁴Jupyter 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] ←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] ←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] ←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
>>> s = sp.Symbol('x')
>>> 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 29 に示す。

表 29: 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	数値かどうかを検査する。 (←こちらが推奨される)	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_Number		is_Symbol	
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、

$$\lim_{x \rightarrow 1} \frac{1}{x - 1}$$

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式として記述可能⁵⁵ 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
```

⁵⁵ 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. $\frac{d}{dx}f(x) - \frac{1}{\sin(x)} = 0$ 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, $f(x) = C_1 + \frac{1}{2} \log(\cos(x) - 1) - \frac{1}{2} \log(\cos(x) + 1)$ 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 $u(x, t) = F(x - ct) \cdot \exp\left(-\frac{a(cx + t)}{c^2 + 1}\right)$ 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
[ a  b ]
[ c  d ] ←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
[ e  f ]
[ g  h ] ←表示結果
>>> sp.pprint( m1 + m2 ) [Enter] ←行列同士の和
[ a+e  b+f ]
[ c+g  d+h ] ←和
>>> sp.pprint( m1 - m2 ) [Enter] ←行列の差
[ a-e  b-f ]
[ c-g  d-h ] ←差
>>> sp.pprint( m1 * m2 ) [Enter] ←行列の積
[ ae+bg  af+bh ]
[ ce+dg  cf+dh ] ←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
[ a  b  e  f ]
[ c  d  g  h ] ←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
[ a  b ]
[ c  d ]
[ e  f ]
[ g  h ] ←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⁵⁶ 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  
[ c d ] ← 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  
[ a ]  
[ c ]  
[ e ]  
[ g ] ← 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  
[ a b ]  
[ 2 3 ] ← 指定した位置に要素が与えられている  
[ e f ]  
[ g h ]
```

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。

⁵⁶NumPy 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  

$$\begin{bmatrix} \frac{d}{ad-bc} & -\frac{b}{ad-bc} \\ -\frac{c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}$$
 ←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  $v_1 = (a, b, c)$  の作成  
>>> v2 = sp.Matrix([d,e,f]) [Enter] ←ベクトル  $v_2 = (d, e, f)$  の作成  
>>> sp pprint(v1) [Enter] ←ベクトル  $v_1$  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  $v_1 \cdot v_2$  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57 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
[ (2, 1, [ [-1], [ 1 ] ] ), (5, 1, [ [ 1/2 ], [ 1 ] ] ) ] ←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, (变数, 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, \dots, 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  $a_1 + (k - 1)d$  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. モジュールの読み込みと対象となる式の用意

```
>>> import sympy as sp [Enter] ←SymPy モジュールの読み込み
>>> 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。

⁵⁷algebraic multiplicity

手順 2. マッチングの実行

```
>>> r = f.match( P1*x**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。Wild 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

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  
3.141592653589793238462643383279502884197169399375105820974944592307816
```

例. Matrix オブジェクトの数値近似

```
>>> s = sp.sympify('sqrt(2)'); t = sp.sympify('sqrt(3)') [Enter] ←  $s = \sqrt{2}, t = \sqrt{3}$   
>>> u = sp.sympify('sqrt(5)'); v = sp.sympify('sqrt(7)') [Enter] ←  $u = \sqrt{5}, v = \sqrt{7}$   
>>> 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 式を数値化する際の工夫

SymPy の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」には対象の式を与える。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と返す。「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 30）のものを使用する。

表 30: 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 の式 'sin(x)' を NumPy 用の関数に変換する

```
>>> import sympy as sp [Enter] ← SymPy の読み込み
>>> f = sp.parse_expr('sin(x)') [Enter] ← SymPy の式 '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 の読み込み
>>> 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 $-4\pi \sim 4\pi$ 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 の読み込み
>>> 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 106 のようなグラフが表示される。

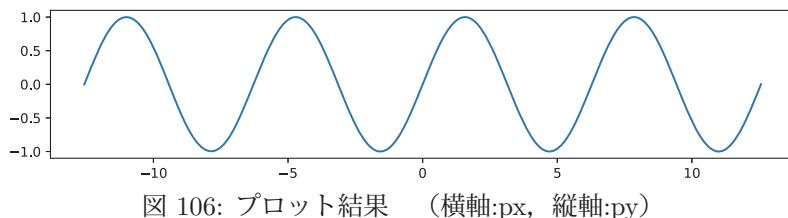


図 106: プロット結果（横軸:px, 縦軸:py）

上に示した事例は単純な式 '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
```

```

5 # 2変数を取る関数 (3D描画用)
6 def w3df(x,y):
7     r = sp.sqrt(x**2 + y**2)
8     z = sp.exp(-r/5) * sp.cos(2*r)
9     return z
10
11 sp.var('x y')    # 上記関数のための引数のSymbol
12 f = w3df(x,y)   # SymPy式として取得
13 print('SymPyの式',f)
14 print('をNumPy用関数に変換して一括計算し, matplotlibで可視化します。')
15
16 # SymPyの式をNumPy用の関数に変換
17 w3df_np = sp.lambdify( (x, y), f, modules='numpy' )
18
19 # NumPyで一括計算
20 px0 = np.linspace(-8,8,120)      # x軸データの配列
21 py0 = np.linspace(-8,8,120)      # y軸データの配列
22 px, py = np.meshgrid(px0, py0)   # 3Dプロット用の平面に変換
23 pz = w3df_np(px,py)             # z軸データを一括計算
24
25 # matplotlibで可視化
26 fig, ax = plt.subplots(subplot_kw={'projection':'3d'})
27 ax.plot_surface(
28     px, py, pz,                  # 座標データ
29     cmap='hot',                  # 面のカラーマップ
30     alpha=0.7,                   # 不透明度 (0=完全透明, 1=不透明)
31     edgecolor='black',           # エッジ線の色
32     linewidth=0.3                # エッジ線の太さ
33 )
34 ax.set_xlabel('px')
35 ax.set_ylabel('py')
36 ax.set_zlabel('pz')
37 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 107 のようなグラフが表示される。

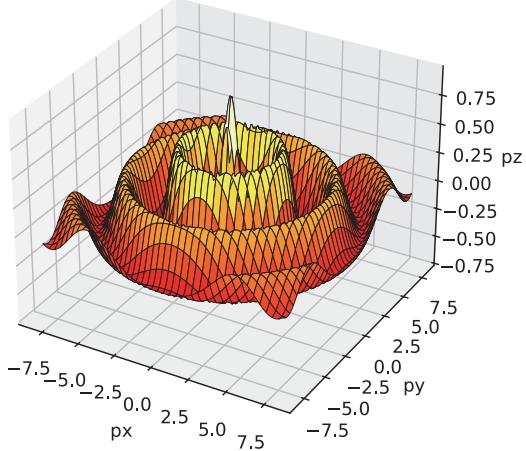


図 107: sympy3Dplt01.py の実行によって表示されるグラフ

3 次元のプロットに関しては「3.1.16 データの可視化：3 次元プロット」(p.108) 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 lambdify を使わずに SymPy の式 '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 の読み込み  
>>> f = sp.parse_expr('sqrt(x)') [Enter] ← SymPy の式 '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 の読み込み  
>>> 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 108 のようなグラフが表示される。

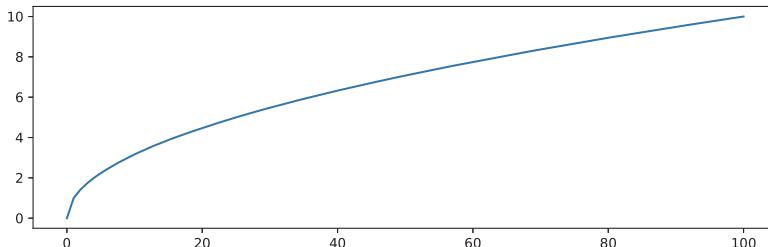


図 108: 表示されるグラフ

上に示した方法は、SymPy の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, L^AT_EX など）の表現に変換する方法が用意されている。

3.3.10.1 L^AT_EX

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 L^AT_EX の式に変換する例を次に示す。

例. L^AT_EX 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 L^AT_EX 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.180) 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 の読み込み
>>> 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 109 のようなグラフが表示される。

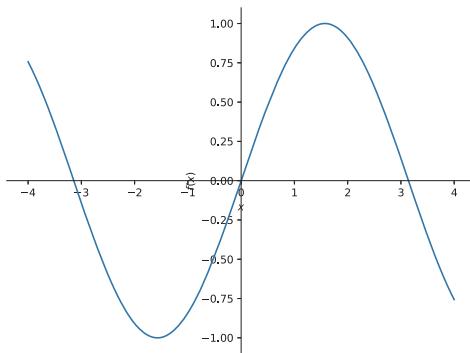


図 109: プロットの表示

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. $\cos(\sqrt{x^2 + y^2})$ のプロット

```
>>> import sympy as sp [Enter] ← SymPy の読み込み
>>> 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 110 のようなグラフが表示される。

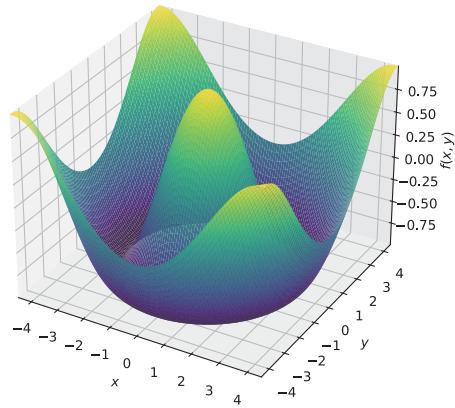


図 110: 3 次元のプロット

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)

<code>>>> g.save('sympyFig01.eps')</code>	<code>Enter</code>	← eps 形式で保存
<code>>>> g.save('sympyFig01.png')</code>	<code>Enter</code>	← 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⁵⁸ 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⁵⁹ を Python から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⁶⁰ によるインストールの例： pip install gmpy2

conda コマンド⁶¹ 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 クラスの読み込み  
>>> 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)

⁵⁸<https://pypi.org/project/gmpy2/>, <https://github.com/aleaxit/gmpy>

⁵⁹<https://gmplib.org/>, <https://www.mpfr.org/>

⁶⁰PSF 版 Python における標準的なライブラリ管理コマンド。

⁶¹Anaconda 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 クラスの読み込み  
>>> 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] ←  $\sqrt{2}$   
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] ←  $\sqrt{2}$  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] ←  $\sqrt{2}$   
mpfr('1.414213562373095048801688724209698078569671875' ←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。

$$\text{精度のビット長} = \lceil n \times \log_2 10 \rceil \quad (n \text{ は } 10 \text{ 進桁数})$$

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] ←  $\sqrt{2}$   
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 31）を演算環境の round 属性に設定することができる。

表 31: 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⁶² 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 = gmpy2mpz(2) [Enter] ←整数
>>> vq = gmpy2mpq(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 = gmpy2mpfr(3.14) [Enter] ←小数点数
>>> vq + vf [Enter] ←有理数+小数点数
mpfr('3.47333333333336') ←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)

```
>>> gmpy2const_pi() [Enter]
mpfr('3.1415926535897931')
>>> gmpy2const_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

```
>>> gmpy2cos( gmpy2const_pi() ) [Enter] ← cos( $\pi$ ) 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)

```
>>> gmpy2sqrt( gmpy2mpfr(2) ) [Enter] ←  $\sqrt{2}$  を求める
mpfr('1.4142135623730951')
>>> gmpy2sqrt( gmpy2mpfr(-2) ) [Enter] ←  $\sqrt{-2}$  を求める
mpfr('nan') ←引数に実数を与えると算出できない
```

⁶²PEP 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] ←  $\sqrt{2}$  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] ←  $\sqrt{-2}$  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⁶³。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方が 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)

$x \div y$ 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。

⁶³Python 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)

$x \div y$ 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)： $[x/y]$ による整数除算に基づく商と剰余

書き方： c_divmod(x, y)

$x \div y$ 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 p_1, p_2, p_3 の合成数 $x = p_1 * p_2, y = p_2 * p_3, z = p_1 * p_2 * p_2$ を作り, x と y の `gcd` が p_2 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 p_1, p_2, p_3 の生成

```
>>> p1 = gmpy2.next_prime( 2**60 ) [Enter] ← 素数  $p_1$  の生成
>>> p2 = gmpy2.next_prime( 2**61 ) [Enter] ← 素数  $p_2$  の生成
>>> p3 = gmpy2.next_prime( 2**62 ) [Enter] ← 素数  $p_3$  の生成
>>> print(p1,p2,p3,sep='\n') [Enter] ← 出力して確認
1152921504606847009 ←  $p_1$ 
2305843009213693967 ←  $p_2$ 
4611686018427388039 ←  $p_3$ 
```

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
10633823966279327363694553002502260713 ← y
12259964326927111656428205713442516863792538781422257417 ← 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⁶⁴ である。

■ 亂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。

⁶⁴疑似乱数という。

プログラム：numSpdTest01.py

```

1 import gmpy2, timeit, sys, math
2 from mpmath import mp
3
4 #--- 階乗計算の関数 ---
5 sys.setrecursionlimit(11000)      # 関数の再帰呼び出しの上限を再設定
6
7 def fct(n):
8     if n <= 0: return 1
9     return n * fct(n-1)
10
11 n = 10000
12 x = fct(n)                      # 事前実行
13 dgt = int(math.floor(math.log10(abs(x)))+1)    # 桁数調査
14 print(f'{n}! は {dgt} 桁 ({int(dgt*3.33)} bit) 速度比, ')
15 print('*'*44)
16
17 #--- intの場合の実行時間 ---
18 t1 = timeit.timeit('fct(n)', globals=globals(), number=10) / 10
19 print(f'  int による計算時間(sec): {t1:7.4f} {t1:5.2f} 倍')
20
21 #--- mpzの場合の実行時間 ---
22 t2 = timeit.timeit('fct(gmpy2.mpz(n))', globals=globals(), number=10) / 10
23 print(f'  mpz による計算時間(sec): {t2:7.4f} {t1/t2:5.2f} 倍')
24
25 #--- mpmathの場合の実行時間 ---
26 mp.dps = dgt                      # 演算精度設定
27 t3 = timeit.timeit('fct(mp.mpf(n))', globals=globals(), number=10) / 10
28 print(f'  mpmath による計算時間(sec): {t3:7.4f} {t1/t3:5.2f} 倍')
29
30 #--- mpfrの場合の実行時間 ---
31 gmpy2.get_context().precision = int(dgt*3.33)  # 演算精度設定
32 t4 = timeit.timeit('fct(gmpy2.mpfr(n))', globals=globals(), number=3) / 3
33 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

```

1 import gmpy2, timeit
2 from fractions import Fraction
3
4 #--- i/(i+1) の総和 ---
5 def sumQ(f,n):      # f の型で n 回加算
6     s = 0
7     for i in range(1,n+1):
8         s += f(i,i+1)

```

```

9     return s
10
11 n = 10000
12 print('sum_{i=1}^{n}{i/(i+1)} の計算: n = , n, , 速度比')
13 print('-*51)
14 #--- Fractionで計算 ---
15 t1 = timeit.timeit('sumQ(Fraction,n)',globals=globals(),number=10) / 10
16 print( f' Fractionによる計算時間: {t1:8.5f} (sec), {t1/t2:5.2f} 倍, ')
17
18 #--- mpqで計算 ---
19 t2 = timeit.timeit('sumQ(gmpy2.mpq,n)',globals=globals(),number=10) / 10
20 print( f'gmpy2.mpqによる計算時間: {t2:8.5f} (sec), {t1/t2:5.2f} 倍, ')
21
22 print('-*51)
23 #--- 計算結果 ---
24 s1 = str(gmpy2.mpfr(sumQ(Fraction,n)))
25 s2 = str(gmpy2.mpfr(sumQ(gmpy2.mpq,n)))
26 print( f' Fractionによる計算結果の近似値: {s1} ')
27 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イ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

```

1 import gmpy2, timeit
2 from mpmath import mp
3
4 #--- mpmathの場合の実行時間 ---
5 mp.dps = 20000                                     # 演算精度設定
6 t1 = timeit.timeit('mp.exp(mp.exp(1))',globals=globals(),number=10) / 10
7 print(f'mpmath による e**e の計算時間(sec):{t1:7.4f}, 速度比:{1.0:5.2f}')
8
9 #--- mpfrの場合の実行時間 ---
10 gmpy2.get_context().precision = int(20000*3.33) # 演算精度設定
11 t2 = timeit.timeit('gmpy2.exp(gmpy2.exp(1))',globals=globals(),number=10) / 10
12 print(f' mpfr による e**e の計算時間(sec):{t2:7.4f}, 速度比:{t1/t2:5.2f}')
13
14 #--- 計算結果 ---
15 print('-*78)
16 print(' mpmathによる計算結果の近似値 :',str(mp.exp(mp.exp(1)))[::47])
17 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イ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 : 15.15426224147926418976043027262991190552854853
mpfr による計算結果の近似値 : 15.15426224147926418976043027262991190552854853

```

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の方が遙かに遅い事例である。

プログラム：numSpdTest04.py

```
1 import timeit
2 import gmpy2
3 from mpmath import mp
4
5 digits = 1000          # 精度（桁数）の設定
6 mp.dps = digits
7 gmpy2.get_context().precision = int(digits * 3.33)
8
9 # --- mpmathによる ζ(3) ---
10 t1 = timeit.timeit('mp.zeta(3)', globals=globals(), number=10) / 10
11 print(f'mpmath による ζ(3) 計算時間 (sec): {t1:.5f}, 速度比:{1.0:8.4f}')
12
13 # --- mpfrによる ζ(3) ---
14 t2 = timeit.timeit('gmpy2.zeta(3)', globals=globals(), number=10) / 10
15 print(f' mpfr による ζ(3) 計算時間 (sec): {t2:.5f}, 速度比:{t1/t2:8.4f}')
16
17 # 計算結果の出力
18 print('---計算結果（近似値）' + '-' * 39)
19 print('mpmath:', str(mp.zeta(3))[:52])
20 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 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⁶⁵を生成する。

プログラム：numSpdTest06.py

```
1 import gmpy2, timeit
2
3 #--- 素数リストの生成 ---
4 primeList = []      # 素数を蓄積するリスト
5
6 def genPrimes(s,n):
7     global primeList
8     p = s
9     for _ in range(n):
10         p = gmpy2.next_prime(p)
11         primeList.append(p)
12
13 #--- 実行時間の測定 ---
14 s = 10**78          # 開始位置
15 n = 10000           # 生成個数
16 print(n,'個の素数を生成します…')
17 print('開始ポイント:\n',s,sep='')
18 t = timeit.timeit('genPrimes(s,n)',globals=globals(),number=1) / 1
19 print('生成時間 :',t,'(sec)')
20
21 #--- 素数をファイルに出力 ---
22 fname = 'primenumbers.txt'
23 print('ファイル :',fname,'に出力します。')
24 f = open(fname,'w',encoding='utf-8')
```

⁶⁵ 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。

```
25 |     for p in primeList:  
26 |         print(int(p),file=f)  
27 | 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

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

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⁶⁶ 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

```

1 import gmpy2, secrets, time
2 import matplotlib.pyplot as plt
3
4 lenR = 128 # 乱数のビット長
5 numR = 10000 # 生成する乱数の個数
6
7 #--- gmpy2による乱数生成 ---
8 t1 = time.time()
9 rs = gmpy2.random_state(1) # 亂数状態オブジェクト
10 R1 = []
11 for _ in range(numR):
12     R1.append(gmpy2mpz_urandomb(rs, lenR))
13 t2 = time.time()
14
15 #--- secretsによる乱数生成 ---
16 t3 = time.time()
17 R2 = []
18 for _ in range(numR):
19     R2.append(secrets.randbits(lenR))
20 t4 = time.time()
21
22 #--- 生成時間の比較 ---
23 print(f'{lenR}ビットの乱数を{numR}個生成: ')
24 print(f' gmpy2mpz_urandomb:{t2-t1:.9f}(sec) ')
25 print(f' secrets.randbits:{t4-t3:.9f}(sec) ')
26
27 #--- 乱数の散布図 ---
28 rat = 2**lenR # 正規化用係数
29 R1_n = [float(r)/rat for r in R1]
30 R2_n = [float(r)/rat for r in R2]
31 fig, ax = plt.subplots(1, 2, figsize=(14, 5))
32 ax[0].scatter(R1_n[::2], R1_n[1::2], alpha=0.4)
33 ax[0].set_title('gmpy2mpz_urandomb')
34 ax[1].scatter(R2_n[::2], R2_n[1::2], alpha=0.4)

```

$^{66}\text{Python}$ 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表示される。(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:
gmpy2mpz_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。

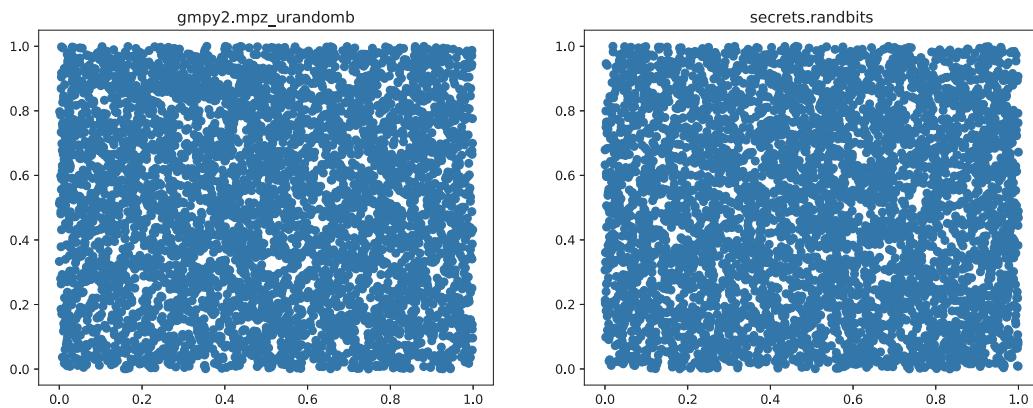


図 111: 生成した乱数を散布図にして眺めるところ

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は、MD5, SHA1, SHA224, SHA256, SHA384, SHA512 である。

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⁶⁷ 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（MD5 による）

```
>>> import hashlib [Enter] ←モジュールの読み込み
>>> m = hashlib.md5(b'MyPassword') [Enter] ←ダイジェスト生成用オブジェクトの生成（MD5）
>>> m.digest() [Enter] ←ダイジェスト生成
b'HP=¥ffdXr¥x0b¥xd5¥xff5¥xc1¥x02¥x06ZR¥xd7' ←得られたダイジェスト（バイト列）
>>> m.hexdigest() [Enter] ←16進数表現でダイジェスト生成
'48503dfd58720bd5ff35c102065a52d7' ←得られたダイジェスト（文字列）
```

このように、ハッシュ化のアルゴリズム（この例では md5）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⁶⁸ (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は passlib.hosts モジュールが提供する次のような関数群（表 32）で調べることができる。

⁶⁷ 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。

⁶⁸ 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 32: passlib で使用できる暗号化アルゴリズムを調べる関数

対象の OS	関数
Linux	passlib.hosts.linux_context.schemes()
FreeBSD	passlib.hosts.freebsd_context.schemes()
OpenBSD	passlib.hosts.openbsd_context.schemes()

表 32 の関数を実行した例を次に示す。

例. 暗号化アルゴリズムを調べる

```
>>> import passlib.hosts [Enter] ← モジュールの読み込み
>>> passlib.hosts.linux_context.schemes() [Enter] ← Linux 用
('sha512_crypt', 'sha256_crypt', 'md5_crypt', 'des_crypt', 'unix_disabled')
↑ 使用できるアルゴリズムのタプル
>>> passlib.hosts.freebsd_context.schemes() [Enter] ← FreeBSD 用
('bcrypt', 'md5_crypt', 'bsd_nthash', 'des_crypt', 'unix_disabled')
↑ 使用できるアルゴリズムのタプル
>>> passlib.hosts.openbsd_context.schemes() [Enter] ← OpenBSD 用
('bcrypt', 'md5_crypt', 'bsdi_crypt', 'des_crypt', 'unix_disabled')
↑ 使用できるアルゴリズムのタプル
```

次に、平文のパスワードとソルトからメッセージダイジェストを生成する例を示す。

例. MD5 アルゴリズム⁶⁹ によるメッセージダイジェストの生成

```
>>> import passlib.hash [Enter] ← モジュールの読み込み
>>> passlib.hash.md5_crypt.hash( 'MyPassword', salt='abc' ) [Enter] ← 暗号化を実行
'$1$abc$1KbExFu3EBz4Nynw0YLsN0' ← 生成されたメッセージダイジェスト
```

この例では、md5_crypt アルゴリズムを用いて、平文パスワード ‘MyPassword’ とソルト ‘abc’ からメッセージダイジェスト ‘\$1\$abc\$1KbExFu3EBz4Nynw0YLsN0’ を生成している。

⁶⁹MD5：多くの UNIX 系 OS のパスワード管理で採用されているアルゴリズム。

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⁷⁰ 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⁷¹ 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⁷² 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み込み
>>> 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。

⁷⁰例：Windows 環境では Visual Studio、Apple Macintosh の場合は Xcode。

⁷¹Cython の日本語ドキュメントサイト：<http://omake.accense.com/static/doc-ja/cython/>

⁷²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み込み
>>> 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み込み
>>> 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 LLVM⁷³ 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⁷⁴ 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()

```

⁷³C 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。

⁷⁴「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。

```

16     for i in range(n):
17         print( fib1(i), end=', ' )
18     else: print()
19     t = time.time() - t1
20     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み込み
>>> 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

```

1 from numba import njit, jit, i8
2 import time
3
4 # フィボナッチ数列の生成
5 @njit('i8(i8)')
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()
16     for i in range(n):
17         print( fib1(i), end=', ' )
18     else: print()
19     t = time.time() - t1
20     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

```
1 #include <stdio.h>
2
3 long fib0( long n )
4 {
5     if ( n == 0 ) {
6         return( 1 );
7     } else if ( n == 1 ) {
8         return( 1 );
9     } else {
10        return( fib0(n-1) + fib0(n-2) );
11    }
12 }
13
14 __declspec(dllexport) void fib( long n )
15 {
16     long c;
17
18     for ( c = 0; c < n; c++ ) {
19         printf("%ld,",fib0(c));
20     }
21     printf("\n");
22 }
```

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み込み
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 /***** 整数 (int) の受け渡し *****/
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 /***** 浮動小数点数 (double) の受け渡し *****/
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み込み
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 33 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 33: 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み込み
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)                 # 値域の配列をリストに変換
25 ay = list(ay2)
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 $0 \sim 2\pi$ 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 112 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 )
```

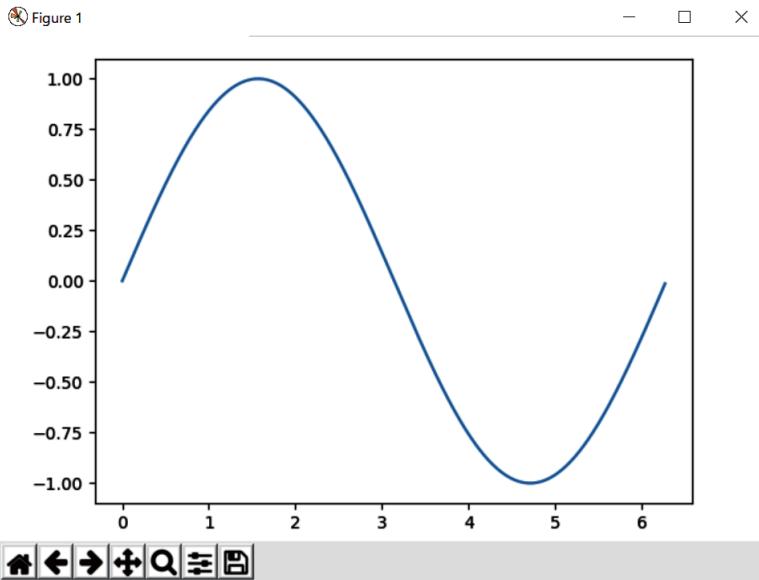


図 112: プロットの表示

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) この形式の読み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 113)



図 113: 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

```
1 import tkinter as tk
2
3 ##### ボタンクリックに対応する処理 #####
4 def on_click(ch):
5     if ch == 'C':           # クリアボタン
6         var.set('')
7     elif ch == '=':         # 「=」ボタンで計算実行
8         expr = var.get()
9         try:
10             # ディスプレイの内容を eval で評価する
11             result = eval(expr, {'__builtins__': None}, {})
12             var.set(str(result))
13         except Exception:
14             var.set('Error')
15     else:                  # 数値と計算記号の入力
16         var.set(var.get() + ch)
17
18 ##### GUI構築 #####
19 root = tk.Tk()
20 root.title('Tkinterの電卓')
21
22 # ディスプレイ部
23 var = tk.StringVar()
24 entry = tk.Entry(root, textvariable=var, justify='right',
25                   font=('TkDefaultFont', 16), bd=5, relief='sunken')
26 entry.grid(row=0, column=0, columnspan=4, sticky='nsew', padx=5, pady=5)
27 entry.focus()
28 # ボタン配列
29 layout = [ ['7', '8', '9', '/'],
30            ['4', '5', '6', '*'],
31            ['1', '2', '3', '-'],
32            ['0', '.', 'C', '+'] ]
33 # ボタンの登録
34 for r, row in enumerate(layout, start=1):
35     for c, ch in enumerate(row):
36         tk.Button(root, text=ch, command=lambda s=ch: on_click(s),
37                    font=('TkDefaultFont', 14)).grid(row=r, column=c,
38                    sticky='nsew', padx=2, pady=2)
39 # ボタンがクリックされた際の処理の登録
40 tk.Button(root, text='=', command=lambda: on_click('='),
41            font=('TkDefaultFont', 14)).grid(row=5, column=0, columnspan=4,
42            sticky='nsew', padx=2, pady=2)
43
44 # 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 114 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 114: 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 114 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 115）も表示される。

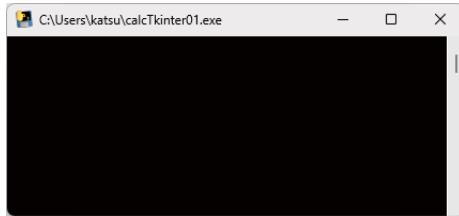


図 115: GUI とは別に表示されるコンソールウィンドウ

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⁷⁵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。

JupyterLab は、IPython⁷⁶を取り込んで構築された、Web ベースのインターフェースを持つ作業環境であり、プログラムの開発と実行に関するほぼ全ての作業（ファイルやディレクトリに関する操作も含む）を Web ブラウザ上で行うことができる。また、matplotlib や pandas をはじめとするパッケージの利用においては、作業を実行している表示領域の中に図や表を表示することができ、視認性を向上する効果も得られる。更に、一連の作業内容をノートブック（Jupyter Notebook）というデータファイルとして保存管理することができ、過去に行った処理の再現や、作業の継続が簡単な形で実現できる。

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⁷⁷から得られる。

6.1 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。

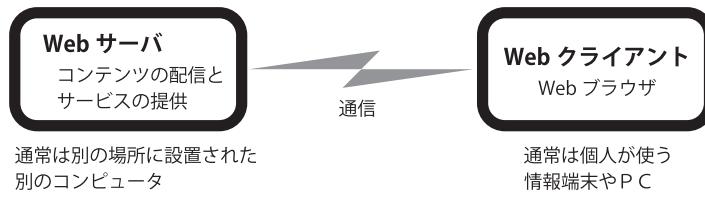


図 116: サーバとクライアントの連携 (通常の場合)

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 116）⁷⁸ 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 117）となる。

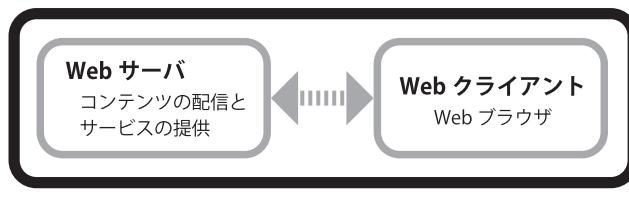


図 117: 同一のコンピュータ内でサーバとクライアントが連携するケース

⁷⁵Windows 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。

⁷⁶対話作業環境の中心的機能を提供するソフトウェア。[\(http://ipython.org/\)](http://ipython.org/)

⁷⁷<http://jupyter.org/>

⁷⁸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。

6.1.1 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 18:47:58.048 LabApp] JupyterLab beta preview extension loaded from c:\program files\python36\lib\site-pac...
[I 18:47:58.048 LabApp] JupyterLab application directory is c:\program files\python36\share\jupyter\lab
:
(途中省略)
:
[I 18:47:58.313 LabApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 18:47:58.315 LabApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
http://localhost:8888/?token=7420a0d25ccefb7cdf9ae1406e8a63bdefc0372237157220
[I 18:48:00.405 LabApp] Accepting one-time-token-authenticated connection from ::1
[I 18:48:03.440 LabApp] Build is up to date
```

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。上に挙げた起動メッセージの例の中に

<http://localhost:8888/?token=7420a0d25ccefb7cdf9ae1406e8a63bdefc0372237157220>

という部分があるが、それがこの例におけるサーバプロセスの URL⁷⁹ 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）

6.1.2 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 118 に示す。

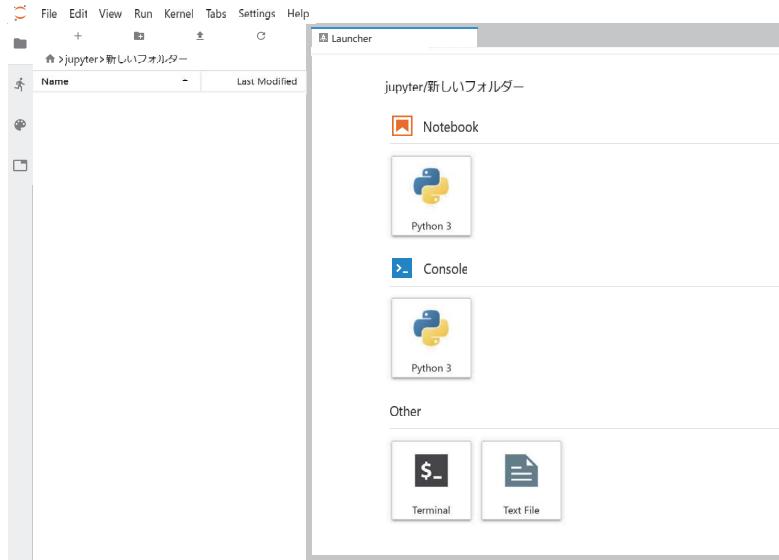


図 118: Web ブラウザで表示した JupyterLab のウィンドウ

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 119）。

⁷⁹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。



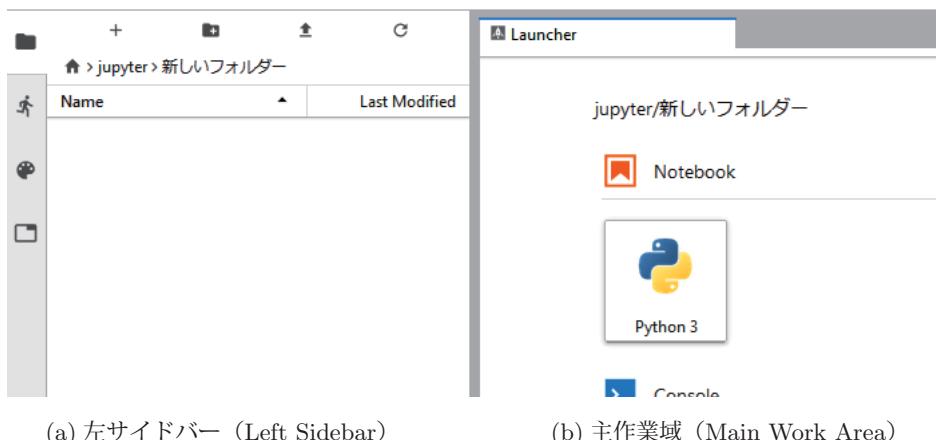
図 119: メニューバー (Menu Bar)

● 左サイドバー (Left Sidebar)

各種の管理機能の領域。(図 120 (a))

● 主作業域 (Main Work Area)

主たる作業領域。(図 120 (b))



(a) 左サイドバー (Left Sidebar)

(b) 主作業域 (Main Work Area)

図 120: 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 を開始した時点では、この部分には**ノートブック** (Notebook), **コンソール** (Console), **Terminal**, **Text Editor** の 4 つのものが表示されている。(後述:「ランチャー」タブ) 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⁸⁰ 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⁸¹ することができる。

6.1.2.1 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 121 のようなノートブックが表示される。

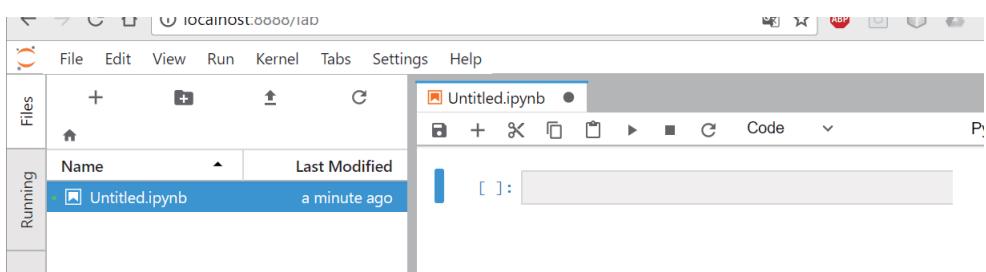


図 121: ノートブックの開始

⁸⁰macOS 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。

⁸¹コード体系や Tab の扱いに関する設定に注意すること。

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、この右側に Python の文や式を入力することができる。

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。

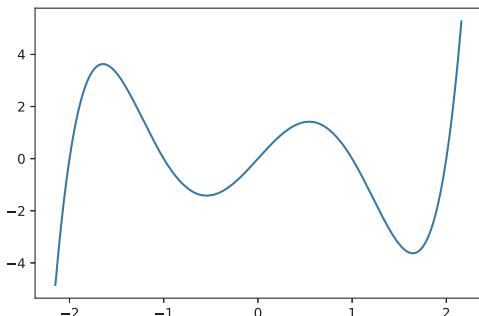
例。matplotlib のグラフのインライン表示

[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み込みを行っている。'[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。

6.1.2.2 カーネル (Kernel) について

ノートブックやコンソールは Python インタプリタを介して文の実行や式の評価を行っている。この際、JupyterLab は Python インタプリタを **カーネル** (Kernel) として起動している。すなわち、ノートブックやコンソールはあくまでインターフェースとして利用者との対話を保持するものであり、実際の実行や評価は JupyterLab の背後で起動された Python インタプリタで行っている。この場合の Python インタプリタが JupyterLab から見た場合のカーネルである。

カーネルは、開いているそれぞれのノートブックやコンソールに対して 1 つずつ起動され、それぞれ別々の実行状態を保持している。すなわち、変数への値の割当てといった 1 連の処理の脈絡は、開いているノートブックやコンソールで固有のものとなる。従って、1 つのノートブックにおける変数への値の割当てや、生成されたオブジェクトは、別のノートブック上では無効である。

ノートブックやコンソールの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「Run」メニューの中にある「Restart Kernel And Run All Cells...」を選択する。

6.1.3 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 関数を使用すると Notebook 独自の GUI の形式による入力となる。

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、入力セルの直下に 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 GUI 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）

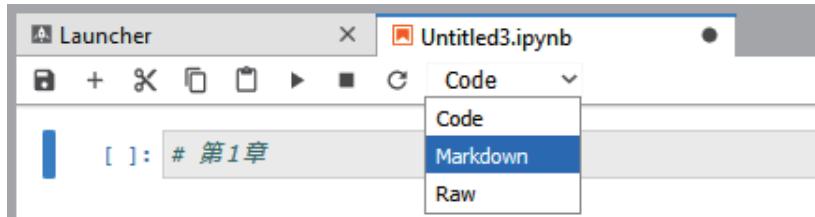
例. 先の例の続き

```
[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 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」とすることができます。

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)

A screenshot of a Jupyter Notebook interface. The title bar says 'Launcher' and 'Untitled3.ipynb'. A cell toolbar is visible above a cell containing '# 第1章'. The output of the cell is '第1章', displayed in large, bold black font.

Markdown のセル内では、先頭の「#」の個数によって文書構造（章・節・項…）を表現できる。以下に例を示す。

例. Markdown セル内で「#」で記述を開始した例（続き）

第1章

```
[1]: # 整形表示関数  
print('Jupyter NotebookのMarkdownです. ')  
print('# 1つで章を表現しています. ')
```

Jupyter NotebookのMarkdownです。
1つで章を表現しています。

例. Markdown セル内で「##」で記述を開始した例（続き）

第1章第1節

```
[2]: # 文書構造  
print('# 2つで節を表現しています. ')  
  
# 2つで節を表現しています。
```

例. Markdown セル内で「###」で記述を開始した例（続き）

第1章第1節第1項

```
[3]: # 文書構造  
print('# 3つで項を表現しています. ')  
  
# 3つで項を表現しています。
```

第1章第1節第2項

```
[4]: # 文書構造  
print('これも項です. ')  
  
これも項です。
```

6.3 使用例

6.3.1 MathJax による SymPy の式の整形表示

数式処理パッケージ SymPy 「3.3 SymPy」は MathJax⁸² 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 `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

```
sympy.init_printing(use_latex='mathjax')
```

とする。SymPy の式を整形表示する例を示す。

例. モジュールの読み込みと、数式整形表示の設定

```
[1]: import sympy      # SymPyモジュール  
# 表示にLaTeX整形を適用する設定  
sympy.init_printing(use_latex='mathjax')
```

例. 不定積分（続き）

```
[2]: # 不定積分の式  
sympy.sympify( 'Integral(f(x),x)' )
```

$$[2]: \int f(x) dx$$

例. 行列（続き）

```
[3]: # 行列  
m = sympy.sympify( 'Matrix([[a,b],[c,d]])' )  
m
```

$$[3]: \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

```
[4]: m.inv() # 逆行列
```

$$[4]: \begin{bmatrix} \frac{d}{ad-bc} & -\frac{b}{ad-bc} \\ -\frac{c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}$$

例. 総和（続き）

```
[5]: # 総和の式  
sympy.sympify('Sum(f(x),(x,0,n))')
```

$$[5]: \sum_{x=0}^n f(x)$$

例. 極限（続き）

```
[6]: # 極限の式  
sympy.sympify( 'Limit(1/f(x),x,oo)' )
```

$$[6]: \lim_{x \rightarrow \infty} \frac{1}{f(x)}$$

例. 導関数（続き）

```
[7]: # 導関数  
sympy.sympify( '(f(x)*g(x)).diff(x)' )
```

$$[7]: f(x) \frac{d}{dx} g(x) + g(x) \frac{d}{dx} f(x)$$

⁸²Web ブラウザ上で数式を整形表示するための JavaScript ライブライ。公式サイトは <https://www.mathjax.org/>

6.3.2 IPython.display モジュールによるサウンドの再生

IPython は JupyterLab の対話用シェルの機能を提供するものであり、IPython.display モジュールは対話シェルに関するメソッドや関数を提供する。ここでは特に

- WAV 形式サウンドファイルの読み込みと再生
- 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。

■ モジュールの読み込み

NumPy モジュールと IPython.display モジュールを読み込む例を次に示す。

例.

```
[1]: import numpy as np          # NumPyモジュール
      import IPython.display as IPyDisp    # IPython関連モジュール
```

IPython.display モジュールを読み込んで、それを別名 ‘IPyDisp’ として扱う形にしている。これにより当該モジュール配下の関数やメソッドを呼び出す際の記述を簡略化できる。

次に、WAV 形式サウンドのファイルを読み込んで再生する例を示す。まず WAV 形式ファイル ‘aaa.wav’ を読み込む例が次のものである。

例. WAV 形式ファイル ‘aaa.wav’ の読み込み（続き）

```
[2]: # Audioオブジェクトの生成(1) : WAV形式サウンドファイルから読み込み
      aud1 = IPyDisp.Audio("aaa.wav")  # Audioオブジェクト
```

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 aud1 にしている。このクラスのタイプ名の確認と再生作業の例を次に示す。

例. Audio オブジェクトの型確認と再生（続き）

```
[3]: print('データタイプ:', type(aud1))
      aud1      # 再生準備
```

データタイプ: <class 'IPython.lib.display.Audio'>

```
[3]: ▶ 0:00 / 0:00 ● 🔊 ⟲ ⟳
```

Audio オブジェクトの値を評価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（続き）

```
[4]: # Audioオブジェクトの生成(2) : 波形データの生成
      r = 44100          # サンプリング周波数: 44.1KHz
      t = np.arange(0,3,1/r)   # 時間軸: 0~3 (秒) まで 1/r 刻み
      f = 440            # 音の周波数: 440Hz
      s = np.sin( 2*np.pi*f*t )  # 波形データ: sin(2πft)
```

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⁸³。また、量子化ビット数も Audio オブジェクト生成時に自動的に設定（16 ビット=2 バイト）される。

⁸³モノラルの場合は単純な一次元配列、ステレオの場合は

[[右 1, 右 2, …, 右 n],
 [左 1, 左 2, …, 左 n]]
のような形式の配列として扱う。

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（続き）

```
[5]: aud2 = IPyDisp.Audio( s, rate=r )    # Audioプロジェクト生成  
aud2    # 再生準備
```

この例でも ▶ で再生できる。

7 psutil ライブライ

psutil は Python の標準ライブラリではなく、サードパーティ⁸⁴ 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
```

7.1 CPU 関連の情報の取得

7.1.1 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⁸⁵ 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
```

7.1.2 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。

7.1.3 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 34) 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`)

⁸⁴<https://github.com/giampaolo/psutil>

⁸⁵ハイパスレッディング (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 34: 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。

7.1.4 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
```

7.2 メモリ関連の情報の取得

7.2.1 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 35 に示す。

表 35 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。

7.2.2 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⁸⁶ の使用状況を sswap オブジェクトとして返す。

⁸⁶ 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 35: 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 36 に示す。

表 36: 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	スワッピンの量（単位：バイト）. ディスクからメモリへの移動量
sout	スワッパウトの量（単位：バイト）. メモリからディスクへの移動量

7.3 ディスク関連の情報の取得

7.3.1 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 37 に示す。

表 37: 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。

7.3.2 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。

7.3.3 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 38 に示す。

表 38: 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。

7.4 ネットワーク関連の情報の取得

7.4.1 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 39 に示す。

表 39: 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)}
```

7.4.2 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⁸⁷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 40 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 42 に挙げる。

⁸⁷これに関しては拙書「Python3 入門」でも解説しています。

表 40: 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 41 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 41: 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 42: 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')

7.5 プロセス関連の情報の取得

7.5.1 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
```

7.5.2 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。

7.5.2.1 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 43 に示す。

表 43: 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
13    return (minf.rss, minf.vms)
14
15 # システムの
16
17 if __name__ == '__main__':
18     m = pstat()
19     print('Python処理系のメモリ：')
20     print('  物理メモリ(Bytes):', m[0])
21     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

7.5.2.2 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.226 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..

7.5.2.3 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 pcpus times 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。

7.5.2.4 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 44 に示すような属性を持つ。

表 44: 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。
```

7.5.2.5 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⁸⁸に関するものである。

⁸⁸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.229) で解説した sconn オブジェクトに準ずる。

7.5.2.6 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> [Enter] ←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。

7.5.3 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	← 状態出力：1回目
memory_info:vms	4190208	2392064	425984	
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	← 状態出力：2回目
memory_info:vms	12255232	11874304	12128256	
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	← 状態出力：3回目
memory_info:vms	12378112	11874304	12169216	
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	← 状態出力：4回目
memory_info:vms	12435456	11935744	12242944	
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

8 その他

8.1 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⁸⁹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。

8.1.1 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。

8.1.2 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。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。(次の例参照)

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
```

8.1.2.1 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)

例。JSON のファイルへの出力 (先の例の続き)

```
>>> f = open( 'dic01.js', 'w' ) [Enter] ←ファイル 'dic01.js' 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.js' に出力される。

⁸⁹RFC 8259

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.js', 'r' ) [Enter] ←ファイル 'dic01.js' 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 [Enter] ←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
```

8.1.2.2 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
```

8.2 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。

8.2.1 他バイト文字の扱い（‘%’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` モジュールを用いる。このモジュールを読み込むには

```
import urllib.parse
```

とする。

例. 他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] ←モジュールの読み込み  
>>> 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. 文字コード列を他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
```

8.2.1.1 文字コード体系の指定

`quote`, `unquote` 共に、文字コード体系を明に指定する場合はキーワード引数 ‘encoding=文字コード体系’ を与える。

例. 文字コード体系（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
```

8.3 zenhan：全角 ⇄ 半角変換

zenhan⁹⁰ は英数字や記号文字などの全角、半角を変換するためのモジュール⁹¹ である。このモジュールは使用に際して次のようにして読み込む。

```
import zenhan
```

半角文字→全角文字の変換には h2z 関数を、全角文字→半角文字の変換には z2h 関数を使用する。

例. 全角、半角の変換

```
>>> import zenhan [Enter] ←モジュールの読み込み  
>>> h = 'abcdABCD0123#&%' [Enter] ←半角文字を  
>>> z = zenhan.h2z( h ) [Enter] ←全角文字に変換  
>>> z [Enter] ←結果の確認  
' a b c d A B C D 0 1 2 3 # & %,' ←変換結果（全角）  
>>> zenhan.z2h( z ) [Enter] ←全角文字を半角文字に変換  
'abcdABCD0123#&%' ←変換結果（半角）
```

全角 ⇄ 半角の変換機能は、次に紹介する jaconv モジュールも提供している。

8.4 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/>

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  
'コノ文ハヒラガナヲ含ンデイマス。' ←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。

⁹⁰<https://pypi.org/project/zenhan/>

⁹¹全角、半角の変換のためのモジュールとしては、zenhan 以外にも mojimoji (<https://pypi.org/project/mojimoji/>) というモジュールが存在する。

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 45 のようなキーワード引数を与える。

表 45: 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	<code>kana=True/False</code>	<code>digit=True/False</code>	<code>ascii=True/False</code>

太字はデフォルト

例. 全角→半角

```
>>> jaconv.z2h(' a b c d 1 2 3 4 # $ % & アイウエ') [Enter] ←全角を半角に変換  
' a b c d 1 2 3 4 # $ % & イウエ' ←カタカナのみが変換されている  
>>> jaconv.z2h(' a b c d 1 2 3 4 # $ % & 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

/etc/shadow, 201
&, 69
~, 69
@, 124
‘%’エンコーディング, 238
@ jit, 205
@ njit, 205
32-bit floating-point の WAV 形式サウンドデータ, 160
3 次元の散布図, 113
3 次元の棒グラフ, 112

absolute, 123
adaptiveThreshold, 11
add_collection3d, 148
add_patch, 145
add_subplot, 112
all, 135
allow_complex, 191
angle, 124
any, 135
apart, 167
append, 62
arange, 55
Arc, 146
arc, 26
argmax, 70, 71
argmin, 70
args, 165
argsort, 73
array, 52
arrowedLine, 13
asarray, 141
astype, 61
atoms, 164
Audio, 223
AudioSegment, 49
Axes3D, 110
Axes3DSubplot, 112
AxesSubplot, 85

bar, 76, 103, 121
bar3d, 112
barh, 103
BGR, 5
bincount, 75, 95
blit, 30, 31

BMP, 20
bool, 51
boxplot, 107

c_, 64
c_divmod, 192
c_mod, 192
cancel, 167
channels, 49
circle, 15, 31
clip, 154
Clock, 29
close, 77
CMY, 5
col, 176
collect, 167
complex128, 51
complex192, 51
complex256, 51
complex64, 51
concatenate, 62, 63
conj, 123
conjugate, 128
convert, 24
copy, 23, 61
corrcoef, 100
count_nonzero, 75
cpu_count, 225
cpu_freq, 225
cpu_percent, 226, 232
cpu_times, 225, 233
CPU のクロック周波数, 225
CPU の構成, 225
CPU 使用率, 226
CPU 時間, 225
crop, 22
CSV, 129
ctypes, 207
cvtColor, 6, 10
Cython, 203

datetime64, 149
delete, 66
Derivative, 170
destroyAllWindows, 2
det, 127, 176

diag, 126
diff, 73, 169
digest, 201
digitize, 94
disk_io_counters, 228
disk_partitions, 227
disk_usage, 228
divmod, 191
doit, 170, 178
dot, 124, 126, 177
Draw, 25
drawMarker, 18
dsolve, 172
dtype, 52
dump, 236
duration_seconds, 49

E, 166
Eel, 45
eig, 127
eigenvals, 177
eigenvects, 177
Ellipse, 145
ellipse, 15, 16, 26, 30
EPS, 20, 21
Eq, 172
evalf, 180
expand, 23, 166
expand_dims, 68
export, 50

f_divmod, 191
f_mod, 192
f_mod_2exp, 193
factor, 167
factorint, 179
fadeout, 37
FFmpeg, 49
fft, 119
fftfreq, 119
figsize, 122
Figure, 77, 85
figure, 77, 110, 122
fill, 26, 125
fillConvexPoly, 17
fillPoly, 16
finfo, 54
flatten, 59
flip, 60
float128, 51
float16, 51
float32, 51
float64, 51
float96, 51
Font, 32
font_manager, 89
FontProperties, 89
format, 20
frame_rate, 49
free_symbols, 165
from_mp3, 49
from_wav, 49
fromarray, 141
frombuffer, 134
full, 125
func, 165
Function, 165

gca, 86, 88
gcd, 194
gcext, 195
gcf, 86
get, 2, 30
get_array_of_samples, 49
get_busy, 37
get_context, 188
get_fonts, 32
get_height, 30, 36
get_length, 38
get_pos, 37
get_volume, 37, 38
get_width, 30, 36
GIF, 20
GMP, 186
gmpy2, 186
GRAY, 5
grid, 79
Group, 42

h2z, 239, 240
hashlib, 201
hexdigest, 201
hira2hkata, 239
hist, 101
hlines, 77
hstack, 63, 175
HSV, 5, 8

I, 166
ICNS, 20
ICO, 20
identity, 125
ifft, 119
iinfo, 54
IM, 20
imag, 123
Image, 141
image, 20
Image.BICUBIC, 22
Image.BILINEAR, 22
Image.BOX, 22
Image.FLIP_LEFT_RIGHT, 23
Image.FLIP_TOP_BOTTOM, 23
Image.HAMMING, 22
Image.LANCZOS, 22
Image.NEAREST, 22
Image.ROTATE_180, 23
Image.ROTATE_270, 23
Image.ROTATE_90, 23
ImageDraw, 25
imread, 3
imshow, 2, 140
imwrite, 3
in32, 51
inf, 55
info, 20
init, 28, 46
init_printing, 222
inner, 126
input, 220
insert, 64
int16, 51
int64, 51
int8, 51
Integral, 170
integrate, 170
inv, 127, 176
invert, 193
IPython, 216, 223
IQR, 108
is_divisible, 194
is_プロパティ, 169
iscomplex, 123
isfinite, 55
isinf, 55
isnan, 55
ISO8601, 149
isprime, 179
isreal, 123
jaconv, 239
JPEG, 20
JPEG2000, 20
JSON, 236
Jupyter, 216
Jupyter Notebook, 216
JupyterLab, 216
KEYDOWN, 36
KEYUP, 36
Lab, 5, 8
lambdify, 181
latex, 184
lcm, 194
legend, 77, 79
limit, 169
linalg, 127, 128
line, 13, 26, 31
lines, 31
linspace, 56
load, 21, 30, 31, 37, 132, 237
loadtxt, 131
logspace, 56
maen, 93
Markdown, 220
MathJax, 222
mathml, 184
matplotlib, 51, 76
Matrix, 175
matrix_rank, 128
matshow, 115
max, 54, 70, 71, 93
maximum, 152
MD5, 201, 202
memory_info, 231
merge, 7, 24
meshgrid, 108, 109
min, 54, 70, 93
minimum, 152
mode, 20
mod 演算, 191
MOUSEBUTTONDOWN, 36
MOUSEBUTTONUP, 36

MOUSEMOTION, 36
MP3, 37
mpc, 189
MPFR, 186
mpfr, 188
mpq, 187
mpz, 186
mpz_urandomb, 195
MSP, 20

nan, 55, 166
ndarray, 1, 52
ndim, 57
net_connections, 229, 233
net_io_counters, 228
new, 21
newaxis, 67
next_prime, 193, 198
NINF, 55
norm, 128
normal, 91
NpzFile, 133
Numba, 205
Number, 165
NumPy, 1, 51

object, 51
Ogg Vorbis, 37
ones, 125
oo, 166
open, 20
open_files, 233
OpenCV, 1

parse_expr, 164
passlib, 201
paste, 23
pause, 37
pcolor, 114
pconn, 233
pcputimes, 233
PCX, 20
pdsolve, 173
percentile, 94
permutation, 101
pi, 166
PID, 231
pids, 230
pie, 105
pieslice, 26
Pillow, 20
play, 37, 38
plot, 77, 184
plot3d, 184
plot_surface, 111
plot_wireframe, 110
pmem, 231
PNG, 20
point, 26
PolarAxesSubplot, 88
Poly3DCollection, 148
Polygon, 147
polygon, 26, 31
polylines, 16
popenfile, 233
PPM, 20
pprint, 175
precision, 188
prime, 179
primepi, 180
primerange, 179
primorial, 180
Process, 231
psutil, 225
putText, 17
PWM, 157
PyDub, 49
pygame, 28
PyInstaller, 213
pylab, 122
quantile, 94
QUIT, 30
quit, 30
r_, 64
rand, 91
randint, 91
random_state, 195
RandomState, 92
ratsimp, 168
ravel, 59
read, 2, 160
real, 123
rect, 30
Rectangle, 145
rectangle, 14, 26
RegularPolygon, 145

release, 2
render, 32
reshape, 58
resize, 4, 22
rewind, 37
RGB, 4, 5
roll, 60
rotate, 23, 34
round, 189
row, 176
rsolve, 174

sample_width, 49
save, 21, 31, 132, 185
savefig, 90
savetxt, 129
sawtooth, 157
scale, 34
scatter, 103
SciPy, 156
sconn, 229
scpufreq, 225
scputimes, 225
sdiskio, 228
sdiskpart, 227
sdiskusage, 228
seed, 92
series, 171
set_caption, 30
set_pos, 37
set_thetagrids, 89
set_title, 86
set_visible, 87, 89
set_volume, 37, 38
set_xlabel, 86, 110
set_xlim, 86
set_xticks, 86
set_ylabel, 86, 110
set_ylim, 86
set_yticks, 86
set_zlabel, 110
SGI, 20
SHA1, 201
SHA224, 201
SHA256, 201
SHA384, 201
SHA512, 201
shape, 4, 57, 176

show, 22, 77
shuffle, 101
simplify, 163
size, 20
snetio, 228
solve, 171
sort, 72
Sound, 38
sparse matrix, 125
SPIDER, 20
Spine, 87
spines, 87, 89
split, 7, 23, 143
Sprite, 40
sqrt, 54, 123
square, 156
sswap, 226
start, 46
std, 93
stop, 37, 38
subplot, 88
subplots, 81
subplots_adjust, 82
subs, 168
Sum, 178
sum, 93
Surface, 28
surface plot, 111
Surface オブジェクトのサイズ, 36
svmem, 226
swap_memory, 226
Symbol, 163, 165
symbols, 163
sympify, 164
SymPy, 162
SysFont, 32

T, 60, 127
t_divmod, 192
t_mod, 192
table, 117
terminate, 234
threshold, 10
thumbnail, 22
tich, 30
TIFF, 20
tile, 64
timedelta64, 149, 150

title, 77, 79, 86
tobytes, 134
transpose, 23, 127, 177
 uin32, 51
 uint16, 51
 uint64, 51
 uint8, 51, 141
 unicode, 51
 unique, 74
 unpause, 37
 update, 30
 urllib, 238
 urllib.parse.quote, 238
 urllib.parse.unquote, 238
var, 93, 163
vectorize, 136, 137
VideoCapture, 2
virtual_memory, 226
vlines, 77
vstack, 63, 175
waitKey, 2
wave, 159
WAV 形式, 159
WebP, 20
where, 69
width, 26
Wild, 178
write, 159
XBM, 20
xlabel, 77, 79, 86
xlim, 79, 86
xscale, 81
xticks, 80, 86
ylabel, 77, 79, 86
ylim, 79, 86
yscale, 81
yticks, 80, 86
z2h, 239, 240
zenhan, 239
zeros, 125
zeros_like, 125
zoo, 166
アスペクト比, 122
アニメーション GIF, 26
位相, 124
一様乱数, 91
イベントキュー, 28, 30
イベント種別, 30
色空間, 4, 5
因数分解, 167
閏年, 150
エルミート共役行列, 128
円グラフ, 105
算タワー, 190
音声の再生, 37
階級, 94
階差方程式, 174
回転, 33
拡縮, 33
拡張ユークリッドの互除法, 195
加色混合, 4
仮想記憶の状況, 226
カテゴリデータ, 75, 130, 131
加法混色, 4
カラーバーの表示, 116
カラーマップ, 111
関数のグラフ, 184
画像モード, 20
共役複素数, 123
共有ライブラリ, 207
極座標, 88
鋸歯状波, 157
近似値, 180
疑似素数, 198
疑似乱数, 195
逆行列, 176
行列, 124
行列式, 176
行列のランク, 128
区間, 94
矩形波, 156
グラフの枠を非表示にする方法, 87
グラフを画像ファイルとして保存, 90
グリッド線, 79
クリッピング, 160
グレゴリオ暦, 150
現在の通信状況, 229
減色混合, 5
減法混色, 5
降順, 72
固有値, 177
固有ベクトル, 177

合計, 93
最小公倍数, 194
最小値, 70, 93
最大公約数, 194
最大値, 70, 93
最頻値, 97
サウンドの合成, 223
サウンドの再生, 223
差分, 73
差分方程式, 174
3次元の散布図, 113
散布図, 103
色相, 9
式の型, 168
式の展開, 166
質的データ, 75
指定した区間で度数を集計する, 96, 102
シャッフル, 101
集計, 94
振幅スペクトル, 121
真理値, 51, 53
次元の拡大, 67
実行中のプロセス, 230
数式処理システム, 162
数値以外のデータの保存, 130
数値以外のデータの読み込み, 131
スカラ一値の型の変換, 61
ステレオサウンドの出力, 159
スペース行列, 125
スプライト, 40
スライス, 57
スライスオブジェクト, 66
スワップ領域, 226
正規乱数, 91
整列, 72
線形合同法, 195
線形代数, 124
選言, 69
全角 \leftrightarrow 半角変換, 239
漸化式, 174
素因数分解, 179
相関行列, 100
相関係数, 100
総和, 178
素数, 179
素数階乗, 180
素数の生成, 193
ソルト, 201
ソート, 72
対角行列, 126
対角成分, 126
対数軸, 81
対数正規分布, 97
タイムスタンプ, 149
タイムゾーン, 149
対話作業環境, 216
多倍長精度の数値演算, 186
代数方程式, 171
遅延実行, 170
チャネル, 4
チャネルの分解と合成, 7
定数, 166
転置, 177
ディスクのI/O状況, 228
ディスクの使用状況, 228
デジタル署名, 201
データの整列, 72
データの抽出, 68
統計, 93
頭部と引数列の取り出し, 165
度数調査, 94
内積, 126, 177
日本語の見出し・ラベル, 89
ネットワークのI/O状況, 228
ノコギリ波, 157
ノルム, 123
ノートブック, 216
ハイレゾ, 159
配列の形状, 57
配列の連結, 62
箱ひげ図, 107
汎関数, 170
パスワードクラック, 201
パスワード文字列の秘匿化, 201
パターンマッチ, 178
ハルス幅変調, 157
パーセント点, 94
パーティションの構成, 227
ヒストグラム, 101
否定, 69
標準偏差, 93
標本標準偏差, 93
標本分散, 93
ヒートマップ, 114
微分方程式, 172
フィルタ, 22

フォント名の取得, 32
複素共役行列, 128
複素数, 123
不偏標準偏差, 93
不偏分散, 93
フレームのサイズ, 4
フーリエ逆変換, 119
フーリエ変換, 119
プロードキャスト, 54
分位点, 94
分散, 93
プロセス ID, 231
平均, 93
平方根, 123
偏角, 124
ベクトル, 124, 177
ベクトル化, 138
ベクトルのノルム, 128
方形波, 156
棒グラフ, 76, 103
ポリゴン, 147
窓関数, 122
マーカー, 18
見出し行, 131
メッセージダイジェスト, 201
面プロット, 111
モジュロ演算, 191
約数の検査, 194
要素の置換, 70
ラスタライズ, 21
乱数状態オブジェクト, 195
量子化ビット数, 159
連言, 69
連立方程式, 172
レーダーチャート, 88
ワイヤフレーム, 110

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：2025年8月18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

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

● 連絡先

nkatsu2012@gmail.com

中村勝則